. . .

Introduction to Computational Data Science Using ScalaTion

. . .

John A. Miller

Department of Computer Science

University of Georgia

. . .

February 10, 2024

Brief Table of Contents

Contents

Preface

Applied Mathematics accelerated starting with the differential equations of Euler's analytical mechanics published in early 1700s [45, 117]. Over time increasingly accurate *mathematical models* of *natural phenomena* were developed. The models are scrutinized by how well they match empirical data and related models. Theories were developed that featured a collection of consistent, related models. In his theory of Universal Gravity [132], Newton argues the sufficiency of this approach, while others seek to understand the underlying substructures and causal mechanisms [117].

Data Science can trace its linage back to Applied Mathematics. One way to represent a mathematical model is as a function $f : \mathbb{R}^n \to \mathbb{R}$.

$$y \;=\; f(\mathbf{x}, \mathbf{b}) + \epsilon$$

This illustrates that a response variable y is functionally related to other predictive variables $\mathbf{x}$ (vector in bold font). Uncertainty in the relationship is modeled as a random variable ϵ (blue font) that follows some probability distribution.

Making useful predictions or even inferences that one product lasts longer than another product are clouded by this uncertainty. DeMoivre developed a limiting distribution for the Binomial Distribution. Laplace derived a central limit theorem that showed that the sample means from several distributions follow this same distribution. Gauss [180] studied this uncertainty and deduced a distribution for measurement errors from basic principles. This distribution is now known as the Gaussian or Normal distribution. Inferences such as which of two products has the longer expected lifetimes can now be made to a certain level of confidence. Gauss also developed the method of least squares estimation.

Momentum in using probability distributions to analyze data, fit parameters and make inferences under uncertainty lead to mathematical statistics emerging from applied mathematics in the late 1800s. In particular, Galton and Pearson collected and transformed statistical techniques into a mathematical discipline (e.g., Pearson correlation coefficient, method of moments estimation, p-value, Chi-square test, statistical hypothesis testing, principal component analysis). In the early 1900s, Gosset and Fisher expanded mathematical statistics (e.g., analysis of variance, design of experiments, maximum likelihood estimation, Student's t-distribution, F-distribution).

With the increasing capabilities of computers, the amount of data available for training models grew rapidly. This lead Computer Scientists into the fray with machine learning coined in 1959 and data mining beginning in the late 1980s. Machine Learning developed slowly over the decades until the invention of the back-propagation algorithm for neural networks in the mid 1980s lead to important advances. Data Mining billed itself as finding patterns in data. Databases are often utilized and data preprocessing is featured in the sense that mining through large amounts of data should be done with care.

With greater computing capabilities and larger amounts of data, statistics and machine learning are leaning toward each other: The emphasis is to develop of accurate, interpretable and explainable models for prediction, classification and forecasting. Data may also be clustered and simulation models that mimic phenomena or systems may be created. Training a model is typically done using an optimization algorithm (e.g., gradient descent) to minimize the errors in the model's predictions. These constitute the elements of data science.

This book is an introduction to data science that includes mathematical and computational foundations. It is divided into three parts: (I) Foundations, (II) Modeling, and (III) Simulation. A review of Optimization from the point of view of data science is included in the Appendix. The level of the book is College Junior through beginning Graduate Student. The ideal mathematical background includes Differential, Integral and Vector Calculus, Applied Linear Algebra, Probability and Mathematical Statistics. The following advanced topics may be found useful for Data Science: Differential Equations, Nonlinear Optimization, Measure Theory, Functional Analysis and Differential Geometry. Data Science also involves Computer Programming, Database Management, Data Structures and Algorithms. Advanced topics include Parallel Processing, Distributed Systems and Big Data frameworks (e.g., Hadoop and Spark). This book has been used in the Data Science I and Data Science II courses at the University of Georgia.

Chapter 1

Introduction to Data Science

1.1 Data Science

The field of Data Science can be defined in many ways. To its left is Machine Learning that emphasizes algorithms for learning, while to its right is Statistics that focuses on procedures for estimating parameters of models and determining statistical properties of those parameters. Both fields develop *models* to describe/predict reality based on one or more datasets. Statistics has a greater interest in making inferences or testing hypotheses based upon datasets. It also has a greater interest in fitting probability distributions (e.g., are the residuals normally or exponentially distributed).

The common thread is modeling. A model should be able to make *predictions* (where is the hurricane likely to make landfall, when will the next recession occur, etc.). In addition, it may be desirable for a model to enhance the *understanding* of the system under study and to address *what-if* type questions (perspective analytics), e.g., how will traffic flow improve/degrade if a light-controlled intersection is replaced with a round-about.

A **model** may be viewed as replacement for a real system, phenomena to process. A model will map inputs into outputs with the goal being that for a given input, the model will produce output that approximates the output that the real system would produce. In addition to inputs and outputs, some models include state information. For example, the output of a heat pump will depend if it is in the heating or cooling state (internally this determines the direction of flow of the refrigerant). Further, some types of models are intended to mimic the behavior of the actual system and facilitate believable animation. Examples of such models are simulation models. They support prescriptive analytics which enables changes to a system to tested on the model, before the often costly changes to the actual system are under taken.

Broad categories of modeling are dependent of the type output (also called response) of the model. When the response is treated as a continuous variable, a **predictive model** (e.g., regression) is used. If the goal is to forecast into the future (or there is dependency among the response values), a **forecasting model** (e.g., ARIMA) is used. When the response is treated as a categorical variable, a **classification model** (e.g., support vector machine) is used. When the response values are largely missing, a **clustering model** may be used. Finally, when values are missing from a data matrix, an **imputation model** (k-nearest neighbors) or **recommendation model** (e.g., low-rank approximation using singular value decomposition) may be used. **Dimensionality reduction** (e.g., principal component analysis) can be useful across categories.

Computational Data Science puts more emphasis on computational issues, such as optimization algorithms for used for learning. Mathematical derivations are provided for the *loss functions* that are used to train the models. Short Scala code snippets are provided to illustrate how the algorithms work. The Scala object-oriented, functional language allows the creation of coincide code that looks very much like the mathematical expressions. Modeling based on ordinary differential equations and simulation models are also provided.

The prerequisite material for data science includes Vector Calculus, Applied Linear Algebra and Calculus-based Probability and Statistics. Datasets can be stored as vectors and matrices, learning/parameter estimation often involves taking gradients, and probability and statistics are needed to handle uncertainty.

1.2 ScalaTion

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics.

In ScalaTion, the `modeling` package provides tools for performing data analytics. Datasets are becoming so large that statistical analysis or machine learning software should utilize parallel and/or distributed processing. Databases are also scaling up to handle greater amounts of data, while at the same time increasing their analytics capabilities beyond the traditional On-Line Analytic Processing (OLAP). ScalaTion provides many analytics techniques found in tools like MATLAB, R and Weka. The analytics component contains six types of tools: **predictors**, **classifiers**, **forecasters**, **clusterers**, **recommenders** and **reducers**. A trait is defined for each type.

To use ScalaTion, go to the Website `http://www.cs.uga.edu/~jam/scalation.html` and click on the most recent version of ScalaTion and follow the first three steps: download, unzip, build.

Current projects are targeting Big Data Analytics in four ways: (i) use of sparse matrices, (ii) parallel implementations using Scala's support for parallelism (e.g., `.par` methods, parallel collections and actors), (iii) distributed implementations using Akka, and (iv) high performance data stores including columnar databases (e.g., Vertica), document databases (e.g., MongoDB), graph databases (e.g., Neo4j) and distributed file systems (e.g., HDFS).

1.2.1 Package Structure

The package structure of ScalaTion is devided in four modules. Each modules can be independently compiled (e.g., package directories can be made removed or made inaccessible using the `chmod 000` command).

The core module of ScalaTion consists of the following packages:

1. `scalation` - general utilities for the rest of ScalaTion packages.
2. `mathstat` - vectors, matrices, tensors and basic statistics
3. `random` - random number and random variate generators
4. `scala2d` - basic UI widgets and 2D graphics

The intermediate module of ScalaTion consists of the following packages:

1. `animation` - general purpose animation code
2. `database` - basic implementations for relational and graph databases

The modeling module of ScalaTion consists of the following packages:

1. `calculus` - derivatives, gradients, Jacobians, Hessians and integrals
2. `modeling` - regression models with sub-packages for classification, clustering, neural networks, and time series
3. `optimization` - derivative-free, first-order, and second-order optimizers

The simulation module of ScalaTion consists of the following packages:

1. `dynamics` - differential equations: ODE and PDE solvers

2. `simulation` - multiple simulation engines

The `scala3d` package is under development.

1.2.2 Scala 3 Control Structures

This section gives the Scala 3 control structures along with their Python equivalents:

- `if`

```
    if x < y then                                  if x < y:
        x += 1                                         x += 1
    else if x > y then                             elsif x > y:
        y += 1                                         y += 1
    else                                           else:
        x += 1                                         x += 1
        y += 1                                         y += 1
    end if
```

The `else` and `end` are optional, as are the line breaks. Note, the `x += 1` shortcut simply means `x = x + 1` for both languages.

- `match`

```
    z = c match                                    match c:
    case '+' => x + y                                  case '+':
                                                           z = x + y
    case '-' => x - y                                  case '-':
                                                           z = x - y
    case '*' => x * y                                  case '*':
                                                           z = x * y
    case '/' => x / y                                  case '/':
                                                           z = x / y
    case _   => println ("not supported")              case -:
                                                           print ("not supported")
```

In Scala 3, the `case` may be indented like Python. Also an `end` may be added.

- `while`

```
    while x <= y do                                while x <= y:
        x += 0.5                                       x += 0.6
    end while
```

The `end` is optional, as are the line breaks.

- `for`

```
    for i <- 0 until 10 do                          for i in range (0, 10):
        a(i) = 0.5 * i~^2                               a[i] = 0.5 * i**2
    end for
```

The `end` is optional, as are the line breaks. Note: `for i <- 0 to 10 do` will include 10, while `until` will stop at 9. Both Scala and Python support other variaties of `for`. The `for-yield` collects all the computed values into `a`.

```
    val a = for i <- 0 until 10 yield 0.5 * i~^2
```

- `cfor`

```
    var i = 0
    cfor (i < 10, i += 1) {
        a(i) = 0.5 * i~^2
    } // cfor
```

  This `for` follows more of a C-style, provides improved efficiency and allows `return`s inside the loop. It is defined as follows:

```
    inline def cfor (pred: => Boolean, step: => Unit) (body: => Unit): Unit =
        while pred do { body; step }
    end cfor
```

- `try`

```
    try                                                    try:
       file = new File ("myfile.csv")                          x = 1 / 0
    catch                                                  except ZeroDivisionError:
       case ex: FileNotFound => println ("not found")          print ("division by zero")
    end try
```

  The `end` is optional and a `finally` clause is available. Both support a `finally` clause and Python provides a shortcut `with` statement that comes in handy for opening files and automatically closing them at the statement's end of scope.

- assign with `if`

```
    val y = if x < 1 then sqrt (x) else x~^2            y = sqrt (x) if x < 1 else x**2
```

  All Scala control structures return values and so can be used in assignment statements. Note, prefix `sqrt` with `math` for Python.

Note, the `end` tags are optional since Scala 3 uses *significant indentation* like Python.

1.2.3 Scala 3 Top-Level Functions

Both Scala 3 and Python support top level functions as well as methods inside classes. Here are functions to compute the length of the hypotenuse of a right triangle with lengths `a` and `b`.

```
   import scala.math.sqrt

   def hypotenuse (a: Double, b: Double): Double =
       sqrt (a ~^ 2 + b ~^ 2)
```

Optionally, an `end hypotenuse` may be added and is often useful for functions which include several lines of code. The Python code below is very similar, with the exception of the exponentiation operator `~^` for ScalaTion and `**` in Python. Outside of ScalaTion import `scalation.~^`. Both `Double` in Scala and `float` in Python indicate 64-bit floating point numbers.

```
import math

def hypotenuse (a: float, b: float) -> float =
    math.sqrt (a ** 2 + b ** 2)
```

The `dot` product operator on vectors is used extensively in data science. It multiplies all the elements in the two vectors and then sums the products. An implementation in ScalaTion is given followed by a similar implementation in Python that includes type annotations for improved readability and type checking.

```
import scalation.mathstat.VectorD

def dot (x: VectorD, y: VectorD): Double =
    (x * y).sum
```

```
import numpy as np

def dot (x: np.ndarray, y: np.ndarray) -> float:
    return float (np.sum (x * y))
```

Note, see the Chapter on Linear Algebra for more efficient implementations of `dot` product. Also, both `numpy.ndarray` and `VectorD` directly provide `dot` product.

```
val z = x dot y
```

```
z = x.dot (y)
```

In cases where the arguments are 2D arrays, `np.dot` is the same as matrix multiplication (`x @ y`) and for scalars it is simple multiplication (`x * y`). ScalaTion supports several forms of multiplication for both vectors and matrices (see the Linear Algebra Chapter).

Executable Top-Level Functions

Executable top-level functions can also be defined in similar ways in both Scala 3 and Python.

```
@main def hello (): Unit =
    val message = "Hello Data Science"
    println (s"message = $message")
```

```
def main () -> None:
    message = "Hello Data Science"
    print ("message = ", message)
```

1.2.4 Classes

Defining a `class` is a good way to combine a data structure with its natural operations. The class will consists of fields/attributes for maintaining data and methods for retrieving and updating the data.

An example of a class in Scala 3 is the `Complex` class that supports complex number (e.g., 2.1 + 3.2i) and operations on complex numbers such as the + method. Of course, the actual implementation provides many methods (see `scalation.mathstat.Complex`).

```
@param re  the real part (e.g., 2.1)
@param im  the imaginary part (e.g., 3.2)

class Complex (re: Double, im: Double = 0.0)
```

```
        extends Fractional [Complex] with Ordered [Complex]:

    def + (c: Complex): Complex = Complex (re + c.re, im + c.im)
```

Notice that second argument `im` provides a default value of `0.0`, so the class can be instantiated using either one or two arguments/parameters.

```
val c1 = new Complex (2.1, 3.2)
var c2 = new Complex (2.0)
```

Also observe that first variable cannot be reassigned as it is declared `val`, while the second variable `c2` can be as it is declared `var`. Finally, notice that the `Complex` class `extends` both `Fractional` and `Ordered`. These are *traits* that the class `Complex` inherits. Some of the functionality (e.g., method implementations) can be provided by the trait itself. The class must implement those that are not implemented or override the ones with implementations to customize their behavior, if need be. Classes can extend several traits (multiple inheritance), but may only extend one class (single inheritance).

Although Python already has a class called `Complex`, one could image coding one as follows:

```
class Complex:
    def __init__ (self, re: float, im: float = 0.0):
        self.re = re
        self.im = im

    def __add__ (self, c: Complex) -> Complex:
        return Complex (self.re + c.re, self.im + c.im)
```

Notice there are few differences: The constructor for Scala is any code at the top level of the class and arguments to the constructor are given in the class definition, while Python has an explicit constructor called `__init__`. Scala has an implicit reference to the instance object called `this`, while Python has an explicit reference to the instance object called `self`. Furthermore, naming the method `__add__` makes it so `+` can be used to add two complex numbers. Another difference (not shown here) is that fields/attributes as well as methods in Scala can be made internal using the `private` access modifier. In Python, a code convention of having the first character of an identifier be underscore (_) indicates that it should not be used externally.

1.2.5 Basic Types

The basic data-types in Scala are integer types: `Byte` (8 bits), `Short` (16), `Int` (32) and `Long` (64), floating point types: `Float` (32) and `Double` (64), character types: `Char` (single quotes) and `String` (double quotes), and `Boolean`.

Corresponding Python data types are integer types: `int` (unlimited), floating point types: `float32` (32) and `float` (64), complex (128), character types: `str` (single or double quotes), and `bool`.

There are many operators that can be applied to these data-types, see `https://docs.scala-lang.org/tour/operators.html` for the precedence of the operators. ScalaTion adds a few itself such as `~^` for exponentiation. Also, ScalaTion provides complex numbers via the `Complex` class in the `mathstat` package.

1.2.6 Collection Types

The most commonly used collection types in Scala are `Array`, `ArrayBuffer`, `Range`, `List`, `Map`, `Set`, and `Tuple`. The Python rough equivalents (in lower case) are on the right (`Map` becomes `dict`).

```
val a = Array.ofDim [Double] (10)              a = np.zeros (10)
val b = ArrayBuffer (2, 3, 3)
val r = 0 until 10                             r = range (10)
val l = List (2, 3, 3)                         l = [2, 3, 3]
val m = Map ("no" -> 0, "yes" -> 1)            m = {"no": 0, "yes": 1}}
val s = Set (1, 2, 3, 5, 7)                    s = {1, 2, 3, 5, 7}
val t = (firstName, lastName)                  t = (firstName, lastName)
```

For more collection types consult their documentation: `https://scala-lang.org/api/3.x/` for Scala and `https://docs.python.org/3/library/collections.html` for Python. Scala typically has mutable and immutable versions of most collection types.

1.2.7 ScalaTion: Vectors, Matrices and Tensors

It is easy to make vectors, matrices and tensors in ScalaTion, via the `VectorD`, `MatrixD`, and `TensorD` classes provided in the `mathstat` package. The following is a vector (1D array) consisting of 9 `Double`s, corresponding to `float` in Python.

```
val y = VectorD (1, 2, 4, 7, 9, 8, 6, 5, 3)
```

A matrix is a 2D array, that in this case is a 9-by-2 matrix holding two variables/features x_0 and x_1 in columns of the matrix.

```
//                       col0 col1
val x = MatrixD ((9, 2), 1,   8,    // row 0
                         2,   7,    // row 1
                         3,   6,    // row 2
                         4,   5,    // row 3
                         5,   5,    // row 4
                         6,   4     // row 5
                         7,   4,    // row 6
                         8,   3,    // row 7
                         9,   2)    // row 8
```

As practice, try to find a vector `b` of length/dimension 2, so that `x * b` is close to `y`. The `*` operator does matrix-vector multiplication. It takes the dot product of the i^{th} row of matrix `x` and vector `b` to obtain the i^{th} element in the resulting vector.

In Python, `numpy` arrays can be used to do the same thing. The following 1D array can represent a vector. Note the use of period “1.” to make the elements be floating point numbers. The “D” indicates such for ScalaTion.

```
y = np.array ([1., 2., 4., 7., 9., 8., 6., 5., 3.])
```

Using double square brackets “[[”, numpy can be used to represent matrices. Each “[...]” corresponds to a row in the matrix.

```
#               col0 col1
x = np.array ([[1., 8.],          # row 0
               [2., 7.],          # row 1
               [3., 6.],          # row 2
```

```
                    [4., 5.],           # row 3
                    [5., 5.],           # row 4
                    [6., 4.],           # row 5
                    [7., 4.],           # row 6
                    [8., 3.],           # row 7
                    [9., 2.]])          # row 8
```

Matrix multiplication-vector is similar in Python `x.dot (b)`.

The following is a ScalaTion tensor (3D array).

```
    // 4 rows, 3 columns, 2 sheets - x_ijk
    //                                                      row  columns  sheet
    val z = TensorD ((4, 3, 2), 1,  2,  3,               //  0   0,1,2    0
                                4,  5,  6,               //  1   0,1,2    0
                                7,  8,  9,               //  2   0,1,2    0
                               10, 11, 12,               //  3   0,1,1    0

                               13, 14, 15,               //  0   0,1,2    1
                               16, 17, 18,               //  1   0,1,2    1
                               19, 20, 21,               //  2   0,1,2    1
                               22, 23, 24)               //  3   0,1,2    1
```

In Python, the above tensor can be defined as a 3D numpy array. Each row and column position has two sheet values, e.g., ”[1., 13.]”.

```
    #                 column 0     column 1     column 2
    z = np.array ([[[1.,  13.], [2.,  14.], [3.,  15.]],      # row 0
                   [[4.,  16.], [5.,  17.], [6.,  18.]],      # row 1
                   [[7.,  19.], [8.,  20.], [9.,  21.]],      # row 2
                   [[10., 22.], [11., 23.], [12., 24.]]])     # row 3
```

Vectors, matrices and tensors will discussed in greater detail in the Linear Algebra Chapter.

1.3 A Data Science Project

The orientation of this textbook is that of developing modeling techniques and the understanding of how to apply them. A secondary goal is to explain the mathematics behind the models in sufficient detail to understand the algorithms implementing the modeling techniques. Concise code based on the mathematics is included and explained in the textbook. Readers may drill down to see the actual ScalaTion code.

The textbook is intended to facilitate trying out the modeling techniques as they are learned and to support a group-based term project that includes the following ten elements. The term project is to culminate in a presentation that explains what was done concerning these ten elements.

1. **Problem Statement**. Imagine that your group is hired as consultants to solve some problem for a company or government agency. The answers and recommendations that your group produces should not depend solely on prior knowledge, but rather on sophisticated analytics performed on multiple large-scale datasets. In particular, the study should be focused and the **purpose of the study** should clearly stated. What not to do: The following datasets are relevant to the company, so we ran them through an analytics package (e.g., R) and obtained the following results.

2. **Collection and Description of Datasets**. To reduce the chances of results being relevant only to a single dataset, multiple datasets should be used for the study (at least two). Explanation must be given to how each dataset relates to the other datasets as well as to the problem statement. When a dataset in the form of a matrix, **metadata** should be collected for each column/variable. In some cases the response column(s)/variable(s) will be obvious, in others it will depend on the purpose of the study. Initially, the result of columns/variables may be considered as features that may be useful in predicting responses. Ideally, the datasets should loaded into a well-designed database. ScalaTion provides two high-performance database systems: a **relational database system** and a **graph database system** in `scalation.database.table` and `scalation.database.graph`, respectively.

3. **Data Preprocessing Techniques Applied**. During the preprocessing phase (before the modeling techniques are applied), the data should be cleaned up. This includes elimination of features with zero variance or too many missing values, as well as the elimination of key columns (e.g., on the **training data**, the `employee-id` could perfectly predict the salary of an employee, but is unlikely to be of any value in making predictions on the **test data**). For the remaining columns, strings should be converted to integers and imputation techniques should be used to replace missing values.

4. **Visual Examination**. At this point, **Exploratory Data Analysis** (EDA) should be applied. Commonly, one column of a dataset in the combined data matrix will be chosen as the response column, call it the response vector $\mathbf{y}$, and the rest of the columns that remain after preprocessing form m-by-n data matrix X. In general models are of the form

$$y = f(\mathbf{x}) + \epsilon \tag{1.1}$$

where f is function mapping feature vector $\mathbf{x}$ into a predicted value for response y. The last term may be viewed as *random error* ϵ. In an ideal model, the last term will be error (e.g., white noise). Since most models are approximations, technically the last term should be referred to as a **residual** (that which is not explained by the model). During exploratory data analysis, the value of $\mathbf{y}$, should be plotted against each feature/column $\mathbf{x}_{:j}$ of data matrix X. The relationships between the columns should be examined by computing a **correlation matrix**. Two columns that are very highly correlated are supplying redundant information, and typically, one should be removed. For a regression type problem, where y is treated as continuous random variable, a *simple linear regression model* should be created for each feature x_j,

$$y \ = \ b_0 \ + \ b_1 x_j \ + \ \epsilon \tag{1.2}$$

where the parameters $\mathbf{b} = [b_0, b_1]$ are to be estimated. The line generated by the model should be plotted along with the $\{(x_{ij}, y_i)\}$ data points. Visually, look for patterns such white noise, linear relationship, quadratic relationship, etc. Plotting the residuals $\{(x_{ij}, \epsilon_i)\}$ will also be useful. One should also create Histograms and Box-Plots for each variable as well as consider removing outliers.

5. **Modeling Techniques Chosen**. For every type of modeling problem, there is the notions of a `NullModel`: For prediction it is guess the mean, i.e., given a feature vector $\mathbf{z}$, predict the value $\mathbb{E}[y]$, regardless of the value of $\mathbf{z}$. The **coefficient of determination** R^2 for such models will be zero. If a more sophisticated model cannot beat the `NullModel`, it is not helpful in predicting or explaining the phenomena. Projects should include four classes of models: (i) `NullModel`, (ii) simple, easy to explain models (e.g., Multiple Linear Regression), (iii) complex, performant models (e.g., Quadratic Regression, Extreme Learning Machines) (iv) complex, time-consuming models (e.g., Neural Networks). If classes (ii-iv) do not improve upon class (i) models, new datasets should be collected. If this does not help, a new problem should be sought. On the flip side, if class (ii) models are nearly perfect (R^2 close to 1), the problem being addressed may be too simple for a term project. At least one modeling technique should be chosen from each class.

6. **Explanation of Why Techniques Were Chosen**. As a consultant to a company, a likely question will be, "why did you chose those particular modeling techniques"? There are an enormous number of possible modeling techniques. Your group should explain how the candidate techniques were narrowed down and ultimately how the techniques were chosen. A review of how well the selected modeling techniques worked, as well as suggested changes for future work, should be given at the end of the presentation.

7. **Feature Selection**. Although feature selection can occur during multiple phases in a modeling study, an overview should be given at this point in the presentation. Explain which features were eliminated and why they were eliminated prior to building the models. During model building, what features were eliminated, e.g., using forward selection, backward elimination, Lasso Regression, dimensionality reduction, etc. Also address and quantify the relative *importance* of the remaining features. Explain how features that *categorical* are handled.

8. **Reporting of Results**. First the experimental setup should be described in sufficient detail to facilitate **reproducibility** of your results. One way to show overall results is to plot predicted responses $\hat{\mathbf{y}}$ and actual responses $\mathbf{y}$ versus the instance index $i = 0$ until m. Reports are to include the Quality of Fit (QoF) for the various models and datasets in the form of tables, figures and explanation of the results. Besides the overall model, for many modeling techniques the importance/significance of model parameters/variables may be assessed as well. Tables and figures must include descriptive captions and color/shape schemes should be consistent across figures.

9. **Interpretation of Results**. With the results clearly presented, they need to be given insightful interpretations. What are the ramifications of the results? Are the modeling techniques useful in making predictions, classifications or forecasts?

10. **Recommendations of Study**. The organization that hired your group would like some take home messages that may result in improvements to the organization (e.g., what to produce, what processes to adapt, how to market, etc.). A brief discussion of how the study could be improved (possibly leading to further consulting work) should be given.

1.4 Additional Textbooks

More detailed development of this material can be found in textbooks on statistical learning, such as

- "An Introduction to Statistical Learning" (ISL) [85]
- "The Elements of Statistical Learning" (ESL) [72]
- "Mathematics for Machine Learning" (MML) [37]

See Table 1.1 for a mapping between the chapters in the four textbooks.

Table 1.1: Source Material Chapter Mappings

Topic	ScalaTion	**ISL**	**ESL**	**MML**
Linear Algebra	Ch. 2	-	-	Ch. 2-5
Probability	Ch. 3	-	-	Ch. 6
Data Management	Ch. 4	-	-	-
Data Preprocessing	Ch. 5	-	-	-
Prediction	Ch. 6	Ch. 3, 5, 6	Ch. 3	Ch 8-9
Classification	Ch. 7	Ch. 2, 5, 8	Ch. 4, 12, 13, 15	Ch. 8.5
Classification: Continuous	Ch. 8	Ch. 4, 8, 9	Ch. 4, 12, 13, 15	Ch. 12
Generalized Linear Models	Ch. 9	-	-	-
Nonlinear Models/Neural Networks	Ch. 10	Ch. 7	Ch. 11	-
Time Series/Temporal Models	Ch. 11	-	-	-
Multivariate Time Series Models	Ch. 12	-	-	-
Dimensionality Reduction	Ch. 13	Ch. 6, 10	Ch. 14	Ch. 10
Clustering	Ch. 14	Ch. 10	Ch. 14	-
Simulation Foundations	Ch. 15	-	-	-
State Space Models	Ch. 16	-	-	-
Event-Oriented Models	Ch. 17	-	-	-
Process-Oriented Models	Ch. 18	-	-	-
Simulation Output Analysis	Ch. 19	-	-	-
Optimization in Data Science	Appendix A	-	- -	Ch. 7
Graph Databases and Analytics	Appendix C	-	-	-

The next two chapters serve as quick reviews of the two principal mathematical foundations for data science: linear algebra and probability.

Part I

Foundations

Chapter 2

Linear Algebra

Data science and analytics make extensive use of *linear algebra*. For example, let y_i be the income of the i^{th} individual and x_{ij} be the value of the j^{th} predictor/feature (*age*, *education*, etc.) for the i^{th} individual. The responses (outcomes of interest) are collected into a vector $\mathbf{y}$, the values for predictors/features are collected in a matrix X and the parameters/coefficients $\mathbf{b}$ are fit to the data.

2.1 Linear System of Equations

The study of linear algebra starts with solving systems of equations, e.g.,

$$\begin{aligned} y_0 &= x_{00}b_0 + x_{01}b_1 \\ y_1 &= x_{10}b_0 + x_{11}b_1 \end{aligned}$$

This linear system has two equations with two variables having unknown values, b_0 and b_1. Such linear systems can be used to solve problems like the following: Suppose a movie theatre charges 10 dollars per child and 20 dollars per adult. The evening attendance is 100, while the revenue is 1600 dollars. How many children (b_0) and adults (b_1) were in attendance?

$$\begin{aligned} 100 &= 1b_0 + 1b_1 \\ 1600 &= 10b_0 + 20b_1 \end{aligned}$$

The solution is $b_0 = 40$ children and $b_1 = 60$ adults.

In general, linear systems may be written using matrix notation.

$$\mathbf{y} \;=\; X\mathbf{b} \tag{2.1}$$

where $\mathbf{y}$ is an m-dimensional vector, X is an m-by-n dimensional matrix and $\mathbf{b}$ is an n-dimensional vector.

2.2 Matrix Inversion

If the matrix is of full rank with $m = n$, then the unknown vector $\mathbf{b}$ may be uniquely determined by multiplying both sides of the equation by the *inverse* of X, X^{-1}

$$\mathbf{b} \;=\; X^{-1}\mathbf{y} \tag{2.2}$$

Multiplying matrix X and its inverse X^{-1}, $X^{-1}X$ results in an n-by-n identity matrix $I_n = [\mathbb{1}_{i=j}]$, where the indicator function $\mathbb{1}_{i=j}$ equals 1 when $i = j$ and 0 otherwise.

A faster and more numerically stable way to solve for $\mathbf{b}$ is to perform *Lower-Upper (LU) Factorization.* This is done by factoring matrix X into lower L and upper U triangular matrices.

$$X = LU \tag{2.3}$$

Then $LU\mathbf{b} \;=\; \mathbf{y}$, so multiplying both sides by L^{-1} gives $U\mathbf{b} = L^{-1}\mathbf{y}$. Taking an augmented matrix

$$\left[\begin{array}{cc|c} 1 & 3 & 1 \\ 2 & 1 & 7 \end{array}\right]$$

and performing row operations to make it upper right triangular has the effect of multiplying by L^{-1}. In this case, the first row multiplied by -2 is added to second row to give.

$$\left[\begin{array}{cc|c} 1 & 3 & 1 \\ 0 & -5 & 5 \end{array}\right]$$

From this, backward substitution can be used to determine $b_1 = -1$ and then that $b_0 = 4$, i.e.,

$$\mathbf{b} \;=\; \begin{bmatrix} 4 \\ -1 \end{bmatrix}$$

In cases where $m > n$, the system may be overdetermined, and no solution will exist. Values for $\mathbf{b}$ are then often determined to make $\mathbf{y}$ and $X\mathbf{b}$ agree as closely as possible, e.g., minimize absolute or squared differences.

Vector notation is used in this book, with vectors shown in boldface and matrices in uppercase. Note, matrices in ScalaTion are in lowercase, since by convention, uppercase indicates a type, not a variable. ScalaTion supports vectors and matrices in its `mathstat` package. A commonly used operation is the dot (inner) product, $\mathbf{x} \cdot \mathbf{y}$, or in ScalaTion, `x dot y`.

2.3 Vector

A *vector* may be viewed a point in multi-dimensional space, e.g., in three space, we may have

$$\begin{array}{lclcl} \mathbf{x} & = & [x_0,\ x_1,\ x_2] & = & [0.57735,\ 0.55735,\ 0.57735] \\ \mathbf{y} & = & [y_0,\ y_1,\ y_2] & = & [1.0,\ 1.0,\ 0.0] \end{array}$$

where $\mathbf{x}$ is a point on the unit sphere and $\mathbf{y}$ is a point in the plane determined by the first two coordinates.

2.3.1 Vector Addition and Subtraction

Vectors may be added $(\mathbf{x} + \mathbf{y})$ and subtracted $(\mathbf{x} - \mathbf{y})$. For example, $[1, 2] + [3, 4] = [4, 6]$.

2.3.2 Element-wise Multiplication and Division

Vectors may be multiplied element-by-element (like a Hadamard product) $(\mathbf{x} * \mathbf{y})$, and divided element-by-element $(\mathbf{x}/\mathbf{y})$. These operations are also supported when one of the arguments is a *scalar.*

2.3.3 Vector Dot Product

A particularly important operation, the *dot product* (or inner product) of two vectors is simply the sum of the products of their elements.

$$\mathbf{x} \cdot \mathbf{y} \ = \ \sum_{i=0}^{n-1} x_i y_i \ = \ 1.1547 \tag{2.4}$$

Note, the inner product applies more generally, e.g., $\langle \mathbf{x}, \mathbf{y} \rangle$ may be applied when $\mathbf{x}$ and $\mathbf{y}$ are infinite sequences or functions. See the exercises for the definition of an inner product space.

2.3.4 Norm

The *norm* of a vector is its length. Assuming Euclidean distance, the norm is

$$\|\mathbf{x}\| \ = \ \sqrt{\sum_{i=0}^{n-1} x_i^2} \ = \ 1 \tag{2.5}$$

The norm of $\mathbf{y}$ is $\sqrt{2}$. If θ is the angle between the $\mathbf{x}$ and $\mathbf{y}$ vectors, then the dot product is the product of their norms and the cosine of the angle.

$$\mathbf{x} \cdot \mathbf{y} \ = \ \|\mathbf{x}\| \|\mathbf{y}\| \, cos(\theta) \tag{2.6}$$

Thus, the cosine of θ is,

$$\cos(\theta) \ = \ \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \ = \ \frac{1.1547}{1 \cdot \sqrt{2}} \ = \ 0.8165$$

so the angle $\theta \ = \ .616$ radians. Vectors $\mathbf{x}$ and $\mathbf{y}$ are *orthogonal* if the angle $\theta \ = \ \pi/2$ radians (90 degrees).

In general there are ℓ^p norms. The two that are used here are the ℓ^2 norm $\|\mathbf{x}\| \ = \ \|\mathbf{x}\|_2$ (Euclidean distance) and the ℓ^1 norm $\|\mathbf{x}\|_1$ (Manhattan distance).

$$\|\mathbf{x}\|_1 \;=\; \sum_{i=0}^{n-1} |x_i| \tag{2.7}$$

Vector notation facilitates concise mathematical expressions. Many common statistical measures for populations or samples can be given in vector notation. For an m dimensional vector (m-vector) the following may be defined.

$$\begin{aligned}
\mu(\mathbf{x}) &= \mu_{\mathbf{x}} &&= \frac{\mathbf{1}\cdot\mathbf{x}}{m} \\
\sigma^2(\mathbf{x}) &= \sigma^2_{\mathbf{x}} &&= \frac{(\mathbf{x} - \mu_{\mathbf{x}})\cdot(\mathbf{x} - \mu_{\mathbf{x}})}{m} \\
& &&= \frac{\mathbf{x}\cdot\mathbf{x}}{m} - \mu^2_{\mathbf{x}} \\
\sigma(\mathbf{x},\mathbf{y}) &= \sigma_{\mathbf{x},\mathbf{y}} &&= \frac{(\mathbf{x} - \mu_{\mathbf{x}})\cdot(\mathbf{y} - \mu_{\mathbf{y}})}{m} \\
& &&= \frac{\mathbf{x}\cdot\mathbf{y}}{m} - \mu_{\mathbf{x}}\,\mu_{\mathbf{y}} \\
\rho(\mathbf{x},\mathbf{y}) &= \rho_{\mathbf{x},\mathbf{y}} &&= \frac{\sigma_{\mathbf{x},\mathbf{y}}}{\sigma_{\mathbf{x}}\sigma_{\mathbf{y}}}
\end{aligned}$$

which are the population *mean*, *variance*, *covariance* and *correlation*, respectively.

The size of the population is m, which corresponds to the number of elements in the vector. A vector of all ones is denoted by $\mathbf{1}$. For an m-vector $\|\mathbf{1}\|^2 = \mathbf{1}\cdot\mathbf{1} = m$. Note, the sample mean uses the same formula, while the sample variance and covariance divide by $m-1$, rather than m (sample indicates that only some fraction of population is used in the calculation).

Vectors may be used for describing the motion of an object through space over time. Let $\mathbf{u}(t)$ be the location of an object (e.g., golf ball) in three dimensional space $\mathbb{R}^3$ at time t,

$$\mathbf{u}(t) \;=\; [x(t), y(t), z(t)]$$

To describe the motion, let $\mathbf{v}(t)$ be the velocity at time t, and $\mathbf{a}$ be the constant acceleration, then according to Newton's Second Law of Motion,

$$\mathbf{u}(t) \;=\; \mathbf{u}(0) + \mathbf{v}(0)\,t + \frac{1}{2}\mathbf{a}\,t^2$$

The time varying function $\mathbf{u}(t)$ over time will show the trajectory of the golf ball.

2.3.5 Vector Operations in ScalaTion

Vector operations are implemented by multiple classes, such as the `VectorD` class.

```
  @param dim   the dimension/size of the vector
  @param v     the 1D array used to store vector elements

  class VectorD (val dim: Int,
                 private [mathstat] var v: Array [Double] = null)
        extends IndexedSeq [Double]
           with PartiallyOrdered [VectorD]
```

```
            with Cloneable [VectorD]
            with DefaultSerializable:
```

`VectorD` includes methods for size, indices, set, copy, filter, select, concatenate, vector arithmetic, power, square, reciprocal, abs, sum, mean variance, rank, cumulate, normalize, dot, norm, max, min, mag, argmax, argmin, indexOf, indexWhere, count, contains, sort and swap.

Table 2.1: Vector Arithmetic Operations

op	**vector op vector**	**vector op scalar**	**vector element op scalar**
+	def + (b: VectorD): VectorD	def + (s: Double): VectorD	def + (s: (Int, Double)): VectorD
+=	def += (b: VectorD): VectorD	def += (s: Double): VectorD	-
-	def - (b: VectorD): VectorD	def - (s: Double): VectorD	def - (s: (Int, Double)): VectorD
-=	def -= (b: VectorD): VectorD	def -= (s: Double): VectorD	-
*	def * (b: VectorD): VectorD	def * (s: Double): VectorD	def * (s: (Int, Double)): VectorD
*=	def *= (b: VectorD): VectorD	def *= (s: Double): VectorD	-
/	def / (b: VectorD): VectorD	def / (s: Double): VectorD	def / (s: (Int, Double)): VectorD
/=	def /= (b: VectorD): VectorD	def /= (s: Double): VectorD	-

2.4 Vector Calculus

Data science uses optimization to fit parameters in models, where for example a quality of fit measure (e.g., sum of squared errors) is minimized. Typically, gradients are involved. In some cases, the gradient of the measure can be set to zero allowing the optimal parameters to be determined by matrix factorization. For complex models, this may not work, so an optimization algorithm that moves in the direction opposite to the gradient can be applied.

2.4.1 Gradient Vector

Consider the following function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ of vector $\mathbf{u} = [x, y]$,

$$f(\mathbf{u}) \ = \ (x-2)^2 + (y-3)^2$$

For example, the functional value at the point $[3, 2]$, $f([3, 2]) = 1 + 1 = 2$ and at the point $[1, 1]$, $f([1, 1]) = 1 + 4 = 5$. The following *contour curves* illustrate the how the elevation of the function increases with distance from the point $[2, 3]$.

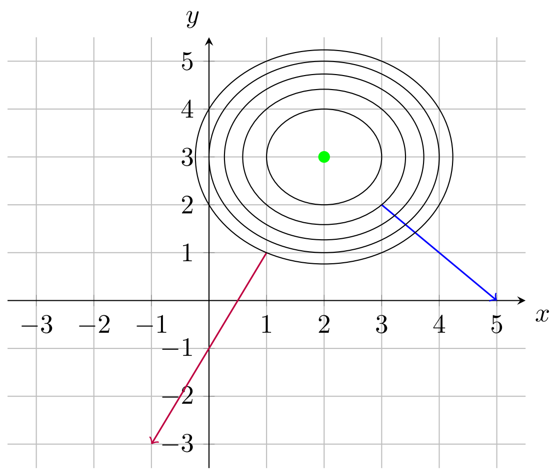


Figure 2.1: Contour Curves for Function f: elevation = 1, 2, 3, 4 and 5

The *gradient* of function f consists of a vector formed from the two partial derivatives.

$$\text{grad} f \ = \ \nabla f \ = \ \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

The gradient evaluated at point/vector $\mathbf{u} \in \mathbb{R}^2$ is

$$\nabla f(\mathbf{u}) \ = \ \left[\frac{\partial f}{\partial x}(\mathbf{u}), \frac{\partial f}{\partial y}(\mathbf{u})\right]$$

The gradient indicates the direction of steepest increase/ascent. For example, the gradient at the point $[3, 2]$, $\nabla f([3, 2]) = [2, -2]$ (in blue), while at $[1, 1]$, $\nabla f([1, 1]) = [-2, -4]$ (in purple).

A gradient's norm indicates the magnitude of the rate of change (or steepness). When the elevation changes are fixed (here they differ by one), the closeness of the contours curves also indicates steepness. Notice that the gradient vector at point $[x, y]$ is orthogonal to the contour curve intersecting that point.

By setting the gradient equal to zero, in this case

$$\frac{\partial f}{\partial x} = 2(x-2)$$
$$\frac{\partial f}{\partial y} = 2(y-3)$$

one may find the vector that *minimizes* function f, namely $\mathbf{u} = [2, 3]$ where $f = 0$. For more complex functions, repeatedly moving in the opposite direction to the gradient, may lead to finding a minimal value.

In general, the gradient (or gradient vector) of function $f : \mathbb{R}^n \to \mathbb{R}$ is

$$\boxed{\nabla f \;=\; \frac{\partial f}{\partial \mathbf{x}} \;=\; \left[\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_{n-1}}\right]} \tag{2.8}$$

or evaluated at point/vector $\mathbf{x} \in \mathbb{R}^n$ is

$$\boxed{\nabla f(\mathbf{x}) \;=\; \frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}) \;=\; \left[\frac{\partial f}{\partial x_0}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_{n-1}}(\mathbf{x})\right]} \tag{2.9}$$

In data science, it is often convenient to take the gradient of a dot product of two functions of $\mathbf{x}$, in which case the following product rule can be applied.

$$\nabla(f(\mathbf{x}) \cdot g(\mathbf{x})) \;=\; \nabla f(\mathbf{x}) \cdot g(\mathbf{x}) + f(\mathbf{x}) \cdot \nabla g(\mathbf{x}) \tag{2.10}$$

2.4.2 Jacobian Matrix

The *Jacobian Matrix* is an extension of the gradient vector to the case where the value of the function is multi-dimensional, i.e., $\mathbf{f} = [f_0, f_1, \ldots, f_{m-1}]$. In general, the Jacobian of function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ of vector $\mathbf{x} \in \mathbb{R}^n$ is

$$\mathbb{J}_{\mathbf{f}}(\mathbf{x}) \;=\; \left[\frac{\partial f_i}{\partial x_j}\right]_{0 \le i < m, 0 \le j < n} \;=\; \tag{2.11}$$

$$\begin{bmatrix} \nabla f_0(\mathbf{x}) \\ \nabla f_1(\mathbf{x}) \\ \ldots \\ \nabla f_{m-1}(\mathbf{x}) \end{bmatrix}$$

This follows the numerator layout where the functions correspond to rows (the opposite is called the denominator layout which is the transpose of the numerator layout).

Consider the following function $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ that maps vectors in $\mathbb{R}^2$ into other vectors in $\mathbb{R}^2$.

$$\mathbf{f}(\mathbf{x}) \;=\; [(x_0 - 2)^2 + (x_1 - 3)^2, (2x_0 - 6)^2 + (3x_1 - 6)^2]$$

The Jacobian of the function, $\mathbb{J}_{\mathbf{f}}(\mathbf{x})$, is

$$\begin{bmatrix} \dfrac{\partial f_0}{\partial x_0}, \dfrac{\partial f_0}{\partial x_1} \\ \dfrac{\partial f_1}{\partial x_0}, \dfrac{\partial f_1}{\partial x_1} \end{bmatrix}$$

Taking the partial derivatives gives the following Jacobian matrix.

$$\begin{bmatrix} 2x_0 - 4,\ 2x_1 - 6) \\ 4x_0 - 12,\ 6x_1 - 12 \end{bmatrix}$$

2.4.3 Hessian Matrix

While the gradient is a vector of first partial derivatives, the Hessian is a symmetric matrix of second partial derivatives. The *Hessian Matrix* of a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ of vector $\mathbf{x} \in \mathbb{R}^n$ is

$$\mathbb{H}_f(\mathbf{x}) \;=\; \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right]_{0 \le i < n, 0 \le j < n} \tag{2.12}$$

Consider the following function $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}$ that maps vectors in $\mathbb{R}^2$ into scalars in $\mathbb{R}$.

$$f(\mathbf{x}) \;=\; (2x_0 - 6)^2 + (3x_1 - 6)^2$$

The Hessian of the function, $\mathbb{H}_f(\mathbf{x})$, is

$$\begin{bmatrix} \dfrac{\partial^2 f}{\partial x_0^2}, \dfrac{\partial^2 f}{\partial x_0 \partial x_1} \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_0}, \dfrac{\partial^2 f}{\partial x_1^2} \end{bmatrix}$$

Taking the second partial derivatives gives the following Hessian matrix.

$$\begin{bmatrix} 4,\ 0 \\ 0,\ 6 \end{bmatrix}$$

Consider a differentiable function of n variables, $f : \mathbb{R}^n \to \mathbb{R}$. The points at which its gradient vector ∇f is zero are referred to as *critical points*. In particular, they may be local minima, local maxima or saddle points/inconclusive, depending on whether the Hessian matrix $\mathbb{H}$ is positive definite, negative definite, or otherwise. A symmetric matrix A is positive definite if $\mathbf{x}^\intercal A\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$ (alternatively, all of A's eigenvalues are positive). Note: a positive/negative semi-definite Hessian matrix may or may not indicate an optimal (minimal/maximal) point.

2.5 Matrix

A *matrix* may be viewed as a collection of vectors, one for each row in the matrix. Matrices may be used to represent *linear transformations*

$$\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m \tag{2.13}$$

that map vectors in R^n to vectors in $\mathbb{R}^m$. For example, in ScalaTion an m-by-n matrix A with $m = 3$ rows and $n = 2$ columns may be created as follows:

```
    val a = MatrixD ((3, 2), 1, 2,
                             3, 4,
                             5, 6)
```

to produce matrix A.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Matrix A will transform $\mathbf{u}$ vectors in $\mathbb{R}^2$ into $\mathbf{v}$ vectors in $\mathbb{R}^3$.

$$A\mathbf{u} \;=\; \mathbf{v} \tag{2.14}$$

For example,

$$A \begin{bmatrix} 1 \\ 2 \end{bmatrix} \;=\; \begin{bmatrix} 5 \\ 11 \\ 17 \end{bmatrix}$$

ScalaTion supports retrieval of row vectors, column vectors and matrix elements. In particular, the following access operations are supported.

A	=	`a`	=	matrix
$\mathbf{a}_i$	=	`a(i)`	=	row vector i
$\mathbf{a}_{:j}$	=	`a(?, j)`	=	column vector j
a_{ij}	=	`a(i, j)`	=	the element at row i and column j
$A_{i:k,j:l}$	=	`a(i to k, j to l)`	=	row and column matrix slice

Note, `i to k` does not include `k`. Common operations on matrices are supported as well.

2.5.1 Matrix Operation in ScalaTion

Matrix operations in ScalaTion are implemented in the `MatrixD` class for dense matrices.

```
    @param dim    the first (row) dimension of the matrix
    @param dim2   the second (column)dimension of the matrix
    @param v      the 2D array used to store matrix elements

    class MatrixD (val dim:  Int,
                   val dim2: Int,
                   private [mathstat] var v: Array [Array [Double]] = null):
```

Matrix Addition and Subtraction

Matrix addition `val c = a + b`

$$c_{ij} \;=\; a_{ij} + b_{ij} \tag{2.15}$$

and matrix subtraction `val c = a - b` are supported.

Matrix Multiplication

A frequently used operation in data science is matrix multiplication `val c = a * b`.

$$c_{ij} \;=\; \sum_{k=0}^{n-1} a_{ik} b_{kj} \;=\; \mathtt{a_i} \cdot \mathtt{b_{:j}} \tag{2.16}$$

Mathematically, this is written as $C = AB$. The ij element in matrix C is the vector dot product of the i^{th} row of A with the j^{th} column of B.

Matrix Transpose

The *transpose* of matrix A, written $A^\intercal$ (`val t = a.transpose` or `val t = a.`$\mathcal{T}$), simply exchanges the roles of rows and columns.

```
    def transpose: MatrixD =
        val a = Array.ofDim [Double] (dim2, dim)
        for j <- indices do
            val v_j = v(j)
            var i = 0
            cfor (i < dim2, i += 1) { a(i)(j) = v_j(i) }
        end for
        new MatrixD (dim2, dim, a)
    end transpose
```

Matrix Determinant

The *determinant* of square $(m = n)$ matrix A, written $|A|$ (`val d = a.det`), indicates whether a matrix is singular or not (and hence invertible), based on whether the determinant is zero or not.

Trace of a Matrix

The trace of matrix $A \in \mathbb{R}^{n \times n}$ is simply the sum of its diagonal elements.

$$\text{tr}(A) \;=\; \sum_{i=0}^{n-1} a_{ii} \tag{2.17}$$

In ScalaTion, the trace is computed using the `trace` method (e.g., `a.trace`).

Matrix Dot Product

ScalaTion provides several types of dot products on both vectors and matrices, two of which are shown below. The first method computes the usual dot product between two vectors. Note, the parameter `y` is generalized to take any vector-like data type.

```scala
    def dot (y: IndexedSeq [Double]): Double =
        var sum = 0.0
        for i <- v.indices do sum += v(i) * y(i)
        sum
    end dot
```

When relevant a n-vector (e.g., $\mathbf{x} \in \mathbb{R}^n$) may be viewed as an n-by-1 matrix (column vector), in which case $\mathbf{x}^\intercal$ would be viewed as an 1-by-n matrix (row vector). Consequently, dot product (and outer product) can be defined in terms of matrix multiplication and transpose operations.

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\intercal \mathbf{y} \qquad \text{dot (inner) product} \tag{2.18}$$

$$\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^\intercal \qquad \text{outer product} \tag{2.19}$$

The second method takes the dot product two matrices. The second method extends the notion of matrices and is an efficient way to compute $A^\intercal B = A \cdot B =$ `a.transpose * b = a dot b`.

```scala
    def dot (y: MatrixD): MatrixD =
        if dim2 != y.dim then
            flaw ("dot", s"matrix dot matrix - incompatible cross dimensions:
                              dim2 = $dim2, y.dim = ${y.dim}")

        val a = Array.ofDim [Double] (dim, y.dim)
        for ii <- 0 until dim by TSZ do
            for jj <- 0 until y.dim2 by TSZ do
                for kk <- 0 until dim2 by TSZ do
                    val k2 = math.min (kk + TSZ, dim2)

                    for i <- ii until math.min (ii + TSZ, dim) do
                        val v_i = v(i); val a_i = a(i)
                        for j <- jj until math.min (jj + TSZ, y.dim2) do
                            val y_j = y.v(j)
                            var sum = 0.0
                            var k = kk
                            cfor (k < k2, k += 1) { sum += v_i(k) * y_j(k) }
                            a_i(j) += sum
                        end for
                    end for

                end for
            end for
        end for
        new MatrixD (dim, y.dim, a)
    end dot
```

2.6 Matrix Factorization

Many problems in data science involve matrix factorization to for example solve linear systems of equations or perform Ordinary Least Squares (OLS) estimation of parameters. ScalaTion supports several factorization techniques, including the techniques shown in Table 2.2

Table 2.2: Matrix Factorization Techniques

Factorization	**Factors**	**Factor 1**	**Factor 2**	**Class**
LU	$A = LU$	lower left triangular	upper right triangular	`Fac_LU`
Cholesky	$A = LL^{\intercal}$	lower left triangular	its transpose	`Fac_Cholesky`
QR	$A = QR$	orthogonal	upper right triangular	`Fac_QR`
SVD	$A = U\Sigma V^{\intercal}$	orthogonal	diagonal, orthogonal	`Fac_SVD`
Eigen	$A = Q\Lambda Q^{-1}$	eigenvectors	diagonal, inverse eigen	`Fac_Eigen`

These algorithms are faster or more numerically stable than algorithms for matrix inversion. See the Prediction chapter to see how matrix factorization is used in Ordinary Least Squares estimation.

2.6.1 Eigenvalues and Eigenvectors

Consider the following matrix A and two vectors $\mathbf{x} = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $\mathbf{z} = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$.

$$\begin{bmatrix}2 & 0\\0 & 3\end{bmatrix}$$

Multiplying A and $\mathbf{x}$ yields $\left[\begin{smallmatrix}2\\0\end{smallmatrix}\right]$, while multiplying A and $\mathbf{z}$ yields $\left[\begin{smallmatrix}0\\3\end{smallmatrix}\right]$. Thus, letting $\lambda_0 = 2$ and $\lambda_1 = 3$, we see that $A\mathbf{x} = \lambda_0\mathbf{x}$ and $A\mathbf{z} = \lambda_1\mathbf{z}$. In general, a matrix $A^{n\times n}$ of rank r will have r non-zero eigenvalues λ_i with corresponding eigenvector $\mathbf{x^{(i)}}$ such that

$$A\mathbf{x^{(i)}} \;=\; \lambda_i\mathbf{x^{(i)}} \tag{2.20}$$

In other words, there will be r unit eigenvectors, for which multiplying by the matrix simply rescales the eigenvector $\mathbf{x^{(i)}}$ by its eigenvalue λ_i. The same will happen for any non-zero vector in alignment with one of the r unit eigenvectors.

Given an eigenvalue λ_i, an eigenvector may be found by noticing that

$$A\mathbf{x^{(i)}} - \lambda_i\mathbf{x^{(i)}} \;=\; [A - \lambda_i]\,\mathbf{x^{(i)}} \;=\; \mathbf{0} \tag{2.21}$$

Any vector in the nullspace of the matrix $A - \lambda_i I$ is an eigenvector corresponding to λ_i. Note, if the above equation is transposed, it is called a left eigenvalue problem (see the section on Markov Chains).

In low dimensions, the eigenvalues may be found as roots of the characteristic polynomial/equation derived from taking the determinant of $A - \lambda_i I$. Software like ScalaTion, however, use iterative algorithms that convert a matrix into Hessenburg and tridiagonal forms.

2.7 Internal Representation

The current internal representation used for storing the elements in a dense matrix is `Array [Array [Double]]` in row major order (row-by-row). Depending on usage, operations may be more efficient using column major order (column-by-column). Also, using a one dimensional array `Array [Double]` mapping (i, j) to the k^{th} location may be more efficient. Furthermore, having operations access through sub-matrices (blocks) may improve performance because of caching efficiency or improved performance for parallel and distributed versions.

The `mathstat` package provides several classes implementing multiple types of vectors and matrices as shown in Table 2.3 including `VectorD` and `MatrixD`.

Table 2.3: Types of Vectors and Matrices: Implementing Classes

trait	**VectorD**	**MatrixD**
dense	VectorD	MatrixD
sparse	SparseVectorD	SparseMatrixD
compressed	RleVectorD	RleMatrixD
tridiagonal	-	SymTriMatrixD
bidiagonal	-	BidMatrixD

The suffix 'D' indicates the base element type is `Double`. There are also implementations for `Complex 'C'`, `Int 'I'`, `Long 'L'`, `Rational 'Q'`, `Real 'R'`, `String 'S'`, and `TimeNum 'T'`.

Note, ScalaTion 2.0 currently only supports dense vectors and matrices. See older versions for the other types of vectors and matrices.

ScalaTion supports many operations involving matrices and vectors, including the following show in Table 2.5.

Table 2.4: Types of Vector and Matrix Products

Product	**Method**	**Example**	**in Math**
vector dot	def dot (y: VectorD): Double	x dot y	$\mathbf{x} \cdot \mathbf{y}$
vector element-wise	def * (y: VectorD): VectorD	x * y	$\mathbf{x}\,\mathbf{y}$
vector outer	def outer (y: VectorD): MatrixD	x outer y	$\mathbf{x} \otimes \mathbf{y}$
matrix mult	def * (y: MatrixD): MatrixD	x * y	$X\,Y$
matrix mdot	def dot (y: MatrixD): MatrixD	x dot y	$X^{\intercal}\,Y$
matrix vector	def * (y: VectorD): VectorD	x * y	$X\,\mathbf{y}$
matrix vector	def * (y: VectorD): MatrixD	x * y	$X\,\mathrm{diag}(\mathbf{y})$

2.8 Tensor

Loosely speaking, a *tensor* is a generalization of scalar, vector and matrix. The order of the tensor indicates the number dimensions. In this text, tensors are treated as *hyper-matrices* and issues such as basis independence, contravariant and covariant vectors/tensors, and the rules for index notation involving super and subscripts are ignored [111]. To examine the relationship between order 2 tensors and matrices more deeply, see the last exercise.

For data science, input into a model may be a vector (e.g., simple regression, univariate time series), a matrix (e.g., multiple linear regression, neural networks), a tensor with three dimensions (e.g., monochromatic/greyscale images), and a tensor with four dimensions (e.g., color images).

Table 2.5: Tensors of Different Orders

Order	**Analog/Name**	**Example**
zeroth	scalar	FICA score
first	vector	customer financial record
second	matrix	collection of financial records
third	tensor	collection of grayscale images
fourth	tensor4	collection color images

2.8.1 Three Dimensional Tensors

In ScalaTion, tensors with three dimensions are supported by the `TensorD` class. The default names for the dimensions [111] were chosen to follow a common convention (row, column, sheet). In data science, the first index usually indicates which instance, e.g., i^{th} element of a vector, i^{th} row of a matrix, i^{th} row of a tensor.

```
    @param dim    size of the 1st level/dimension (row) of the tensor (height)
    @param dim2   size of the 2nd level/dimension (column) of the tensor (width)
    @param dim3   size of the 3rd level/dimension (sheet) of the tensor (depth)

    class TensorD (val dim: Int, val dim2: Int, val dim3: Int,
                   private [mathstat] var v: Array [Array [Array [Double]]] = null)
          extends Error with Serializable
```

A tensor $\mathbf{T}$ is stored in a triple array $[t_{ijk}]$. Below is an example of a 2-by-2-by-2 tensor, $\mathbf{T} = [T_{::0} | T_{::1}]$

$$\begin{bmatrix} t_{000} & t_{010} & | & t_{001} & t_{011} \\ t_{100} & t_{110} & | & t_{101} & t_{111} \end{bmatrix}$$

where each sheet $T_{::k}$ is a 2-by-2 matrix.

Note, ScalaTion allows the default names for the dimensions to be changed, so they are more suggestive given the application, e.g., (row, column, channel) for one color image or (sheet, row, column) for spreadsheets.

Ragged order 3 tensors `RTensorD` are also supported which allow the middle dimension to vary (be ragged).

2.8.2 Four Dimensional Tensors

In ScalaTion, tensors with four dimensions are supported by the `Tensor4D` class. The default names for the dimensions [111] were chosen to follow a common convention (row, column, sheet, channel).

```
    @param dim    size of the 1st level/dimension (row) of the tensor (height)
    @param dim2   size of the 2nd level/dimension (column) of the tensor (width)
    @param dim3   size of the 3rd level/dimension (sheet) of the tensor (depth)
    @param dim3   size of the 4rd level/dimension (channel) of the tensor (spectra)

    class Tensor4D (val dim: Int, val dim2: Int, val dim3: Int,, dim4: Int,
                   private [mathstat] var v: Array [Array [Array [Array [Double]]]] = null)
          extends Error with Serializable
```

Such a tensor $\mathbf{T}$ is stored in a quad array $[t_{ijkl}]$.

Ragged order 4 tensors `RTensor4D` are also supported which allow the middle two dimensions to vary (be ragged).

2.9 Exercises

1. Draw two 2-dimensional non-zero vectors, $\mathbf{x}$ and $\mathbf{y}$, whose dot product $\mathbf{x} \cdot \mathbf{y}$ is zero.

2. A vector can be transformed into a unit vector in the same direction by dividing by its norm, $\frac{\mathbf{x}}{\|\mathbf{x}\|}$. Let, $\mathbf{y} = 2\mathbf{x}$ and show that the dot of the corresponding unit vectors equals one. This means that their Cosine Similarity equals one.

$$\cos_{\mathbf{xy}} \;=\; \cos(\theta) \;=\; \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \qquad \text{where } \theta \text{ is the angle between the vectors}$$

   When would the Cosine Similarity be -1? When would it be 0?

3. Correlation $\rho_{\mathbf{xy}}$ vs. Cosine Similarity $\cos_{\mathbf{xy}}$. What does it mean when the correlation (cosine similarity) is 1, 0, -1, respectively. In general, does $\rho_{\mathbf{xy}} = \cos_{\mathbf{xy}}$? What about in special cases?

4. Given the matrix X and the vector $\mathbf{y}$, solve for the vector $\mathbf{b}$ in the equation $\mathbf{y} = X\mathbf{b}$ using matrix inversion and LU factorization.

```
import scalation.mathstat.{MatrixD, VectorD, Fac_LU}
val x = MatrixD ((2, 2), 1, 3,
                         2, 1)
val y = VectorD (1, 7)
println ("using inverse: b = X^-1 y = " + x.inverse * y)
println ("using LU fact: Lb = Uy    = " + { val lu = new Fac_LU (x); lu.factor ().solve
    (y) } )
```

   Modify the code to show the inverse matrix X^{-1} and the factorization into the L and U matrices.

5. If Q is an orthogonal matrix, then $Q^{\intercal}Q$ becomes what type of matrix? What about $QQ^{\intercal}$? Illustrate with an example 3-by-3 matrix. What is the inverse of Q?

6. Show that the Hessian matrix of a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ is the transpose of the Jacobian of the gradient, i.e.,

$$\mathbb{H}_f(\mathbf{x}) \;=\; [\mathbb{J}\,\nabla f(\mathbf{x})]^{\intercal}$$

7. Critical points for a function $f : \mathbb{R}^n \to \mathbb{R}$ occur when $\nabla f(\mathbf{x}) = \mathbf{0}$. How can the Hessian Matrix can be used to decide whether a particular critical point is a local minimum or maximum?

8. Define three functions, $f_1(x, y)$, $f_2(x, y)$ and $f_3(x, y)$, that have critical points (zero gradient) at the point $[2, 3]$ such that this point is (a) a minimal point, (b) a maximal point, (c) a saddle point, respectively. Compute the Hessian matrix at this point for each function and use it to explain the type of critical point. Plot the three surfaces in 3D.

   Hint: see `https://www.math.usm.edu/lambers/mat280/spr10/lecture8.pdf`

9. Determine the eigenvalues for the matrix A given in the section on eigenvalues and eigenvectors, by setting the determinant of $A - \lambda I$ equal to zero.

$$\begin{bmatrix} 2-\lambda & 0 \\ 0 & 3-\lambda \end{bmatrix}$$

to obtain the following characteristics polynomial.

$$(2-\lambda)(3-\lambda) - 0 \;=\; 0$$

Solve for all roots of this polynomial to determine the eigenvalues.

10. A *vector space* $\mathcal{V}$ over field K (e.g., $\mathbb{R}$ or $\mathbb{C}$) is a set of objects, e.g., vectors $\mathbf{x}, \mathbf{y}$, and $\mathbf{z}$, and two operations, addition and scalar multiplication,

$$\mathbf{x}, \mathbf{y} \in \mathcal{V} \implies \mathbf{x} + \mathbf{y} \in \mathcal{V} \tag{2.22}$$

$$\mathbf{x} \in \mathcal{V} \text{ and } a \in K \implies a\mathbf{x} \in \mathcal{V} \tag{2.23}$$

satisfying the following conditions/axioms

$$\begin{aligned}
(\mathbf{x}+\mathbf{y})+\mathbf{z} &= \mathbf{x}+(\mathbf{y}+\mathbf{z}) \\
\mathbf{x}+\mathbf{y} &= \mathbf{y}+\mathbf{x} \\
\exists\, \mathbf{0} \in \mathcal{V} \;\; \mathbf{s.t.} \;\; \mathbf{x}+\mathbf{0} &= \mathbf{x} \\
\exists - \mathbf{x} \in \mathcal{V} \;\; \mathbf{s.t.} \;\; \mathbf{x}+(-\mathbf{x}) &= \mathbf{0} \\
(ab)\mathbf{x} &= a(b\mathbf{x}) \\
a(\mathbf{x}+\mathbf{y}) &= a\mathbf{x}+a\mathbf{y} \\
(a+b)\mathbf{x} &= a\mathbf{x}+b\mathbf{x} \\
\exists 1 \in K \; s.t. \; 1\mathbf{x} &= \mathbf{x}
\end{aligned}$$

Give names to these axioms and illustrate them with examples.

11. A *normed vector space* $\mathcal{V}$ over field K is a vector space with a function defined that gives the length (norm) of a vector,

$$\mathbf{x} \in \mathcal{V} \implies \|\mathbf{x}\| \in \mathbb{R}^{+}$$

satisfying the following conditions/axioms

$$\begin{aligned}
\|a\mathbf{x}\| &= |a|\,\|\mathbf{x}\| \\
\|\mathbf{x}\| &> 0 \quad \text{unless} \;\; \mathbf{x} = \mathbf{0} \\
\|\mathbf{x}+\mathbf{y}\| &\le \|\mathbf{x}\| + \|\mathbf{y}\|
\end{aligned}$$

A norm induced metric called distance can be defined,

$$d(\mathbf{x}, \mathbf{y}) \;=\; \|\mathbf{x} - \mathbf{y}\|$$

The ℓ^p-norm is defined as follows:

$$\|\mathbf{x}\|_p \;=\; \left(\sum_{i=0}^{n-1} |x_i|^p \right)^{\frac{1}{p}}$$

Norms and distances are very useful in data science, for example, *loss functions* used to judge/optimize models are often defined in terms of norms or distances.

Show that the last axiom called the *triangle inequality* hold for ℓ^2-norms.

Hint: $\|\mathbf{x}\|_2^2$ is the sum of the elements in $\mathbf{x}$ squared.

12. An *inner product space* $\mathcal{H}$ over field K is a vector space with one more operation, inner product,

$$\mathbf{x}, \mathbf{y} \in \mathcal{H} \implies \langle \mathbf{x}, \mathbf{y} \rangle \in K$$

satisfying the following conditions/axioms

$$\begin{aligned} \langle \mathbf{x}, \mathbf{y} \rangle &= \langle \mathbf{y}, \mathbf{x} \rangle^* \\ \langle a\mathbf{x} + b\mathbf{y}, \mathbf{z} \rangle &= a\,\langle \mathbf{x}, \mathbf{z} \rangle + b\,\langle \mathbf{y}, \mathbf{z} \rangle \\ \langle \mathbf{x}, \mathbf{x} \rangle &> 0 \quad \text{unless} \;\; \mathbf{x} = \mathbf{0} \end{aligned}$$

Note, the complex conjugate negates the imaginary part of a complex number, e.g., $(c + di)^* = c - di$

Show that an n-dimensional Euclidean vector space using the definition of dot product given in this chapter is an inner product space over $\mathbb{R}$.

13. Explain the meaning of the following statement, "a tensor of order 2 for a given coordinate system can be represented by a matrix."

Hint: see "Tensors: A Brief Introduction" [32]

2.10 Further Reading

1. Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares [21]
2. Matrix Computations [58]
3. Tensors and Hypermatrices [111]

Chapter 3

Probability

Probability is used to measure the likelihood of certain events occurring, such as flipping a coin and getting a head, rolling a pair of dice and getting a sum of 7, or getting a full house in five card draw. Given a random experiment, the *sample space* Ω is the set of all possible outcomes.

3.1 Probability Measure

Definition: A *probability measure* P can be defined axiomatically as follows:

$$\begin{aligned}
&P(A) \geq 0 \;\text{ for any event } A \subseteq \Omega \\
&P(\Omega) = 1 \\
&P(\cup A_i) = \sum P(A_i) \;\text{ for a countable collection of disjoint events}
\end{aligned} \tag{3.1}$$

Technically speaking, an *event* is a measurable subset of Ω (see [41] for a measure-theoretic definition). Letting $\mathcal{F}$ be the set of all possible events, one may define a probability space as follows:

Definition: A *probability space* is defined as a triple $(\Omega, \mathcal{F}, P)$.

Given an event $A \in \mathcal{F}$, the probability of its occurrence is restricted to the unit interval, $P(A) \in [0,\ 1]$. Thus, P may be viewed as a function that maps events to the unit interval.

$$P : \mathcal{F} \to [0, 1] \tag{3.2}$$

3.1.1 Joint Probability

Given two events A and B, the *joint probability* of their co-occurrence is denoted by

$$P(AB) = P(A \cap B) \ \in \ [0,\ min(P(A),\ P(B))] \tag{3.3}$$

If events A and B are *independent*, simply take the product of the individual probabilities,

$$P(AB) = P(A)P(B) \tag{3.4}$$

3.1.2 Conditional Probability

The *conditional probability* of the occurrence of event A, given it is known that event B has occurred/will occur is

$$P(A|B) \;=\; \frac{P(AB)}{P(B)} \tag{3.5}$$

If events A and B are independent, the conditional probability reduces to

$$P(A|B) \;=\; \frac{P(AB)}{P(B)} \;=\; \frac{P(A)P(B)}{P(B)} \;=\; P(A) \tag{3.6}$$

In other words, the occurrence of event B has no effect on the probability of event A occurring.

Bayes Theorem

An important theorem involving conditional probability is *Bayes Theorem.*

$$P(A|B) \;=\; \frac{P(B|A)P(A)}{P(B)} \tag{3.7}$$

When determining conditional probability $A|B$ is difficult, one may try going the other direction and first determine $B|A$.

Example

Consider flipping two coins. What is the Sample/Outcome Space Ω?

$$\Omega \;=\; \{[T,T],[T,H],[H,T],[H,H]\} \;=\; \{\omega_1,\omega_2,\omega_3,\omega_4\}$$

The size of the outcome space is 4 and since the event space $\mathcal{F}$ contains all subsets of Ω, its size is $2^4 = 16$. Define the following two events:

- event $A =$ first coin showing heads and
- event $B =$ at least one head was rolled.

What is the probability that event A occurred, given that you know that event B occurred? If fair coins are used, the probability of a head (or tail) is 1/2 and the probabilities reduce to the ratios of set sizes.

$$P(A|B) \;=\; \frac{P(AB)}{P(B)} \;=\; \frac{|A \cap B|}{|B|} \;=\; \frac{|\{\omega_3,\omega_4\} \cap \{\omega_2,\omega_3,\omega_4\}|}{|\{\omega_2,\omega_3,\omega_4\}|} \;=\; 2/3$$

This simplification can be done whenever all outcomes are *equi-probable.*

3.2 Random Variable

Rather than just looking at individual events, e.g., E_1 or E_2, one is often more interested in the probability that random variables take on certain values.

Definition: A *random variable* y is a function that maps outcomes in the sample space Ω into a set/domain of numeric values D_y.

$$y : \Omega \to D_y \tag{3.8}$$

Some commonly used domains are real numbers $\mathbb{R}$, integers $\mathbb{Z}$, natural numbers $\mathbb{N}$, or subsets thereof. An example of a mapping from outcomes to numeric values is `tail` $\to 0$, `head` $\to 1$. In other cases such as the roll of one dice, the map is the identity function.

One may think of a random variable y (blue font) as taking on values from a given domain D_y. With a random variable, its value is uncertain, i.e., its value is only known probabilistically.

For $A \subseteq D_y$ one can measure the probability of the random variable y taking on a value from the set A. This is denoted by

$$P(y \in A) \tag{3.9}$$

This really means the probability of event E which maps to set A

$$E = y^{-1}(A) \tag{3.10}$$

$$P(E) \tag{3.11}$$

where $y^{-1}(A)$ is the inverse image of A.

3.2.1 Discrete Random Variable

A discrete random variable is defined on finite or countably infinite domains. For example, the probability of rolling a natural in dice (sum of 7 or 11 with two dice) is given by

$$P(y \in \{7, 11\}) = 6/36 + 2/36 = 8/36 = 2/9$$

3.2.2 Continuous Random Variable

A continuous random variable is defined on uncountably infinite domains. For example, the probability of my tee shot on a par-3 golf hole ending up within 10 meters of the hole is 0.1 or 10 percent.

$$P(y \in [0, 10]) = 0.1$$

3.3 Probability Distribution

A random variable y is characterized by how its probability is distributed over its domain D_y. This can be captured by functions that map D_y to $\mathbb{R}^+$.

3.3.1 Cumulative Distribution Function

The most straightforward way to do this is to examine the probability measure for a random variable in terms of a *Cumulative Distribution Function* (CDF).

$$F_y : D_y \to [0,1] \tag{3.12}$$

It measures the amount probability or mass accumulated over the domain up to and including the point y. The color highlighted symbol y is the random variable, while y simply represents a value.

$$F_y(y) = P(y \le y) \tag{3.13}$$

To illustrate the concept, let x_1 and x_2 be the number on dice 1 and dice 2, respectively. Let $y = x_1 + x_2$, then $F_y(6) = P(y \le 6) = 5/12$. The entire CDF for the discrete random variable y (roll of two dice), $F_y(y)$ is

$$\{(2,1/36),(3,3/36),(4,6/36),(5,10/36),(6,15/36),(7,21/36),(8,26/36),(9,30/36),(10,33/36),(11,35/36),(12,36/36)\}$$

As another example, the CDF for a continuous random variable y that is defined to be uniformly distributed on the interval $[0,2]$ is

$$F_y(y) = \frac{y}{2} \quad \text{on} \quad [0,2]$$

When random variable y follows this CDF, we may say that y is distributed as `Uniform (0, 2)`, symbolically, $y \sim$ `Uniform (0, 2)`.

3.3.2 Probability Mass Function

While the CDF indicates accumulated probability or mass (totaling 1), examining probability or mass locally can be more informative. In case the random variable is discrete (i.e., D_y is discrete), a *probability mass function* (pmf) may be defined.

$$p_y : D_y \to [0,1] \tag{3.14}$$

This function indicates the amount of mass/probability at point $y_i \in D_y$,

$$p_y(y_i) \;=\; F_y(y_i) - F_y(y_{i-1}) \tag{3.15}$$

It can be calculated as the first difference of the CDF, i.e., the amount of accumulated mass at point y_i minus the amount of accumulated mass at the previous point y_{i-1}.
For one dice x_1, the pmf is

$$\{(1,1/6),(2,1/6),(3,1/6),(4,1/6),(5,1/6),(6,1/6)\}$$

A second dice x_2 will have the same pmf. Both random variables follow the Discrete Uniform Distribution, `Randi (1, 6)`.

$$p_x(x) \;=\; \frac{1}{6}\,\mathbb{1}_{\{1\le x\le 6\}} \tag{3.16}$$

where $\mathbb{1}_{\{c\}}$ is the indicator function (`if c then 1 else 0`).

If the two random variables are added $y = x_1 + x_2$, the pmf for the random variable y (roll of two dice), $p_y(y)$ is

$$\{(2,1/36),(3,2/36),(4,3/36),(5,4/36),(6,5/36),(7,6/36),(8,5/36),(9,4/36),(10,3/36),(11,2/36),(12,1/36)\}$$

The random variable y follows the Discrete Triangular Distribution (that peaks in the middle) and not the flat Discrete Uniform Distribution.

$$p_y(y) \;=\; \frac{\min(y-1, 13-y)}{36}\,\mathbb{1}_{\{2\le y\le 12\}} \tag{3.17}$$

Using the absolute value, this may be written as follows:

$$p_y(y) \;=\; \frac{6-|7-y|}{36} \qquad \text{for } y \in \{2,\ldots,12\} \tag{3.18}$$

3.3.3 Probability Density Function

Suppose y is defined on the continuous domain, e.g., $D_y = [0,2]$, and that mass/probability is uniformly spread amongst all the points in the domain. In such situations, it is not productive to consider the mass at one particular point. Rather one would like to consider the mass in a small interval and scale it by dividing by the length of the interval. In the limit this is the derivative which gives the density. For a continuous random variable, if the function F_y is differentiable, a *probability density function* (pdf) may be defined.

$$f_y : D_y \to \mathbb{R}^+ \tag{3.19}$$

It is calculated as the first derivative of the CDF, i.e.,

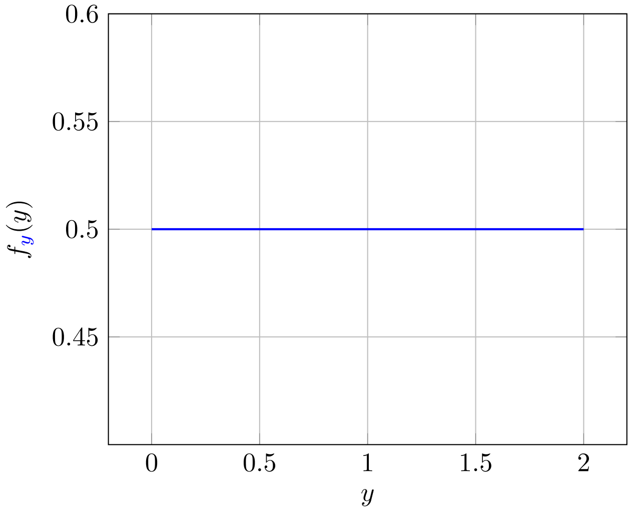
$$f_y(y) \;=\; \frac{dF_y(y)}{dy} \tag{3.20}$$

For example, the pdf for a uniformly distributed random variable y on $[0,2]$ is

$$f_y(y) \;=\; \frac{d}{dy}\,\frac{y}{2} \;=\; \frac{1}{2} \;\;\text{on}\;\; [0,2]$$

The pdf for the Uniform Distribution is shown in the figure below.

pdf for Uniform Distribution



Random variates of this type may be generated using SCALATION's `Uniform (0, 2)` class within the `scalation.random` package.

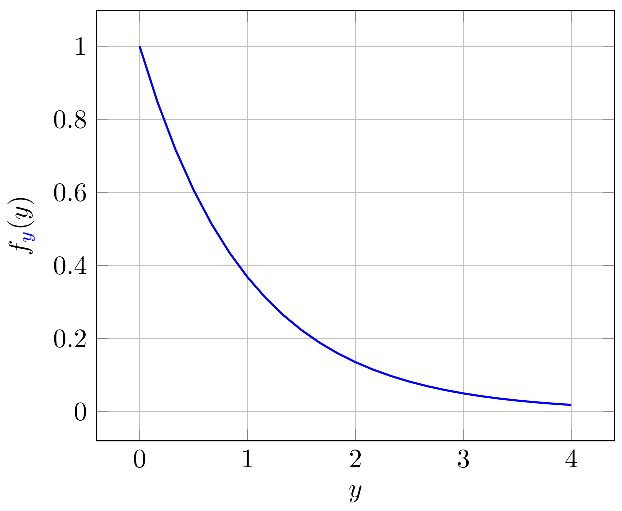
```
    val rvg = Uniform (0, 2)
    val yi  = rvg.gen
```

For another example, the pdf for an exponentially distributed random variable y on $[0, \infty)$ with rate parameter λ is

$$f_y(y) \; = \; \lambda e^{-\lambda y} \;\; \text{on} \;\; [0, \infty)$$

The pdf for the Exponential ($\lambda = 1$) Distribution is shown in the figure below.

pdf for Exponential Distribution



Going the other direction, the CDF $F_y(y)$ can be computed by summing the pmf $p_y(y)$

$$F_y(y) = \sum_{x_i \le y} p_y(x_i) \tag{3.21}$$

or integrating the pdf $f_y(y)$.

$$F_y(y) = \int_{-\infty}^{y} f_y(x)dx \tag{3.22}$$

3.4 Empirical Distribution

An empirical distribution may be used to describe a dataset probabilistically. Consider a dataset $(X, \mathbf{y})$ where $X \in \mathbb{R}^{m \times n}$ is the data matrix collected about the predictor variables and $\mathbf{y} \in \mathbb{R}^m$ is the data vector collected about the response variable. In other words, the dataset consists of m instances of an n-dimensional predictor vector $\mathbf{x}_i$ and a response value y_i.

The joint empirical probability mass function (epmf) may be defined on the basis of a given dataset $(X, \mathbf{y})$.

$$p_{data}(\mathbf{x}, y) \;=\; \frac{\nu(\mathbf{x}, y)}{m} \;=\; \frac{1}{m} \sum_{i=0}^{m-1} \mathbb{1}_{\{\mathbf{x}_i = \mathbf{x}, y_i = y\}} \tag{3.23}$$

where $\nu(\mathbf{x}, y)$ is the frequency count and $\mathbb{1}_{\{c\}}$ is the indicator function (`if c then 1 else 0`).

The corresponding Empirical Cumulative Distribution Function (ECDF) may be defined as follows:

$$F_{data}(\mathbf{x}, y) \;=\; \frac{1}{m} \sum_{i=0}^{m-1} \mathbb{1}_{\{\mathbf{x}_i \le \mathbf{x}, y_i \le y\}} \tag{3.24}$$

3.5 Expectation

Using the definition of a CDF, one can determine the *expected value* (or *mean*) for random variable y using a Riemann-Stieltjes integral.

$$\mathbb{E}[y] \ = \ \int\limits_{D_y} y \, dF_y(y) \tag{3.25}$$

The mean specifies the center of mass, e.g., a two-meters rod with the mass evenly distributed throughout, would have a center of mass at 1 meter. Although it will not affect the center of mass calculation, since the total probability is 1, unit mass is assumed (one kilogram). The center of mass is the balance point in the middle of the bar.

3.5.1 Continuous Case

When y is a continuous random variable, we may write the mean as follows:

$$\mathbb{E}[y] \ = \ \int\limits_{D_y} y \, f_y(y) dy \tag{3.26}$$

The mean of $y \sim$ `Uniform (0, 2)` is

$$\mathbb{E}[y] \ = \ \int_0^2 y \, \frac{1}{2} \, dy \ = \ 1.$$

3.5.2 Discrete Case

When y is a discrete random variable, we may write

$$\mathbb{E}[y] \ = \ \sum_{y \in D_y} y \, p_y(y) \tag{3.27}$$

The mean for rolling two dice is $\mathbb{E}[y] = 7$. One way to interpret this is to imagine winning y dollars by playing a game, e.g., two dollars for rolling a 2 and twelve dollars for rolling a 12, etc. The expected earnings when playing the game once is seven dollars. Also, by the *law of large numbers*, the average earnings for playing the game n times will converge to seven dollars as n gets large.

3.5.3 Variance

The *variance* of random variable y is given by

$$\mathbb{V}[y] \ = \ \mathbb{E}\left[(y - \mathbb{E}[y])^2\right] \tag{3.28}$$

The variance specifies how the mass spreads out from the center of mass. For example, the variance of $y \sim$ `Uniform (0, 2)` is

$$\mathbb{V}[y] \ = \ \mathbb{E}\left[(y-1)^2\right] \ = \ \int_0^2 (y-1)^2 \, \frac{1}{2} \, dy \ = \ \frac{1}{3}$$

That is, the variance of the one kilogram, two-meter rod is $\frac{1}{3}$ kilogram meter^2. Again, for probability to be viewed as mass, unit mass (one kilogram) must be used, so the answer may also be given as $\frac{1}{3}$ meter^2 Similarly to interpreting the mean as the center of mass, the variance corresponds to the moment of inertia. The *standard deviation* is simply the square root of variance.

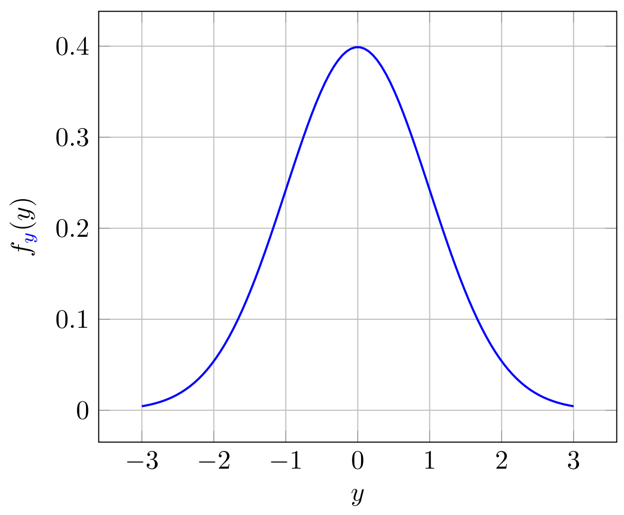
$$\mathbb{SD}[y] \;=\; \sqrt{\mathbb{V}[y]} \tag{3.29}$$

For the two-meter rod, the standard deviation is $\frac{1}{\sqrt{3}} = 0.57735$. The percentage of mass within one standard deviation unit of the center of mass is then 58%. Many distributions, such as the Normal (Gaussian) distribution concentrate mass closer to the center. For example, the *Standard Normal Distribution* has the following pdf.

$$f_y(y) \;=\; \frac{1}{\sqrt{2\pi}}\, e^{-y^2/2} \tag{3.30}$$

The mean for this distribution is 0, while the variance is 1. The percentage of mass within one standard deviation unit of the center of mass is 68%. The pdf for the Normal ($\mu = 0$, $\sigma^2 = 1$) Distribution is shown in the figure below.

pdf for Standard Normal Distribution



Note, the uncentered variance (or mean square) of the random variable y is simply $\mathbb{E}\left[y^2\right]$.

3.5.4 Covariance

The *covariance* of two random variable x and y is given by

$$\mathbb{C}[x, y] \;=\; \mathbb{E}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])] \tag{3.31}$$

The covariance specifies whether the two random variables have similar tendencies. If the random variables are *independent*, the covariance will be zero, while similar tendencies show up as positive covariance and dissimilar tendencies as negative covariance. *Correlation* normalizes covariance to the domain $[-1, 1]$. Covariance can be extended to more than two random variables. Let $\mathbf{z}$ be a vector of k random variables, then a *covariance matrix* is produced.

$$\mathbb{C}\left[\mathbf{z}\right] \;=\; \left[\mathbb{C}\left[z_i, z_j\right]\right]_{0\leq i,j<k} \tag{3.32}$$

3.6 Algebra of Random Variables

When random variables x_1 and x_2 are added to create a new random variable y,

$$y = x_1 + x_2$$

how is y described in terms of mean, variance and probability distribution? Also, what happens when a random variable is multiplied a constant?

$$y = ax$$

3.6.1 Expectation is a Linear Operator

The expectation/mean of the sum is simply the sum of the means.

$$\mathbb{E}[y] = \mathbb{E}[x_1] + \mathbb{E}[x_2] \tag{3.33}$$

The expectation of a random variable multiplied by a constant, is the constant multiplied by the random variable's expectation.

$$\mathbb{E}[ay] = a\mathbb{E}[y] \tag{3.34}$$

The last two equations imply that expectation is a linear operator.

3.6.2 Variance is not a Linear Operator

The variance of the sum is the sum of variances plus twice the covariance.

$$\mathbb{V}[y] = \mathbb{V}[x_1] + \mathbb{V}[x_2] + 2\,\mathbb{C}[x_1, x_2] \tag{3.35}$$

When the random variable are independent, the covariance is zero, so the variance of sum is just the sum of variances.

$$\mathbb{V}[y] = \mathbb{V}[x_1] + \mathbb{V}[x_2] \tag{3.36}$$

The variance of a random variable multiplied by a constant, is the constant squared multiplied by the random variable's variance.

$$\mathbb{V}[ay] = a^2\mathbb{V}[y] \tag{3.37}$$

See the exercises for derivations.

3.6.3 Convolution of Probability Distributions

Determination the new probability distribution of the sum of two random variables is more difficult.

Convolution: Discrete Case

Assuming the random variables are independent and discrete, the pmf of the sum p_y is the *convolution* of two pmfs p_{x_1} and p_{x_2}.

$$p_y \;=\; p_{x_1} * p_{x_2} \tag{3.38}$$

$$p_y(y) \;=\; \sum_{x \in D_x} p_{x_1}(x)\, p_{x_2}(y - x) \tag{3.39}$$

For example, letting $x_1, x_2 \sim \text{Bernoulli}(p)$, i.e., $p_{x_1}(x) = p^x(1-p)^{1-x}$ on $D_x = \{0, 1\}$, gives

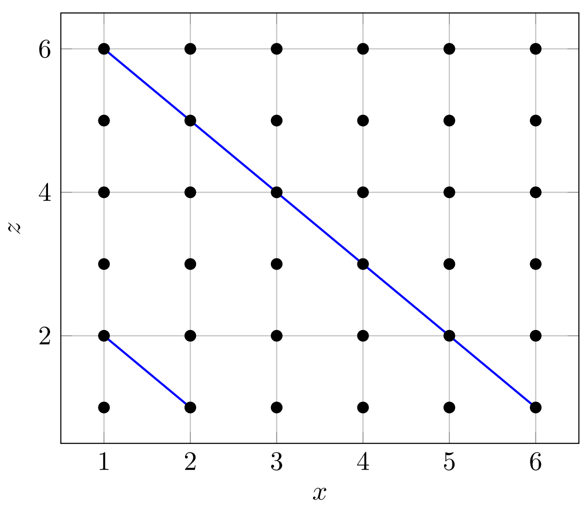
$$\begin{aligned}
p_y(0) &= \sum_{x \in D_x} p_{x_1}(x)\, p_{x_2}(0 - x) &&= p^2 \\
p_y(1) &= \sum_{x \in D_x} p_{x_1}(x)\, p_{x_2}(1 - x) &&= 2p(1-p) \\
p_y(2) &= \sum_{x \in D_x} p_{x_1}(x)\, p_{x_2}(2 - x) &&= (1-p)^2
\end{aligned}$$

which indicates that $y \sim \text{Binomial}(p, 2)$. The pmf for the Binomial(p, n) distribution is

$$p_y(y) \;=\; \binom{n}{y} p^y\,(1-p)^{n-y} \tag{3.40}$$

Consider the sum of two dice, $y = x + z$, where $x, z \sim \text{DiscreteUniform}(1, 6)$.

Joint Probability for Two Dice



As the joint pmf $p_{xz}(x_i, z_j) = p_x(x_i)p_z(z_j) = 1/36$ is constant over all points, the convolution sum for a particular value of y corresponds to the downward diagonal sum where the dice sum to that value, e.g., $p_y(3) = 2/36$, $p_y(7) = 6/36$.

Convolution: Continuous Case

Now, assuming the random variables are independent and continuous, the pdf of the sum f_y is the *convolution* of two pdfs f_{x_1} and f_{x_2}.

$$f_y \;=\; f_{x_1} * f_{x_2} \tag{3.41}$$

$$f_y(y) \;=\; \int_{D_x} f_{x_1}(x)\, f_{x_2}(y-x)\, dx \tag{3.42}$$

For example, letting $x_1, x_2 \sim \text{Uniform}(0,1)$, i.e., $f_{x_1}(x) = 1$ on $D_x = [0,1]$, gives

$$\begin{aligned}
\text{for } y \in [0,1] \quad f_y(y) \;&=\; \int_{[0,y]} f_{x_1}(x)\, f_{x_2}(y-x)dx \;=\; y \\
\text{for } y \in [1,2] \quad f_y(y) \;&=\; \int_{[0,2-y]} f_{x_1}(x)\, f_{x_2}(y-x)dx \;=\; 2-y
\end{aligned}$$

which indicates that $y \sim \text{Triangular}(0,1,2)$.

3.6.4 Central Limit Theorem

When several random variables are added (or averaged), interesting phenomena occurs, e.g., consider the distribution of y as the sum of m random variables.

$$y \;=\; \sum_{i=0}^{m-1} x_i$$

When $x_i \sim \text{Uniform}(0,1)$ with mean $\frac{1}{2}$ and variance $\frac{1}{12}$, then for m large enough y will follow a Normal distribution

$$y \;\sim\; \text{Normal}(\mu, \sigma^2)$$

where $\mu = \frac{m}{2}$ and $\sigma^2 = \frac{m}{12}$. The pdf for the Normal Distribution is

$$f_y(y) \;=\; \frac{1}{\sqrt{2\pi}\sigma}\, e^{-\frac{1}{2}(\frac{y-\mu}{\sigma})^2} \tag{3.43}$$

For most distributions, summed random variables will be approximately distributed as Normal, as indicated by the Central Limit Theorem (CLT); for proofs see [47, 11]. Suppose $x_i \sim F$ with mean μ_x and variance $\sigma_x^2 < \infty$, then the sum of m independent and identically distributed (*iid*) random variables is distributed as follows:

$$y \;=\; \sum_{i=0}^{m-1} x_i \;\sim\; N(m\mu_x, m\sigma_x^2) \qquad \text{as } m \to \infty \tag{3.44}$$

This is one simple form of the CLT. See the exercises of a visual illustration of the CLT.

Similarly, the sum of m independent and identically distributed random variables (with mean μ_x and variance ${\sigma_x}^2$) divided by m will also be Normally distributed for sufficiently large m.

$$y = \frac{1}{m} \sum_{i=0}^{m-1} x_i$$

The expectation of $y = \frac{1}{m} m\mu_x = \mu_x$, while variance is σ_x^2/m, so

$$y \sim \text{Normal}(\mu_x, \sigma_x^2/m)$$

As, $\mathbb{E}[y] = \mu_x$, y can serve as an unbiased estimator of μ_x. This can be transformed to the Standard Normal Distribution with the following transformation.

$$z = \frac{y - \mu_x}{\sigma_x/\sqrt{m}} \sim \text{Normal}(0, 1)$$

The Normal distribution is also referred to as the Gaussian distribution. See the exercises for related distributions: Chi-square, Student's t and F.

3.7 Median, Mode and Quantiles

As stated, the mean is the expected value, a probability weighted sum/integral of the values in the domain of the random variable. Other ways of characterizing a distribution are based more directly on the probability.

3.7.1 Median

Moving along the distribution, the place at which half of the mass is below you and half is above you is the *median*.

$$P(y \leq median) \geq \frac{1}{2} \;\text{ and }\; P(y \geq median) \geq \frac{1}{2} \tag{3.45}$$

Given equally likely values (1, 2, 3), the median is 2. Given equally likely values (1, 2, 3, 4), there are two common interpretations for the median: The smallest value satisfying the above equation (i.e., 2) or the average of the values satisfying the equation (i.e., 2.5) The median for two dice (with the numbers summed) which follow the Triangular distribution is 7.

3.7.2 Quantile

The *median* is also referred to as the half quantile.

$$\mathbb{Q}\,[y] \;=\; F_y^{-1}(\frac{1}{2}) \tag{3.46}$$

More generally, the $p \in [0, 1]$ quantile is given by

$$_p\mathbb{Q}\,[y] \;=\; F_y^{-1}(p) \tag{3.47}$$

where F_y^{-1} is the inverse CDF (iCDF). For example, recall the CDF for `Uniform (0, 2)` is

$$p \;=\; F_y(y) \;=\; \frac{y}{2} \;\text{ on }\; [0, 2]$$

Taking the inverse yields the iCDF.

$$F_y^{-1}(p) \;=\; 2p \;\text{ on }\; [0, 1]$$

Consequently, the median $\mathbb{Q}\,[y] = F_y^{-1}(\frac{1}{2}) = 1$.

3.7.3 Mode

Similarly, one may be interested in the *mode*, which is the average of the points of maximal probability mass.

$$\mathbb{M}\,[y] \;=\; \underset{y \in Dy}{\text{argmax}}\; p_y(y) \tag{3.48}$$

The mode for rolling two dice is $y = 7$. For continuous random variables, it is the average of points of maximal probability density.

$$\mathbb{M}\,[y] \;=\; \underset{y \in Dy}{\text{argmax}}\; f_y(y) \tag{3.49}$$

For the two-meter rod, the mean, median and mode are all equal to 1.

3.8 Joint, Marginal and Conditional Distributions

Knowledge of one random variable may be useful in narrowing down the possibilities for another random variable. Therefore, it is important to understand how probability is distributed in multiple dimensions. There are three main concepts: joint, marginal and conditional.

In general, the *joint* CDF for two random variables x and y is

$$F_{xy}(x, y) \;=\; P(x \le x, y \le y) \tag{3.50}$$

3.8.1 Discrete Case: Joint and Marginal Mass

In the discrete case, the *joint* pmf for two random variables x and y is

$$p_{xy}(x_i, y_j) \;=\; F_{xy}(x_i, y_j) - [F_{xy}(x_{i-1}, y_j) + F_{xy}(x_i, y_{j-1}) - F_{xy}(x_{i-1}, y_{j-1})] \tag{3.51}$$

See the exercises to check this formula for the matrix shown below.

Imagine nine weights placed in a 3-by-3 grid with the number indicating the relative mass.

```
    val mat = MatrixD ((3, 3), 1, 2, 3,
                                4, 5, 6,
                                7, 8, 9)
```

Dividing the matrix by 45 or calling `toProbability` in the `scalation.mathstat.Probability` object yields a probability matrix representing a joint probability mass function (pmf), $p_{xy}(x_i, y_j)$,

```
    MatrixD(0.0222222,    0.0444444,    0.0666667,
            0.0888889,    0.111111,     0.133333,
            0.155556,     0.177778,     0.200000)
```

The marginal pmfs are computed as follows:

$$p_x(x_i) \;=\; \sum_{y_j \in D_y} p_{xy}(x_i, y_j) \qquad \text{sum out y} \tag{3.52}$$

$$p_y(y_j) \;=\; \sum_{x_i \in D_x} p_{xy}(x_i, y_j) \qquad \text{sum out x} \tag{3.53}$$

Carrying out the summations or calling `margProbX (pxy)` for $p_x(x_i)$ and `margProbY (pxy)` for $p_y(y_j)$ gives,

```
Marginal X: px = VectorD(0.13333333333333333, 0.33333333333333337, 0.5333333333333334)
Marginal Y: py = VectorD(0.26666666666666666, 0.33333333333333337, 0.4)
```

Scaling back by multiplying by 45 produces,

```
45*Marginal X: px = VectorI(6, 15, 24)
45*Marginal Y: py = VectorI(12, 15, 18)
```

It is now easy to see that `px` is based on row sums, while `py` is based on column sums.

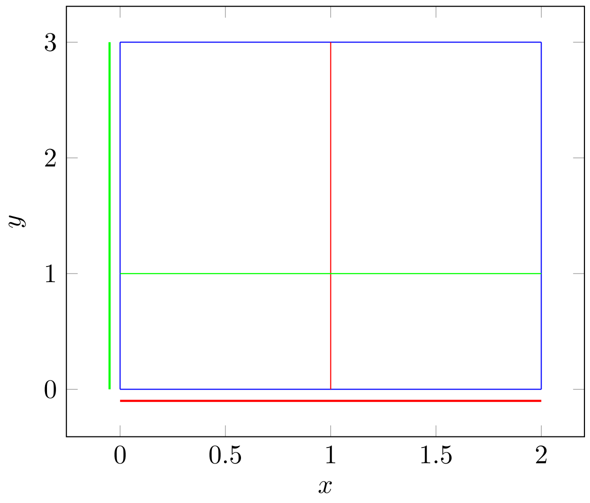
3.8.2 Continuous Case: Joint and Marginal Density

In the continuous case, the *joint* pdf for two random variables x and y is

$$f_{xy}(x,y) \;=\; \frac{\partial^2 F_{xy}}{\partial x \partial y}(x,y) \tag{3.54}$$

Consider the following joint pdf that specifies the distribution of one kilogram of mass (or probability) uniformly over a 2-by-3 meter plate.

$$f_{xy}(x,y) \;=\; \frac{1}{6} \qquad \text{on} \;\; [0,2] \times [0,3]$$



The joint CDF is then a double integral,

$$F_{xy}(x,y) \;=\; \int_0^x \int_0^y \frac{1}{6}\, dy dx \;=\; \frac{xy}{6}$$

There are two marginal pdfs that are single integrals: Think of the mass of the vertical red line being collected into the thick red bar at the bottom. Collecting all such lines creates the **red bar** at the bottom and its mass is distributed as follows:

$$f_x(x) \;=\; \int_0^3 \frac{1}{6}\, dy \;=\; \frac{3}{6} \;=\; \frac{1}{2} \;\; \text{on } [0,2] \qquad \text{integrate out y}$$

Now think of the mass of the horizontal green line being collected into the thick green bar on the left. Collecting all such lines creates the **green bar** on the left and its mass is distributed as follows:

$$f_y(y) \;=\; \int_0^2 \frac{1}{6}\, dx \;=\; \frac{2}{6} \;=\; \frac{1}{3} \;\; \text{on } [0,3] \qquad \text{integrate out x}$$

For more details and examples, see Class 7 of [138].

3.8.3 Discrete Case: Conditional Mass

Conditional probability can be examined locally. Given two discrete random variables x and y, the *conditional mass function* of x given y is defined as follows:

$$p_{x|y}(x_i, y_j) \;=\; P(x = x_i | y = y_j) \;=\; \frac{p_{xy}(x_i, y_j)}{p_y(y_j)} \tag{3.55}$$

where $p_{xy}(x_i, y_j)$ is the *joint mass function.* Again, the *marginal mass functions* are

$$\begin{aligned} p_x(x_i) &= \sum_{y_j \in D_y} p_{xy}(x_i, y_j) \\ p_y(y_j) &= \sum_{x_i \in D_x} p_{xy}(x_i, y_j) \end{aligned}$$

Consider the following example: Roll two dice. Let x be the value on the first dice and y be the sum of the two dice. Compute the conditional pmf for x given that it is known that $y = 2$.

$$p_{x|y}(x_i, 2) \;=\; P(x = x_i | y = 2) \;=\; \frac{p_{xy}(x_i, 2)}{p_y(2)} \tag{3.56}$$

Try this problem for each possible value for y.

3.8.4 Continuous Case: Conditional Density

Similarly, for two continuous random variables x and y, the *conditional density function* of x given y is defined as follows:

$$f_{x|y}(x, y) \;=\; \frac{f_{xy}(x, y)}{f_y(y)} \tag{3.57}$$

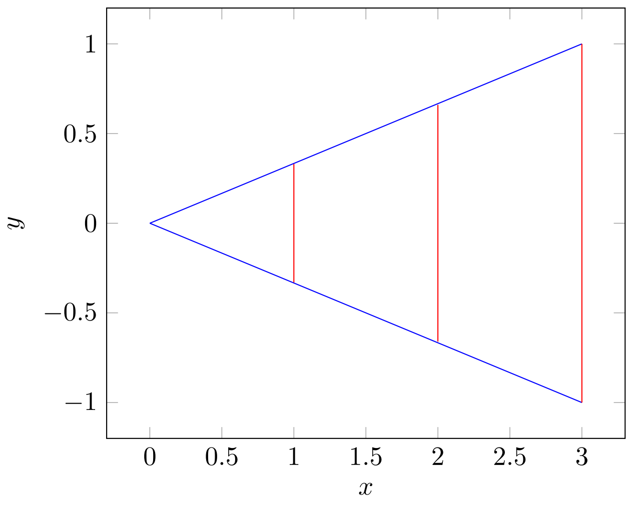
where $f_{xy}(x, y)$ is the *joint density function.* The *marginal density functions* are

$$f_x(x) \;=\; \int_{y \in D_y} f_{xy}(x, y)dy \tag{3.58}$$

$$f_y(y) \;=\; \int_{x \in D_x} f_{xy}(x, y)dx \tag{3.59}$$

The marginal density function in the x-dimension is the probability mass projected onto the x-axis from all other dimensions, e.g., for a bivariate distribution with mass distributed in the first xy quadrant, all the mass will fall onto the x-axis.

Consider the example below where the random variable x indicates how far down the center-line of a straight golf hole the golf ball was driven in units of 100 yards. The random variable y indicates how far left (positive) or right (negative) the golf ball ends up from the center of the fairway. Let us call these random variable distance and dispersion. The golfer teed the ball up at location $[0, 0]$. For simplicity, assume the probability is uniformly distributed within the triangle.



As the area of the triangle is 3, the joint density function is

$$f_{xy}(x, y) = \frac{1}{3} \qquad \text{on} \;\; x \in [0, 3], y \in [-x/3, x/3]$$

The distribution (density) of the driving distance down the center-line is given the marginal density for the random variable x

$$f_x(x) = \int_{-x/3}^{x/3} \frac{1}{3}\, dy = \frac{y}{3}\bigg|_{-x/3}^{x/3} = \frac{2x}{9}$$

Therefore, the conditional density of dispersion y given distance x is given by

$$f_{y|x}(x, y) = \frac{f_{xy}(x, y)}{f_x(x)} = \frac{1/3}{2x/9} = \frac{3}{2x} \tag{3.60}$$

3.8.5 Independence

The two random variables x and y are said to *independent* denoted $x \perp y$ when the joint CDF (equivalently pmf/pdf) can be factored into the product of its marginal CDFs (equivalently pmf/pdf).

$$F_{xy}(x, y) = F_x(x)\, F_y(y) \tag{3.61}$$

For example, determine which of the following two joint density functions defined on $[0, 1]^2$ signify independence.

$$\begin{aligned} f_{xy}(x, y) &= 4xy \\ f_{xy}(x, y) &= 8xy - x - y \end{aligned}$$

For the first joint density, the two marginal densities are the following:

$$f_x(x) = \int_0^1 4xy\, dy = \frac{4xy^2}{2}\bigg|_0^1 = 2x$$

$$f_y(y) \;=\; \int_0^1 4xy\,dx \;=\; \frac{4x^2y}{2}\Big|_0^1 \;=\; 2y$$

The product of the marginal densities $f_x(x)\,f_y(y) \;=\; 4xy$ is the joint density.

Compute the conditional density under the assumption that the random variables, x and y, are independent.

$$f_{x|y}(x,y) \;=\; \frac{f_{xy}(x,y)}{f_y(y)} \tag{3.62}$$

As the joint density can be factored, $f_{xy}(x,y) = f_x(x)\,f_y(y)$, we obtain,

$$f_{x|y}(x,y) \;=\; \frac{f_x(x)\,f_y(y)}{f_y(y)} \;=\; f_x(x) \tag{3.63}$$

showing that the value of random variable y has no effect on x. See the exercises for a proof that independence implies zero covariance (and therefore zero correlation).

3.8.6 Conditional Expectation

The value of one random variable may influence the expected value of another random variable The *conditional expectation* of random variable x given random variable y is defined as follows:

$$\mathbb{E}\left[x|y=y\right] \;=\; \int\limits_{D_x} x\,dF_{x|y}(x,y) \tag{3.64}$$

When y is a discrete random variable, we may write

$$\mathbb{E}\left[x|y=y\right] = \sum_{x\in D_x} x\,p_{x|y}(x,y) \tag{3.65}$$

When y is a continuous random variable, we may write

$$\mathbb{E}\left[x|y=y\right] = \int\limits_{D_x} x\,f_{x|y}(x,y)dx \tag{3.66}$$

Consider the previous example on the dispersion y of a golf ball conditioned on the driving distance y. Compute the conditional mean and the conditional variance for y given x.

$$\begin{aligned}
\mu_{y|x} \;&=\; \mathbb{E}\left[y|x=x\right] \;=\; \int_{-x/3}^{x/3} y\,f_{y|x}(x,y)dy \\
\sigma^2_{y|x} \;&=\; \mathbb{E}\left[(y-\mu_{y|x})^2|x=x\right] \;=\; \int_{-x/3}^{x/3} (y-\mu_{y|x})^2\,f_{y|x}(x,y)dy
\end{aligned}$$

3.8.7 Conditional Independence

A wide class of modeling techniques are under the umbrella of probabilistic graphical models (e.g., Bayesian Networks and Markov Networks). They work by factoring a joint probability based on conditional independencies. Random variables x and y are conditionally independent given z, denoted

$$x \perp y \,|\, z$$

means that

$$F_{x,y|z}(x,y,z) \;=\; F_{x|z}(x,z)\, F_{y|z}(y,z)$$

3.9 Odds

Another way of looking a probability is *odds*. This is the ratio of probabilities of an event A occurring over the event not occurring $S - A$.

$$\text{odds}(y \in A) \;=\; \frac{P(y \in A)}{P(y \in S - A)} \;=\; \frac{P(y \in A)}{1 - P(y \in A)} \tag{3.67}$$

For example, the odds of rolling a pair dice and getting natural is 8 to 28.

$$\text{odds}(y \in \{7, 11\}) \;=\; \frac{8}{28} \;=\; \frac{2}{7} \;=\; .2857$$

Of the 36 individual outcomes, eight will be a natural and 28 will not. Odds can be easily calculated from probability.

$$\text{odds}(y \in \{7, 11\}) \;=\; \frac{P(y \in \{7, 11\})}{1 - P(y \in \{7, 11\})} \;=\; \frac{2/9}{7/9} \;=\; \frac{2}{7} \;=\; .2857$$

Calculating probability from odds may be done as follows:

$$P(y \in \{7, 11\}) \;=\; \frac{\text{odds}(y \in \{7, 11\})}{1 + \text{odds}(y \in \{7, 11\})} \;=\; \frac{2/7}{9/7} \;=\; \frac{2}{9} \;=\; .2222$$

3.10 Example Problems

Understanding of some of techniques to be discussed requires some background in conditional probability.

1. Consider the probability of rolling a natural (i.e., 7 or 11) with two dice where the random variable y is the sum of the dice.

$$P(y \in \{7, 11\}) \;=\; 1/6 + 1/18 \;=\; 2/9$$

If you knew you rolled a natural, what is the conditional probability that you rolled a 5 or 7?

$$P(y \in \{5, 7\} \,|\, y \in \{7, 11\}) \;=\; \frac{P(y \in \{5, 7\},\; y \in \{7, 11\})}{P(y \in \{7, 11\})} \;=\; \frac{1/6}{2/9} \;=\; 3/4$$

This is the conditional probability of rolling a 5 or 7 given that you rolled a natural.

More generally, the conditional probability that $y \in A$ given that $x \in B$ is the joint probability divided by the probability that $x \in B$.

$$P(y \in A \,|\, x \in B) \;=\; \frac{P(y \in A,\; x \in B)}{P(x \in B)}$$

where

$$P(y \in A,\; x \in B) \;=\; P(x \in B \,|\, y \in A)\, P(y \in A)$$

Therefore, the conditional probability of y given x is

$$P(y \in A \,|\, x \in B) \;=\; \frac{P(x \in B \,|\, y \in A)\, P(y \in A)}{P(x \in B)}$$

This is Bayes Theorem written using random variables, which provides an alternative way to compute conditional probabilities, i.e., $P(y \in \{5, 7\} \,|\, y \in \{7, 11\})$ is

$$\frac{P(y \in \{7, 11\} \,|\, y \in \{5, 7\})\, P(y \in \{5, 7\})}{P(y \in \{7, 11\})} \;=\; \frac{(3/5) \cdot (5/18)}{2/9} \;=\; 3/4$$

2. To illustrate the usefulness of Bayes Theorem, consider the following problem from John Allen Paulos that is hard to solve without it. Suppose you are given three coins, two fair and one counterfeit (always lands heads). Randomly select one of the coins. Let x indicate whether the selected coin is fair (0) or counterfeit (1). What is the probability that you selected the counterfeit coin?

$$P(x = 1) \;=\; 1/3$$

Obviously, the probability is 1/3, since the probability of picking any of the three coins is the same. This is the *prior probability*.

Not satisfied with this level of uncertainty, you conduct experiments. In particular, you flip the selected coin three times and get all heads. Let y indicate the number of heads rolled. Using Bayes Theorem, we have,

$$P(x = 1 \mid y = 3) \;=\; \frac{P(y = 3 \mid x = 1)\, P(x = 1)}{P(y = 3)} \;=\; \frac{1 \cdot (1/3)}{5/12} \;=\; 4/5$$

where $P(y = 3) = (1/3)(1) + (2/3)(1/8) = 5/12$. After conducting the experiments (collecting evidence) the probability estimate may be improved. Now the *posterior probability* is 4/5.

3.11 Estimating Parameters from Samples

Given a model for predicting a response value for y from a feature/predictor vector $\mathbf{x}$,

$$y \;=\; f(\mathbf{x}; \mathbf{b}) + \epsilon$$

one needs to pick a functional form for f and collect a sample of data to estimate the parameters $\mathbf{b}$. The sample will consist of m instances $(y_i, \mathbf{x}_i)$ that form the response/output vector $\mathbf{y}$ and the data/input matrix X.

$$\mathbf{y} \;=\; f(X; \mathbf{b}) + \boldsymbol{\epsilon}$$

There are multiple types of estimation procedures. The central ideas are to minimize error or maximize the likelihood that the model would generate data like the sample. A common way to minimize error is to minimize the **Mean Squared Error** (MSE). The error vector $\boldsymbol{\epsilon}$ is the difference between the actual response vector $\mathbf{y}$ and the predicted response vector $\hat{\mathbf{y}}$.

$$\boldsymbol{\epsilon} \;=\; \mathbf{y} - \hat{\mathbf{y}} \;=\; y - f(\mathbf{x}; \mathbf{b})$$

The mean squared error on the length (Euclidean norm) of the error vector $\|\boldsymbol{\epsilon}\|$ is given by

$$\mathbb{E}\left[\|\boldsymbol{\epsilon}\|^2\right] \;=\; \mathbb{V}\left[\|\boldsymbol{\epsilon}\|\right] + \mathbb{E}\left[\|\boldsymbol{\epsilon}\|\right]^2 \tag{3.68}$$

where $\mathbb{V}[\|\boldsymbol{\epsilon}\|]$ is error variance and $\mathbb{E}[\|\boldsymbol{\epsilon}\|]$ is the error mean. If the model is unbiased the error mean will be zero, in which case the goal is to minimize the error variance.

3.11.1 Sample Mean

Suppose the speeds of cars on an interstate highway are Normally distributed with a mean at the speed limit of 70 mph (113 kph) and a standard deviation of 8 mph (13 kph), i.e., $\mathbf{y} \sim N(\mu, \sigma^2)$ in which case the model is

$$\mathbf{y} \;=\; \mu + \epsilon$$

where $\epsilon \sim N(0, \sigma^2)$. Create a sample of size $m = 100$ data points, using a Normal random variate generator. The population values for the mean μ and standard deviation σ are typically unknown and need to be estimated from the sample, hence the names sample mean $\hat{\mu}$ and sample standard deviation $\hat{\sigma}$. Show the generated sample, by plotting the data points and displaying a histogram.

```scala
@main def sampleStats (): Unit =

    val (mu, sig) = (70.0, 8.0)                            // pop. mean and std dev
    val m       = 100                                      // sample size
    val rvg     = Normal (mu, sig * sig)                   // Normal random variate
    val sample = VectorD (for i <- 0 until m yield rvg.gen)  // sample from Normal dist
    val (mu_, sig_) = (sample.mean, sample.stdev)          // sample mean and std dev
    println (s"(mu_, sig_) = ($mu_, $sig_)")
    new Plot (null, sample)
    new Histogram (sample)

end sampleStats
```

Imports: `scalation.mathstat._`, `scalation.random._`.

3.11.2 Confidence Interval

Now that you have an estimate for the mean, you begin to wonder if is correct or rather close enough. Generally, an estimate is considered close enough if its confidence interval contains the population mean. Collect an iid sample of values into a vector $\mathbf{y}$. Then the *sample mean* is simply

$$\hat{\mu} \;=\; \frac{\mathbf{1} \cdot \mathbf{y}}{m} \;=\; \frac{1}{m} \sum_{i=0}^{m-1} y_i$$

To create a confidence interval, we need to determine the variability or variance in the estimate $\hat{\mu}$.

$$\mathbb{V}[\hat{\mu}] \;=\; \mathbb{V}\left[\frac{1}{m} \sum_{i=0}^{m-1} y_i\right] \;=\; \frac{1}{m^2} \sum_{i=0}^{m-1} \mathbb{V}[y_i] \;=\; \frac{\sigma^2}{m}$$

The difference between the estimate from the sample and the population mean is Normally distributed and centered at zero (show that $\hat{\mu}$ is an unbiased estimator for μ, i.e., $\mathbb{E}[\hat{\mu}] = \mu$).

$$\hat{\mu} - \mu \;\sim\; N(0, \frac{\sigma^2}{m})$$

We would like to transform the difference so that the resulting expression follows a Standard Normal distribution. This can be done by dividing by $\frac{\sigma}{\sqrt{m}}$.

$$\frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}} \;\sim\; N(0, 1)$$

Consequently, the probability that the expression is greater than z is given by the CDF of the Standard Normal distribution, $F_N(z)$.

$$P\left(\frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}} > z\right) \;=\; 1 - F_N(z)$$

One might consider that if $z = 2$, two standard deviation units, then the estimate is not close enough. The same problem can exist on the negative side, so we should require

$$\left|\frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}}\right| \leq 2$$

In other words,

$$|\hat{\mu} - \mu| \leq \frac{2\sigma}{\sqrt{m}}$$

This condition implies that μ would likely be inside the following confidence interval.

$$\left[\hat{\mu} - \frac{2\sigma}{\sqrt{m}},\, \hat{\mu} + \frac{2\sigma}{\sqrt{m}}\right]$$

In this case it is easy to compute values for the lower and upper bounds of the confidence interval. The interval half width is simply $\frac{2 \cdot 8}{10} = 1.6$, which is to be subtracted and added to the sample mean.

Use ScalaTion to determine the probability that μ is within such confidence intervals?

```
    println (s"1 - F(2) = ${1 - normalCDF (2)}")
```

The probability is one minus twice this value. If 1.96 is used instead of 2, what is the probability, expressed as a percentage.

Typically, the population standard deviation is unlikely to be known. It would need to estimated by using the sample standard deviation, where the *sample variance* is

$$\hat{\sigma}^2 \;=\; \frac{1}{m-1} \sum_{i=0}^{m-1} (y_i - \hat{\mu})^2 \tag{3.69}$$

Note, this textbook uses $\hat{\theta}$ to indicate an estimator for parameter θ, regardless of whether it is a Maximum Likelihood (MLE) estimator. This substitution introduces more variability into the estimation of the confidence interval and results in the Standard Normal distribution (z-distribution)

$$\left[\hat{\mu} - \frac{z^*\sigma}{\sqrt{m}},\, \hat{\mu} + \frac{z^*\sigma}{\sqrt{m}}\right] \tag{3.70}$$

being replace by the Student's t distribution

$$\left[\hat{\mu} - \frac{t^*\hat{\sigma}}{\sqrt{m}},\, \hat{\mu} + \frac{t^*\hat{\sigma}}{\sqrt{m}}\right] \tag{3.71}$$

where z^* and t^* represent distances from zero, e,g., 1.96 or 2.09, that are large enough so that the analyst is comfortable with the probability that they may be wrong.

The numerators for the interval half widths (ihw) are calculated by the following top-level functions in `Statistics.scala`. The `z_sigma` function is used for the z-distribution.

```scala
def z_sigma (sig: Double, p: Double = .95): Double =
    val pp = 1.0 - (1.0 - p) / 2.0                         // e.g., .95 --> .975 (two tails)
    val z  = random.Quantile.normalInv (pp)
    z * sig
end z_sigma
```

The `t_sigma` function is used for the t-distribution.

```scala
def t_sigma (sig: Double, df: Int, p: Double = .95): Double =
    if df < 1 then { flaw ("interval", "must have at least 2 observations"); return 0.0 }
    val pp = 1.0 - (1.0 - p) / 2.0                         // e.g., .95 --> .975 (two tails)
    val t  = random.Quantile.studentTInv (pp, df)
    t * sig
end t_sigma
```

Does the probability you determined in the last example problem make any sense. Seemingly, if you took several samples, only a certain percentage of them would have the population mean within their confidence interval.

```scala
@main def confidenceIntervalTest (): Unit =

    val (mu, sig) = (70.0, 8.0)                               // pop. mean and std dev
    val m    = 100                                            // sample size
    val rm   = sqrt (m)
    val rvg = Normal (mu, sig * sig)                          // Normal random variate
    var count_z, count_t = 0

    for it <- 1 to 100 do                                     // test several datasets
        val y = VectorD (for i <- 0 until m yield rvg.gen)    // sample from Normal dist
        val (mu_, sig_) = (y.mean, y.stdev)                  // sample mean and std dev

        val ihw_z = z_sigma (sig_) / rm                      // interval half width: z
        val ci_z  = (mu_ - ihw_z, mu_ + ihw_z)               // z-confidence interval
        println (s"mu = muinci_z =ci_z?")
        if mu in ci_z then count_z += 1

        val ihw_t = t_sigma (sig_, m-1) / rm                 // interval half width: t
        val ci_t  = (mu_ - ihw_t, mu_ + ihw_t)               // z-confidence interval
        println (s"mu = muinci_t =ci_t?")
        if mu in ci_t then count_t += 1
    end for

    println (s"mu inside count_zprintln(s"muinsidecount_t % t-confidence intervals")

end confidenceIntervalTest
```

Imports: `scalation._`, `scalation.mathstat._`, `scalation.random._`.
Try various values for m starting with $m = 20$. Compute percentages for both the t-distribution and the z-distribution. Given the default confidence level used by ScalaTion is 0.95 (or 95%) what would you expect your percentages to be?

3.11.3 Estimation for Discrete Outcomes/Responses

Explain why the probability mass function (pmf) for flipping a coin n times with the experiment resulting in the discrete random variable $y = k$ heads is given by the *Binomial Distribution* having unknown parameter p, the probability of getting a head for any particular coin flip,

$$p_n(k) \;=\; P(y = k) \;=\; \binom{n}{k} p^k (1-p)^{n-k}$$

i.e., $y \sim \text{Binomial}(n, p)$.

Now suppose an experiment is run and $y = k$, a fixed number, e.g., $n = 100$ and $k = 60$. For various values of p, plot the following function.

$$L(p) \;=\; \binom{n}{k} p^k (1-p)^{n-k}$$

What value of p maximizes the function $L(p)$? The function $L(p)$ is called the *Likelihood function* and it is used in Maximum Likelihood Estimation (MLE) [139].

The `VectorD` class provides methods for computing statistics on vectors.

3.12 Entropy

The entropy of a discrete random variable y with probability mass function (pmf) $p_y(y)$ is the negative of the expected value of the log of the probability.

$$H(y) \;=\; H(p_y) \;=\; -\,\mathbb{E}\left[\log_2 p_y\right] \;=\; -\sum_{y\in D_y} p_y(y)\log_2 p_y(y) \tag{3.72}$$

The following single loop is used in ScalaTion to compute entropy.

```
    def entropy (px: VectorD): Double =
        var sum = 0.0
        for p <- px if p > 0.0 do sum -= p * log2 (p)
        sum
    end entropy
```

For finite domains of size $k = |D_y|$, entropy $H(y)$ ranges from 0 to $\log_2(k)$. Low entropy (close to 0) means that there is low uncertainty/risk in predicting an outcome of an experiment involving the random variable y, while high entropy (close to $\log_2 k$) means that there is high uncertainty/risk in predicting an outcome of such an experiment. For binary classification ($k = 2$), the upper bound on entropy is 1.

The entropy may be normalized by setting the base of the logarithm to the size of the domain k, in which case, the entropy will be in the interval $[0, 1]$.

$$H_k(y) \;=\; H_k(p_y) \;=\; -\,\mathbb{E}\left[\log_k p_y\right] \;=\; -\sum_{y\in D_y} p_y(y)\log_k p_y(y)$$

A random variable $y \sim \text{Bernoulli}(p)$ may be used to model the flip of a single coin that has a probability of success/head (1) of p. Its pmf is given by the following formula.

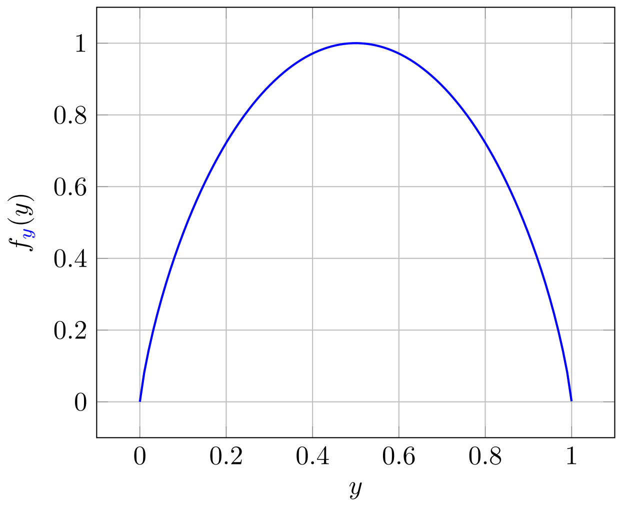
$$p(y) \;=\; p^y(1-p)^{1-y}$$

The pmf p_y can be captured in a probability vector $\mathbf{p}_y$

$$H(y) \;=\; H(p_y) \;=\; H(\mathbf{p}_y) \;=\; H([p, 1-p]) \;=\; p\log_2 p + (1-p)\log_2 1-p$$

The figure below plots the entropy $H([p, 1-p])$ as probability of a head p ranges from 0 to 1.

Entropy for Bernoulli pmf



A random variable $y = z_1 + z_2$ where z_1, z_2 are distributed as Bernoulli(p) may be used to model the sum of flipping two coins.

$$H(y) \;=\; H(\mathbf{p}_y) \;=\; H([p^2, 2p(1-p), (1-p)^2])$$

See the exercises for how to extend entropy to continuous random variables.

3.12.1 Positive Log Probability

Entropy can also be expressed in terms of positive log probability [57] (or plog).

$$\text{plog}(y) \;=\; -\log_2 p_y \tag{3.73}$$

Then entropy is simply the expected values of the plog.

$$H(y) \;=\; \mathbb{E}\,[\text{plog}(y)] \tag{3.74}$$

The concept of plog can also be used in place of probability and offers several advantages: (1) multiplying many small probabilities may lead to round off error or underflow; (2) independence leads to addition of plog values rather than multiplication of probabilities; and (3) its relationship to log-likelihood in Maximum Likelihood Estimation.

$$\text{plog}(\mathbf{x}) \;=\; \sum_j \text{plog}(x_j) \qquad \text{for independent random variables} \tag{3.75}$$

The greater the plog the less likely the occurrence, e.g., the plog of rolling snake eyes (1, 1) with two dice is about 5.17, while probability of rolling 7 is 2.58. Note, probability 1 and .5 give plog of 0 and 1, respectively.

3.12.2 Joint Entropy

Entropy may be defined for multiple random variables as well. Given two discrete random variables, x, y, with a joint pmf $p_{x,y}(x, y)$ the joint entropy is defined as follows:

$$H(x, y) \;=\; H(p_{x,y}) \;=\; -\,\mathbb{E}\left[\log_2 p_{x,y}\right] \;=\; -\sum_{x\in D_x}\sum_{y\in D_y} p_{x,y}(x, y)\, \log_2 p_{x,y}(x, y) \tag{3.76}$$

3.12.3 Conditional Entropy

Replacing the joint pmf, with the conditional pmf gives conditional entropy.

$$H(x|y) \;=\; H(p_{x|y}) \;=\; -\,\mathbb{E}\left[\log_2 p_{x|y}\right] \;=\; -\sum_{x\in D_x}\sum_{y\in D_y} p_{x,y}(x, y)\, \log_2 p_{x|y}(x, y) \tag{3.77}$$

Suppose an experiment involves two random variables x and y. Initially, the overall entropy is given by the joint entropy $H(x, y)$. Now, partial evidence allows the value of y to be determined, so the overall entropy should decrease by y's entropy.

$$H(x|y) \;=\; H(x, y) \;-\; H(y) \tag{3.78}$$

When there is *no dependency* between x and y (i.e., they are independent), H(x, y) = H(x) + H(y)), so

$$H(x|y) \;=\; H(x) \tag{3.79}$$

At the other extreme, when there is *full dependency* (i.e., they value of x can be determined from the value of y).

$$H(x|y) \;=\; 0 \tag{3.80}$$

3.12.4 Relative Entropy

Relative entropy, also known as *Kullback-Leibler (KL) divergence*, measures the dissimilarity between two probability distributions.

Discrete Random Variable

Given a discrete random variables, y, with two candidate probability mass functions (pmf)s $p_y(y)$ and $q_y(y)$ the relative entropy is defined as follows:

$$H(p_y||q_y) \;=\; \mathbb{E}\left[\log_2 \frac{p_y}{q_y}\right] \;=\; \sum_{y\in D_y} p_y(y) \log_2 \frac{p_y(y)}{q_y(y)} \tag{3.81}$$

One way to look at relative entropy is that it measures the uncertainty that is introduced by replacing the true/empirical distribution p_y with an approximate/model distribution q_y. If the distributions are identical, then the relative entropy is 0, i.e., $H(p_y||p_y) = 0$. The larger the value of $H(p_y||q_y)$ the greater the *dissimilarity* between the distributions p_y and q_y.

As an example, assume the true distribution for a coin is [.6, .4], but it is thought that the coin is fair [.5, .5]. The relative entropy is computed as follows:

$$H(p_y || q_y) \;=\; .6 \log_2 .6/.5 \,+\, .4 \log_2 .4/.5 \;=\; 0.029$$

Continuous Random Variable

Given a continuous random variables, y, with two candidate probability density functions (pdf)s $f_y(y)$ and $g_y(y)$ the relative entropy is defined as follows:

$$H(f_y || g_y) \;=\; \mathbb{E}\left[\log_2 \frac{f_y}{g_y}\right] \;=\; \int_{D_y} f_y(y) \log_2 \frac{f_y(y)}{g_y(y)} dy \tag{3.82}$$

Maximum Likelihood Estimation

In this subsection, we examine the relationship between KL Divergence and Maximum Likelihood. Consider the dissimlarity of an empirical distribution $p_{data}(\mathbf{x}, y)$ and a model generated distribution $p_{mod}(\mathbf{x}, y, \mathbf{b})$.

$$H(p_{data}(\mathbf{x}, y) || p_{mod}(\mathbf{x}, y, \mathbf{b})) \;=\; \mathbb{E}\left[\log \frac{p_{data}(\mathbf{x}, y)}{p_{mod}(\mathbf{x}, y, \mathbf{b})}\right] \tag{3.83}$$

$$\;=\; \sum_{i=0}^{m-1} p_{data}(\mathbf{x}_i, y_i) \log \frac{p_{data}(\mathbf{x}_i, y_i)}{p_{mod}(\mathbf{x}_i, y_i, \mathbf{b})} \tag{3.84}$$

Note, that $p_{data}(\mathbf{x}_i, y_i)$ is unaffected by the choice of parameters $\mathbf{b}$, so it represents a constant C.

$$H(p_{data}(\mathbf{x}, y) || p_{mod}(\mathbf{x}, y, \mathbf{b})) \;=\; C - \sum_{i=0}^{m-1} p_{data}(\mathbf{x}_i, y_i) \log p_{mod}(\mathbf{x}_i, y_i, \mathbf{b}) \tag{3.85}$$

The probability for the i^{th} data instance is $\frac{1}{m}$, thus

$$H(p_{data}(\mathbf{x}, y) || p_{mod}(\mathbf{x}, y, \mathbf{b})) \;=\; C - \frac{1}{m} \sum_{i=0}^{m-1} \log p_{mod}(\mathbf{x}_i, y_i, \mathbf{b}) \tag{3.86}$$

The second term is the negative log-likelihood (the Chapter on Generalized Linear Models for details).

3.12.5 Cross Entropy

Relative entropy can be adjusted to capture overall entropy by adding the entropy of p_y.

$$H(p_y \times q_y) \;=\; H(p_y) + H(p_y || q_y) \tag{3.87}$$

It is the sum of the entropy of the empirical distribution and the model distribution's relative entropy to the empirical distribution. It can be calculated using the following formula (see exercises for details):

$$H(p_y \times q_y) \;=\; -\sum_{y \in D_y} p_y(y) \log_2 q_y(y) \tag{3.88}$$

Since cross entropy is more efficient to calculate than relative entropy, it is a good candidate as a loss function for machine learning algorithms. The smaller the cross entropy, the more the model (e.g., Neural Network) agrees with the empirical distribution (dataset). The formula looks like the one for ordinary entropy with q_y substituted in as the argument for the log function. Hence the name cross entropy.

3.12.6 Mutual Information

Recall that if x and y are independent, then for all $x \in D_x$ and $y \in D_y$,

$$p_{x,y}(x,y) \;=\; p_x(x)\, p_y(y)$$

A possibly more interesting and practical question is to measure how close two random variables are to being independent. One approach is to look at the covariance (or correlation) between x and y.

$$\mathbb{C}\left[x, y\right] \;=\; \mathbb{E}\left[(x-\mu_x)(y-\mu_y)\right] \;=\; \sum_{x\in D_x}\sum_{y\in D_y}(x-\mu_x)(y-\mu_y)p_{x,y}(x,y)$$

If x and y are independent, then

$$\mathbb{C}\left[x, y\right] \;=\; \sum_{x\in D_x}\sum_{y\in D_y}\left[(x-\mu_x)p_x(x)\right]\left[(y-\mu_y)p_y(y)\right] \;=\; 0$$

An alternative is to look at the relative entropy of $p_{x,y}$ and $p_x\, p_y$.

$$H(p_{x,y}||p_x\, p_y) \;=\; \mathbb{E}\left[\log_2 \frac{p_{x,y}}{p_x\, p_y}\right] \;=\; \sum_{x\in D_x}\sum_{y\in D_y} p_{x,y}(x,y)\log_2 \frac{p_{x,y}(x,y)}{p_x(x)\, p_y(y)} \tag{3.89}$$

The relative entropy (KL divergence) of the joint distribution to the product of the marginal distributions is referred to as mutual information.

$$I(x;y) \;=\; H(p_{x,y}||p_x\, p_y) \tag{3.90}$$

The following double loop is used in ScalaTion to compute mutual information.

```
    def muInfo (pxy: MatrixD, px: VectorD, py: VectorD): Double =
        var sum = 0.0
        for i <- pxy.indices; j <- pxy.indices2 do
            val p = pxy(i, j)
            if p > 0.0 then sum += p * log2 (p / (px(i) * py(j)))
        end for
        sum
    end muInfo
```

As with covariance (or correlation) mutual information will be zero when x and y are independent. While independence implies zero covariance, independence is equivalent to zero mutual information. Mutual information is symmetric and non-negative. See the exercises for additional comparisons between covariance/-correlation and mutual information.

While mutual information measures the dependence between two random variables, relative entropy (KL divergence) measures the dissimilarity of two distribution.

Mutual Information corresponds to Information Gain, i.e., the drop in entropy of one random variable due to knowledge of the value of the other random variable.

$$I(x;y) \;=\; H(x) \;-\; H(x|y) \;=\; H(y) \;-\; H(y|x) \tag{3.91}$$

The `Probability` object in the `scalation.stat` package provides methods to compute probabilities from frequencies, compute joint, marginal, conditional and log probabilities, as well as entropy, normalized entropy, relative entropy, cross entropy, and mutual information.

3.12.7 Probability Object

Class Methods:

```
object Probability:

def isProbability (px: VectorD): Boolean = px.min >= 0.0 && abs (px.sum - 1.0) < EPSILON
def isProbability (pxy: MatrixD): Boolean = pxy.mmin >= 0.0 && abs (pxy.sum - 1.0) <
EPSILON
def freq (x: VectorI, vc: Int, y: VectorI, k: Int): MatrixD =
def freq (x: VectorI, y: VectorI, k: Int, vl: Int): (Double, VectorI) =
def freq (x: VectorD, y: VectorI, k: Int, vl: Int, cont: Boolean,
def count (x: VectorD, vl: Int, cont: Boolean, thres: Double): Int =
def toProbability (nu: VectorI): VectorD = nu.toDouble / nu.sum.toDouble
def toProbability (nu: VectorI, n: Int): VectorD = nu.toDouble / n.toDouble
def toProbability (nu: MatrixD): MatrixD = nu / nu.sum
def toProbability (nu: MatrixD, n: Int): MatrixD = nu / n.toDouble
def probY (y: VectorI, k: Int): VectorD = y.freq (k)._2
def jointProbXY (px: VectorD, py: VectorD): MatrixD = outer (px, py)
def margProbX (pxy: MatrixD): VectorD =
def margProbY (pxy: MatrixD): VectorD =
def condProbY_X (pxy: MatrixD, px_ : VectorD = null): MatrixD =
def condProbX_Y (pxy: MatrixD, py_ : VectorD = null): MatrixD =
inline def plog (p: Double): Double = - log2 (p)
def plog (px: VectorD): VectorD = px.map (plog (_))
def entropy (px: VectorD): Double =
def entropy (nu: VectorI): Double =
def entropy (px: VectorD, b: Int): Double =
def nentropy (px: VectorD): Double =
def rentropy (px: VectorD, qx: VectorD): Double =
def centropy (px: VectorD, qx: VectorD): Double =
def entropy (pxy: MatrixD): Double =
def entropy (pxy: MatrixD, px_y: MatrixD): Double =
def muInfo (pxy: MatrixD, px: VectorD, py: VectorD): Double =
def muInfo (pxy: MatrixD): Double = muInfo (pxy, margProbX (pxy), margProbY (pxy))
```

For example, the following **freq** method is used by Naïve Bayes Classifiers. It computes the Joint Frequency Table (JFT) for all value combinations of vectors **x** and **y** by counting the number of cases where $x_i = v$ and $y_i = c$.

```
@param x   the variable/feature vector
@param vc  the number of distinct values in vector x (value count)
@param y   the response/classification vector
@param k   the maximum value of y + 1 (number of classes)

def freq (x: VectorI, vc: Int, y: VectorI, k: Int): MatrixD =
    val jft = new MatrixD (vc, k)
    for i <- x.indices do jft(x(i), y(i)) += 1
    jft
end freq
```

3.13 Exercises

Several random number and random variate generators can be found in ScalaTion's `random` package. Some of the following exercises will utilize these generators.

1. Let the random variable h be the number heads when two coins are flipped. Determine the following conditional probability: $P(h = 2|h \geq 1)$.

2. Prove Bayes Theorem.

$$P(A|B) \ = \ \frac{P(B|A)P(A)}{P(B)}$$

3. Compute the mean and variance for the *Bernoulli Distribution* with success probability p.

$$p_y(y) \ = \ p^y\,(1-p)^{1-y} \quad \text{for } y \in \{0, 1\}$$

4. Use the `Randi` random variate generator to run experiments to check the pmf and CDF for rolling two dice.

```
import scalation.mathstat._
import scalation.random.Randi

@main def diceTest (): Unit =
    val dice = Randi (1, 6)
    val x    = VectorD.range (0, 13)
    val freq = new VectorD (13)
    for i <- 0 until 10000 do
        val sum = dice.igen + dice.igen
        freq(sum) += 1
    end for
    new Plot (x, freq)
end diceTest
```

5. Show that the variance may be written as follows:

$$\mathbb{V}\,[y] \ = \ \mathbb{E}\left[(y - \mathbb{E}\,[y])^2\right] \ = \ \mathbb{E}\left[y^2\right] - \mathbb{E}\,[y]^2$$

6. Show that the covariance may be written as follows:

$$\mathbb{C}\,[x, y] \ = \ \mathbb{E}\,[(x - \mathbb{E}\,[x])(y - \mathbb{E}\,[y])] \ = \ \mathbb{E}\,[xy] - \mathbb{E}\,[x]\,\mathbb{E}\,[y]$$

7. Show that the covariance of two independent, continuous random variables, x and y, is zero.

$$\mathbb{C}\,[x, y] \ = \ \mathbb{E}\,[(x - \mu_x)(y - \mu_y)] \ = \ \int_{D_y}\int_{D_x} (x - \mu_x)(y - \mu_y) f_{xy}(x, y)dxdy$$

where $\mu_x = \mathbb{E}\,[x]$ and $\mu_y = \mathbb{E}\,[y]$.

8. Derive the formula for the expectation of the sum of random variables.

$$\mathbb{E}[x_1 + x_2] \;=\; \mathbb{E}[x_1] + \mathbb{E}[x_2]$$

9. Derive the formula for the variance of the sum of random variables.

$$\mathbb{V}[x_1 + x_2] \;=\; \mathbb{V}[x_1] + \mathbb{V}[x_2] + 2\mathbb{C}[x_1, x_2]$$

   Hint: use $\mathbb{V}[x_1 + x_2] = \mathbb{E}\left[(x_1 + x_2)^2\right] - \mathbb{E}[x_1 + x_2]^2$

10. Use the `Uniform` random variate generator and the `Histogram` class to run experiments illustrating the Central Limit Theorem (CLT).

```scala
import scalation.mathstat._
import scalation.random.Uniform

@main def cLTTest (): Unit =

    val rg = Uniform ()
    val x = VectorD (for i <- 0 until 100000 yield rg.gen + rg.gen + rg.gen + rg.gen)
    new Histogram (x)

end cLTTest
```

   Try with other distributions such as `Exponential`.

11. Chi-square distribution: Show that if $z \sim \text{Normal}(0, 1)$, then

$$z^2 \sim \chi^2_1$$

12. Student's t distribution: Show that if $z \sim \text{Normal}(0, 1)$ and $v \sim \chi^2_k$, then

$$\frac{z}{\sqrt{v/k}} \;\sim\; t_k$$

13. F distribution: Show that if $u \sim \chi^2_{k_1}$ and $v \sim \chi^2_{k_2}$, then

$$\frac{u/k_1}{v/k_2} \;\sim\; F_{k_1,k_2}$$

14. Run the `confidenceIntervalTest` main function (see the Confidence Interval section) for values of m = 20 to 40, 60, 80 and 100. Report the confidence interval and the number cases when the true values was inside the confidence interval for (a) the z-distribution and (b) the t-distribution. Explain.

15. Given three random variables such that $x \perp y \,|\, z$, show that

$$F_{x|y,z}(x, y, z) \;=\; F_{x|z}(x, z)$$

16. Show that formula for computing the joint probability mass function (pmf) for the 3-by-3 grid of weights is correct. Hint: Add/subtract rectangular regions of the grid and make sure nothing is double counted.

17. Show for $k = 2$ where $\mathbf{pp} = [p, 1-p]$, that $H(\mathbf{pp}) = p \log_2(p) + (1-p)\log_2(1-p)$. Plot the entropy $H(\mathbf{pp})$ versus p.

```
val p = VectorD.range (1, 100) / 100.0
val h = p.map (p => -p * log2 (p) - (1-p) * log2 (1-p))
new Plot (p, h)
```

18. Plot the entropy H and normalized entropy H_k for the first 16 Binomial(p, n) distributions, i.e., for the number of coins $n = 1, \ldots, 16$. Try with $p = .6$ and $p = .5$.

19. Entropy can be defined for continuous random variables. Take the definition for discrete random variables and replace the sum with an integral and the pmf with a pdf. Compute the entropy for $y \sim$ Uniform$(0, 1)$.

20. Using the summation formulas for entropy, relative entropy and cross entropy, show that cross entropy is the sum of entropy and relative entropy.

21. Show that mutual information equals the sum of marginal entropies minus the joint entropy, i.e.,

$$I(x; y) = H(x) + H(x) - H(x, y)$$

22. Compare correlation and mutual information in terms of how well they measure dependence between random variables x and y. Try various functional relationships: negative exponential, reciprocal, constant, logarithmic, square root, linear, right-arm quadratic, symmetric quadratic, cubic, exponential and trigonometric.

$$y \ = \ f(x) + \epsilon$$

Other types of relationships are also possible. Try various constrained mathematical relations: circle, ellipse and diamond.

$$f(x, y) + \epsilon \ = \ c$$

What happens as the noise ϵ increases?

23. Consider an experiment involving the roll of two dice. Let x indicate the value of dice 1 and x_2 indicate of the value of dice 2. In order to examine dependency between random variables, define $y = x + x_2$. The joint pmf $p_{x,y}$ can be recorded in a 6-by-11 matrix that can be computed from the following feasible occurrence matrix ($0 \rightarrow$ cannot occur, $1 \rightarrow$ can occur), since all the non-zero probabilities are the same (equal likelihood).

```
// X  - dice 1: 1, 2, 3, 4, 5, 6
// X2 - dice 2: 1, 2, 3, 4, 5, 6
// Y  = X + X2: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
val nuxy = MatrixD ((6, 11), 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
                              0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
                              0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0,
                              0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
                              0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,
                              0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1)
```

Use methods in the `Probability` object to compute the joint, marginal and conditional probability distributions, as well as the joint, marginal, conditional and relative entropy, and mutual information. Explore the independence between random variables x and y.

24. **Convolution**. The convolution operator may be applied to vectors as well as functions (including mass and density functions). Consider two vectors $\mathbf{c} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$. Without loss of generality let $m \leq n$, then their convolution is defined as follows:

$$\mathbf{y} \;=\; \mathbf{c} \star \mathbf{x} \;=\; \left[y_k = \sum_{j=0}^{m-1} c_j x_{k-j} \right]_{k=0,m+n-2} \tag{3.92}$$

Compute the ('full') convolution of $\mathbf{c}$ and $\mathbf{x}$.

$$\mathbf{y} \;=\; \mathbf{c} \star \mathbf{x} \;=\; [1,2,3,4,5] \star [1,2,3,4,5,6,7] \tag{3.93}$$

Note, there are also 'same' and 'valid' versions of convolution operators.

25. Consider a distribution with density on the interval $[0,2]$. Let the probability density function (pdf) for this distribution be the following:

$$f_y(y) \;=\; \frac{y}{2} \qquad \text{on} \quad [0,2]$$

(i) Draw/plot the pdf $f_y(y)$ vs. y for the interval $[0,2]$.

(ii) Determine the Cumulative Distribution Function (CDF), $F_y(y)$.

(iii) Draw/plot the CDF $F_y(y)$ vs y for the interval $[0,2]$.

(iv) Determine the expected value of the Random Variable (RV) y, i.e., $\mathbb{E}[y]$.

26. Take the limit of the difference quotient of monomial x^n to show that

$$\frac{d}{dx} x^n \;=\; n x^{n-1}$$

Recall the definition of *derivative* as the limit of the difference quotient.

$$\frac{d}{dx} f(x) \;=\; \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Recall the notations due to Leibniz, Lagrange, and Euler.

$$\frac{d}{dx} f(x) \;=\; f'(x) \;=\; D_x f(x)$$

27. Take the *integral* and then the derivative of the monomial x^n to show that

$$\frac{d}{dx}\int x^n dx \;=\; x^n$$

3.14 Further Reading

1. Probability and Mathematical Statistics [163].

2. Entropy, Relative Entropy and Mutual Information [35].

3.15 Notational Conventions

With respect to random variables, vectors and matrices, the following notational conventions shown in Table 3.1 will be used in this book.

Table 3.1: Notational Conventions Followed

variable type	case	font	color
scalar	lower	italics	black
vector	lower	bold	black
matrix	upper	italics	black
tensor	upper	bold	black
random scalar	lower	italics	blue
random vector	lower	bold	blue

Built on the Functional Programming features in Scala, ScalaTion support several function types:

```
    type FunctionS2S = Double   => Double        // function of a scalar
    type FunctionS2V = Double   => VectorD       // vector-valued function of a scalar

    type FunctionV2S = VectorD => Double         // function of a vector
    type FunctionV2V = VectorD => VectorD        // vector-valued function of a vector
    type FunctionV2M = VectorD => MatrixD        // matrix-valued function of a vector

    type FunctionM2V = MatrixD => VectorD        // vector-valued function of a matrix
    type FunctionM2M = MatrixD => MatrixD        // matrix-valued function of a matrix
```

These function types are defined in the `scalation` and `scalation.mathstat` packages. A scalar-valued function type ends in `'S'`, a vector-valued function type ends in `'V'`, and a matrix-valued function type ends in `'M'`.
Mathematically, the scalar-valued functions are denoted by a symbol, e.g., f.

$$\begin{aligned} &\texttt{S2S function} \quad f : \mathbb{R} \to \mathbb{R} \\ &\texttt{V2S function} \quad f : \mathbb{R}^n \to \mathbb{R} \end{aligned}$$

Mathemtically, the vector-valued functions are denoted by a bold symbol, e.g., $\mathbf{f}$.

$$\begin{aligned} &\texttt{S2V function} \quad \mathbf{f} : \mathbb{R} \to \mathbb{R}^n \\ &\texttt{V2V function} \quad \mathbf{f} : \mathbb{R}^m \to \mathbb{R}^n \\ &\texttt{M2V function} \quad \mathbf{f} : \mathbb{R}^{m \times p} \to \mathbb{R}^n \end{aligned}$$

Mathemtically, the matrix-valued functions are denoted by a bold symbol, e.g., $\mathbf{f}$.

V2M function	$\mathbf{f} : \mathbb{R}^p \to \mathbb{R}^{m \times n}$
M2M function	$\mathbf{f} : \mathbb{R}^{p \times q} \to \mathbb{R}^{m \times n}$

3.16 Model

Models are about making predictions such as given certain properties of a car, predict the car's mileage, given recent performance of a stock index fund, forecast its future value, or given a person's credit report, classify them as either likely to repay or not likely to repay a loan. The thing that is being predicted, forecasted or classified is referred to the response/output variable, call it y. In many cases, the "given something" is either captured by other input/feature variables collected into a vector, call it $\mathbf{x}$,

$$y \;=\; f(\mathbf{x}; \mathbf{b}) + \epsilon \tag{3.94}$$

or by previous values of y. Some functional form f is chosen to map input vector $\mathbf{x}$ into a predicted value for response y. The last term indicates the difference between actual and predicted values, i.e., the residuals ϵ. The function f is parameterized and often these parameters can be collected into a matrix $\mathbf{b}$.

If values for the parameter vector $\mathbf{b}$ are set randomly, the model is unlikely to produce accurate predictions. The model needs to be *trained* by collecting a dataset, i.e., several (m) instances of $(\mathbf{x}_i, y_i)$, and optimizing the parameter vector $\mathbf{b}$ to minimize some *loss* function, such as *mean squared error* (mse),

$$mse \;=\; \frac{1}{m}\,\|\mathbf{y} - \hat{\mathbf{y}}\|^2 \tag{3.95}$$

where $\mathbf{y}$ is the vector from all the response instances and $\hat{\mathbf{y}} = f(X; \mathbf{b})$ is the vector of predicted response values and X is the matrix formed from all the input/feature vector instances.

Estimation Procedures

Although there are many types of parameter estimation procedures, this text only utilizes the three most commonly used procedures [14].

Table 3.2: Estimation Procedures

Procedure	**Full Name**	**Inventor**
LSE	Least Squares Estimation	Gauss
MoM	Method of Moments	Pearson
MLE	Maximum Likelihood Estimation	Fisher

The method of moments develops equations the relate the moments of a distribution to the parameters of the model, in order to create estimates for the parameters. Least Squares Estimation takes the sum of squared errors and sets the parameter values to minimize this sum. It has three main varieties: Ordinary Least Squares (OLS), Weighted Least Squares (WLS), and Generalized Least Squares (GLS). Finally, Maximum Likelihood Estimation sets the parameter values so that the observed data is likely to occur. The easiest way to think about this is to imagine that one wants to create a generative model (a model that generates data). One would want to set the parameters of the model so it generates data that looks like the given dataset.

Setting of parameters is done by solving a system of equations for the simpler models, or by using an optimization algorithm for more complex models.

Quality of Fit (QoF)

After a model is trained, its Quality of Fit (QoF) should be evaluated. One way to perform the evaluation is to train the model on the full dataset and test as well on the full dataset. For complex models with many parameters, *over-fitting* will likely occur. Then its excellent evaluation is unlikely to be reproduced when the model is applied in the real-world. To avoid overly optimistic evaluations due to over-fitting, it is common to divide a dataset $(X, \mathbf{y})$ into a training dataset and testing dataset where training is conducted on the training dataset $(X_r, \mathbf{y}_r)$ and evaluation is done on the test dataset $(X_e, \mathbf{y}_e)$. The conventions used in this book for the full, training and test datasets are shown in Table 3.3

Table 3.3: Convention for Datasets

Math Symbol	Code	Description
X	x	full data/input matrix
$X_{\_}$	x_	training data/input matrix (maybe full)
X_e	x_e	test data/input matrix (maybe full)
$\mathbf{y}$	y	full response/output vector
$\mathbf{y}_{\_}$	y_	training response/output vector (maybe full)
$\mathbf{y}_e$	y_e	test response/output vector (maybe full)

Note, when training and testing on the full dataset, the training and test dataset are actually the same, i.e., they are the full dataset. If a model has many parameters, the Quality of Fit (QoF) found from training and testing on the full dataset should be suspect. See the section on cross-validation for more details.

In ScalaTion, the `Model` trait severs as base trait for all the modeling techniques in the `modeling` package and its sub-packages `classifying`, `clustering`, `fda`, `forecasting`, and `recommeneder`.

`Model` **Trait**

Trait Methods:

```
    trait Model:

    def getFname: Array [String]
    def train (x_ : MatrixD, y_ : VectorD): Unit
    def test (x_ : MatrixD, y_ : VectorD): (VectorD, VectorD)
    def predict (z: VectorD): Double | VectorD
    def hparameter: HyperParameter
    def parameter: VectorD | MatrixD
    def report (ftVec: VectorD): String =
    def report (ftMat: MatrixD): String =
```

The `getFname` method returns the predictor variable/feature names in the model. The `train` method will use a training or full dataset to train the model, i.e., optimize its parameter vector **b** to minimize a given loss function. After training, the quality of the model may be assessed using the `test` method. The evaluation may be performed on a test or full dataset. Finally, information about the model may be extracted by the following three methods: (1) `hparameter` showing the hyper-parameters, (2) `parameter` showing the parameters, and (3) `report` showing the hyper-parameters, the parameter, and the Quality of Fit (QoF) of the model. Note, hyper-parameters are used by some modeling techniques to influence either the result or how the result is obtained.

Classes that implement (directly or indirectly) the `Model` trait should default `x_` and `x_e` to the full data/input matrix `x`, and `y_` and `y_e` to the full response/output vector `y` that are passed into the class constructor.

Implementations of the `train` method take a training data/input matrix `x_` and a training response/output vector `y_` and optimize the parameter vector `b` to, for example, minimize error or maximize likelihood. Implementations of the `test` method take a test data/input matrix `x_e` and the corresponding test response/output vector `y_e` to compute errors and evaluate the Quality of Fit (QoF). Note that with cross-validation (to be explained later), there will be multiple training and test datasets created from one full dataset. Implementations of the `hparameter` method simply return the hyper-parameter vector `hparam`, while implementations of the `parameter` method simply return the optimized parameter vector `b`. (The `fname_` and `technique` parameters for `Regression` are the feature names and the solution/optimization technique used to estimate the parameter vector, respectively.)

Associated with the `Model` trait is the `FitM` trait that provides QoF measures common to all types of models. For prediction, `Fit` extends `FitM` with several additional QoF measures and they are explained on the Prediction Chapter. Similarity, `FitC` extends `FitM` for classification models.

`FitM` Trait

Trait Methods:

```
trait FitM:

def sse_ : Double = sse
def rSq_ : Double  = rSq                                        // using mean
def rSq0_ : Double = rSq0                                       // using 0
def diagnose (y: VectorD, yp: VectorD, w: VectorD = null): VectorD =
def fit: VectorD
def help: String
def summary (x_ : MatrixD, fname: Array [String], b: VectorD, vifs: VectorD = null):
    String
```

The `diagnose` method takes the actual response/output vector `y` and the predictions from the model `yp` and calculates the basic QoF measures.

```
@param y   the actual response/output vector to use (test/full)
@param yp  the predicted response/output vector (test/full)
@param w   the weights on the instances (defaults to null)

def diagnose (y: VectorD, yp: VectorD, w: VectorD = null): VectorD =
    m = y.dim                                          // size of response vector
    if m < 2 then flaw ("diagnose", s"requires at least 2 responses to evaluate m = $m")
    if yp.dim != m then flaw ("diagnose", s"yp.dim = ${yp.dim} != y.dim = $m")

```

```
        val mu = y.mean                                       // mean of y (may be zero)
        val e  = y - yp                                       // residual/error vector
        sse    = e.normSq                                     // sum of squares for error
        if w == null then
            sst = y.cnormSq                                   // sum of squares total
            ssr = sst - sse                                   // sum of squares model
        else
            ssr = (w * (yp - (w * yp / w.sum).sum)~^2).sum  // regression sum of squares
            sst = ssr + sse
        end if

        mse0   = sse / m                                      // raw/MLE mean squared error
        rmse   = sqrt (mse0)                                  // root mean squared error
        mae    = e.norm1 / m                                  // mean absolute error
        rSq    = ssr / sst                                    // R^2 using mean
        rSq0   = 1 - sse / y.normSq                           // R^2 using 0
        e                                                     // returns error vector
    end diagnose
```

Note, `~^` is the exponentiation operator provided in ScalaTion, where the first character is `~` to give the operator higher precedence than multiplication (*).

One of the measures is based on absolute errors, Mean Absolute Error (MAE), and is computed as the ℓ^1 norm of the error vector divided by the number of elements in the response vector (`m`). The rest are based on squared values. Various squared ℓ^2 norms may be taken to compute these quantities, i.e., `sst = y.cnormSq` is the centered norm squared of `y`, while `sse = e.normSq` is the norm squared of `e`. Then `ssr`, the sum of squares model/regression, is the difference. The idea being that one started with the variation in the response, some of which can accounted for by the model, with the remaining part considered errors. As models are less than perfect, what remains are better referred to as residuals, part of which a better model could account for. The fraction of the variation accounted for by the model to the total variation is called the coefficient of determination $R^2 = ssr/sst \leq 1$. A measure that parallel MAE is the Root Mean Squared Error (RMSE). It is typically higher as a large squared term has more of an effect. Both are interpretable as they are in the units of the response variable, e.g., imagine one hits a golf ball at 150 mph with an MAE of 7 mph and an RMSE of 10 mph. Further explanations are given in the Prediction Chapter.

Chapter 4

Data Management

4.1 Introduction

Data Science relies on having large amounts of quality data. Collecting data and handling data quality issues are of utmost importance. Without support from a system or framework, this can be very time-consuming and error-prone. This chapter provides a quick overview of the support provided by ScalaTion for data management.

In the era of big data, a variety of database management technologies have been proposed, including those under the umbrella of Not-only-SQL (NoSQL). These technologies include the following:

- Key-value stores (e.g., Memcached). When the purpose the data store is very rapid lookup and not advanced query capabilities, a key-value store may be ideal. They are often implemented as distributed hash tables.
- Document-oriented databases (e.g., MongoDB). These databases are intended for storage and retrieval of unstructured (e.g., text) and semi-structured (e.g., XML or JSON) data.
- Columnar databases (e.g., Vertica). Such databases are intended for structured data like traditional relational databases, but to better facilitate data compression and analytic operations. Data is stored in columns rather rows as in traditional relational databases.
- Graph databases (e.g., Neo4j). These make the implicit relationships (via foreign-key, primary-key pairs) in relational databases explicit. A tuple in a relational database is mapped to a node in a graph database, while an implicit relationship is mapped to edge in a graph database. The database then consists of a collection directed graphs, each consisting of nodes and edges connecting the nodes. These database are particularly suited to social networks.

The purpose of these database technologies is to provide enhanced performance over traditional, row-oriented relational database and each of the above are best suited to particular types of data.

Data management capabilities provided by ScalaTion include Relational Databases, Columnar Databases and Graph Databases. All include extensions making them suitable as a Time Series DataBase (TSDB). Graph databases are discussed in the Appendix.

Preprocessing of data should be done before applying analytics techniques to ensure they are working on quality data. ScalaTion provides a variety of preprocessing techniques, as discussed in the next chapter.

4.1.1 Analytics Databases

In data science, it is convenient to collect data from multiple sources and store the data in a database. Analytics databases are organized to support efficient data analytics.

A database supporting data science should make it easy and efficient to view and select data to be feed into models. The structures supported by the database should make it easy to extract data to create vectors, matrices and tensors that are used by data science tools and packages.

Multiple systems, including ScalaTion's TSDB, are built on top of columnar, main memory databases in order to provide high performance. ScalaTion's TSDB is a Time Series DataBase that has built-in capabilities for handling time series data. It is able to store non-time series data as well. It provides multiple Application Programming Interfaces (APIs) for convenient access to the data [**?**].

4.1.2 The `Tabular` Trait

A common interface in the form of a Scala trait is provided for both Relational and Columnar Relational databases. A `Tabular` database will have a `name`, a `schema` or array of attribute/column names, a `domain` or array of domains/data types, and a `key` (primary key).

```
    @param name    the name of the table
    @param schema  the attributes for the table
    @param domain  the domains/data-types for the attributes ('D', 'I', 'L', 'S', 'X', 'T')
    @param key     the attributes forming the primary key

    trait Tabular [T <: Tabular [T]] (val name: String, val schema: Schema,
                                      val domain: Domain, val key: Schema)
          extends Serializable:
```

For convenience, the following two Scala type definitions are utilized.

```
    type Schema = Array [String]
    type Domain = Array [Char]
```

`Tabular` structures are logically linked together via foreign keys. A *foreign key* is an attribute that references a *primary key* in some table (typically another table). In ScalaTion, the foreign key specification is added via the following method call after the Tabular structure is created.

```
    def addForeignKey (fkey: String, refTab: T): Unit
```

ScalaTion supports the following domains/data-types: 'D'ouble, 'I'nt, 'L'ong, 'S'tring, and 'T'imeNum.

```
    'D' - `Double`   - `VectorD` -  64 bit double precision floating point number
    'I' - `Int`      - `VectorI` -  32 bit integer
    'L' - `Long`     - `VectorL` -  64 bit long integer
    'S' - `String`   - `VectorS` -  variable length numeric string
    'T' - `TimeNum`  - `VectorT` -  time numbers for date-time
```

These data types are generalized into a `ValueType` as a Scala union type.

```
    type ValueType = ( Double | Int | Long | String | TimeNum )
```

4.2 Relational Data Model

A relational database table may be built up as follows: A cell in the table holds an atomic value of type `ValueType`. A tuple (or row) in the table is simply an array of `ValueType`. A relational `Table` consists of a bag (or multi-set) of tuples. Each column in the `Table` is restricted to a particular domain. Note, uniqueness of primary keys is enforced by creating a primary index.

```
    type Tuple = Array [ValueType]
```

4.2.1 Data Definition Language

The Data Definition Language consists of a `Table` class constructor and associated `apply` methods in the companion object. The following example illustrates the creation of four tables based on the example Bank schema given in [74].

```
    val customer = Table ("customer", "cname, street, ccity",
                                      "S, S, S", "cname")
    val branch   = Table ("branch", "bname, assets, bcity",
                                    "S, D, S", "bname")
    val deposit  = Table ("deposit", "accno, balance, cname, bname",
                                     "I, D, S, S", "accno")
    val loan     = Table ("loan", "loanno, amount, cname, bname",
                                  "I, D, S, S", "loanno")
```

4.2.2 Data Manipulation Language

As with many database systems, the Data Manipulation Language consists of methods for insertion, update and deletion.

```
   def add (t: Tuple): Table =
   def update (atr: String, newVal: ValueType, matchVal: ValueType): Boolean =
   def update (atr: String, func: ValueType => ValueType, matchVal: ValueType): Boolean =
   def delete (predicate: Predicate): Boolean =
```

Using the operator `+=` as an alias for the `add` method the following code may be used to populate the Bank database.

```
    customer += ("Peter", "Oak St",   "Bogart")
             += ("Paul",  "Elm St",   "Watkinsville")
             += ("Mary",  "Maple St", "Athens")
    customer.show ()

    branch += ("Alps",     20000000.0, "Athens")
           += ("Downtown", 30000000.0, "Athens")
           += ("Lake",     10000000.0, "Bogart")
    branch.show ()

    deposit += (11, 2000.0, "Peter", "Lake")
            += (12, 1500.0, "Paul",  "Alps")
            += (13, 2500.0, "Paul",  "Downtown")
            += (14, 2500.0, "Paul",  "Lake")
            += (15, 3000.0, "Mary",  "Alps")
            += (16, 1000.0, "Mary",  "Downtown")
    deposit.show ()

    loan += (21, 2200.0, "Peter", "Alps")
         += (22, 2100.0, "Peter", "Downtown")
         += (23, 1500.0, "Paul",  "Alps")
         += (24, 2500.0, "Paul",  "Downtown")
         += (25, 3000.0, "Mary",  "Alps")
         += (26, 1000.0, "Mary",  "Lake")
    loan.show ()
```

4.2.3 Relational Algebra

Relational Algebra provides a set of operators for writing queries on tables, including extracting columns (project), rows (select), performing set operations on tables (union, minus, intersect and Cartesian product). Several forms of join operations for composing a new table from two existing tables are provided as well as division, group-by and order-by operations.

Table 4.1: Relational Algebra Operators (Tables r and $r2$)

Operator	**Unicode**	**Signature**
rename	ρ	def ρ (newName: String): T = rename (newName)
project	π	def π (x: String): T = project (strim (x))
project	π	def π (cPos: IndexedSeq [Int]): T = project (cPos)
selproject	$\sigma\pi$	def $\sigma\pi$ (a: String, apred: APredicate): T = selproject (a, apred)
select	σ	def σ (a: String, apred: APredicate): T = select (a, apred)
select	σ	def σ (predicate: Predicate): T = select (predicate)
select	σ	def σ (condition: String): T = select (condition)
select	σ	def σ (pkey: KeyType): T = select (pkey)
union	$\cup$	def $\cup$ (r2: T): T = union (r2)
minus	$-$	def $-$ (r2: T): T = minus (r2)
intersect	$\cap$	def $\cap$ (r2: T): T = intersect (r2)
product	$\times$	def $\times$ (r2: T): T = product (r2)
join	$\bowtie$	def $\bowtie$ (predicate: Predicate2, r2: T): T = join (predicate, r2)
join	$\bowtie$	def $\bowtie$ (condition: String, r2: T): T = join (condition, r2)
join	$\bowtie$	def $\bowtie$ (x: String, y: String, r2: T): T = join (strim (x), strim (y), r2)
join	$\bowtie$	def $\bowtie$ (fkey: (String, T)): T = join (fkey)
join	$\bowtie$	def $\bowtie$ (r2: T): T = join (r2)
leftJoin	$\ltimes$	def $\ltimes$ (x: Schema, y: Schema, r2: T): T = leftJoin (x, y, r2)
rightJoin	$\rtimes$	def $\rtimes$ (x: Schema, y: Schema, r2: T): T = rightJoin (x, y, r2)
divide	$/$	def $/$ (r2: T): T = divide (r2)
groupBy	γ	def γ (ag: String): T = groupBy (ag)
aggregate	$\mathcal{F}$	def $\mathcal{F}$ (ag: String, f_as: (AggFunction, String)*): T = aggregate (ag, f_as :_*)
orderBy	$\uparrow$	def $\uparrow$ (x: String*): T = orderBy (x :_*)
orderByDesc	$\downarrow$	def $\downarrow$ (x: String*): T = orderByDesc (x :_*)(true)

Fundamental Relational Algebra Operators

The following six relational algebra operators form the *fundamental operators* for ScalaTion's `table` package and are shown in Table 4.1. They are fundamental in sense that rest of operators, although convenient, do not increase the power of the query language.

1. **Rename Operator**. The *rename* operator renames table *customer* to *client*.

$$customer\, \rho\, (\text{“}client\text{”})$$

```
    customer ρ ("client")
```

2. **Project Operator**. The *project* operator will return the specified columns in table *customer*.

$$\pi_{\text{street, ccity}}(customer)$$

```
    customer π ("street, ccity")
```

3. **Select Operator**. The *select* operator will return the rows that match the predicate in table *customer*.

$$\sigma_{\text{ccity} == \text{‘Athens’}}(customer)$$

```
    customer σ ("ccity == 'Athens'")
```

4. **Union Operator**. The *union* operator will return the union of rows from *deposit* and *loan*. Duplicate tuples may be eliminated by creating an index. For this operator the textbook syntax and ScalaTion syntax are identical.

$$deposit \cup loan$$

```
    deposit ∪ loan
```

5. **Minus Operator**. The *minus* operator will return the rows from *account* (result of the union) that are not in *loan*. For this operator the textbook syntax and ScalaTion syntax are identical.

$$account - loan$$

```
    account - loan
```

6. **Cartesian Product Operator**. The *product* operator will return all combinations of rows in *customer* with rows in *deposit*. For this operator the textbook syntax and ScalaTion syntax are identical.

$$customer \times deposit$$

```
    customer × deposit
```

Additional Relational Algebra Operators

The next eight operators, although not fundamental, are important operators in SACALATION's `table` package and are shown in Table 4.1.

1. **Join Operator**. In order to combine information from two tables, *join* operators are preferred over products, as they are much more efficient and only combine related rows. SCALATION's `table` package supports natural-join, equi-join, theta-join, left outer join, and right outer join, as shown below. For each tuple in the left table, the equi-join pairs it with all tuples in the right table that match it on the given attributes (in this case *customer*.bname = *deposit*.bname). The natural-join is an equi-join on the common attributes in the two tables, followed by projecting away any duplicate columns. The theta-join generalizes an equi-join by allowing any comparison operator to be used (in this case $deposit_1$.balance < $deposit_2$.balance). The symbol for semi-join is adopted for outer joins as it is a Unicode symbol. The left join keeps all tuples from the left (null padding if need be), while the right join keeps all tuples from the right table.

$$\begin{array}{lr}
customer \bowtie deposit & \text{natural} - \text{join} \\
customer \bowtie_{cname\,==\,cname}\; deposit & \text{equi} - \text{join} \\
deposit \bowtie_{balance\,<\,balance}\; deposit & \text{theta} - \text{join} \\
customer \ltimes deposit & \text{left outer join} \\
customer \rtimes deposit & \text{right outer join}
\end{array}$$

```
    customer ⋈ deposit
    customer ⋈ ("cname == cname", deposit)
    deposit ⋈ ("balance < balance", deposit)
    customer ⋉ deposit
    customer ⋊ deposit
```

   Additional forms of joins are also available in the `Table` class. Join is not fundamental as its result can be made by combining product and select.

2. **Divide Operator**. For the query below, the *divide* operator will return the cnames where the customers has a deposit account at all branches (of course it would make sense to first select on the branches).

$$\pi_{cname,\,bname}(deposit) / \pi_{bname}(branch)$$

```
    deposit.π ("cname, bname") / branch.π ("bname")
```

   The divide operator requires the other attributes (in this case cname) in the left table to be paired up with all the attribute values (in this case bname) in the right table.

3. **Intersect Operator**. The *intersect* operator will return the rows in *account* that are also in *loan*. For this operator the textbook syntax and SCALATION syntax are identical.

$$account \cap loan$$

```
    account ∩ loan
```

Intersection is not fundamental as its result can be made by successive minuses.

4. **GroupBy Operator**. The *groupBy* operator forms groups among the relation based on the equality of attributes. The following example groups the tuples in the deposit table based on the value of the bname attribute.

$$\gamma_{bname}(deposit)$$

```
    deposit γ "bname"
```

5. **Aggregate Operator**. The *aggregate* operator returns values for the grouped-by attribute (e.g., bname) and applies aggregate operators on the specified columns (e.g., avg (balance)). Typically it is called after the *groupBy* operator.

$$\mathcal{F}_{bname,\, count(accno),\, avg(balance)}(deposit)$$

```
    deposit ℱ ("bname", (count, "accno"), (avg, "balance"))
```

6. **OrderBy Operator**. The *orderBy* operator effectively puts the rows into ascending order based on the given attributes.

$$\uparrow_{bname}(deposit)$$

```
    deposit ↑ "bname"
```

7. **OrderByDesc Operator**. The *orderByDesc* operator effectively puts the rows into descending order based on the given attributes.

$$\downarrow_{bname}(deposit)$$

```
    deposit ↓ "bname"
```

8. **Select-Project Operator**. The *selproject* is a combination operator added for convenience and efficiency, especially for columnar relation databases (see the next section). As whole columns are stored together, this operator only requires one column to be accessed.

```
    customer σπ ("ccity", _ == 'Athens')
```

4.2.4 Example Queries

1. List the names of customers who live in the city of Athens.

```
    val liveAthens = customer.σ ("ccity == 'Athens'").π ("cname")
    liveAthens.show ()
```

2. List the names of customers who live in Athens or bank (have deposits in branches located) in Athens.

```
    val bankAthens = (deposit ⋈ branch).σ ("bcity == 'Athens'").π ("cname")
    bankAthens.show ()
```

3. List the names of customers who live and bank in the same city.

```
    val sameCity = (customer ⋈ deposit ⋈ branch).σ ("ccity == bcity").π ("cname")
    sameCity.create_index ()
    sameCity.show ()
```

4. List the names and account numbers of customers with the largest balance.

```
    val largest = deposit.π ("cname, accno") - (deposit ⋈ ("balance < balance",
    deposit)).π ("cname, accno")
    largest.show ()
```

5. List the names of customers who are silver club members (have loans where they have deposits).

```
    val silver = (loan.π ("cname, bname") ∩ deposit.π ("cname, bname")).π ("cname")
    silver.create_index ()
    silver.show ()
```

6. List the names of customers who are gold club members (have loans only where they have deposits).

```
    val gold = loan.π ("cname") - (loan.π ("cname, bname") - deposit.π ("cname, bname")
    ).π ("cname")
    gold.create_index ()
    gold.show ()
```

7. List the names of branches located in Athens.

```
    val inAthens = branch.σ ("bcity == 'Athens'").π ("bname")
    inAthens.show ()
```

8. List the names of customers who have deposits at all branches located in Athens.

```
    val allAthens = deposit.π ("cname, bname") / inAthens
    allAthens.create_index ()
    allAthens.show ()
```

9. List the branch names and their average balances.

```
    val avgBalance = deposit.γ ("bname").aggregate ("bname", (count, "accno"), (avg, "
    balance"))
    avgBalance.show ()
```

4.2.5 Persistence

Modern databases do much of the processing in main-memory due to its large size and high speed. Although using MRAM, main-memories may be persistent, typically they are volatile, meaning if the power is lost, so is the data. It is therefore essential to provide efficient mechanisms for making and maintaining the persistence of data.

Traditional database management systems achieve this by having a persistence data store in non-volatile storage (e.g., Hard-Disk Drives (HDD) or Solid-State Devices (SSD)) and a large database cache in main-memory. Complex page management algorithms are used to ensure persistence and transactional correctness (see the next subsection).

A simple way to provide persistence is to design the database management system to operate in main-memory and then provide `load` and `save` methods that utilize built-in serialization to save to or load from persistent storage. This is what ScalaTion does.

The `load` method will read a table with a given name into main-memory using serialization.

```
@param name  the name of the table to load

def load (name: String): Table =
    val ois = new ObjectInputStream (new FileInputStream (STORE_DIR + name + SER))
    val tab = ois.readObject.asInstanceOf [Table]
    ois.close ()
    tab
end load
```

The `save` method will write the entire contents of this table into a file using serialization.

```
def save (): Unit =
    val oos = new ObjectOutputStream (new FileOutputStream (STORE_DIR + name + SER))
    oos.writeObject (this)
    oos.close ()
end save
```

For small databases, this approach is fine, but as database become large, greater efficiency must be sought. One cannot `save` a whole table ever time there is a change. See the exercises for alternatives.

4.2.6 Transactions

The idea of a transaction is to bundle a sequence of operations into a meaningful action that one wants to succeed, such as transferring money from one bank account to another.

Making the action a transaction has the main benefit of making it *atomic*, the action either completes successfully (called a *commit*) or is completely undone having no effect on the database state (called a *rollback*). The third option, a partially completed action in this case would lead to a bank customer losing their money.

Making a transaction atomic can be achieved by maintaining a *log*. Operations can be written to the log and then only saved once the transaction commits. If a transaction cannot commit, it must be rolled back. There must also be a recover procedure to handles the situation when volatile storage is lost. For this to function, committed log records must be flushed to persistent storage.

A second important advantage of making an action a transaction is to protect it from other transactions, so it can think of itself as if it is running in `isolation`. Rather than worrying about how other transactions may corrupt the action, this worry is turned over to database management system to handle it. One form of potential interference involves two transactions running concurrently and accessing the same back accounts. It one transaction accesses all the accounts first, there will be no corruption. Such an execution of two transactions is called a *serial* execution (one transaction executes at a time). Unfortunately, modern high-performance database management systems could not operate at the slow speed this would dictate. Transaction must be run concurrently, not serially. The correction condition caller *serializability* allows transaction to run with their concurrency controlled by a protocol that ensures their effects on the database are equivalent to one of their slow-running, serially-executing cousin schedules. In other words, the fast running serializable schedule for a set of transactions must be equivalent to some serial execution of the same set of transactions. See the exercise for more details on equivalence (e.g., conflict and view equivalence) and various *concurrency control protocols* that can be used to ensure correctness with minimal impact on performance.

4.2.7 `Table` Class

```
    @param name     the name of the table
    @param schema   the attributes for the table
    @param domain   the domains/data-types for the attributes ('D', 'I', 'L', 'S', 'X', 'T')
    @param key      the attributes forming the primary key

    class Table (override val name: String, override val schema: Schema,
                 override val domain: Domain, override val key: Schema)
         extends Tabular [Table] (name, schema, domain, key)
            with Serializable:
```

Internally, the `Table` class maintains a collection of `tuples`. Using a `Bag` allows for duplicates, if wanted. Creating an index on the primary will efficiently eliminate any duplicates. Foreign key relationships are specified in `linkTypes`. It also provides a `groupMap` used by the `groupBy` operator.

The `Table` class supports three types of indices:

1. Primary Index. A uniques index on the primary key (may be composite).

```
    private [table] val index = IndexMap [KeyType, Tuple] ()
```

2. Secondary Unique Indices. A unique index on a single attribute (other than the primary key). For example, a `student_id` may be used as a primary for a `Student` table, while `email` may also be required to be unique. Since there can be multiple such indices a `Map` is used to name each index.

```
    private [table] val sindex = Map [String, IndexMap [ValueType, Tuple]] ()
```

3. Non-Unique Indices. When fast-lookup is required based on an attribute/column that is not required to be unique (e.g., `name`) such an index may be used. Again, since there can be multiple such indices a `Map` is used to name each index.

```
    private [table] val mindex = Map [String, MIndexMap [ValueType, Tuple]] ()
```

The following methods may be used to create the various types of indices: primary unique index, secondary unique index, or non-unique index, respectively.

```
    def create_index (rebuild: Boolean = false): Unit =
    def create_sindex (atr: String): Unit =
    def create_mindex (atr: String): Unit =
```

The following factory method in the companion object provides a more convenient way to create a table. The `strim` method splits a string into an array of strings based on a separation character and then trims away any white-space.

```
    def apply (name: String, schema: String, domain_ : String, key: String): Table =
        new Table (name, strim (schema), strim (domain_).map (_.head), strim (key))
    end apply
```

The following two classes extend the `Table` class in the direction of the Graph Data Model, see Appendix C.

4.2.8 `LTable` Class

```
    @param name_    the name of the linkable-table
    @param schema_  the attributes for the linkable-table
    @param domain_  the domains/data-types for attributes ('D', 'I', 'L', 'S', 'X', 'T')
    @param key_     the attributes forming the primary key

    case class LTable (name_ : String, schema_ : Schema, domain_ : Domain, key_ : Schema)
         extends Table (name_, schema_, domain_, key_)
            with Serializable:
```

The `LTable` class (for Linked-Table) simply adds an explicit link from the foreign key to the primary key that it references. For each tuple in a linked-table, add a link to the referenced table, so that the foreign key is linked to the primary key. Caveat: `LTable` does not handle composite foreign keys. Although in general primary keys may be composite, a foreign key is conceptualized as a column value and its associated link.

```
    @param fkey    the foreign key column
    @param refTab  the referenced table being linked to

    def addLinks (fkey: String, refTab: Table): Unit =
```

The `LTable` class makes many-to-one relationships/associations explicit and improves the efficiency of the most common form of join operation which is based on equating a foreign key (fkey) to a primary key (pkey). Without an index, these are performed using a Nest-Loop Join algorithm. The existence of an index on the primary key allows a much more efficient Indexed Join algorithm to be utilized. The direct linkage provides for additional speed up of such join operations (see the exercises for a comparison). Note that the linkage is only in one direction, so joining from the primary key table to the foreign key table would require a non-unique index on the foreign key column, or resorting to a slow nested loop join.

Note, the link and foreign key value are in some sense redundant. Removing the foreign key column is possible, but may force the need for an additional join for some queries, so the database designer may wish to keep the foreign key column. ScalaTion leaves this issue up to the database designer.

The next class moves further in the direction of the Graph Data Model.

4.2.9 `VTable` Class

```
    @param name_    the name of the vertex-table
    @param schema_  the attributes for the vertex-table
    @param domain_  the domains/data-types for attributes ('D', 'I', 'L', 'S', 'X', 'T')
    @param key_     the attributes forming the primary key

    case class VTable (name_ : String, schema_ : Schema, domain_ : Domain, key_ : Schema)
         extends Table (name_, schema_, domain_, key_)
            with Serializable:
```

The `VTable` class (for Vertex-Table) supports many-to-many relationships with efficient navigation in both directions. Supporting this is much more completed than what is needed for `LTable`, but provides for *index-free adjacency*, similar to what is provided by Graph Database systems.

The `VTable` model is graph-like in that it elevates tuples into vertices as first-class citizens of the data model. However, edges are embedded inside of vertices and are there to establish adjacency. Edges do not have labels, attributes or properties. Although this simplifies the data model and makes it more relation-like, it is not set up to naturally support finding for example shortest paths.

The `Vertex` class extends the notion of `Tuple` into values stored in the tuple part, along with foreign keys links captured as outgoing edges.

```
@param tuple  the tuple part of a vertex

case class Vertex (tuple: Tuple):

    val edge = Map [String, Set [Vertex]] ()

end Vertex
```

For data models where edges become first-class citizens, see the Appendix on Graph Data Models.

4.3 Columnar Relational Data Model

Of the NoSQL database management systems, columnar databases are closest to traditional relational databases. Rather than tuples/rows taking center stage, columns/vectors take center stage.

A columnar database is made up of the following components:

- Element - a value from a given Domain or Datatype (e.g., `Int`, `Long`, `Double`, `Rational`, `Real`, `Complex`, `String`, `TimeNum`)
- Column/Vector - a collection of values from the same Datatype (e.g., forming `VectorI`, `VectorL`, `VectorD`, `VectorQ`, `VectorR`, `VectorC`, `VectorS`, `VectorT`)
- Columnar Relation - a heterogeneous collection of columns/vectors put into a table-like structure.
- Columnar Database - a collection of columnar relations.

Table 4.2 shows the first 10 rows (out of 392) for the well-known Auto_MPG dataset (see `https://archive.ics.uci.edu/ml/datasets/Auto+MPG`).

Table 4.2: Example Columnar Relation: First 10 Rows of Auto_MPG Dataset

mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
Double	Int	Double	Double	Double	Double	Int	Int	String
18.0	8	307.0	130.0	3504.0	12.0	70	1	"chevrolet chevelle"
15.0	8	350.0	165.0	3693.0	11.5	70	1	"buick skylark 320"
18.0	8	318.0	150.0	3436.0	11.0	70	1	"plymouth satellite"
16.0	8	304.0	150.0	3433.0	12.0	70	1	"amc rebel sst"
17.0	8	302.0	140.0	3449.0	10.5	70	1	"ford torino"
15.0	8	429.0	198.0	4341.0	10.0	70	1	"ford galaxie 500"
14.0	8	454.0	220.0	4354.0	9.0	70	1	"chevrolet impala"
14.0	8	440.0	215.0	4312.0	8.5	70	1	"plymouth fury iii"
14.0	8	455.0	225.0	4425.0	10.0	70	1	"pontiac catalina"
15.0	8	390.0	190.0	3850.0	8.5	70	1	"amc ambassador dpl"

Since each column is stored as a vector, they can be readily compressed. Due to the high repetition in the `cylinders` column it can be effectively compressed using Run Length Encoding (RLE) compression. In addition, a column can be efficiently extracted since it already stored as a vector in the database. These vectors can be used in aggregate operators or passed into analytic models.

Data files in various formats (e.g., comma separated values (csv)) can be loaded into the database.

```
val auto_mpg = Relation ("auto_mpg", "auto_mpg.csv")
```

It is easy to create a Multiple Linear Regression model for this dataset. Simply pick the response column, in this case `mpg` and the predictor columns, in this case all other columns besides `car_name`. The connection between `car_name` and `mpg` is coincidental. The response column/variable goes into a vector.

```
val y = auto_mpg.toVectorD (0)
```

The predictor columns/variables goes into a matrix.

```
    val x = auto_mpg.toMatrixD (1 to 7))
```

Then the matrix `x` and vector `y` can be passed into a `Regression` model constructor.

```
    val rg = new Regression (x, y)
```

See the next chapter for how to train a model, evaluate the quality of fit and make predictions.

The first API is a *Columnar Relational Algebra* that includes the standard operators of relational algebra plus those common to column-oriented databases. It consists of the `Table` trait and two implementing classes: `Relation` and `MM_Relation`. Persistence for `Relation` is provided by the `save` method, while `MM_Relation` utilizes memory-mapped files.

4.3.1 Data Definition Language

A `Relation` object is created by invoking a constructor or factory apply function. For example, the following six `Relations` may be useful in a traffic forecasting study.

```
val sensor  = Relation ("sensor",
                        Seq ("sensorId", "model", "latitude", "longitude", "roadId"),
                        Seq (), 0, "ISDDI")
val road    = Relation ("road",
                        Seq ("roadId", "rdName", "lat1", "long1", "lat2", "long2"),
                        Seq (), 0, "ISDDDD")
val mroad   = Relation ("mroad",
                        Seq ("roadId", "rdName", "lanes", "lat1", "long1", "lat2", "long2"),
                        Seq (), 0, "ISIDDDD")
val traffic = Relation ("traffic",
                        Seq ("time", "sensorId", "count" "speed"),
                        Seq (), Seq (0, 1), "TIID")
val wsensor = Relation ("wsensor",
                        Seq ("sensorId", "model", "latitude", "longitude"),
                        Seq (), 0, "ISDD")
val weather = Relation ("weather",
                        Seq ("time", "sensorId", "precipitation" "wind"),
                        Seq (), Seq (0, 1), "TIDD")
```

The name of the first relation is “sensor” and it stores information about traffic sensors.

- The first argument is the name of the relation (`name`).
- The second argument is the sequence of attribute/column names (`colName`).
- The third argument is the sequence of data, currently empty (`col`),
- The fourth argument is the column number for the primary key (`key`),
- The fifth argument, “ISDDI”, indicates the domains (`domain`) for the attributes (`Integer`, `String`, `Double`, `Double`, `Integer`).
- The sixth and optional argument can be used to define foreign keys (`fKeys`).
- The seventh and optional argument indicates whether to `enter` that relation is the system `Catalog`.

The second relation `road` stores the Id, name, beginning and ending latitude-longitude coordinates.

The third relation `mroad` is for multi-lane roads.

The fourth relation `traffic` stores the data collected from traffic sensors. The primary key in this case is composite, Seq (0, 1), as both the time and the sensorId are required for unique identification.

The fifth relation `wsensor` stores information about weather sensors.

Finally, the sixth relation `weather` stores data collected from the weather sensors.

4.3.2 Data Manipulation Language

There are several ways to populate the `Relations`. A row/tuple can be added one at a time using `def add (tuple: Row)`. Population may also occur during relation construction (via a constructor or apply method). There are factory apply functions that take a file or URL as input.

For example to populate the `sensor` relation with information about Austin, Texas' traffic sensors stored in the file `austin_traffic_sensors.csv` the following line of code may be used.

```
    val sensor = Relation (‘‘sensor", ‘‘austin\_traffic\_sensors.csv")
```

Data files are stored in subdirectories of ScalaTion's `data` directory.

4.3.3 Columnar Relational Algebra

Table 4.3 shows the thirteen operators supported (the first six are considered fundamental). Operator names as well as Unicode symbols may be used interchangeably (e.g., r *union* s or $r \cup s$ compute the union of relations r and s). Note, the extended projection operator *eproject* (Π) provides a convenient mechanism for applying aggregate functions. It is often called after the *groupBy* operator, in which case multiple rows will be returned. Multiple columns may be specified in *eproject* as well. There are also several varieties of *join* operators. As an alternative to using the Unicode symbol when they are Greek letters, the letter may be written out in English (*pi*, *sigma*, *rho*, *gamma*, *epi*, *omega*, *zeta*, *unzeta*).

The subsections below present the columnar relational algebra operators, first showing the textbook notation followed by the syntax in ScalaTion's `column_db` package. To make the examples complex more concise, let r = road, s = sensor, t = traffic, q = mroad, v = wsensor and w = weather.

Select Operator

The *select* operator will return the rows that match the predicate, in this case $rdName ==$ "$I285$".

$$\sigma_{rdName == \text{``}I285\text{''}}(r)$$

$$r.\sigma(\text{``}rdName\text{''}, \_ == \text{``}I285\text{''})$$

Project Operator

The *project* operator will return the specified columns, in this case $rdName, lat1, long1$.

$$\pi_{rdName, lat1, long1}(r)$$

$$r.\pi(\text{``}rdName\text{''}, \text{``}lat1\text{''}, \text{``}long1\text{''})$$

Union Operator

The *union* operator will return the rows from r and s with no duplicates. For this operator the textbook syntax and `column_db` syntax are identical.

$$r \cup s$$

Minus Operator

The *minus* operator will return the rows from r that are not in s. For this operator the textbook syntax and `column_db` syntax are identical.

$$r - s$$

Cartesian Product Operator

The *product* operator will return all combinations of rows in r with rows in s. For this operator the textbook syntax and `column_db` syntax are identical.

$$r \times s$$

Rename Operator

The *rename* operator renames relation r's name to $r2$.

$$r.\rho(\text{“}r2\text{”})$$

The above six operators form the fundamental operators for SACALATION's `column_db` package and are shown as the first group in Table 4.3.

Table 4.3: Columnar Relational Algebra (r = road, s = sensor, t = traffic, q = mroad, w = weather)

Operator	**Unicode**	**Example**	**Return**
select	σ	$r.\sigma$ (“rdName”, _ == “I285”)	rows of r where rdName == “I285”
project	π	$r.\pi$ (“rdName”, “lat1”, “long1”)	the rdName, lat1, and long1 columns of r
union	$\cup$	$r \cup q$	rows that are in r or q
minus	-	$r - q$	rows that are in r but not q
product	$\times$	$r \times t$	concatenation of each row of r with those of t
rename	ρ	$r.\rho$(“$r2$”)	a copy of r with new name $r2$
join	$\bowtie$	$r \bowtie s$	rows in natural join of r and s
intersect	$\cap$	$r \cap q$	rows that are in r and q
groupBy	γ	$t.\gamma$ (“sensorId”)	rows of t grouped by sensorId
eproject	Π	$t.\Pi$ (avg, “acount”, “count”)(“sensorId”)	the average of the count column of t
orderBy	ω	$t.\omega$ (“sensorId”)	rows of t ordered by sensorId
compress	ζ	$t.\zeta$ (“count”)	compress the count column of t
uncompress	Z	$t.Z$ (“count”)	uncompress the count column of t

The next seven operators, although not fundamental, are important operators in SACALATION's `column_db` package and are shown as the second group in Table 4.3.

Join Operators

In order to combine information from two relations, *join* operators are preferred over products, as they are much more efficiently and only combine related rows. SCALATION's `column_db` package supports natural-join, equi-join, general theta join, left outer join, and right outer join, as shown below.

$$\begin{aligned} &r \bowtie s && \text{natural} - \text{join} \\ &r \bowtie (\text{“}roadId\text{”}, \text{“}roadId\text{”}, s) && \text{equi} - \text{join} \\ &r \bowtie [\text{Int}](s, (\text{“}roadId\text{”}, \text{“}roadId\text{”}, \_ == \_)) && \text{theta join} \\ &t \ltimes (\text{“}time\text{”}, \text{“}time\text{”}, w) && \text{left outer join} \\ &t \rtimes (\text{“}time\text{”}, \text{“}time\text{”}, w) && \text{right outer join} \end{aligned}$$

Intersect Operator

The *intersect* operator will return the rows in r that are also in s. For this operator the textbook syntax and `column_db` syntax are identical.

$$r \cap s$$

GroupBy Operator

The *groupBy* operator forms groups among the relation based on the equality of attributes. The following example groups traffic data based in the value of the "*sensorId*" attribute.

$$t.\gamma(\text{“}sensorId\text{”})$$

Extended Projection Operator

The *extended projection* operator *eproject* applies aggregate operators on aggregation columns (first arguments) and regular project on the other columns (second arguments). Typically it is called after the *groupBy* operator.

$$t.\gamma(\text{“}sensorId\text{”}).\Pi(avg, \text{“}acount\text{”}, \text{“}count\text{”})(\text{“}sensorId\text{”})$$

OrderBy Operator

The *orderBy* operator effectively puts the rows into ascending (descending) order based on the given attributes.

$$t.\omega(\text{“}sensorId\text{”})$$

Compress Operator

The *compress* operator will compress the given columns of the relation.

$$t.\zeta(\text{“}count\text{”})$$

Uncompress Operator

The *uncompress* operator will uncompress the given columns of the relation.

$$t.Z(\text{“}count\text{”})$$

4.3.4 Example Queries

Several example queries for the traffic study are given below.

1. Retrieve the automobile mileage data for cars with 8 cylinders.

   auto_mpg.select (“cylinders”, _ == 8)

   Note, select and σ may be use interchangeably

2. Retrieve the automobile mileage data for cars with 8 cylinders, returning the car_name and mpg.

   auto_mpg.select (“cylinders”, _ == 8).project (“car_name”, “mpg”)

   Note, project and π may be use interchangeably

3. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).

   val austin = latLong2UTMxy (LatitudeLongitude (30.266667, -97.733333))
   val alat = (austin._1 - 100000, austin._1 + 100000)
   val along = (austin._2 - 100000, austin._2 + 100000)
   traffic ⋈ sensor.σ [Double] (“latitude”, _ ∈ alat).σ [Double] (“longitude” _ ∈ along)

4.3.5 `Relation` Class

Class Methods:

```
    @param name      the name of the relation
    @param colName   the names of columns
    @param col       the Scala Vector of columns making up the columnar relation
    @param key       the column number for the primary key (< 0 => no primary key)
    @param domain    an optional string indicating domains for columns (e.g., 'SD' = 'String'
    , 'Double')
    @param fKeys     an optional sequence of foreign keys
                       - Seq (column name, ref table name, ref column position)
    @param enter     whether to enter the newly created relation into the ‘Catalog‘

    class Relation (val name: String, val colName: Seq [String], var col: Vector [Vec] =
    null,
                    val key: Int = 0, val domain: String = null,
                    var fKeys: Seq [(String, String, Int)] = null, enter: Boolean = true)
          extends Table with Error with Serializable
```

4.4 SQL-Like Language

The SQL-Like API in ScalaTion provides many of the language constructs of SQL in a functional style.

4.4.1 Relation Creation

A `RelationSQL` object is created by invoking a constructor or factory apply function. For example, the following six `RelationSQL`s may be useful in a traffic forecasting study.

```
    val sensor  = RelationSQL ("sensor",
                               Seq ("sensorId", "model", "latitude", "longitude", "roadId"),
                               null, 0, "ISDDI")
    val road    = RelationSQL ("road",
                               Seq ("roadId", "rdName", "lat1", "long1", "lat2", "long2"),
                               null, 0, "ISDDDD")
    val mroad   = RelationSQL ("mroad",
                               Seq ("roadId", "rdName", "lanes", "lat1", "long1", "lat2", "
long2"),
                               null, 0, "ISIDDDD")
    val traffic = RelationSQL ("traffic",
                               Seq ("time", "sensorId", "count", "speed"),
                               null, 0, "TIID")
    val wsensor = RelationSQL ("wsensor",
                               Seq ("sensorId", "model", "latitude", "longitude"),
                               null, 0, "ISDD")
    val weather = RelationSQL ("weather",
                               Seq ("time", "sensorId", "precipitation", "wind"),
                               null, 0, "TIDD")
```

4.4.2 Sample Queries

The ScalaTion columnar database provides a functional SQL-like query language.

1. Retrieve the vehicle traffic counts over time from all sensors on the road with Id = 101.

```
    (traffic join sensor).where [Int] ("roadId", _ == 101)
                         .select ("sensorId", "time", "count")
```

   In SQL, this would be written as follows:

```
    select sensorId, time, count
    from traffic natural join sensor
    where roadId == 101
```

2. Retrieve the vehicle traffic counts averaged over time from all sensors on the road with Id = 101.

```
    (traffic join sensor).where [Int] ("roadId", _ == 101)
                         .groupBy ("sensorId")
                         .eselect ((avg, "acount", "count"))("sensorId")
```

4.4.3 RelationSQL Class

Class Methods:

```
@param name      the name of the relation
@param colName   the names of columns
@param col       the Scala Vector of columns making up the columnar relation
@param key       the column number for the primary key (< 0 => no primary key)
@param domain    an optional string indicating domains for columns (e.g., 'SD' = 'String'
, 'Double')
@param fKeys     an optional sequence of foreign keys - Seq (column name, ref table name,
 ref column position)

class RelationSQL (name: String, colName: Seq [String], col: Vector [Vec],
                   key: Int = 0, domain: String = null, fKeys: Seq [(String, String, Int
)] = null)
      extends Tabular with Serializable

def repr: Relation = r
def this (r: Relation) = this (r.name, r.colName, r.col, r.key, r.domain, r.fKeys)
def select (cName: String*): RelationSQL =
def eselect (aggCol: AggColumn*)(cName: String*): RelationSQL =
def join (r2: RelationSQL): RelationSQL =
def join (cName1: String, cName2: String, r2: RelationSQL): RelationSQL =
def join (cName1: Seq [String], cName2: Seq [String], r2: RelationSQL): RelationSQL =
def where [T: ClassTag] (cName: String, p: T => Boolean): RelationSQL =
def where2 [T: ClassTag] (p: Predicate [T]*): RelationSQL =
def groupBy (cName: String*): RelationSQL =
def orderBy (cName: String*): RelationSQL =
def orderByDesc (cName: String*): RelationSQL =
def union (r2: RelationSQL): RelationSQL =
def intersect (r2: RelationSQL): RelationSQL =
def intersect2 (r2: RelationSQL): RelationSQL =
def minus (r2: RelationSQL): RelationSQL =
def minus2 (r2: RelationSQL): RelationSQL =
def stack (cName1: String, cName2: String): RelationSQL =
def insert (rows: Row*)
def materialize ()
def exists: Boolean = r.exists
```

```
def toMatrixD (colPos: Seq [Int], kind: MatrixKind = DENSE): MatrixD =
def toMatrixDD (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatrixD,
VectorD) =
def toMatrixDI (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatrixD,
VectorI) =
def toMatrixI (colPos: Seq [Int], kind: MatrixKind = DENSE): MatrixI =
def toMatrixI2 (colPos: Seq [Int] = null, kind: MatrixKind = DENSE): MatrixI =
def toMatrixII (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatrixI,
VectorI) =
def toVectorD (colPos: Int = 0): VectorD = r.toVectorD (colPos)
def toVectorD (colName: String): VectorD = r.toVectorD (colName)
def toVectorI (colPos: Int = 0): VectorI = r.toVectorI (colPos)
def toVectorI (colName: String): VectorI = r.toVectorI (colName)
def toVectorL (colPos: Int = 0): VectorL = r.toVectorL (colPos)
```

```
    def toVectorL (colName: String): VectorL = r.toVectorL (colName)
    def toVectorS (colPos: Int = 0): VectorS = r.toVectorS (colPos)
    def toVectorS (colName: String): VectorS = r.toVectorS (colName)
    def toVectorT (colPos: Int = 0): VectorT = r.toVectorT (colPos)
    def toVectorT (colName: String): VectorT = r.toVectorT (colName)
    def show (limit: Int = Int.MaxValue) r.show (limit)
    def save () r.save ()
    def generateIndex (reset: Boolean = false) r.generateIndex (reset)
```

4.5 Exercises

1. Use Scala 3 to complete the implementation of the following ScalaTion data models: `Table`, `LTable`, and VTable in the `scalation.table` package. A group will work on one the data models. See Appendix C for two more data models: `GTable` and `PGraph`.

   - Test all the operators.
   - Test all types of unique indices (`IndexMap`). Use the `import` scheme shown in the beginning of `Table.scala`.

   Table 4.4: Types of Indices (for Unique, Non-Unique Indices)

IndexMap	**MIndexMap**	**Description**
LinHashMap	LinHashMultiMap	ScalaTion's Linear Hash Map
HashMap	HashMultiMap	Scala's Hash Map
JHashMap	JHashMultiMap	Java's Hash Map
BpTreeMap	BpTreeMultiMap	ScalaTion's B^+Tree Map
TreeMap	TreeMultiMap	Scala's Tree Map
JTreeMap	JTreeMultiMap	Java's Tree Map

   - Test all types of non-unique indices (`MIndexMap`). Use the `import` scheme shown in the beginning of `Table.scala`.
   - Add use of indexing to speed up as many operations as possible.
   - Speed up joins by using Unique Indices and Non-Unique Indices.
   - Use index-free adjacency when possible for further speed-up.
   - Make the `save` operation efficient, by only serializing tuples/vertices that have changed since the last load. One way to approach this would be to maintain a map in persistent storage,

     ```
         Map [KeyType , [TimeNum , Tuple]]
     ```

     where the key for a tuple/vertex may be used to check the timestamp of a tuple/vertex. Unless the timestamp of the volatile tuple/vertex is larger, there is no need to save it. Further speed improvement may be obtained by switching from Java's text-based serialization to Kryo's binary serialization.

2. Conflict vs. View Equivalence. TBD.

3. Comparison of Concurrency Control Protocols. TBD.

4. Create the sensor schema using the `RelationSQL` class in the `columnar_db` package.

5. Populate the sensor database with sample data. See
   `https://data.austintexas.gov/Transportation-and-Mobility/Traffic-Count-Study-Area/cqdh-farx`

6. Retrieve the sensors that are on I35.

7. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).

8. Consider the following schema:

```
    val student   = Table ("student",   "sid, sname, street, city, dept, level",
                                        "I, S, S, S, S, I", "sid")
    val professor = Table ("professor", "pid, pname, street, city, dept",
                                        "I, S, S, S, S", "pid")
    val course    = Table ("course",    "cid, cname, hours, dept, pid",
                                        "I, X, I, S, I", "cid")
    val takes     = Table ("takes",     "sid, cid",
                                        "I, I", "sid, cid")
```

Formulate a relation algebra expression to list the names of the professors of courses taken by Peter.

Chapter 5

Data Preprocessing

5.1 Basic Operations

Using the ScalaTion TSDB, data scientists may write queries that extract data from one or more columnar relations. These data are used to create vectors and matrices that may be passed to various analytics techniques. Before the vectors and matrices are created the data need to be preprocessed to improve data quality and transform the data into a form more suitable for analytics.

5.1.1 Remove Identifiers

Any column that is unique (e.g., a primary key) with arbitrary values should be removed before applying a modeling/analytics technique. For example, an employee ID in a Neural Network analysis to predict salary could result in a perfect fit. Upon knowing the employee ID, the salary is a known. As the ID itself (e.g., ID = 1234567) is arbitrary, such a model has little value.

5.1.2 Convert String Columns to Numeric Columns

In ScalaTion, columns with strings (`VectorS`) should be converted to integers. For displaying final results, however, is often useful to convert the integers back to the original strings. The capabilities are provided by the `map2Int` function in the `VectorS` class (see the section on `RegressionCat`).

5.1.3 Identify Missing Values

Missing Values are common is real datasets. For some datasets, a question mark character '?' is used to indicate that a value is missing. In Comma Separated Value (CSV) files, repeated commas may indicate missing values, e.g., 10.1, 11.2,,,9.8. If zero or negative numbers are not valid for the application, these may be used to indicate missing values.

5.1.4 Preliminary Feature Selection

Before selecting a modeling/analytics technique, certain columns may be thrown away. Examples include columns with too many missing values or columns with near zero variance. Further discuss on this topic can be found in the section on Exploratory Data Analysis (EDA).

5.2 Methods for Outlier Detection

Data points that are considered outliers may happen because of errors or highly unusual occurrences. For example, suppose a dataset records the times for members of a football team to run a 100-yard dash and one of the recorded values is 3.2 seconds. This is an outlier. Some analytics techniques are less sensitive to outliers, e.g., ℓ^1 Regression, while others, e.g., ℓ^2 Regression, are more sensitive. Detection of outliers suffers from the obvious problems of being too strict (in which case good data may be thrown away) or too lenient (in which case outliers are passed to an analytics technique). One may choose to handle outliers separately, or turn them into missing values, so that both outliers and missing values may be handled together.

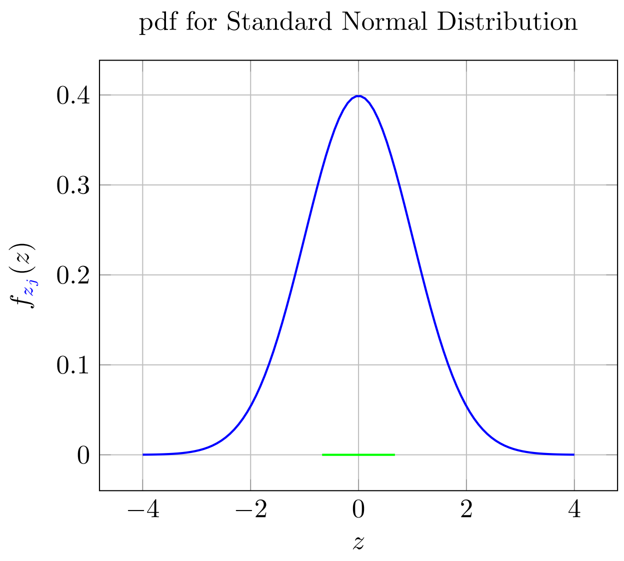
5.2.1 Based on Standard Deviation

If measured values for a random variable x_j are approximately Normally distributed and are several standard deviation units away form the center (μ_{x_j}), they are rare events. Depending on the situation, this may be important information to examine, but may often indicate incorrect measurement. Table 5.1 shows how unlikely it is to obtain data points in distant tails of a Normal distribution. The standard way to detect outliers using the standard deviation method is to examine points beyond three standard deviation (σ_{x_j}) units for being outliers. This is also called the z-score method as x_j needs to be transformed to z_j that follows the Standard Normal distribution.

$$z_j \quad = \quad \frac{x_j - \mu_{x_j}}{\sigma_{x_j}} \tag{5.1}$$

Table 5.1: Probabilities/Percentiles for the Standard Normal Distribution

± distance	**percent inside**	**percent in tails**	**outside per 10,000**
0.67448	50.00	50.00	5000
1.00000	68.27	31.73	3173
1.50000	86.64	13.36	1336
1.95996	95.00	5.00	500
2.00000	95.45	4.55	455
2.50000	98.76	1.24	124
2.57500	99.00	1.00	100
2.70000	99.31	0.69	69
3.00000	99.73	0.27	27
3.50000	99.95	0.05	5



5.2.2 Based on InterQuartile Range

The InterQuartile Range (IQR) shown in green for the Standard Normal distribution is 1.34896 (±0.67448). It includes the second (${}_{.25}\mathbb{Q}\,[x_j]$) and third (${}_{.75}Qx_j$) quartiles, i.e., the middle two out of four quartiles. The IQR gives a basic distance or yardstick for measuring when points are too far away from the median. A data point x_j should be examined as an outlier when the following rule is true.

$$x_j \;\notin\; [\,{}_{.25}\mathbb{Q}\,[x_j] - \delta \cdot \text{IQR},\; {}_{.75}\mathbb{Q}\,[x_j] + \delta \cdot \text{IQR}\,] \tag{5.2}$$

For the Normal distribution case, when the scale factor $\delta = 1.5$, it corresponds to 2.69792 standard deviation units and at 2.0 it corresponds to 3.3724 standard deviation units (see the exercises). The advantage of this method over the previous one, is that it can work when the data points are not approximately Normal. This includes the cases where the distribution is not symmetric (a problematic situation for the previous method). A weakness of the IQR method occurs when data are concentrated near the median, resulting in an IQR that is in some sense too small to be useful.

Use of Box-Plots provides visual support for looking for outliers. The IQR is shown as a box with whiskers extending in both directions, extending $\delta \cdot$IQR units beyond the box, with indications of locations of extreme data points beyond the whiskers.

5.2.3 Based on Quantiles/Percentiles

A simple method for detecting outliers is to assume that the most extreme 1% of data points are outliers (they may well not be). This would include the 0.5% smallest and 0.5% largest data points. Under the Normality assumption this would correspond to 2.575 standard deviation units. Given that this method does not look for how far points are from a mean or median, it should not be used as the sole evidence that a data point is an outlier.

In the `Outlier.scala` file, ScalaTion currently provides the following techniques for outlier detection:

- **Standard Deviation Method**: data points too many standard deviation units (typically 2.5 to 3.5, defaults to 2.7) away from the mean, `DistanceOutlier`;

- **InterQuartile Range Method**: data points a scale factor/expansion multiplier (typically 1.5 to 2.0, defaults to 1.5) times the IQR beyond the middle two quartiles, `QuartileXOutlier`; and
- **Quantiles/Percentile Method**: data points in the extreme percentages (typically 0.7 to 10 percent, defaults to 0.7), i.e., having the smallest or largest values, `QuantileOutlier`.

Note: These **defaults** put these three outlier detection methods in alignment when data points are approximately Normally distributed.

The following function will turn outliers in missing values, by reassigning the outliers to `noDouble`, ScalaTion's indicator of a missing value of type `Double`.

DistanceOutlier.rmOutlier (traffic.column ("speed"))

An alternative to eliminating outliers during data preprocessing, is to eliminate them during modeling by looking for extreme residuals. In addition to looking at the magnitude of a residual ϵ_i, some argue only to remove data points that also have high influence on the model's parameters/coefficients, using techniques such as DFFITS, Cook's Distance, or DFBETAS [34].

5.3 Imputation Techniques

The two main ways to handle missing values are (1) throw them away, or (2) use imputation to replace them with reasonable guesses. When there is a gap in time series data, imputation may be used for short gaps, but is unlikely to be useful for long gaps. This is especially true when imputation techniques are simple. The alternative could be to use an advanced modeling technique like SARIMA for imputation, but then results of a modeling study using SARIMA are likely to be biased. Imputation implementations are based on the `Imputation` trait in the `scalation.modeling` package.

5.3.1 `Imputation` Trait

Trait Methods:

```
    trait Imputation

    def setMissVal (missVal_ : Double) { missVal = missVal_ }
    def setDist (dist_ : Int) { dist = dist_ }
    def imputeAt (x: VectorD, i: Int): Double
    def impute (x: VectorD, i: Int = 0): (Int, Double) = findMissing (x, i)
    def imputeAll (x: VectorD): VectorD =
    def impute (x: MatrixD): MatrixD =
    def imputeCol (c: Vec, i: Int = 0): (Int, Any) =
```

ScalaTion currently supports the following imputation techniques:

1. `object ImputeRegression extends Imputation`: Use `SimpleRegression` on the instance index to estimate the next missing value.

2. `object ImputeForward extends Imputation`: Use the previous value and slope to estimate the next missing value.

3. `object ImputeBackward extends Imputation`: Use the subsequent value and slope to estimate the previous missing value.

4. `object ImputeMean extends Imputation`: Use the filtered mean to estimate the next missing value.

5. `object ImputeMovingAvg extends Imputation`: Use the moving-average of the last 'dist' values to estimate the next missing value.

6. `object ImputeNormal extends Imputation`: Use the median of three Normally distributed, based on filtered mean and variance, random values to estimate the next missing value.

7. `object ImputeNormalWin extends Imputation`: Same as `ImputeNormal` except mean and variance are recomputed over a sliding window.

5.4 Align Multiple Time Series

When the data include multiple time series, there are likely to be time alignment problems. The frequency and/or phase may not be in agreement. For example, traffic count data may be recorded every 15 minutes and phased on the hour, while weather precipitation data may be collected every 30 minutes and phased to 10 minutes past the hour.

ScalaTion supports the following alignments techniques: (1) approximate left outer join and (2) dynamic time warping. The first operator will perform a left outer join between two relations based on their time (`TimeNum`) columns. Rather than the usual matching based on equality, approximately equal times are considered sufficient for alignment. For example, to align traffic data with the weather data, the following approximate left outer join may be used.

$$\text{traffic} \ltimes (0.01)(\text{“time”}, \text{“time”}, \text{weather}) \qquad\qquad \text{approximate left outer join}$$

The second operator ...

5.5 Creating Vectors and Matrices

Once the data have been preprocessed, columns may be projected out to create a matrix that may be passed to analytics/modeling techniques.

val mat = $\pi_{\text{“time”},\text{“count”}}$ (traffic).toMatrixD

This matrix may then be passed into multiple modeling techniques: (1) a Multiple Linear Regression, (2) a Auto-Regressive, Integrated, Moving-Average (ARIMA) model.

val model1 = Regression (mat)
val model2 = ARIMA (mat)

By default in ScalaTion the rightmost columns are the response/output variables. As many of the modeling techniques have a single response variable, it will be assumed to in the last column. There are also constructors and factory apply functions that take explicit vector and matrix parameters, e.g., a matrix of predictor variables and a response vector.

5.6 Exercises

1. Assume random variable x_j is distributed $N(\mu, \sigma)$.

   (a) Show that when the scale factor $\delta = 1.5$, the InterQuartile Range method corresponds to the Standard Deviation method at 2.69792 standard deviation units.

   (b) Show that when the scale factor $\delta = 2.0$, the InterQuartile Range method corresponds to the Standard Deviation method at 3.3724 standard deviation units.

   (c) What should the scale factor δ need to be to correspond to 3 standard deviation units?

2. Randomly generate 10,000 data points from the Standard Normal distribution. Count how many of these data points are considered as outliers for

   (a) the Standard Deviation method set at 3.3724 standard deviation units, and

   (b) the InterQuartile Range method with $\delta = 2.0$.

   (c) the Quantile/Percentile method set at *what?* percent.

3. Load the `auto_mpg.csv` dataset into an `auto_mpg` relation. Perform the preprocessing steps above to create a cleaned-up relation `auto_mpg2` and produce a data matrix called `auto_mat` from this relation. Print out the correlation matrix for `auto_mat`. Which columns have the highest correlation? To predict the miles per gallon `mpg` which columns are likely to be the best predictors.

4. Find a dataset at the UCI Machine Learning Repository and carry out the same steps
   https://archive.ics.uci.edu/ml/index.php.

Part II

Modeling

Chapter 6

Prediction

As the name predictive analytics indicates, the purpose of techniques that fall in this category is to develop models to predict outcomes. For example, the distance a golf ball travels y when hit by a driver depends on several factors or inputs $\mathbf{x}$ such as club head speed, barometric pressure, and smash factor (how square the impact is). The models can be developed using a combination of data (e.g., from experiments) and knowledge (e.g., Newton's Second Law). The modeling techniques discussed in this technical report tend to emphasize the use of data more than knowledge, while those in the simulation modeling technical report emphasize knowledge.

Abstractly, a predictive model can generally be formulated using a prediction function f as follows:

$$y \;=\; f(\mathbf{x},\ t;\ \mathbf{b}) + \epsilon \tag{6.1}$$

where

- y is an response/output scalar,
- $\mathbf{x}$ is an predictor/input vector,
- t is a scalar representing time,
- $\mathbf{b}$ is the vector of parameters of the function, and
- ϵ represents remaining residuals/errors.

Both the response y and residuals/errors ϵ are treated as random variables, while the predictor/feature variables $\mathbf{x}$ may be treated as either random or deterministic depending on context. Depending on the goals of the study as well as whether the data are the product of controlled/designed experiments, the random or deterministic view may be more suitable.

The parameters $\mathbf{b}$ can be adjusted so that the predictive model matches the available data. Note, in the definition of a function, the *arguments* appear before the ";", while the *parameters* appear after. The residuals/errors are typically additive as shown above, but may also be multiplicative. Of course, the formulation could be generalized by turning the output/response into a vector $\mathbf{y}$ and the parameters into a matrix B.

When a model is time-independent or time can be treated as just another dimension within the $\mathbf{x}$ vectors, prediction functions can be represented as follows:

$$y = f(\mathbf{x};\ \mathbf{b}) + \epsilon \tag{6.2}$$

Another way to look at such models, is that we are trying to estimate the conditional expectation of y given $\mathbf{x}$.

$$y = \mathbb{E}[y|\mathbf{x}] + \epsilon$$

$$\epsilon = y - f(\mathbf{x};\ \mathbf{b})$$

Given a dataset (m instances of data), each instance contributes to an overall residual/error vector $\boldsymbol{\epsilon}$. One of the simpler ways to estimate the parameters $\mathbf{b}$ is to minimize the size of the residual/error vector, e.g., its Euclidean norm. The square of this norm is the sum of squared errors (sse)

$$sse = \|\boldsymbol{\epsilon}\|^2 = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} \tag{6.3}$$

This corresponds to minimizing the raw mean square error ($mse = sse/m$). See the section on Generalized Linear Models for further development along these lines.

In ScalaTion, data are passed to the `train` function to train the model/fit the parameters $\mathbf{b}$. In the case of prediction, the `predict` function is used to predict values for the scalar response y.

A key question to address is the possible functional forms that f may take, such as the importance of time, the linearity of the function, the domains for y and $\mathbf{x}$, etc. We consider several cases in the subsections below.

6.1 Predictor

In ScalaTion, the `Predictor` trait provides a common framework for several predictor classes such as `SimpleRegression` or `Regression`. All of the modeling techniques discussed in this chapter extend the `Predictor` trait. They also extend the `Fit` trait to enable Quality of Fit (QoF) evaluation. (Unlike classes, traits support multiple inheritance).

Many modeling techniques utilize several predictor/input variables to predict a value for a response/output variable, e.g., given values for $[x_0, x_1, x_2]$ predict a value for y. The datasets fed into such modeling techniques will collect multiple instances of the predictor variables into a matrix `x` and multiple instances of the response variable into a vector `y`. The `Predictor` trait takes datasets of this form.

6.1.1 `Predictor` Trait

Trait Methods:

```
    @param x        the input/data m-by-n matrix
                        (augment with a first column of ones to include intercept in model)
    @param y        the response/output m-vector
    @param fname    the feature/variable names (if null, use x_j)
    @param hparam   the hyper-parameters for the model

    trait Predictor (x: MatrixD, y: VectorD, protected var fname: Array [String],
                     hparam: HyperParameter)
          extends Model:

    def getX: MatrixD = x
    def getY: VectorD = y
    def getFname: Array [String] = fname
    def numTerms: Int = getX.dim2
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit
    def train2 (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD)
    def trainNtest (x_ : MatrixD = x, y_ : VectorD = y)
                   (xx: MatrixD = x, yy: VectorD = y): (VectorD, VectorD) =
    def predict (z: VectorD): Double = b dot z
    def predict (x_ : MatrixD): VectorD =
    def hparameter: HyperParameter = hparam
    def parameter: VectorD = b
    def residual: VectorD = e

    def buildModel (x_cols: MatrixD): Predictor = null
    def selectFeatures (tech: SelectionTech, idx_q: Int = QoF.rSqBar.ordinal,
                         cross: Boolean = true): (LinkedHashSet [Int], MatrixD) =
    def forwardSel (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal): BestStep =
    def forwardSelAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
                      (LinkedHashSet [Int], MatrixD) =
    def importance (cols: Array [Int], rSq: MatrixD): Array [(Int, Double)] =
    def backwardElim (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal,
                      first: Int = 1): BestStep =
    def backwardElimAll (idx_q: Int = QoF.rSqBar.ordinal, first: Int = 1,
                          cross: Boolean = true): (LinkedHashSet [Int], MatrixD) =
    def stepRegressionAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
                          (LinkedHashSet [Int], MatrixD) =

    def vif (skip: Int = 1): VectorD =
    inline def testIndices (n_test: Int, rando: Boolean): IndexedSeq [Int] =
    def validate (rando: Boolean = true, ratio: Double = 0.2)
                 (idx : IndexedSeq [Int] =
                  testIndices ((ratio * y.dim).toInt, rando)): VectorD =
    def crossValidate (k: Int = 5, rando: Boolean = true): Array [Statistic] =
```

The `Predictor` trait extends the `Model` trait (see the end of the Probability chapter) and has the following methods:

1. The `getX` method returns the actual data/input matrix used by the model. Some complex models expand the columns in an initial data matrix to add for example quadratic or cross terms.
2. The `getY` method returns the actual response/output vector used by the model. Some complex models transform the initial response vector.
3. The `getFname` method returns the names of predictor variable/features, both given and extended.
4. The `numTerms` method returns the number of terms in the model.
5. The `train` method takes the dataset passed into the model (either the full dataset or a training-data) and optimizes the model parameters **b**.
6. The `train2` method takes the dataset passed into the model (either the full dataset or a training dataset) and optimizes the model parameters **b**. It also optimizes the hyper-parameters.
7. The `test` method evaluates the Quality of Fit (QoF) either on the full dataset or a designated test-data using the `diagnose` method.
8. The `trainNtest` method trains on the *training-set* and evaluates on the *test-set*.
9. The `predict` method take a data vector (e.g., a new data instance) and predicts its response. Another `predict` method takes a matrix as input (with each row being an instance) and makes predictions for each row.
10. The `hparameter` method returns the hyper-parameters for the model. Many simple models have none, but more sophisticated modeling techniques such as `RidgeRegression` and `LassoRegression` have them (e.g., a shrinkage hyper-parameter).
11. The `parameter` method returns the estimated parameters for the model.
12. The `residual` method returns the difference between the actual and predicted response vectors. The residual indicates what the model has left to explain/account for (e.g., an ideal model will only leave the noise in the data unaccounted for).
13. The `buildModel` method build a sub-model that is restricted to given columns of the data matrix. This method of called by the following feature selection methods.

14. The `selectFeatures` methods makes it easy to switch between forward, backward and stepwise feature selection.

15. The `forwardSel` method is used for forward selection of variables/features for inclusion into the model. At each step the variable that increases the predictive power of the model the most is selected. This method is called repeatedly in `forwardSelAll` to find "best" combination of features. Not guaranteed to find the optimal combination.

16. The `importance` method is used to indicate the relative importance of the features/variables.

17. The `bakwardElim` method is used for backward elimination of variables/features from the model. At each step the variable that contributes the least to the predictor power of the model is eliminated. This method is called repeatedly in `bakwardElimAll` to find "best" combination of features. Not guaranteed to find the optimal combination.

18. The `stepRegressionAll` method decides to add or remove a variable/feature based on whichever leads to the greater improvement. It continues until there is no further improvement. A swap operation may yield a better combination of features.

19. The `vif` method returns the Variance Inflation Factors (VIFs) for each of the columns in the data/input matrix. High VIF scores may indicate *multi-collinearity.*

20. The `testIndices` method returns the indices of the test-set.

21. The `validate` method divides a dataset into a training-set and a test-set, trains on one and tests on the other to determine `out-of-sample` Quality of Fit (QoF).

22. The `crossValidate` method implements k-fold cross-validation, where a dataset is divided into a training-set and a test-set. The training-set is used by the `train` method, while the test-set is used by the `test` method. The `crossValidate` method is similar to `validate`, but more extensive in that it repeats this process k times and makes sure all the data ends up in one of the k test-sets.

6.2 Quality of Fit for Prediction

The related `Fit` trait provides a common framework for computing Quality of Fit (QoF) measures. The dataset for many models comes in the form of an m-by-n data matrix X and an m response vector $\mathbf{y}$. After the parameters $\mathbf{b}$ (an n vector) have been fit/estimated, the error vector $\boldsymbol{\epsilon}$ may be calculated. The basic QoF measures involve taking either ℓ^1 (Manhattan) or ℓ^2 (Euclidean) norms of the error vector as indicated in Table 6.1.

Table 6.1: Quality of Fit

error/residual	**absolute**	**ℓ^1 norm**	**squared**	**ℓ^2 norm**
sum	sum of absolute errors	$sae = \|\boldsymbol{\epsilon}\|_1$	sum of squared errors	$sse = \|\boldsymbol{\epsilon}\|_2^2$
mean	mean absolute error	$mae^0 = sae/m$	mean squared error	$mse^0 = sse/m$
unbiased mean	mean absolute error	$mae = sae/df$	mean squared error	$mse = sse/df$

Typically, if a model has m instances/rows in the dataset and n parameters to fit, the error vector will live in an $m - n$ dimensional space (ignoring issues related to the rank the data matrix). Note, if $n = m$, there may be a unique solution for the parameter vector $\mathbf{b}$, in which case $\boldsymbol{\epsilon} = \mathbf{0}$, i.e., the error vector lives in a 0-dimensional space. The *Degrees of Freedom* (for error) is the dimensionality of the space that the error vector lives in, namely, $df = m - n$.

6.2.1 `Fit` Trait

Trait Methods:

```
@param dfm  the degrees of freedom for model/regression
@param df   the degrees of freedom for error

trait Fit (private var dfm: Double, private var df: Double)
      extends FitM:

def resetDF (df_update: (Double, Double)): Unit =
def mse_ : Double = mse
override def diagnose (y: VectorD, yp: VectorD, w: VectorD = null): VectorD =
def ll (ms: Double = mse0, s2: Double = sig2e, m2: Int = m): Double =
def fit: VectorD = VectorD (rSq, rSqBar, sst, sse, mse0, rmse, mae,
                            dfm, df, fStat, aic, bic, mape, smape, mase)
def help: String = Fit.help
def summary (x_ : MatrixD, fname: Array [String], b: VectorD, vifs: VectorD = null):
    String =
```

For modeling, a user chooses one the of classes (directly or indirectly) extending the trait `Predictor` (e.g., `Regression`) to instantiate an object. Next the `train` method would be typically called, followed by the `test` method, which computes the residual/error vector and calls the `diagnose` method. Then the fitMap method would be called to return quality of fit statistics computed by the diagnose method. The quality of fit measures computed by the diagnose method in the Fit class are shown below.

```
    @param y   the actual response/output vector to use (test/full)
    @param yp  the predicted response/output vector (test/full)
    @param w   the weights on the instances (defaults to null)

    override def diagnose (y: VectorD, yp: VectorD, w: VectorD = null): VectorD =
        val e = super.diagnose (y, yp, w)

        if dfm <= 0 || df <= 0 then
            flaw ("diagnose", s"degrees of freedom dfm = dfmanddf =df must be > 0")
        if dfm == 0 then dfm = 1                                // must have at least 1 DoF
                                                                // b_0 or b_0 + b_1x_1 or b_1x_1
        msr    = ssr / dfm                                      // mean of squares for model
        mse    = sse / df                                       // mean of squares for error

        rse    = sqrt (mse)                                     // residual standard error
        rSqBar = 1 - (1-rSq) * r_df                             // adjusted R-squared
        fStat  = msr / mse                                      // F statistic (quality of fit)
        p_fS   = 1.0 - fisherCDF (fStat, dfm.toInt, df.toInt)   // p-value for fStat
        if p_fS.isNaN then p_fS = 0.0                           // NaN => error by fisherCDF
        if sig2e == -1.0 then sig2e = e.variance_

        val ln_m = log (m)                                      // natural log of m (ln(m))
        aic    = ll() + 2 * (dfm + 1)                           // Akaike Information Criterion
                                                                // +1 on dfm accounts for sig2e
        bic    = aic + (dfm + 1) * (ln_m - 2)                   // Bayesian Info. Criterion
        mape   = 100 * (e.abs / y.abs).sum / m                  // mean abs. percentage error
        smape  = 200 * (e.abs / (y.abs + yp.abs)).sum / m       // symmetric MAPE
        mase   = Fit.mase (y, yp)                               // mean absolute scaled error
        fit
    end diagnose
```

One may look at the sum of squared errors (sse) as an indicator of model quality.

$$sse \;=\; \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} \tag{6.4}$$

In particular, sse can be compared to the sum of squares total (sst), which measures the total variability of the response y,

$$sst \;=\; \|\mathbf{y} - \mu_{\mathbf{y}}\|^2 \;=\; \mathbf{y} \cdot \mathbf{y} - m\,\mu_{\mathbf{y}}^2 \;=\; \mathbf{y} \cdot \mathbf{y} - \frac{1}{m}\Big[\sum y_i\Big]^2 \tag{6.5}$$

while the sum of squares regression ($ssr = sst - sse$) measures the variability captured by the model, so the *coefficient of determination* measures the fraction of the variability captured by the model.

$$\boxed{R^2 \;=\; \frac{ssr}{sst} \;=\; 1 - \frac{sse}{sst} \;\leq\; 1} \tag{6.6}$$

Values for R^2 would be non-negative, unless the proposed model is so bad (worse than the Null Model that simply predicts the mean) that the proposed model actually adds variability.

6.3 Null Model

The `NullModel` class implements the simplest type of predictive modeling technique. If all else fails it may be reasonable to simply guess that y will take on its expected value or mean.

$$y \;=\; \mathbb{E}\,[y] + \epsilon \tag{6.7}$$

This could happen if the predictors $\mathbf{x}$ are not relevant, not collected in a useful range or the relationship is too complex for the modeling techniques you have applied.

6.3.1 Model Equation

Ignoring the predictor variables $\mathbf{x}$ gives the following simple model equation.

$$\boxed{y \;=\; b_0 + \epsilon} \tag{6.8}$$

This intercept-only model is just a constant term plus the error/residual term.

6.3.2 Training

The training dataset in this case only consists of a response vector $\mathbf{y}$. The error vector in this case is

$$\boldsymbol{\epsilon} \;=\; \mathbf{y} - \hat{\mathbf{y}} \;=\; \mathbf{y} - b_0\mathbf{1} \tag{6.9}$$

For Least Squares Estimation (LSE), the loss function $\mathcal{L}(\mathbf{b})$ can be set to half the sum of squared errors.

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} sse \;=\; \frac{1}{2}\|\boldsymbol{\epsilon}\|^2 \;=\; \frac{1}{2}\boldsymbol{\epsilon}\cdot\boldsymbol{\epsilon} \tag{6.10}$$

Substituting for $\boldsymbol{\epsilon}$ gives

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\,\boxed{\mathbf{y} - b_0\mathbf{1}}\cdot\boxed{\mathbf{y} - b_0\mathbf{1}} \tag{6.11}$$

6.3.3 Optimization - Derivative

A function can be optimized using Calculus by taking the first derivative and setting it equal to zero. If the second derivative is positive (negative) it will be minimal (maximal).

In particular, the derivative product rule (for dot products) may be used.

$$\begin{aligned}(\mathbf{f}\cdot\mathbf{g})' \;&=\; \mathbf{f}'\cdot\mathbf{g} + \mathbf{f}\cdot\mathbf{g}' \\ (\mathbf{f}\cdot\mathbf{f})' \;&=\; 2\,\mathbf{f}'\cdot\mathbf{f}\end{aligned}$$

Dividing by $\frac{1}{2}$ gives,

$$\frac{1}{2}\,(\mathbf{f}\cdot\mathbf{f})' \;=\; \boxed{\mathbf{f}'}\cdot\mathbf{f} \tag{6.12}$$

Taking the derivative w.r.t. b_0, $\dfrac{d\mathcal{L}}{db_0}$, using the derivative product rule and setting it equal to zero yields the following equation.

$$\frac{d\mathcal{L}}{db_0} = \boxed{-\mathbf{1}} \cdot (\mathbf{y} - b_0\mathbf{1}) = 0$$

Therefore, the optimal value for the parameter b_0 is

$$\boxed{b_0 = \frac{\mathbf{1} \cdot \mathbf{y}}{\mathbf{1} \cdot \mathbf{1}} = \frac{\mathbf{1} \cdot \mathbf{y}}{m} = \mu_{\mathbf{y}}} \tag{6.13}$$

This shows that the optimal value for the parameter is the mean of the response vector.

In ScalaTion this requires just one line of code inside the `train` method.

```
def train (x_null: MatrixD = null, y_ : VectorD = y): Unit =
    b = VectorD (y_.mean)                                 // parameter vector [b0]
end train
```

After values for the model parameters are determined, it it important to assess the Quality of Fit (QoF). The `test` method will compute the residual/error vector $\boldsymbol{\epsilon}$ and then call the `diagnose` method.

```
def test (x_null: MatrixD = null, y_ : VectorD = y): (VectorD, VectorD) =
    val yp = VectorD.fill (y_.dim)(b(0))                  // y predicted for (test/full)
    (yp, diagnose (y_, yp))                               // return predictions and QoF
end test
```

The coefficient of determination R^2 for the null regression model is always 0, i.e., none of variance in the random variable y is explained by the model. A more sophisticated model should only be used if it is better than the null model, that is when its R^2 is strictly greater than zero. Also, a model can have a negative R^2 if its predictions are worse than guessing the mean.

Finally, the `predict` method is simply.

```
def predict (z: VectorD): Double = b(0)
```

6.3.4 Example Calculation

For the training data shown below, the optimal value for the intercept parameter $b_0 = \mu_{\mathbf{y}} = \frac{11}{4} = 2.75$. The table below shows the values of $\mathbf{x}$, $\mathbf{y}$, $\hat{\mathbf{y}}$, $\boldsymbol{\epsilon}$, and $\boldsymbol{\epsilon}^2$. for the Null Model,

$$y = 2.75 + \epsilon \tag{6.14}$$

Table 6.2: Null Model: Example Training Data

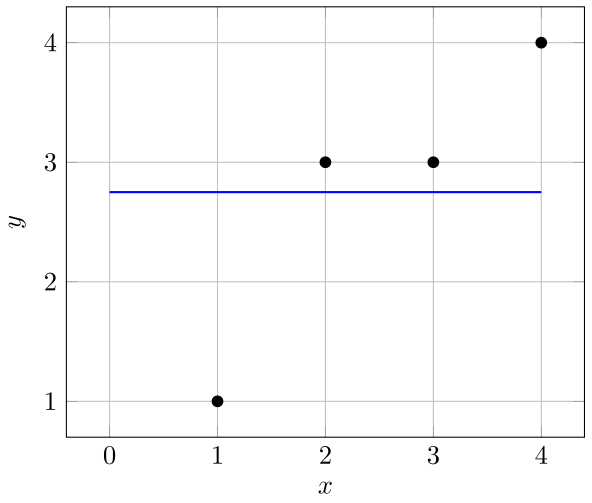
$\mathbf{x}$	$\mathbf{y}$	$\hat{\mathbf{y}}$	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}^2$
1	1	$\frac{11}{4}$	$-\frac{7}{4}$	$\frac{49}{16}$
2	3	$\frac{11}{4}$	$\frac{1}{4}$	$\frac{1}{16}$
3	3	$\frac{11}{4}$	$\frac{1}{4}$	$\frac{1}{16}$
4	4	$\frac{11}{4}$	$\frac{5}{4}$	$\frac{25}{16}$
10	11	11	0	$\frac{19}{4} = 4.75$

The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so

$$R^2 \;=\; 1 - \frac{sse}{sst} \;=\; 1 - \frac{4.75}{4.75} \;=\; 0$$

The plot below illustrates how the Null Model attempts to fit the four given data points.

Null Model Line vs. Data Points



6.3.5 `NullModel` Class

Class Methods:

```
@param y  the response/output vector

class NullModel (y: VectorD)
      extends Predictor (MatrixD.one (y.dim), y, Array ("one"), null)
         with Fit (dfm = 1, df = y.dim)
         with NoSubModels:

def train (x_null: MatrixD = null, y_ : VectorD = y): Unit =
def test (x_null: MatrixD = null, y_ : VectorD = y): (VectorD, VectorD) =
override def predict (z: VectorD): Double = b(0)
override def predict (x_ : MatrixD): VectorD = VectorD.fill (x_.dim)(b(0))
override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                      b_ : VectorD = b, vifs: VectorD = vif ()): String =
```

6.3.6 Exercises

1. Determine the value of the second derivative of the loss function

$$\frac{d^2\mathcal{L}}{d{b_0}^2} \;=\; ?$$

at the critical point $b_0 = \mu_{\mathbf{y}}$. What kind of critical point is this?

2. Let the response vector $\mathbf{y}$ be

```scala
    val y = VectorD (1, 3, 3, 4)
```

and execute the `NullModel`.

For context, assume the corresponding predictor vector $\mathbf{y}$ is

```scala
    val x = VectorD (1, 2, 3, 4)
```

Draw an xy plot of the data points. Give the value for the parameter vector **b**. Show the error distance for each point in the plot. Compare the sum of squared errors sse with the sum of squares total sst. What is the value for the coefficient of determination R^2?

3. Using ScalaTion, analyze the `NullModel` for the following response vector y.

```scala
    val y = VectorD (2.0, 3.0, 5.0, 4.0, 6.0)              // response vector y
    println (s"y = $y")

    val mod = new NullModel (y)                            // create a null model
    mod.trainNtest ()()                                    // train and test the model

    val z  = VectorD (5.0)                                 // predict y for one point
    val yp = mod.predict (z)                               // yp (y-predicted or y-hat)
    println (s"predict (z) = $yp")
```

4. Execute the `NullModel` on the `Auto_MPG` dataset. See `scalation.modeling.Example_AutoMPG`. What is the quality of the fit (e.g., R^2 or `rSq`)? Is this value expected? Is is possible for a model to perform worse than this?

6.4 Simpler Regression

The `SimplerRegression` class supports simpler linear regression. In this case, the predictor vector $\mathbf{x}$ consists of a single variable x_0, i.e., $\mathbf{x} = [x_0]$ and there is only a single parameter that is the coefficient for x_0 in the model.

6.4.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b} = [b_0]$ in the following model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 x_0 + \epsilon} \tag{6.15}$$

where ϵ represents the residuals/errors (the part not explained by the model).

6.4.2 Training

A dataset may be collected for providing an estimate for parameter b_0. Given m data points, stored in an m-dimensional vector $\mathbf{x}_0$ and m response values, stored in an m-dimensional vector $\mathbf{y}$, we may obtain the following vector equation.

$$\mathbf{y} \;=\; b_0 \mathbf{x}_0 + \boldsymbol{\epsilon} \tag{6.16}$$

One way to find a value for parameter b_0 is to minimize the norm of residual/error vector $\boldsymbol{\epsilon}$.

$$\min_{b_0} \|\boldsymbol{\epsilon}\| \tag{6.17}$$

Since $\boldsymbol{\epsilon} \;=\; \mathbf{y} - b_0 \mathbf{x}_0$, we may solve the following optimization problem:

$$\min_{b_0} \|\mathbf{y} \;-\; b_0 \mathbf{x}_0\| \tag{6.18}$$

This is equivalent to minimizing half the dot product ($\frac{1}{2}\|\boldsymbol{\epsilon}\|^2 = \frac{1}{2}\boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} = \frac{1}{2} sse$). Thus the loss function is

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} \boxed{\mathbf{y} - b_0 \mathbf{x}_0} \cdot \boxed{\mathbf{y} - b_0 \mathbf{x}_0} \tag{6.19}$$

6.4.3 Optimization - Derivative

Again, a function can be optimized using Calculus by taking the first derivative and setting it equal to zero. If the second derivative is positive (negative) it will be minimal (maximal). Taking the derivative w.r.t. b_0, $\frac{d\mathcal{L}}{db_0}$, using the derivative product rule (for dot products) gives

$$\frac{1}{2}\,(\mathbf{f} \cdot \mathbf{f})' \;=\; \boxed{\mathbf{f}'} \cdot \mathbf{f}$$

and setting it equal to zero yields the following equation.

$$\frac{d\mathcal{L}}{db_0} \;=\; \boxed{-\mathbf{x}_0} \cdot (\mathbf{y} - b_0 \mathbf{x}_0) = 0 \tag{6.20}$$

Therefore, the optimal value for the parameter b_0 is

$$\boxed{b_0 \;=\; \frac{\mathbf{x}_0 \cdot \mathbf{y}}{\mathbf{x}_0 \cdot \mathbf{x}_0}} \tag{6.21}$$

6.4.4 Example Calculation

Consider the following data points $\{(1,1),(2,3),(3,3),(3,4)\}$ and solve for the parameter (slope) b_0.

$$b_0 \;=\; \frac{[1,2,3,4]\cdot[1,3,3,4]}{[1,2,3,4]\cdot[1,2,3,4]} \;=\; \frac{32}{30} \;=\; \frac{16}{15}$$

Using this optimal value for the parameter $b_0 = \dfrac{16}{15}$, we may obtain predicted values for each of the x-values.

$$\hat{\mathbf{y}} \;=\; \hat{\mathbf{y}} \;=\; predict(\mathbf{x}_0) \;=\; b_0\mathbf{x}_0 \;=\; [1.067, 2.133, 3.200, 4.267]$$

Therefore, the error/residual vector is

$$\boldsymbol{\epsilon} \;=\; \mathbf{y} - \hat{\mathbf{y}} \;=\; [1,3,3,4] \;-\; [1.067, 2.133, 3.200, 4.267] \;=\; [-0.067, 0.867, -0.2, -0.267]$$

The table below shows the values of $\mathbf{x}$, $\mathbf{y}$, $\hat{\mathbf{y}}$, $\boldsymbol{\epsilon}$, and $\boldsymbol{\epsilon}^2$. for the Simpler Regression Model,

$$y \;=\; \left[\frac{16}{15}\right]\cdot[x] + \epsilon \;=\; \frac{16}{15}x + \epsilon$$

Table 6.3: Simpler Regression Model: Example Training Data

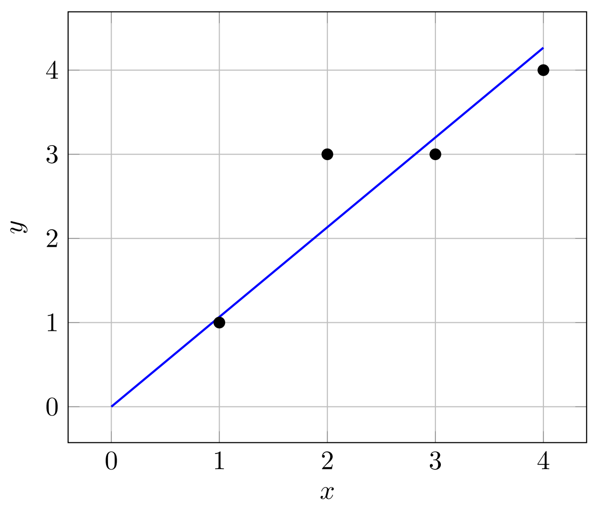
$\mathbf{x}$	$\mathbf{y}$	$\hat{\mathbf{y}}$	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}^2$
1	1	$\frac{16}{15}$	$-\frac{1}{15}$	$\frac{1}{225}$
2	3	$\frac{32}{15}$	$\frac{13}{15}$	$\frac{169}{225}$
3	3	$\frac{48}{15}$	$-\frac{3}{15}$	$\frac{9}{225}$
4	4	$\frac{64}{15}$	$-\frac{4}{15}$	$\frac{16}{225}$
10	11	$\frac{160}{15}$	$\frac{5}{15}$	$\frac{13}{15} = 0.867$

The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so

$$R^2 \;=\; 1 - \frac{sse}{sst} \;=\; 1 - \frac{0.867}{4.75} \;=\; 0.813$$

The plot below illustrates how the Simpler Regression Model attempts to fit the four given data points.

Simpler Regression Model Line vs. Data Points



Note, that this model has no intercept. This makes the solution for the parameter very easy, but may make the model less accurate. This is remedied in the next section. Since no intercept really means the intercept is zero, the regression line will go through the origin. This is referred to as Regression Through the Origin (RTO) and should only be applied when the data scientist has reason to believe it makes sense.

6.4.5 `SimplerRegression` Class

Class Methods:

```
    @param x        the data/input matrix (only use the first column)
    @param y        the response/output vector
    @param fname_   the feature/variable names (only use the first name)

    class SimplerRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null)
          extends Predictor (x, y, if fname_ == null then null else fname_.slice (0, 1),
                                   null)
             with Fit (dfm = 1, df = x.dim - 1)
             with NoSubModels:

    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                          b_ : VectorD = b, vifs: VectorD = vif ()): String =
```

6.4.6 Exercises

1. For $\mathbf{x}_0 = [1, 2, 3, 4]$ and $\mathbf{y} = [1, 3, 3, 4]$, try various values for the parameter b_0. Plot the sum of squared errors (sse) vs. b_0. Note, the code must be completed before it is complied and run.

```scala
    import scalation.mathstat._

    @main def simplerRegression_exer_1 (): Unit =

        val x0  = VectorD (1, 2, 3, 4)
        val y   = VectorD (1, 3, 3, 4)
        val b0  = VectorD.range (0, 50) / 25.0
        val sse = new VectorD (b0.dim)
        for i <- b0.indices do
            val e  = ?
            sse(i) = e dot e
        end for
        new Plot (b0, sse, lines = true)

    end simplerRegression_exer_1
```

Where do you think the minimum occurs?

Note, to run your code you may use `my_scalation` outside of ScalaTion. Make sure its `lib` directory has the ScalaTion's jar file. Create a file called `SimplerRegression_exer_1.scala` in your `src/main/scala` directory. In your project's base directory, type `sbt`. Within `sbt` type `compile` and then `run`.

2. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \leq i < m\}$ and draw the line that is nearest to these points. What is the slope of this line. Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimplerRegression` class to obtain the $\mathbf{b} = [b_0]$ vector.

```scala
    // 4 data points:       x0
    val x = MatrixD ((4, 1), 1,                            // x 4-by-1 matrix
                             2,
                             3,
                             4)
    val y = VectorD (1, 3, 3, 4)                           // y vector

    val mod = new SimplerRegression (x, y)                 // create a simpler regression
    mod.trainNtest ()()                                    // train and test the model

    val yp = mod.predict (x)
    new Plot (x(?, 0), y, yp, lines = true)                // black for y and red for yp
```

An alternative to using the above constructor `new SimplerRegression` is to use a factory method `SimplerRegression`. Substitute in the following lines of code to do this.

```scala
    val x = VectorD (1, 2, 3, 4)
    val rg = SimplerRegression (x, y, null)
    new Plot (x, y, yp, lines = true)
```

3. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \leq i < m\}$ and draw the line that is nearest to these points and intersects the origin $[0, 0]$. What is the slope of this line? Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimplerRegression` class to obtain the $\mathbf{b} = [b_0]$ vector.

```scala
    // 5 data points:       x0
    val x = MatrixD ((5, 1), 0,                            // x 5-by-1 matrix
                             1,
                              2,
                              3,
                              4)
    val y = VectorD (2, 3, 5, 4, 6)                     // y vector

    val mod = new SimplerRegression (x, y)              // create a simpler regression
    model
    mod.trainNtest ()()                                 // train and test the model

    val z  = VectorD (5)                                // predict y for one point
    val yp = rg.predict (z)                             // y-predicted
    println (s"predict (z)=yp")
```

4. Execute the `SimplerRegression` on the `Auto_MPG` dataset. See `scalation.modeling.Example_AutoMPG`. What is the quality of the fit (e.g., R^2 or `rSq`)? Is this value expected? What does it say about this model? Try using different columns for the predictor variable.

5. Compute the second derivative of the loss function w.r.t. b_0, $\dfrac{d^2\mathcal{L}}{d{b_0}^2}$. Under what conditions will it be positive?

6.5 Simple Regression

The `SimpleRegression` class supports simple linear regression. It combines the benefits of the last two modeling techniques: the intercept model `NullModel` and the slope model `SimplerRegression`. It is guaranteed to be at least as good as the better of these two modeling techniques. In this case, the predictor vector $\mathbf{x} \in \mathbb{R}^2$ consists of the constant one and a single variable x_1, i.e., $[1, x_1]$, so there are now two parameters $\mathbf{b} = [b_0, b_1] \in \mathbb{R}^2$ in the model.

6.5.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; [b_0, b_1] \cdot [1, x_1] + \epsilon \;=\; b_0 + b_1 x_1 + \epsilon} \tag{6.22}$$

where ϵ represents the residuals (the part not explained by the model).

6.5.2 Training

The model is trained on a dataset consisting of m data points/vectors, stored row-wise in an m-by-2 matrix $X \in \mathbb{R}^{m\times 2}$ and m response values, stored in an m dimensional vector $\mathbf{y} \in \mathbb{R}^m$.

$$\mathbf{y} \;=\; X\mathbf{b} + \boldsymbol{\epsilon} \tag{6.23}$$

The parameter vector $\mathbf{b}$ may be determined by solving the following optimization problem:

$$\min_{\mathbf{b}} \|\boldsymbol{\epsilon}\| \tag{6.24}$$

Substituting $\boldsymbol{\epsilon} \;=\; \mathbf{y} - \hat{\mathbf{y}} \;=\; \mathbf{y} \,-\, X\mathbf{b}$ yields

$$\min_{\mathbf{b}} \|\mathbf{y} \,-\, X\mathbf{b}\|$$

Using the fact that the matrix X consists of two column vectors $\mathbf{1}$ and $\mathbf{x}_1$, it can be rewritten,

$$\min_{[b_0,b_1]} \|\mathbf{y} \,-\, [\mathbf{1}\,\mathbf{x}_1] \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \|$$

$$\min_{[b_0,b_1]} \|\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x}_1)\| \tag{6.25}$$

This is equivalent to minimizing the dot product ($\|\boldsymbol{\epsilon}\|^2 = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} = sse$)

$$(\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x}_1)) \cdot (\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x}_1)) \tag{6.26}$$

Since $\mathbf{x}_0$ is just $\mathbf{1}$, for simplicity we drop the subscript on $\mathbf{x}_1$. Thus the loss function $\frac{1}{2}sse$ is

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} \boxed{\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})} \cdot \boxed{\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})} \tag{6.27}$$

6.5.3 Optimization - Gradient

A function of several variables can be optimized using Vector Calculus by setting its gradient (see the Linear Algebra Chapter) equal to zero and solving the resulting system of equations. When the system of equations are linear, matrix factorization may be used, otherwise techniques from Nonlinear Optimization may be needed.

Taking the gradient of the loss function $\mathcal{L}$ gives

$$\nabla \mathcal{L} \;=\; \left[\frac{\partial \mathcal{L}}{\partial b_0}, \frac{\partial \mathcal{L}}{\partial b_1}\right] \tag{6.28}$$

The goal is to find the value of the parameter vector $\mathbf{b}$ that yields a zero gradient (flat response surface). Setting the gradient equal to zero ($\mathbf{0} = [0, 0]$) yields two equations.

$$\nabla \mathcal{L}(\mathbf{b}) \;=\; \left[\frac{\partial \mathcal{L}}{\partial b_0}(\mathbf{b}), \frac{\partial \mathcal{L}}{\partial b_1}(\mathbf{b})\right] \;=\; \mathbf{0} \tag{6.29}$$

The gradient (the two partial derivatives) may be determined using the derivative product rule for dot products.

$$\frac{1}{2}\,(\mathbf{f}\cdot\mathbf{f})' \;=\; \boxed{\mathbf{f}'}\cdot\mathbf{f}$$

Partial Derivative w.r.t. b_0

The first equation results from setting $\dfrac{\partial}{\partial b_0}$ of $\mathcal{L}$ to zero.

$$\begin{aligned}\boxed{-\mathbf{1}}\cdot(\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})) &= 0\\ \mathbf{1}\cdot\mathbf{y} \,-\, \mathbf{1}\cdot(b_0\mathbf{1} + b_1\mathbf{x}) &= 0\\ b_0\mathbf{1}\cdot\mathbf{1} &= \mathbf{1}\cdot\mathbf{y} \,-\, b_1\mathbf{1}\cdot\mathbf{x}\end{aligned}$$

Since $\mathbf{1}\cdot\mathbf{1} = m$, b_0 may be expressed as

$$b_0 \;=\; \frac{\mathbf{1}\cdot\mathbf{y} \,-\, b_1\mathbf{1}\cdot\mathbf{x}}{m} \tag{6.30}$$

Partial Derivative w.r.t. b_1

Similarly, the second equation results from setting $\dfrac{\partial}{\partial b_1}$ of $\mathcal{L}$ to zero.

$$\begin{aligned}\boxed{-\mathbf{x}}\cdot(\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})) &= 0\\ \mathbf{x}\cdot\mathbf{y} \,-\, \mathbf{x}\cdot(b_0\mathbf{1} + b_1\mathbf{x}) &= 0\\ b_0\mathbf{1}\cdot\mathbf{x} + b_1\mathbf{x}\cdot\mathbf{x} \;&=\; \mathbf{x}\cdot\mathbf{y}\end{aligned}$$

Multiplying by both sides by m produces

$$m\,b_0\mathbf{1}\cdot\mathbf{x} + m\,b_1\mathbf{x}\cdot\mathbf{x} \;=\; m\,\mathbf{x}\cdot\mathbf{y} \tag{6.31}$$

Substituting for $m\,b_0 = \mathbf{1}\cdot\mathbf{y} \,-\, b_1\mathbf{1}\cdot\mathbf{x}$ yields

$$\begin{aligned}[\mathbf{1}\cdot\mathbf{y} \,-\, b_1\mathbf{1}\cdot\mathbf{x}]\mathbf{1}\cdot\mathbf{x} \,+\, m\,b_1\mathbf{x}\cdot\mathbf{x} \;&=\; m\,\mathbf{x}\cdot\mathbf{y}\\ b_1[m\,\mathbf{x}\cdot\mathbf{x} - (\mathbf{1}\cdot\mathbf{x})^2] \;&=\; m\,\mathbf{x}\cdot\mathbf{y} - (\mathbf{1}\cdot\mathbf{x})(\mathbf{1}\cdot\mathbf{y})\end{aligned}$$

Solving for b_1 gives

$$b_1 \;=\; \frac{m\,\mathbf{x}\cdot\mathbf{y} - (\mathbf{1}\cdot\mathbf{x})(\mathbf{1}\cdot\mathbf{y})}{m\,\mathbf{x}\cdot\mathbf{x} - (\mathbf{1}\cdot\mathbf{x})^2} \tag{6.32}$$

The b_0 parameter gives the *intercept*, while the b_1 parameter gives the *slope* of the line that best fits the data points.

6.5.4 Example Calculation

Consider again the problem from the last section where the data points are $\{(1,1),(2,3),(3,3),(3,4)\}$ and solve for the two parameters, (intercept) b_0 and (slope) b_1.

$$b_1 \;=\; \frac{4[1,2,3,4]\cdot[1,3,3,4] - (\mathbf{1}\cdot[1,2,3,4])(\mathbf{1}\cdot[1,3,3,4])}{4[1,2,3,4]\cdot[1,2,3,4] - (\mathbf{1}\cdot[1,2,3,4])^2} \;=\; \frac{128-110}{120-100} \;=\; \frac{18}{20} \;=\; 0.9$$

$$b_0 \;=\; \frac{\mathbf{1}\cdot[1,3,3,4] - 0.9(\mathbf{1}\cdot[1,2,3,4])}{4} \;=\; \frac{11-0.9*10}{4} \;=\; 0.5$$

Table **??** below shows the values of $\mathbf{x}$, $\mathbf{y}$, $\hat{\mathbf{y}}$, $\boldsymbol{\epsilon}$, and $\boldsymbol{\epsilon}^2$ for the Simple Regression Model,

$$y \;=\; [0.5, 0.9]\cdot[1,x] + \epsilon \;=\; 0.5 + 0.9x + \epsilon$$

Table 6.4: Simple Regression Model: Example Training Data

$\mathbf{x}$	$\mathbf{y}$	$\hat{\mathbf{y}}$	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}^2$
1	1	1.4	-0.4	0.16
2	3	2.3	0.7	0.49
3	3	3.2	-0.2	0.04
4	4	4.1	-0.1	0.01
10	11	11	0	0.7

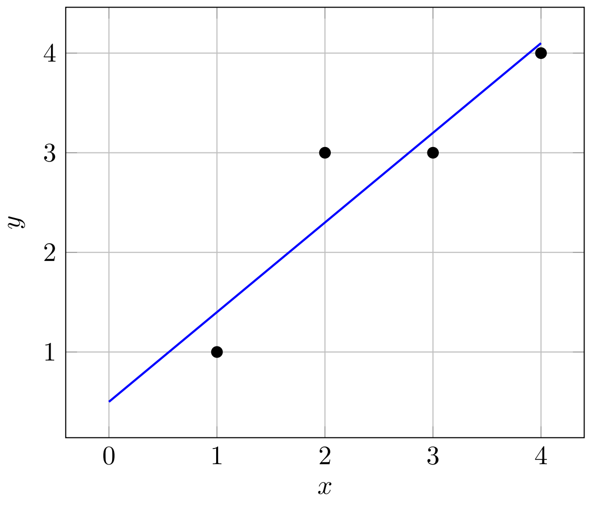
For which models (`NullModel`, `SimplerRegression` and `SimpleRegression`), did the redidual/error vector $\boldsymbol{\epsilon}$ sum to zero?

The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so the Coefficient of Determination,

$$R^2 \;=\; 1 - \frac{sse}{sst} \;=\; 1 - \frac{0.7}{4.75} \;=\; 0.853$$

The plot below illustrates how the Simple Regression Model (`SimpleRegression`) attempts to fit the four given data points.

Simple Regression Model Line vs. Data Points



Concise Formulas for the Parameters

More concise and intuitive formulas for the parameters b_0 and b_1 may be derived.

- Using the definition for mean from Chapter 3 for $\mu_{\mathbf{x}}$ and $\mu_{\mathbf{y}}$, it can be shown that the expression for b_0 shortens to

$$\boxed{b_0 \;=\; \mu_{\mathbf{y}} \,-\, b_1\mu_{\mathbf{x}}} \tag{6.33}$$

  Draw a line through the following two points $[0, b_0]$ (the intercept) and $[\mu_{\mathbf{x}}, \mu_{\mathbf{y}}]$ (the center of mass). How does this line compare to the regression line.

- Now, using the definitions for covariance $\sigma_{\mathbf{x},\mathbf{y}}$ and variance $\sigma^2_{\mathbf{x}}$ from Chapter 3, it can be shown that the expression for b_1 shortens to

$$\boxed{b_1 \;=\; \frac{\sigma_{\mathbf{x},\mathbf{y}}}{\sigma^2_{\mathbf{x}}}} \tag{6.34}$$

  If the slope of the regression line is simply the ratio of the covariance to the variance, what would the slope be if $y = x$. It may also be written as follows:

$$b_1 \;=\; \frac{S_{xy}}{S_{xx}} \tag{6.35}$$

  where $S_{xy} = \sum_i (x_i - \mu_{\mathbf{x}})(y_i - \mu_{\mathbf{y}})$ and $S_{xx} = \sum_i (x_i - \mu_{\mathbf{x}})^2$.

Table 6.5 extends the previous table to facilitate computing the parameters vector **b** using the concise formulas.

Table 6.5: Simple Regression Model: Expanded Table with Centering $\mu_x = 2.5$, $\mu_y = 2.75$

$\mathbf{x}$	$\mathbf{x} - \mu_x$	$\mathbf{y}$	$\mathbf{y} - \mu_x$	$\hat{\mathbf{y}}$	ϵ	ϵ^2
1	-1.5	1	-1.75	1.4	-0.4	0.16
2	-0.5	3	0.25	2.3	0.7	0.49
3	0.5	3	0.25	3.2	-0.2	0.04
4	1.5	4	1.25	4.1	-0.1	0.01
10	0	11	0	11	0	0.7

$$\begin{aligned}
S_{xx} &= \sum_i (x_i - \mu_\mathbf{x})^2 &= 1.5^2 + 0.5^2 + 0.5^2 + 1.5^2 &= 5 \\
S_{yy} &= \sum_i (y_i - \mu_\mathbf{y})^2 &= 1.75^2 + 0.25^2 + 0.25^2 + 1.25^2 &= 4.75 \\
S_{xy} &= \sum_i (x_i - \mu_\mathbf{x})(y_i - \mu_\mathbf{y}) &= (-1.5 \cdot -1.75) + (-0.5 \cdot 0.25) + (0.5 \cdot 0.25) + (1.5 \cdot 1.25) &= 4.5
\end{aligned}$$

Therefore,

$$b_1 = \frac{S_{xy}}{S_{xx}} = \frac{4.5}{5} = 0.9$$

$$b_0 = \mu_\mathbf{y} - b_1 \mu_\mathbf{x} = 2.75 - 0.9 \cdot 2.5 = 2.75 - 2.25 = 0.5 \qquad (6.36)$$

Furthermore, it facilitates computing $sst = S_{yy} = 4.75$.

6.5.5 Exploratory Data Analysis

As discussed in Chapter 1, *Exploratory Data Analysis* (EDA) should be performed after preprocessing the dataset. Once the response variable y is selected, a null model should be created to see in a plot where the data points lie compared to the mean. The code below shows how to do this for the AutoMPG dataset.

```
import Example_AutoMPG.{xy, x, y, x_fname}

banner ("Null Model")
val nm = new NullModel (y)
nm.trainNtest ()()                                    // train and test the model
val yp = nm.predict (x)
new Plot (null, y, yp, "EDA: y and yp (red) vs. t", lines = true)
```

Next the relationships between the predictor variable x_j (the columns in input/data matrix X) should be compared. If two of the predictor variables are highly correlated, their individual effects on the response variable y may be indistinguishable. The correlations between the predictor variable, may be seen by examining the *correlation matrix*. Including the response variable in a combined data matrix `xy` allows one to see how each predictor variable is correlated with the response.

```
banner ("Correlation Matrix for Columns of xy")
println (s"x_fname = ${stringOf (x_fname)}")
```

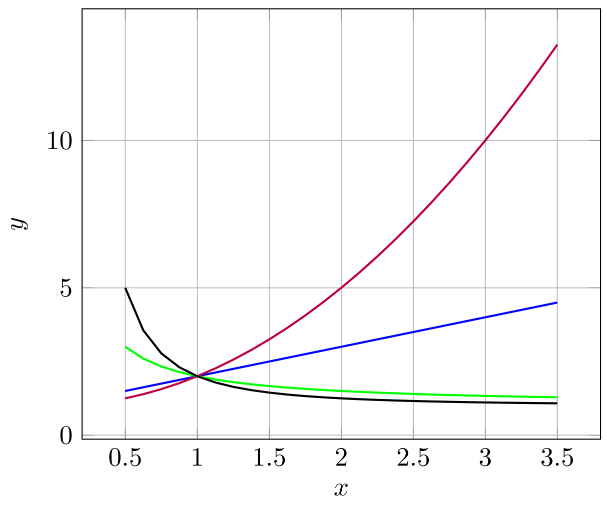
```
    println (s"y_name   = MPG")
    println (s"xy.corr = ${xy.corr}")
```

Although Simple Regression may be too simple for many problems/datasets, it should be used in *Exploratory Data Analysis* (EDA). A simple regression model should be created for each predictor variable x_j. The data points and the best fitting line should be plotted with y on the vertical axis and x_j on the horizontal axis. The data scientist should look for patterns/tendencies of y versus x_j, such as linear, quadratic, logarithmic, or exponential patterns. When there is no relationship, the points will appear to be randomly and uniformly positioned in the plane.

```
  for j <- x.indices2 do
      banner (s"Plot response y vs. predictor variable ${x_fname(j)}")
      val xj = x(?, j)
      val mod = SimpleRegression (xj, y, Array ("one", x_fname(j)))
      mod.trainNtest ()()                                        // train and test model
      val yp = mod.predict (mod.getX)
      new Plot (xj, y, yp, s"EDA: y and yp (red) vs. ${x_fname(j)}", lines = true)
   end for
```

The Figure below shows four possible patterns: Linear (blue), Quadratic (purple), Inverse (green), Inverse-Square (black). Each curve depicts a function $1 + x^p$, for $p = -2, -1, 1, 2$.

Finding a Pattern: Linear (blue), Quadratic (purple), Inverse (green), Inverse-Square (black)



To look for quadratic patterns, the following code regresses on the square of each predictor variable (i.e., x_j^2).

```
   for j <- x.indices2 do
       banner (s"Plot response y vs. predictor variable ${x_fname(j)}")
       val xj = x(?, j)
       val mod = SimpleRegression.quadratic (xj, y, Array ("one", x_fname(j) + "^2"))
       mod.trainNtest ()()                                       // train and test model
       val yp = mod.predict (mod.getX)
       new Plot (xj, y, yp, s"EDA: y and yp (red) vs. ${x_fname(j)}", lines = false)
   end for
```

To determine the effect of having linear and quadratic terms (both x_j and x_j^2) the `Regression` class that supports Multiple Linear Regression or the `SymbolicRegression` object may be used. Generally, one could include both terms if there is sufficient improvement over just using one term. If one term is chosen, use the linear term unless the quadratic term is sufficiently better (see the section on Symbolic Regression for a more detailed discussion).

Plotting

The `Plot` and `PlotM` classes in the `mathstat` package can be used for plotting data and results. Both use `ZoomablePanel` in the `scala2d` package to support zooming and dragging. The mouse wheel controls the amount of zooming (scroll value where up is negative and down is positive), while mouse dragging repositions the objects in the panel (drawing canvas).

```
@param x        the x vector of data values (horizontal), use null to use y s index
@param y        the y vector of data values (primary vertical, black)
@param z        the z vector of data values (secondary vertical, red) to compare with y
@param _title   the title of the plot
@param lines    flag for generating a line plot

class Plot (x: VectorD, y: VectorD, z: VectorD = null, _title: String = "Plot y vs. x",
            lines: Boolean = false)
      extends VizFrame (_title, null):
```

```
@param x_       the x vector of data values (horizontal)
@param y_       the y matrix of data values where y(i) is the i-th vector (vertical)
@param label    the label/legend/key for each curve in the plot
@param _title   the title of the plot
@param lines    flag for generating a line plot

class PlotM (x_ : VectorD, y_ : MatrixD, var label: Array [String] = null,
            _title: String = "PlotM y_i vs. x for each i", lines: Boolean = false)
      extends VizFrame (_title, null):
```

6.5.6 `SimpleRegression` Class

Class Methods:

```
@param x        the data/input matrix augmented with a first column of ones
                (only use the first two columns [1, x1])
@param y        the response/output vector
@param fname_   the feature/variable names (only use the first two names)

class SimpleRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null)
      extends Predictor (x, y, if fname_ == null then null else fname_.slice (0, 2),
                         null)
         with Fit (dfm = 1, df = x.dim - 2)
         with NoSubModels:

def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                             b_ : VectorD = b, vifs: VectorD = vif ()): String =
    def confInterval (x_ : MatrixD = getX): VectorD =
```

6.5.7 Exercises

1. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \le i < m\}$ and draw the line that is nearest to these points (i.e., that minimize $\|\boldsymbol{\epsilon}\|$). Using the formulas developed in this section, what are the intercept and slope $[b_0, b_1]$ of this line.

   Also, pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimpleRegression` class to obtain the $\mathbf{b}$ vector.

```
    // 4 data points:       one x1
    val x = MatrixD ((4, 2), 1, 1,                      // x 4-by-2 matrix
                             1, 2,
                             1, 3,
                             1, 4)
    val y = VectorD (1, 3, 3, 4)                        // y vector

    val mod = new SimpleRegression (x, y)               // create a simple regression
    model
    mod.trainNtest ()()                                 // train and test the model

    val yp = mod.predict (x)
    new Plot (x(?, 1), y, yp, "plot y and yp vs. x", lines = true)
```

2. For more complex models, setting the gradient to zero and solving a system of simultaneous equation may not work, in which case more general optimization techniques may be applied. Two simple optimization techniques are *grid search* and *gradient descent*.

   For grid search, in a spreadsheet set up a 5-by-5 grid around the optimal point for $\mathbf{b}$, found in the previous problem. Compute values for the loss function $\mathcal{L} = \frac{1}{2} sse$ for each point in the grid. Plot h versus b_0 across the optimal point. Do the same for b_1. Make a 3D plot of the surface h as a function b_0 and b_1.

   For gradient descent, pick a starting point $\mathbf{b}^0$, compute the gradient $\nabla\mathcal{L}$ and move $-\eta\nabla\mathcal{L}$ from $\mathbf{b}^0$ where η is the learning rate (e.g., 0.1). Repeat for a few iterations. What is happening to the value of the loss function $\mathcal{L} = \frac{1}{2} see$.

$$\nabla\mathcal{L} \;=\; [-\mathbf{1} \cdot (\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})), -\mathbf{x} \cdot (\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x}))]$$

   Substituting $\boldsymbol{\epsilon} = \mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})$, $\nabla\mathcal{L}$ may be written as

$$[-\mathbf{1} \cdot \boldsymbol{\epsilon}, -\mathbf{x} \cdot \boldsymbol{\epsilon}]$$

3. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \le i < m\}$ and draw the line that is nearest to these points. What are the intercept and slope of this line. Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimpleRegression` class to obtain the $\mathbf{b}$ vector.

```
    // 5 data points:       one  x1
    val x = MatrixD ((5, 2), 1, 0,                         // x 5-by-2 matrix
                             1, 1,
                             1, 2,
                             1, 3,
                             1, 4)
    val y = VectorD (2, 3, 5, 4, 6)                        // y vector

    val mod = new SimpleRegression (x, y)                  // create a simple regression
    mod.trainNtest ()()                                    // train and test the model

    val yp = mod.predict (x)
    new Plot (x(?, 1), y, yp, "plot y and yp vs. x", lines = true)
```

4. Execute `SimpleRegression` on the `Auto_MPG` dataset. See `scalation.modeling.Example_AutoMPG`. What is the quality of the fit (e.g., R^2 or `rSq`)? Is this value expected? Try using different columns for the predictor variable. Plot `y` and `yp` vs. `xj` for each feature/predictor variable. How do the results relate to information given in the correlation matrix?

5. Let errors ϵ_i have $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{V}[\epsilon_i] = \sigma^2$, and be independent of each other. Show that the variances for the parameters b_0 and b_1 are as follows:

$$\mathbb{V}[b_1] = \frac{\sigma^2}{S_{xx}}$$

Hint: $\mathbb{V}[b_1] = \mathbb{V}\left[\frac{S_{xy}}{S_{xx}}\right] = S_{xx}^{-2}\,\mathbb{V}[S_{xy}]$.

$$\mathbb{V}[b_0] = \left[\frac{1}{m} + \frac{\mu_x^2}{S_{xx}}\right]\sigma^2$$

Hint: $\mathbb{V}[b_0] = \mathbb{V}[\mu_y] - \mu_x^2\mathbb{V}[b_1]$

6. Further assume that $\epsilon_i \sim \mathrm{N}(0, \sigma^2)$. Show that the confidence intervals for the parameters b_0 and b_1 are as follows:

$$\left[b_1 \pm t^* \frac{s}{\sqrt{S_{xx}}}\right]$$

Hint: Let the error variance estimator be $s^2 = \dfrac{sse}{m-2} = mse$.

$$\left[b_0 \pm t^* s\sqrt{\frac{1}{m} + \frac{\mu_x^2}{S_{xx}}}\right]$$

7. For the following simple dataset,

```
    val x  = VectorD (1, 2, 3, 4, 5)
    val y  = VectorD (1, 3, 3, 5, 4)
    val ox = MatrixD.one (x.dim) :^+ x
```

estimate the error variance $s^2 = \dfrac{sse}{m-2} = mse$. Take its square root to obtain the residual standard error s. Use these to compute 95% confidence intervals for the parameters: b_0 and b_1.

8. Consider the above simple dataset, but where the y values are reversed so the slope is negative and the fit line is away from the origin,

```
    val x  = VectorD (1, 2, 3, 4, 5)
    val y  = VectorD (4, 5, 3, 3, 1)
    val ox = MatrixD.one (x.dim) :^+ x
```

Compare the `SimplerRegression` model with the `SimpleRegression` model. Examine the QoF measures for each model and make an argument for which model to pick. Also compute R_0^2 (R^2 relative to 0)

$$R_0^2 \;=\; 1 - \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|^2}{\|\mathbf{y}\|^2} \tag{6.37}$$

Recall the previous definition for R^2.

$$R^2 \;=\; 1 - \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|^2}{\|\mathbf{y} - \mu_{\mathbf{y}}\|^2} \tag{6.38}$$

For **Regression Through the Origin** (RTO) some software packages use R_0^2 in place of R^2. See [43] for a deeper discussion of the issues involved, including when it is appropriate to **not include an intercept** b_0 in the model. ScalaTion provides functions for both in the `FitM` trait: `def rSq_` (the default) and `def rSq0_`.

6.6 Regression

The `Regression` class supports multiple linear regression where multiple *input/predictor variables* are used to predict a value for the *response/output variable*. When the response variable has non-zero correlations with multiple predictor variables, this technique tends to be effective, efficient and leads to explainable models. It should be applied typically in combination with more complex modeling techniques. In this case, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ...x_k] \in \mathbb{R}^n$, so the parameter vector $\mathbf{b} = [b_0, b_1, \ldots, b_k]$ has the same dimension as $\mathbf{x}$, while response y is a scalar.

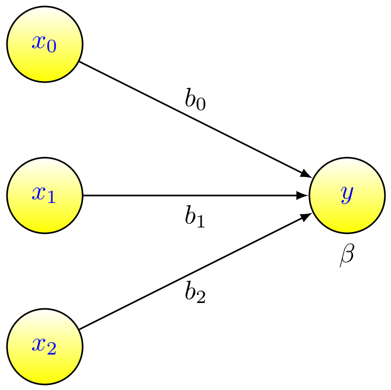


Figure 6.1: Multiple Linear Regression

The intercept can be provided by fixing x_0 to one, making b_0 the intercept. Alternatively, x_0 can be used as a regular input variable by introducing another parameter β for the intercept. In Neural Networks, β is referred to as bias and b_j is referred to as the edge weight connecting input vertex/node j to the output node as shown in Figure 6.1. Note, if a activation function f_a is added to the model, the Multiple Linear Regression model becomes a Perceptron model.

6.6.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... + b_k x_k + \epsilon} \tag{6.39}$$

where ϵ represents the residuals (the part not explained by the model).

6.6.2 Training

Using several data samples as a training set $(X, \mathbf{y})$, the `Regression` class in ScalaTion can be used to estimate the parameter vector $\mathbf{b}$. Each sample pairs an $\mathbf{x}$ input vector with a y response value. The $\mathbf{x}$ vectors are placed into a data/input matrix $X \in \mathbb{R}^{m \times n}$ row-by-row with a column of ones as the first column in X. The individual response values taken together form the response vector $\mathbf{y} \in \mathbb{R}^m$.

The training diagram shown in Figure 6.2 illustrates how the i^{th} instance/row flows through the diagram computing the predicted response $\hat{y} = \mathbf{b} \cdot \mathbf{x}$ and the error $\epsilon = y - \hat{y}$.

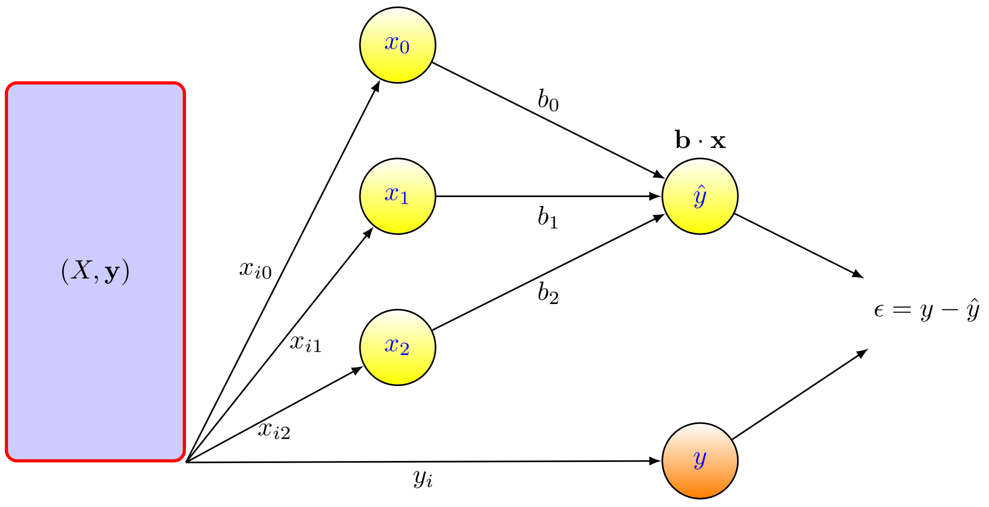


Figure 6.2: Training Diagram for Regression

The matrix-vector product $X\mathbf{b}$ provides an estimate for the response vector $\hat{\mathbf{y}}$.

$$\mathbf{y} \ = \ X\mathbf{b} + \boldsymbol{\epsilon} \tag{6.40}$$

The goal is to minimize the distance between $\mathbf{y}$ and its estimate $\hat{\mathbf{y}}$. i.e., minimize the norm of residual/error vector.

$$\min_{\mathbf{b}} \|\boldsymbol{\epsilon}\| \tag{6.41}$$

Substituting $\boldsymbol{\epsilon} \ = \ \mathbf{y} - \hat{\mathbf{y}} \ = \ \mathbf{y} - X\mathbf{b}$ yields

$$\min_{\mathbf{b}} \|\mathbf{y} - X\mathbf{b}\| \tag{6.42}$$

This is equivalent to minimizing half the dot product of the error vector with itself ($\frac{1}{2}\|\boldsymbol{\epsilon}\|^2 = \frac{1}{2}\boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} = \frac{1}{2}sse$) Thus, the loss function is

$$\mathcal{L}(\mathbf{b}) \ = \ \frac{1}{2} \boxed{\mathbf{y} - X\mathbf{b}} \cdot \boxed{\mathbf{y} - X\mathbf{b}} \tag{6.43}$$

6.6.3 Optimization - Gradient

The gradient of the loss function $\nabla\mathcal{L}$ with respect to the parameter vector $\mathbf{b}$ is the vector of partial derivatives.

$$\nabla\mathcal{L} \ = \ \left[\frac{\partial\mathcal{L}}{\partial b_0}, \frac{\partial\mathcal{L}}{\partial b_1}, \dots \frac{\partial\mathcal{L}}{\partial b_k}\right] \tag{6.44}$$

Again using the product rule for dot products

$$\frac{1}{2}(\mathbf{f} \cdot \mathbf{f})' \ = \ \boxed{\mathbf{f}'} \cdot \mathbf{f} \tag{6.45}$$

yields the j^{th} partial derivative.

$$\frac{\partial \mathcal{L}}{\partial b_j} \;=\; \boxed{-\mathbf{x}_{:j}} \cdot (\mathbf{y} \,-\, X\mathbf{b}) \;=\; -\,\mathbf{x}_{:j}^{\intercal}(\mathbf{y} \,-\, X\mathbf{b}) \tag{6.46}$$

Notice that the parameter b_j is only multiplied by column $\mathbf{x}_{:j}$ in the matrix-vector product $X\mathbf{b}$. The dot product is equivalent a transpose operation followed by matrix multiplication. The gradient is formed by collecting all these partial derivatives together.

$$\nabla\mathcal{L} \;=\; -\,X^{\intercal}(\mathbf{y} \,-\, X\mathbf{b}) \tag{6.47}$$

Now, setting the gradient equal to the zero vector $\mathbf{0} \in \mathbb{R}^n$ yields

$$-X^{\intercal}(\mathbf{y} \,-\, X\mathbf{b}) \;=\; \mathbf{0}$$

$$-X^{\intercal}\mathbf{y} \,+\, (X^{\intercal}X)\mathbf{b} \;=\; \mathbf{0}$$

A more detailed derivation of this equation is given in section 3.4 of "Matrix Calculus: Derivation and Simple Application" [82]. Moving the term involving $\mathbf{b}$ to the left side, results in the *Normal Equations.*

$$\boxed{(X^{\intercal}X)\mathbf{b} \;=\; X^{\intercal}\mathbf{y}} \tag{6.48}$$

Note: equivalent to minimizing the distance between $\mathbf{y}$ and $X\mathbf{b}$ is minimizing the sum of the squared residuals/errors (*Least Squares* method).

ScalaTion provides five techniques for solving for the parameter vector $\mathbf{b}$ based on the Normal Equations: Matrix Inversion, LU Factorization, Cholesky Factorization, QR Factorization and SVD Factorization.

6.6.4 Matrix Inversion Technique

Starting with the Normal Equations

$$(X^{\intercal}X)\mathbf{b} \;=\; X^{\intercal}\mathbf{y}$$

a simple technique is Matrix Inversion, which involves computing the inverse of $X^{\intercal}X$ and using it to multiply both sides of the Normal Equations.

$$\boxed{\mathbf{b} \;=\; (X^{\intercal}X)^{-1}X^{\intercal}\mathbf{y}} \tag{6.49}$$

where $(X^{\intercal}X)^{-1}$ is an n-by-n matrix, $X^{\intercal}$ is an n-by-m matrix and $\mathbf{y}$ is an m-vector. When X is full rank, the expression above involving the X matrix may be referred to as the pseudo-inverse X^+.

$$X^+ \;=\; (X^{\intercal}X)^{-1}X^{\intercal}$$

When X is not full rank, Singular Value Decomposition may be applied to compute X^+. Using the pseudo-inverse, the parameter vector $\mathbf{b}$ may be solved for as follows:

$$\mathbf{b} \;=\; X^+\mathbf{y} \tag{6.50}$$

The pseudo-inverse can be computed by first multiplying X by its transpose. Gaussian Elimination can be used to compute the inverse of this, which can be then multiplied by the transpose of X. In ScalaTion, the computation for the pseudo-inverse (`x_pinv`) looks similar to the math.

```
    val x_pinv = (x.𝒯 * x).inverse * x.𝒯
```

Most of the factorization classes/objects implement matrix inversion, including `Fac_Inv`, `Fac_LU`, `Fac_Cholesky`, and `Fac_QR`. The default `Fac_LU` combines reasonable speed and robustness.

```
    def inverse: MatrixD = Fac_LU.inverse (this)()
```

For efficiency, the code in `Regression` does not calculate `x_pinv`, rather is directly solves for the parameters $\mathbf{b}$.

```
    val b = fac.solve (x.𝒯 * y)
```

The Hat Matrix

Starting the solution to the Normal Equations that takes the inverse for determining the optimal parameter vector $\mathbf{b}$,

$$\mathbf{b} \;=\; (X^{\intercal}X)^{-1}X^{\intercal}\mathbf{y} \tag{6.51}$$

One can substitute the rhs into the prediction equation for $\hat{\mathbf{y}} \;=\; X\mathbf{b}$

$$\hat{\mathbf{y}} \;=\; X(X^{\intercal}X)^{-1}X^{\intercal}\mathbf{y} \;=\; H\mathbf{y} \tag{6.52}$$

where $H = X(X^{\intercal}X)^{-1}X^{\intercal}$ is the hat matrix (puts a hat on $\mathbf{y}$). The hat matrix may be viewed as a projection matrix.

6.6.5 LU Factorization Technique

Lower, Upper Factorization (Decomposition) works like Matrix Inversion, except that is just reduces the matrix to zeroes below the diagonal, so it tends to be faster and less prone to numerical instability. First the product $X^{\intercal}X$, an n-by-n matrix, is factored

$$X^{\intercal}X \;=\; LU$$

where L is a lower left triangular n-by-n matrix and U is an upper right triangular n-by-n matrix. Then the normal equations may be rewritten

$$LU\mathbf{b} \;=\; X^{\intercal}\mathbf{y}$$

Letting $\mathbf{w} = U\mathbf{b}$ allows the problem to solved in two steps. The first is solved by *forward substitution* to determine the vector $\mathbf{w}$.

$$L\mathbf{w} \;=\; X^{\intercal}\mathbf{y}$$

Finally, the parameter vector $\mathbf{b}$ is determined by *backward substitution*.

$$U\mathbf{b} \;=\; \mathbf{w}$$

Example Calculation

Consider the example where the input/data matrix X and output/response vector $\mathbf{y}$ are as follows:

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 1 \\ 3 \\ 3 \\ 4 \end{bmatrix}$$

Putting these values into the Normal Equations $(X^\intercal X)\mathbf{b} = X^\intercal \mathbf{y}$ yields

$$\left[\begin{array}{cc|c} 4 & 10 & 11 \\ 10 & 30 & 32 \end{array}\right]$$

Multiply the first row by -2.5 and add it to the second row,

$$\left[\begin{array}{cc|c} 4 & 10 & 11 \\ 0 & 5 & 4.5 \end{array}\right]$$

This results in the following optimal parameter vector $\mathbf{b} = [.5, .9]$. Note, the product of L and U gives $X^\intercal X$.

$$\begin{bmatrix} 1 & 0 \\ 2.5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 10 \\ 0 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 10 \\ 10 & 30 \end{bmatrix}$$

6.6.6 Cholesky Factorization Technique

A faster and slightly more stable technique is to use Cholesky Factorization. Since the product $X^\intercal X$ is a positive definite, symmetric matrix, it may factored using Cholesky Factorization into

$$X^\intercal X = LL^\intercal$$

where L is a lower triangular n-by-n matrix. Then the normal equations may be rewritten

$$LL^\intercal \mathbf{b} = X^\intercal \mathbf{y}$$

Letting $\mathbf{w} = L^\intercal \mathbf{b}$, we may solve for $\mathbf{w}$ using forward substitution

$$L\mathbf{w} = X^\intercal \mathbf{y}$$

and then solve for $\mathbf{b}$ using backward substitution.

$$L^\intercal \mathbf{b} = \mathbf{w}$$

As an example, the product of L and its transpose $L^\intercal$ gives $X^\intercal X$.

$$\begin{bmatrix} 2 & 0 \\ 5 & \sqrt{5} \end{bmatrix} \begin{bmatrix} 2 & 5 \\ 0 & \sqrt{5} \end{bmatrix} = \begin{bmatrix} 4 & 10 \\ 10 & 30 \end{bmatrix}$$

Therefore, $\mathbf{w}$ can be determined by forward substitution and $\mathbf{b}$ by backward substitution.

$$\begin{bmatrix} 2 & 0 \\ 5 & \sqrt{5} \end{bmatrix} \mathbf{w} \;=\; \begin{bmatrix} 11 \\ 32 \end{bmatrix} \quad , \qquad \begin{bmatrix} 2 & 5 \\ 0 & \sqrt{5} \end{bmatrix} \mathbf{b} \;=\; \mathbf{w}$$

6.6.7 QR Factorization Technique

A slightly slower, but even more robust technique is to use QR Factorization. Using this technique, the m-by-n X matrix can be factored directly, which increases the stability of the technique.

$$X \;=\; QR$$

where Q is an orthogonal m-by-n matrix and R matrix is a right upper triangular n-by-n matrix. Starting again with the Normal Equations,

$$(X^{\intercal} X)\mathbf{b} \;=\; X^{\intercal}\mathbf{y}$$

simply substitute QR for X.

$$(QR)^{\intercal} QR\mathbf{b} \;=\; (QR)^{\intercal}\mathbf{y}$$

Taking the transpose gives

$$R^{\intercal} Q^{\intercal} QR\mathbf{b} \;=\; R^{\intercal} Q^{\intercal}\mathbf{y}$$

and using the fact that $Q^{\intercal} Q = I$, we obtain the following:

$$R^{\intercal} R\mathbf{b} \;=\; R^{\intercal} Q^{\intercal}\mathbf{y}$$

Multiply both sides by $(R^{\intercal})^{-1}$ yields

$$R\mathbf{b} \;=\; Q^{\intercal}\mathbf{y}$$

Since R is an upper triangular matrix, the parameter vector $\mathbf{b}$ can be determined by backward substitution. Alternatively, the pseudo-inverse may be computed as follows:

$$X^{+} \;=\; R^{-1} Q^{\intercal}$$

ScalaTion uses Householder Orthogonalization (alternately Modified Gram-Schmidt Orthogonalization) to factor X into the product of Q and R.

6.6.8 Use of Factorization in Regression

By default, ScalaTion uses QR Factorization for matrix factorization. The other techniques may be selected by changing the hyper-parameter (`algorithm`), setting it to `Cholesky`, `SVD`, `LU` or `Inverse`. For more information see `http://see.stanford.edu/materials/lsoeldsee263/05-ls.pdf`.

Based on the selected algorithm, the appropriate type of matrix factorization is performed. The first part of the code below constructs and returns a factorization object.

```
    private def solver (x_ : MatrixD): Factorization =
        algorithm match
        case "Fac_Cholesky" => new Fac_Cholesky (x_.𝒯 * x_)       // Cholesky Factorization
        case "Fac_LU"       => new Fac_LU (x_.𝒯 * x_)             // LU Factorization
        case "Fac_Inverse"  => new Fac_Inverse (x_.𝒯 * x_)        // Inverse Factorization
        case "Fac_SVD"      => new Fac_SVD (x_)                   // Singular Value Decomp.
        case _              => new Fac_QR (x_)                    // QR Factorization
        end match
    end solver
```

The `train` method below computes parameter/coefficient vector **b** by calling the `solve` method provided by the factorization classes.

```
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
        val fac = solver (x_)
        fac.factor ()                                         // factor the matrix

        b = fac match                                         // RECORD the parameters
            case fac: Fac_QR  => fac.solve (y_)
            case fac: Fac_SVD => fac.solve (y_)
            case _            => fac.solve (x_.𝒯 * y_)

        if b(0).isNaN then flaw ("train", s"parameter b = $b")
        debug ("train", s"$fac estimates parameter b = $b")
    end train
```

After training, the `test` method does two things: First, the residual/error vector $\boldsymbol{\epsilon}$ is computed. Second, several quality of fit measures are computed by calling the `diagnose` method.

```
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
        val yp = predict (x_)                                 // make predictions
        e = y_ - yp                                           // RECORD the residuals/errors
        (yp, diagnose (y_, yp))                               // return predictions and QoF
    end test
```

To see how the `train` and `test` methods work in a `Regression` model see the Collinearity Test and Texas Temperatures examples in subsequent subsections.

6.6.9 Model Assessment

The quality of fit measures includes the coefficient of determination R^2 as well as several others.

Degrees of Freedom

Given m instances, k variables and n parameters in a regression model,

Table 6.6: Degrees of Freedom: Part 1

Instances	m	number of data points
Variables	k	number of non-redundant predictor variables
Parameters	n	number of parameters

the prediction vector $\hat{\mathbf{y}}$ is a projection of the response vector $\mathbf{y} \in \mathbb{R}^m$ onto $\mathbb{R}^k$, the space (hyperplane) spanned by the vectors $\mathbf{x}_1, \ldots \mathbf{x}_k$. Since $\boldsymbol{\epsilon} = \mathbf{y} - \hat{\mathbf{y}}$, one might think that the residual/error $\boldsymbol{\epsilon} \in \mathbb{R}^{m-k}$. As $\sum_i \epsilon_i = 0$ when an intercept parameter b_0 is included in the model ($n = k + 1$), this constraint reduces the dimensionality of the space by one, so $\boldsymbol{\epsilon} \in \mathbb{R}^{m-n}$.

Therefore, the Degrees of Freedom (DoF) captured by the regression model is df_r and left for error is df are indicated in the table below.

Table 6.7: Degrees of Freedom: Part 2

df_r	k	degrees of freedom regression/model
df	$m - n$	degrees of freedom residuals/error

As an example, the equation $\hat{y} = 2x_1 + x_2 + .5$ defines a $df_r = 2$ dimensional hyperplane (or ordinary plane) as shown in Figure 6.3.

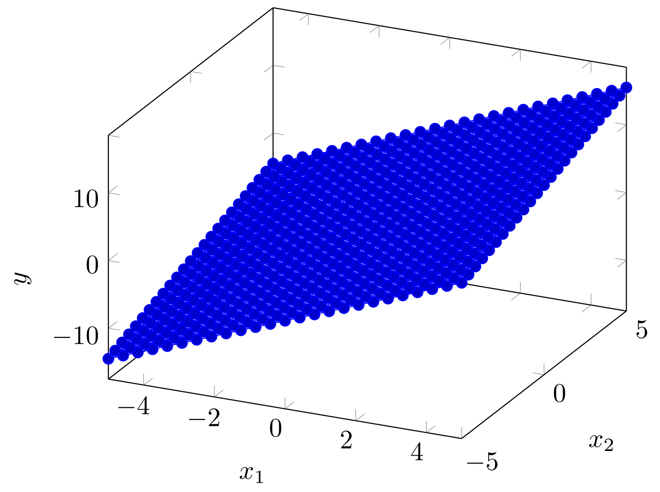


Figure 6.3: Hyperplane: $\hat{y} = 2x_1 + x_2 + .5$

It is important to remember that if the model has an intercept, $k = n - 1$, otherwise $k = n$.
Note, for more complex or regularized models, effective Degrees of Freedom (eDoF) may be used, see the exercises in the section of Ridge Regression.

Adjusted Coefficient of Determination $\bar{R}^2$

The ratio of total Degrees of Freedom to Degrees of Freedom for error is

$$r_{df} \;=\; \frac{df_r + df}{df}$$

`SimplerRegression` is at one extreme of model complexity, where $df = m{-}1$ and $df_r = 1$, so $r_{df} = m/(m{-}1)$ is close to one. For a more complicated model, say with $n = m/2$, r_{df} will be close to 2. This ratio can be used to adjust the Coefficient of Determination R^2 to reduce it with increasing number of parameters. This is called the Adjusted Coefficient of Determination $\bar{R}^2$

$$\bar{R}^2 \;=\; 1 - r_{df}(1 - R^2)$$

Suppose $m = 121, n = 21$ and $R^2 = 0.9$, as an exercise, show that $r_{df} = 1.2$ and $\bar{R}^2 = 0.88$.

Dividing *sse* and *ssr* by their respective Degrees of Freedom gives the mean square error and regression, respectively

$$\begin{aligned} mse &= sse \,/\, df \\ msr &= ssr \,/\, df_r \end{aligned}$$

The mean square error *mse* follows a Chi-square distribution with *df* Degrees of Freedom, while the mean square regression *msr* follows a Chi-square distribution with df_r Degrees of Freedom. Consequently, the ratio

$$\frac{msr}{mse} \;\sim\; F_{df_r,df} \tag{6.53}$$

that is, it follows an F-distribution with (df_r, df) Degrees of Freedom. If this number exceeds the critical value, one can claim that the parameter vector $\mathbf{b}$ is not zero, implying the model is useful. More general quality of fit measures useful for comparing models are the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC).

In ScalaTion the several Quality of Fit (QoF) measures are computed by the `diagnose` method in the `Fit` class, as described in section 1 of this chapter.

```
    def diagnose (y: VectorD, yp: VectorD, w: VectorD = null)
```

It looks at different ways to measure the difference between the actual `y` and predicted `yp` values for the response. The differences are optionally weighted by the vector `w`. Weighting is not applied when `w` is null.

6.6.10 Model Validation

Data are needed to two purposes: First, the characteristics or patterns of the data need to be investigated to select an appropriate modeling technique, features for a model and finally to estimate the parameters and probabilities used by the model. Data Scientists assisted by tools do the first part of this process, while the latter part is called *training*. Hence the `train` method is part of all modeling techniques provided by ScalaTion. Second, data are needed to test the quality of the trained model.

One approach would be to train the model using all the available data. This makes sense, since the more data used for training, the better the model. In this case, the testing data would need to be same as the training leading to whole dataset evaluation (*in-sample*).

Now the difficult issue is how to guard against *over-fitting*. With enough flexibility and parameters to fit, modeling techniques can push quality measures like R^2 to perfection ($R^2 = 1$) by fitting the signal and the noise in the data. Doing so tends to make a model worse in practice than a simple model that just captures the signal. That is where quality measures like $\bar{R}^2$ (or AIC) come into play, but computations of $\bar{R}^2$ require determination of Degrees of Freedom (df), which may be difficult for some modeling techniques. Furthermore, the amount of penalty introduced by such quality measures is somewhat arbitrary.

Would not it be better to measure quality in way in which models fitting noise are downgraded because they perform more poorly on data they have not seen? Is it really a test, if the model has already seen the data? The answers to these questions are obvious, but the solution of the underlying problem is a bit tricky. The first thought would be to divide a dataset in half, but then only half of the data are available for training. Also, picking a different half may result in substantially different quality measures.

This leads to two guiding principles: First, the majority of the data should be used for training. Second, multiple testing should be done. In general, conducting real-world tests of a model can be difficult. There are, however, strategies that attempt to approximate such testing. Two simple and commonly used strategies are the following: *Leave-One-Out* and *Cross-Validation*. In both cases, a dataset is divided into a *training set* and a *test set*.

Leave-One-Out

When fitting the parameters $\mathbf{b}$ the more data available in the training set, in all likelihood, the better the fit. The Leave-One-Out strategy takes this to the extreme, by splitting the dataset into a training set of size $m - 1$ and test set of size 1 (e.g., row t in data matrix X). From this, a test error can be computed $y_t - \mathbf{b} \cdot \mathbf{x}_t$. This can be repeated by iteratively letting t range from the first to the last row of data matrix X. For certain predictive analytics techniques such as Multiple Linear Regression, there are efficient ways to compute the test sse based on the leverage each point in the training set has [85].

k-Fold Cross-Validation

A more generally applicable strategy is called cross-validation, where a dataset is divided into k test sets. For each test set, the corresponding training set is all the instances not chosen for that test set. A simple way to do this is to let the first test dataset be first m/k rows of matrix X, the second be the second m/k rows, etc.

```
    val tsize = m / k                                                   // test set size
    for l <- 0 until k do
        x_e = x(l * tsize until ((l+1) * tsize)                         // l-th test set
        x_  = x.not(l * tsize until ((l+1) * tsize))                    // l-th training set
    end for
```

The model is trained k times using each of the training sets. The corresponding test set is then used to estimate the test sse (or other quality measure such as mse). These are more meaningful *out-of-sample* results. From each of these samples, a mean, standard deviation and confidence interval may be computed for the test sse.

Due to patterns that may exist in the dataset, it is more robust to randomly select each of the test sets. The row indices may be permuted for random selection that ensures that all data instances show up exactly in one test set.

Typically, training QoF (in-sample) measures such as R^2 will be better than testing QoF (out-of-sample) measures such as R^2_{cv}. Adjusted measures such as $\bar{R}^2$ are intending to more closely follow R^2_{cv} than R^2.

ScalaTion support cross-validation via is `crossValidate` method.

```
    @param k       the number of cross-validation iterations/folds (defaults to 5x).
    @param rando   flag indicating whether to use randomized or simple cross-validation

    def crossValidate (k: Int = 5, rando: Boolean = true): Array [Statistic] =
```

It also supports a simpler strategy that only tests once, via its `validate` method defined in the `Predictor` trait. It utilizes the Test-n-Train Split `TnT_Split` from the `mathstat` package.

```
    @param rando   flag indicating whether to use randomized or simple validation
    @param ratio   the ratio of the TESTING set to the full dataset (e.g., 70-30, 80-20)
    @param idx     the prescribed TESTING set indices

    def validate (rando: Boolean = true, ratio: Double = 0.2)
                 (idx : IndexedSeq [Int] =
                  testIndices (rando, (ratio * y.dim).toInt)): VectorD =
        val (x_e, x_, y_e, y_) = TnT_Split (x, y, idx)       // Test-n-Train Split

        train (x_, y_)
        val qof = test (x_e, y_e)._2
        if qof(QoF.sst.ordinal) <= 0.0 then
            flaw ("validate", "chosen testing set has no variability")
        end if
        println (FitM.fitMap (qof, QoF.values.map (_.toString)))
        qof
    end validate
```

6.6.11 Collinearity

Consider the matrix-vector equation used for estimating the parameters $\mathbf{b}$ via the minimization of $\|\boldsymbol{\epsilon}\|$.

$$\mathbf{y} \;=\; X\mathbf{b} + \boldsymbol{\epsilon}$$

The parameter/coefficient vector $\mathbf{b} = [b_0, b_1, \dots, b_k]$ may be viewed as weights on the column vectors in the data/predictor matrix X.

$$\mathbf{y} \;=\; b_0\mathbf{1} \;+\; b_1\mathbf{x}_{:1} \;+\; \dots \;+\; b_k\mathbf{x}_{:k} \;+\; \boldsymbol{\epsilon}$$

A question arises when two of these column vectors are nearly the same (or more generally nearly parallel or anti-parallel). They will affect and may obfuscate each others' parameter values.

First, we will examine ways of detecting such problems and then give some remedies. A simple check is to compute the correlation matrix for the column vectors in matrix X. High (positive or negative) correlation indicates *collinearity*.

Example Problem

Consider the following data/input matrix X and response vector $\mathbf{y}$. This is the same example used for `SimpleRegression` with new variable x_2 added (i.e., $y = b_0 + b_1x_1 + b_2x_2 + \epsilon$). The `collinearityTest` main function allows one to see the effects of increasing the collinearity of features/variables x_1 and x_2.

```
package <your-package>

import scalation.modeling.Regression
import scalation.mathstat.{MatrixD, VectorD}

@main def collinearityTest (): Unit =

//                          one  x1  x2
    val x = MatrixD ((4, 3), 1, 1, 1,              // input/data matrix
                             1, 2, 2,
                             1, 3, 3,
                             1, 4, 0)              // change 0 by adding .5 until it's 4

    val y  = VectorD (1, 3, 3, 4)                  // output/response vector

    val v = x(?, 0 until 2)
    banner (s"Test without column x2")
    println (s"v = $v")
    var mod = new Regression (v, y)
    mod.trainNtest ()()
    println (mod.summary ())

    for i <- 0 to 8 do
        banner (s"Test Increasing Collinearity: x_32 = ${x(3, 2)}")
        println (s"x = $x")
        println (s"x.corr = ${x.corr}")
        mod = new Regression (x, y)
        mod.trainNtest ()()
        println (mod.summary ())
        x(3, 2) += 0.5
    end for

end collinearityTest
```

Try changing the value of element x_{32} from 0 to 4 by .5 and observe what happens to the correlation matrix. What effect do these changes have on the parameter vector $\mathbf{b} = [b_0, b_1, b_2]$ and how do the first two parameters compare to the regression where the last column of X is removed giving the parameter vector $\mathbf{b} = [b_0, b_1]$.

The `corr` method is provided by the `scalation.mathstat.MatrixD` class. For this method, if either column vector has zero variance, when the column vectors are the same, it returns 1.0, otherwise -0.0 (indicating undefined).

Note, perfect collinearity produces a singular matrix, in which case many factorization algorithms will give `NaN` (Not-a-Number) for much of their output. In this case, `Fac_SVD` (Singular Value Decomposition) should be used. This can be done by changing the following hyper-parameter provided by the `Regression` object, before instantiating the `Regression` class.

```
Regression.hp("factorization") = "Fac_SVD"
val mod = new Regression (x, y)
```

Multi-Collinearity

Even if no particular entry in the correlation matrix is high, a column in the matrix may still be nearly a linear combination of other columns. This is the problem of *multi-collinearity*. This can be checked by computing the Variance Inflation Factor (VIF) function (or vif in ScalaTion). For a particular parameter b_j for the variable/predictor x_j, the function is evaluated as follows:

$$\text{vif}(b_j) \;=\; \frac{1}{1 - R^2(x_j)} \tag{6.54}$$

where $R^2(x_j)$ is R^2 for the regression of variable x_j onto the rest of the predictors. It measures how well the variable x_j (or its column vector $\mathbf{x}_{:j}$) can be predicted by all x_l for $l \neq j$. Values above 20 ($R^2(x_j) = 0.95$) are considered by some to be problematic. In particular, the value for parameter b_j may be suspect, since its variance is inflated by vif(b_j).

$$\hat{\sigma}^2(b_j) \;=\; \frac{mse}{k\,\hat{\sigma}^2(x_j)} \cdot \text{vif}(b_j) \tag{6.55}$$

See the exercises for details. Both `corr` and `vif` may be tested in ScalaTion using `RegressionTest4`.

One remedy to reduce collinearity/multi-collinearity is to eliminate the variable with the highest `corr`/`vif` value. Another is to use regularized regression such as `RidgeRegression` or `LassoRegression`.

6.6.12 Feature Selection

There may be predictor variables (features) in the model that contribute little in terms of their contributions to the model's ability to make predictions. The improvement to R^2 may be small and may make $\bar{R}^2$ or other quality of fit measures worse. An easy way to get a basic understanding is to compute the correlation of each predictor variable $\mathbf{x}_{:j}$ (j^{th} column of matrix X) with the response vector $\mathbf{y}$. A more intuitive way to do this would be to plot the response vector $\mathbf{y}$ versus each predictor variable $\mathbf{x}_{:j}$. See the exercises for an example.

Ideally, one would like pick a subset of the k variables that would optimize a selected quality measure. Unfortunately, there are 2^k possible subsets to test. Two simple techniques (greedy algorithms) for selecting features are forward selection and backward elimination. A combination of these two is provided by stepwise regression.

Forward Selection

The `forewordSel` method, coded in the `Predictor` trait, performs *forward selection* by adding the most predictive variable to the existing model, returning the variable to be added and a reference to the new model with the added variable/feature.

```
    @param cols   the columns of matrix x currently included in the existing model
    @param idx_q  index of Quality of Fit (QoF) to use for comparing quality

    def forwardSel (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal): BestStep =
```

The `BestStep` is used to record the best improvement step found so far.

```
    @param col  the column/variable to ADD/REMOVE for this step
    @param qof  the Quality of Fit (QoF) for this step
    @param mod  the model including selected features/variables for this step
```

```scala

    case class BestStep (col: Int = -1, qof: VectorD = null, mod: Predictor = null)
```

Selecting the most predictive variable to add boils down to comparing on the basis of a Quality of Fit (QoF) measure. The default is the Adjusted Coefficient of Determination $\bar{R}^2$. The optional argument `idx_q` indicates which QoF measure to use (defaults to `QoF.rSqBar.ordinal`). To start with a minimal model, set `cols = Set (0)` for an intercept-only model. The method will consider every variable/column `x.indices2` not already in `cols` and pick the best one for inclusion.

```scala
    for j <- x.indices2 if ! (cols contains j) do
```

To find the best model, the `forwardSel` method should be called repeatedly while the quality of fit measure is sufficiently improving. This process is automated in the `forwardSelAll` method.

```scala
    @param idx_q  index of Quality of Fit (QoF) to use for comparing quality
    @param cross  whether to include the cross-validation QoF measure

    def forwardSelAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
                      (LinkedHashSet [Int], MatrixD) =
```

The `forwardSelAll` method takes the QoF measure to use as the selection criterion and whether to apply cross-validation as inputs and returns the best collection of features/columns to include in the model as well as the QoF measures for all steps.

To see how R^2, $\bar{R}^2$, sMAPE, and R^2_{cv} change with the number of features/parameters added to the model by `forwardSelAll` method, run the following test code from the `scalation_modeling` module.

```
sbt> runMain scalation.modeling.regressionTest5
```

sMAPE, symmetric Mean Absolute Percentage Error, is explained in detail in the Time Series/Temporal Models Chapter.

Backward Elimination

The `backwardElim` method, coded in the `Predictor` trait, performs *backward elimination* by removing the least predictive variable from the existing model, returning the variable to eliminate, the new parameter vector and a reference to the new model with the removed variable/feature.

```scala
    @param cols   the columns of matrix x currently included in the existing model
    @param idx_q  index of Quality of Fit (QoF) to use for comparing quality
    @param first  first variable to consider for elimination
                        (default (1) assume intercept x_0 will be in any model)

    def backwardElim (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal,
                      first: Int = 1): BestStep =
```

To start with a maximal model, set `cols = Set (0, 1, ..., k)` for a full model. As with `forwardSel`, the `idx_q` optional argument allows one to choose from among the QoF measures. The last parameter `first` provides immunity from elimination for any variable/parameter that is less than `first` (e.g., to ensure that models include an intercept b_0, set `first` to one). The method will consider every variable/column from `first` until `x.dim2` in `cols` and pick the worst one for elimination.

```scala
    for j <- first until x.dim2 if cols contains j do
```

To find the best model, the `backwardElim` method should be called repeatedly until the quality of fit measure sufficiently decreases. This process is automated in the `backwardElimAll` method.

```
    @param idx_q  index of Quality of Fit (QoF) to use for comparing quality
    @param first  first variable to consider for elimination
    @param cross  whether to include the cross-validation QoF measure

    def backwardElimAll (idx_q: Int = QoF.rSqBar.ordinal, first: Int = 1,
                          cross: Boolean = true):
                         (LinkedHashSet [Int], MatrixD) =
```

The `backwardElimAll` method takes the QoF measure to use as the selection criterion, the index of the first variable to consider for elimination, and whether to apply cross-validation as inputs and returns the best collection of features/columns to include in the model as well as the QoF measures for all steps.

Some studies have indicated that backward elimination can outperform forward selection, but it is difficult to say in general.

More advanced feature selection techniques include using genetic algorithms to find near optimal subsets of variables as well as techniques that select variables as part of the parameter estimation process, e.g., `LassoRegression`.

Stepwise Regression

An improvement over Forward Selection and Backward Elimination is possible with Stepwise Regression. It starts with either no variables or the intercept in the model and adds one variable that improves the selection criterion the most. It then adds the second best variable for step two. After the second step, it determines whether it is better to add or remove a variable. It continues in this fashion until no improvement in the selection criterion is found at which point it terminates. Note, for Forward Selection and Backward Elimination it may instructive to continue all the way to the end (all variables for forward/no variables for backward).

Stepwise regression may lead to coincidental relationships being included in the model, particularly if a t-test is the basis of inclusion or a penalty-free QoF measure such as R^2 is used. Typically, this approach is used when there a penalty for having extra variables/parameters, e.g., R^2 adjusted $\bar{R}^2$, R^2 cross-validation R^2_{cv} or Akaike Information Criterion (AIC). See the section on Maximum Likelihood Estimation for a definition of AIC. Alternatives to Stepwise Regression include Lasso Regression (ℓ^1 regularization) and to a lesser extent Ridge Regression (ℓ^2 regularization).

ScalaTion provides the `stepRegressionAll` method for Stepwise Regression. At each step it calls `forwardSel` and `backwardElim` and chooses the one yielding better improvement.

```
    @param idx_q  index of Quality of Fit (QoF) to use for comparing quality
    @param cross  whether to include the cross-validation QoF measure

    def stepRegressionAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
             (LinkedHashSet [Int], MatrixD) =
```

An option for further improvement is to add a *swapping operation*, which finds the best variable to remove and replace with a variable not in the model. Unfortunately, this may lead to a quadratic number of steps in the worst-case (as opposed to linear for forward, backward and stepwise without swapping). See the exercises for more details.

Categorical Variables/Features

For `Regression`, the variables/features have so far been treated as continuous or ordinal. However, some variables may be categorical in nature, where there is no ordering of the values for a categorical variable.

Although one can encode "English", "French", "Spanish" as 0, 1, and 2, it may lead to problems such as concluding the average of "English" and "Spanish" is 'French".

In such cases, it may be useful to replace a categorical variable with multiple *dummy variables*. Typically, a categorical variable (column in the data matrix) taking on k distinct values is replaced with with $k-1$ dummy variables (columns in the data matrix). For details on how to do this effectively, see the section on `RegressionCat`.

6.6.13 Regression Problem: Texas Temperatures

Solving a regression problem in ScalaTion simply involves creating a data/input matrix $X \in \mathbb{R}^{m \times n}$ and a response/output vector $\mathbf{y} \in \mathbb{R}^m$ and then creating a `Regression` object upon which the `trainNtest` method is called. The `trainNtest` method conveniently calls the `train`, `test` and `report` methods internally.

The Texas Temperature dataset below from `http://www.stat.ufl.edu/~winner/cases/txtemp.ppt` is used to illustrate how to use ScalaTion for a regression problem. The purpose of the model is to predict average January high temperatures for 16 Texas county weather stations based on their Latitude, Elevation and Longitude.

```scala
// 16 data points:         one      x1      x2       x3     y
//                                  Lat     Elev     Long   Temp         County
val xy = MatrixD ((16, 5), 1.0, 29.767,   41.0,  95.367, 56.0,    // Harris
                           1.0, 32.850,  440.0,  96.850, 48.0,    // Dallas
                           1.0, 26.933,   25.0,  97.800, 60.0,    // Kennedy
                           1.0, 31.950, 2851.0, 102.183, 46.0,    // Midland
                           1.0, 34.800, 3840.0, 102.467, 38.0,    // Deaf Smith
                           1.0, 33.450, 1461.0,  99.633, 46.0,    // Knox
                           1.0, 28.700,  815.0, 100.483, 53.0,    // Maverick
                           1.0, 32.450, 2380.0, 100.533, 46.0,    // Nolan
                           1.0, 31.800, 3918.0, 106.400, 44.0,    // El Paso
                           1.0, 34.850, 2040.0, 100.217, 41.0,    // Collington
                           1.0, 30.867, 3000.0, 102.900, 47.0,    // Pecos
                           1.0, 36.350, 3693.0, 102.083, 36.0,    // Sherman
                           1.0, 30.300,  597.0,  97.700, 52.0,    // Travis
                           1.0, 26.900,  315.0,  99.283, 60.0,    // Zapata
                           1.0, 28.450,  459.0,  99.217, 56.0,    // Lasalle
                           1.0, 25.900,   19.0,  97.433, 62.0)    // Cameron

banner ("Texas Temperatures Regression")
val mod = Regression (xy)()                           // create a regression model
mod.trainNtest ()()                                   // train and test the model
println (mod.summary ())                              // parameter/coefficient statistics
```

The `trainNtest` Method

The `trainNtest` method defined in the `Predictor` trait does several things: trains the model on `x_` and `y_`, tests the model on `xx` and `yy`, produces a report about training and testing, and optionally plots y-actual and y-predicted.

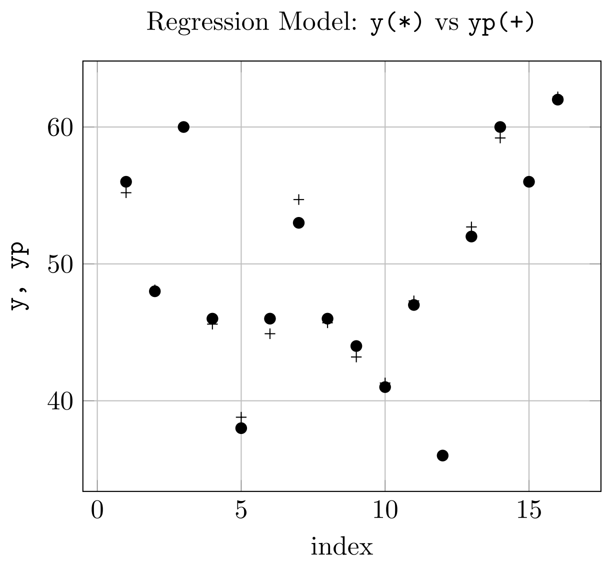
```
    @param x_  the training/full data/input matrix (defaults to full x)
    @param y_  the training/full response/output vector (defaults to full y)
    @param xx  the testing/full data/input matrix (defaults to full x)
    @param yy  the testing/full response/output vector (defaults to full y)

    def trainNtest (x_ : MatrixD = x, y_ : VectorD = y)
                   (xx:  MatrixD = x, yy:  VectorD = y): (VectorD, VectorD) =
        train (x_, y_)
        debug ("trainNTest", s"b = $b")
        val (yp, qof) = test (xx, yy)
        println (report (qof))
        if DO_PLOT then
            val lim = min (yy.dim, LIMIT)
            val (qyy, qyp) = (yy(0 until lim), yp(0 until lim))           // slice to LIMIT
            val (ryy, ryp) = orderByY (qyy, qyp)                          // order by yy
            new Plot (null, ryy, ryp, s"$modelName: y actual, predicted")
        end if
        (yp, qof)
    end trainNtest
```

The `report` Method

The `report` method returns the following basic information: (1) the name of the modeling technique `nm`, (2) the values of the hyper-parameters `hp` (used for controlling the model/optimizer), (3) the feature/predictor variable names `fn`, (4) the values of the parameters `b`, and (5) several Quality of Fit measures `qof`.

```
REPORT
    ----------------------------------------------------------------------------
    modelName  mn  = Regression
    ----------------------------------------------------------------------------
    hparameter hp  = HyperParameter(factorization -> (Fac_QR,Fac_QR))
    ----------------------------------------------------------------------------
    features   fn  = Array(x0, x1, x2, x3)
    ----------------------------------------------------------------------------
    parameter  b   = VectorD(151.298,-1.99323,-0.000955478,-0.384710)
    ----------------------------------------------------------------------------
    fitMap     qof = LinkedHashMap(
                     rSq  -> 0.991921,  rSqBar -> 0.989902,   sst -> 941.937500, sse -> 7.609494,
                     mse0 -> 0.475593,  rmse   -> 0.689633,   mae -> 0.531353,   dfm -> 3.000000,
                     df   -> 12.000000, fStat  -> 491.138015, aic -> -8.757481,  bic -> -5.667126,
                     mape -> 1.095990,  smape  -> 1.094779,  mase -> 0.066419)
```

The plot below shows the results from running the ScalaTion Regression Model in terms of actual (`y`) vs. predicted (`yp`) response vectors.



The summary Method

More details about the parameters/coefficients including standard errors, t-values, p-values, and Variance Inflation Factors (VIFs) are shown by the `summary` method.

```
    println (mod.summary ())
```

For the Texas Temperatures dataset it provides the following information: The `Estimate` is the value assigned to the parameter for the given `Var`. The `Std. Error`, `t-value`, `p-value` and `VIF` are also given.

```
fname = Array(x0 = intercept, x1 = Lat, x2 = Elev, x3 = Long)
SUMMARY
    Parameters/Coefficients:
    Var      Estimate     Std. Error         t value       Pr(>|t|)           VIF
----------------------------------------------------------------------------------
    x0     151.297616      25.133361        6.019792       0.000060            NA
    x1      -1.993228       0.136390      -14.614194       0.000000      4.228079
    x2      -0.000955       0.000568       -1.683440       0.118102     16.481808
    x3      -0.384710       0.228584       -1.683018       0.118185      9.432463

    Residual standard error: 0.796319 on 12.0 degrees of freedom
    Multiple R-squared:  0.991921,    Adjusted R-squared:  0.989902
    F-statistic: 491.1380151109529 on 3.0 and 12.0 DF,  p-value: 8.089084957418891E-13
----------------------------------------------------------------------------------
```

Given the following assumptions: (1) $\boldsymbol{\epsilon} \sim D(\mathbf{0}, \sigma I)$ for some distribution D and (2) for each column j, $\boldsymbol{\epsilon}$ and $\mathbf{x}_j$ are independent, the covariance matrix of the parameter vector $\mathbf{b}$ is

$$\mathbb{C}[\mathbf{b}] \;=\; \sigma^2 (X^{\intercal} X)^{-1} \tag{6.56}$$

See [159] for a derivation. Using $\hat{\sigma}^2$ as an estimate for σ^2,

$$\hat{\sigma}^2 = \frac{\boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}}{df} = mse \tag{6.57}$$

the standard deviation (or standard error) of the j^{th} parameter/coefficient may be given as the square root of the j^{th} diagonal element of the covariance matrix.

$$\hat{\sigma}_{b_j} = \hat{\sigma}\sqrt{(X^\intercal X)^{-1}{}_{jj}} \tag{6.58}$$

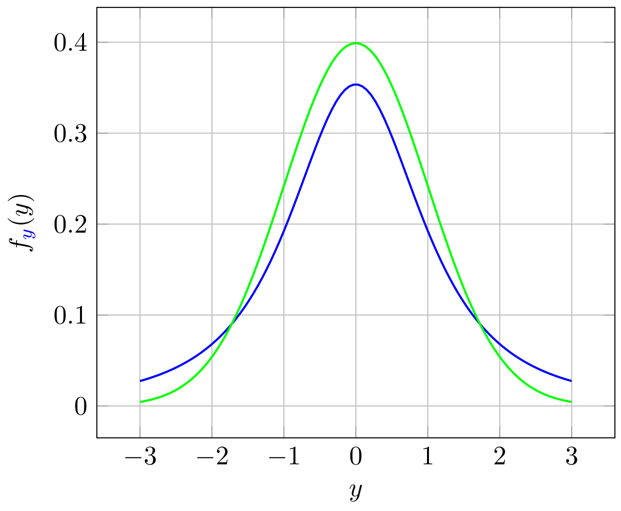
The corresponding t-value is simply the parameter value divided by its standard error, which indicates how many standard deviation units it is away from zero. The farther way from zero the more significant (or more important to the model) the parameter is.

$$t(b_j) = \frac{b_j}{\hat{\sigma}_{b_j}} \tag{6.59}$$

When the error distribution is Normal, then $t(b_j)$ follows the Student's t Distribution. For example, the pdf for the Student's t Distribution with $df = \nu = 2$ Degrees of Freedom is shown in the figure below (the t Distribution approaches the Normal Distribution as ν increases).

$$f_y(y) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1 + \frac{y^2}{\nu}\right)^{-\frac{\nu+1}{2}} = \frac{\Gamma(3/2)}{\sqrt{2\pi}\Gamma(1)}\left(1 + \frac{y^2}{2}\right)^{-3/2} = \frac{1}{2\sqrt{2}}\left(1 + \frac{y^2}{2}\right)^{-3/2} \tag{6.60}$$

pdf for Student's t Distribution (blue) vs. Normal (green)



The corresponding p-value $P(|y| > t)$ measures how significant the t-value is, e.g.,

$$\begin{aligned} F_y(-1.683018) &= 0.0590926 \\ P(|y| > -1.683018) &= 2F_y(-1.683018)) = 0.118185 \quad \text{for} \ \ \nu = df = 12 \end{aligned}$$

Typically, the t-value is only considered significant if is in the tails of the Student's t distribution. The farther out in the tails, the less likely for the parameter to be non-zero (and hence be part of the model) simply by chance. The p-value measures the risk (chance of being wrong) in including parameter b_j and therefore variable x_j in the model.

The predict Method

Finally, a given new data vector **z**, the `predict` method may be used to predict its response value.

```
val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println (s"predict (z) ={mod.predict (z)}")
```

Feature Selection

Feature selection (or Variable Selection) may be carried out by using either `forwrardSel` or `backwardElim`. These methods add or remove one variable at a time. To iteratively add or remove, the following methods may be called.

```
mod.forwardSelAll (cross = false)
mod.backwardElimAll (cross = false)
mod.stepRegressionAll (cross = false)
```

The default criterion for choosing which variable to add/remove is Adjusted R^2. It may be changed via the `idx_q` parameter to the methods (see the `Fit` trait for the possible values for this parameter). Note: The cross-validation is turned off (`cross = false`) due to the small size of the dataset.

The source code for the Texas Temperatures example is a test case in `Regression.scala`.

6.6.14 Regression Class

Class Methods:

```
@param x        the data/input m-by-n matrix
                    (augment with a first column of ones to include intercept in model)
@param y        the response/output m-vector
@param fname_   the feature/variable names (defaults to null)
@param hparam   the hyper-parameters (defaults to Regression.hp)

class Regression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                  hparam: HyperParameter = Regression.hp)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                      b_ : VectorD = b, vifs: VectorD = vif ()): String =
override def predict (x_ : MatrixD): VectorD = x_ * b
override def buildModel (x_cols: MatrixD): Regression =
```

6.6.15 Exercises

1. For Exercise 1 from the last section, compute $A = X^{\intercal} X$ and $\mathbf{z} = X^{\intercal}\mathbf{y}$. Now solve the following linear systems of equations for $\mathbf{b}$.

$$A\mathbf{b} \;=\; \mathbf{z}$$

2. Gradient descent can be used for Multiple Linear Regression as well. For gradient descent, pick a starting point $\mathbf{b}^0$, compute the gradient of the loss function $\nabla\mathcal{L}$ and move $-\eta\nabla\mathcal{L}$ from $\mathbf{b}^0$ where η is the learning rate. Write a Scala program that repeats this for several iterations for the above data. What is happening to the value of the loss function $\mathcal{L}$.

   $$\nabla\mathcal{L} \;=\; -X^{\intercal}(\mathbf{y} - X\mathbf{b})$$

   Substituting $\boldsymbol{\epsilon} = \mathbf{y} - X\mathbf{b}$, allows $\nabla\mathcal{L}$ to be written as

   $$-X^{\intercal}\boldsymbol{\epsilon}$$

   Starting with data matrix `x`, response vector `y` and parameter vector `b`, in ScalaTion, the calculations become

```
    val yp = x * b                                  // y predicted
    val e  = y - yp                                 // error
    val g  = x.𝒯 * e                                // - gradient
    b     += g * eta                                // update parameter b
    val h  = 0.5 * (e dot e)                        // half the sum of squared errors
```

   Unless the dataset is normalized, finding an appropriate learning rate `eta` may be difficult. See the `MatrixTransform` object for details. Do this for the Blood Pressure `Example_BPressure` dataset. Try using another dataset.

3. Consider the relationships between the predictor variables and the response variable in the AutoMPG dataset. This is a well know dataset that is available at multiple websites including the UCI Machine Learning Repository `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`. The response variable is the miles per gallon (`mpg`: continuous) while the predictor variables are `cylinders`: multi-valued discrete, `displacement`: continuous, `horsepower`: continuous, `weight`: continuous, `acceleration`: continuous, `model_year`: multi-valued discrete, `origin`: multi-valued discrete, and `car_name`: string (unique for each instance). Since the `car_name` is unique and obviously not causal, this variable is eliminated, leaving seven predictor variables. First compute the correlations between `mpg` (vector $\mathbf{y}$) and the seven predictor variables (each column vector $\mathbf{x}_{:j}$ in matrix X).

```
    val correlation = y corr x_j
```

   and then plot `mpg` versus each of the predictor variables. The source code for this example is at
   `http://www.cs.uga.edu/~jam/scalation_2.0/src/main/scala/scalation/modeling/Example_AutoMPG.scala` .

   Alternatively, a `.csv` file containing the AutoMPG dataset may be read into a relation called `auto_tab` from which data matrix `x` and response vector `y` may be produced. If the dataset has **missing values**, they may be replaced using a spreadsheet or using the techniques discusses in the Data Preprocessing Chapter.

```
    val auto_tab = Relation (BASE_DIR + "auto-mpg.csv", "auto_mpg", null, -1)
    val (x, y)   = auto_tab.toMatrixDD (1 to 6, 0)
    println (s"x = x")println(s"y =y"
```

4. Apply Regression analysis on the AutoMPG dataset. Compare with results of applying the `NullModel`, `SimplerRegression` and `SimpleRegression`. Try using `SimplerRegression` and `SimpleRegression` with different predictor variables for these models. How does their R^2 values compare to the correlation analysis done in the previous exercise?

5. Examine the collinearity and multi-collinearity of the column vectors in the AutoMPG dataset.

6. For the AutoMPG dataset, repeatedly call the `backwardElim` method to remove the predictor variable that contributes the least to the model. Show how the various quality of fit (QoF) measures change as variables are eliminated. Do the same for the `forwardSel` method. Using $\bar{R}^2$, select the best models from the forward and backward approaches. Are they the same?

7. Compare model assessment and model validation. Compute sse, mse and R^2 for the full and best AutoMPG models trained on the entire data set. Compare this with the results of Leave-One-Out, 5-fold Cross-Validation and 10-fold Cross-Validation.

8. The variance of the estimate of parameter b_j may be estimated as follows:

$$\hat{\sigma}^2(b_j) \;=\; \frac{mse}{k\,\hat{\sigma}^2(x_j)} \cdot \text{vif}(b_j)$$

   Derive this formula. The standard error is the square root of this value. Use the estimate for b_j and its standard error to compute a t-value and p-value for the estimate. Run the AutoMPG model and explain these values produced by the `summary` method.

9. **Singular Value Decomposition Technique**. In cases where the rank of the data/input matrix X is not full or its multi-collinearity is high, a useful technique to solve for the parameters of the model is Singular Value Decomposition (SVD). Based on the derivation given in `http://www.ime.unicamp.br/~marianar/MI602/material%20extra/svd-regression-analysis.pdf`, we start with the equation estimating $\mathbf{y}$ as the product of the data matrix X and the parameter vector $\mathbf{b}$.

$$\mathbf{y} \;=\; X\mathbf{b}$$

   We then perform a singular value decomposition on the m-by-n matrix X

$$X \;=\; U\Sigma V^{\intercal}$$

   where in the full-rank case, U is an m-by-n orthogonal matrix, Σ is an n-by-n diagonal matrix of singular values, and $V^{\intercal}$ is an n-by-n orthogonal matrix The $r = rank(A)$ equals the number of nonzero singular values in Σ, so in general, U is m-by-r, Σ is r-by-r, and $V^{\intercal}$ is r-by-n. The singular values are the square roots of the nonzero eigenvalues of $X^{\intercal}X$. Substituting for X yields

$$\mathbf{y} \;=\; U\Sigma V^{\intercal}\mathbf{b}$$

Defining $\mathbf{d} = \Sigma V^{\intercal}\mathbf{b}$, we may write

$$\mathbf{y} \;=\; U\mathbf{d}$$

This can be viewed as a estimating equation where X is replaced with U and $\mathbf{b}$ is replaced with $\mathbf{d}$. Consequently, a least squares solution for the alternate parameter vector $\mathbf{d}$ is given by

$$\mathbf{d} \;=\; (U^{\intercal}U)^{-1}U^{\intercal}\mathbf{y}$$

Since $U^{\intercal}U = I$, this reduces to

$$\mathbf{d} \;=\; U^{\intercal}\mathbf{y}$$

If $\text{rank}(A) = n$ (full-rank), then the conventional parameters $\mathbf{b}$ may be obtained as follows:

$$\mathbf{b} \;=\; V\Sigma^{-1}\mathbf{d}$$

where Σ^{-1} is a diagonal matrix where elements on the main diagonal are the reciprocals of the singular values.

10. **Improve Stepwise Regression**. Write ScalaTion code to improve the `stepRegressionAll` method by implementing the swapping operation. Then redo exercise 6 using all three: Forward Selection, Backward Elimination, and Stepwise Regression with all four criteria: R^2, $\bar{R}^2$, R^2_{cv}, and AIC. Plot the curve for each criterion, determine the best number of variables and what these variables are. Compare the four criteria.

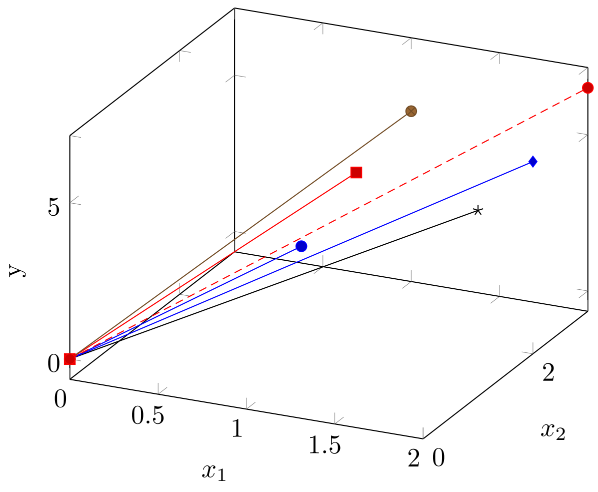
    As part of a larger project compare this form of feature selection with that provided by Ridge Regression and Lasso Regression. See the next two sections.

    Now add features including quadratic terms, cubic terms, and dummy variables to the model using `SymbolicRegression.quadratic`, `SymbolicRegression.cubic`, and `RegressionCat`. See the subsequent sections.

    In addition to the `AutoMPG` dataset, use the `Concrete` dataset and three more datasets from UCI Machine Learning Repository. The UCI datasets should have more instances (m) and variables (n) than the first two datasets. The testing should also be done in R or Python.

11. **Regression as Projection**. Consider the following six vectors/points in 3D space where the response variable y is modeled as a linear function of predictor variables x_1 and x_2.

```
//                              x1 x2  y
    val xy = MatrixD ((6, 3), 1, 1, 2.8,
                              1, 2, 4.2,
                              1, 3, 4.8,
                              2, 1, 5.3,
                              2, 2, 5.5
                              2, 3, 6.5)
```



Consider a regression model equation with no intercept.

$$y \ = \ b_0 x_1 + b_1 x_2 + \epsilon$$

Determine the plane (response surface) that these six points are projected onto.

$$\hat{y} \ = \ b_0 x_1 + b_1 x_2$$

For this problem, the number of instances $m = 6$ and the number of parameters/predictor variables $n = 2$. Determine the number of Degrees of Freedom for the model df_m and the number of Degrees of Freedom for the residuals/errors df.

12. Given a data matrix $X \in \mathbb{R}^{m \times 2}$ and response vector $\mathbf{y} \in \mathbb{R}^m$ where $X = [\mathbf{1}, \mathbf{x}]$, compute $X^\intercal X$ and $X^\intercal \mathbf{y}$. Use these to set up an augmented matrix and then apply LU Factorization to make it upper triangular. Solve for the parameters b_0 and b_1 symbolically. Simply to reproduce formulas for b_0 and b_1 for Simple Regression.

13. Recall that $\hat{\mathbf{y}} = H\mathbf{y}$ where the hat matrix is $X(X^\intercal X)^{-1}X^\intercal$. The *leverage* of point i is defined to be h_{ii}.

$$h_{ii} \ = \ \mathbf{x}_i^\intercal (X^\intercal X)^{-1} \mathbf{x}_i \qquad (6.61)$$

The main diagonal of the hat matrix gives the leverage for each of the points. Points with high leverage are those above a threshold such as

$$h_{ii} \ \geq \ \frac{2\,\mathrm{tr}(H)}{m} \qquad (6.62)$$

Note, that the trace $\mathrm{tr}(H) = \mathrm{rank}(H) = \mathrm{rank}(X)$ will equal n when X has full rank. List the high leverage points for the `Example_AutoMPG` dataset.

14. Points that are influential in determining values for model coefficients/parameters combine high leverage with large residuals. Measures of influence include Cook's Distance, DFFITS, and DFBETAS [34] and see `http://home.iitk.ac.in/~shalab/regression/Chapter6-Regression-Diagnostic%20for%20Leverage%20and%20Influence.pdf`. These measures can also be useful in detecting potential outliers. Compute these measures for the `Example_AutoMPG` dataset.

15. The best two predictor variables for AutoMPG are `weight` and `modelyear` and with the weight given in units of 1000 pounds, the prediction equation for the `Regression` model (with intercept) is

$$\hat{y} \;=\; -14.3473 - 6.63208x_1 + 0.757318x_2$$

The corresponding hyperplane is show in Figure 6.4

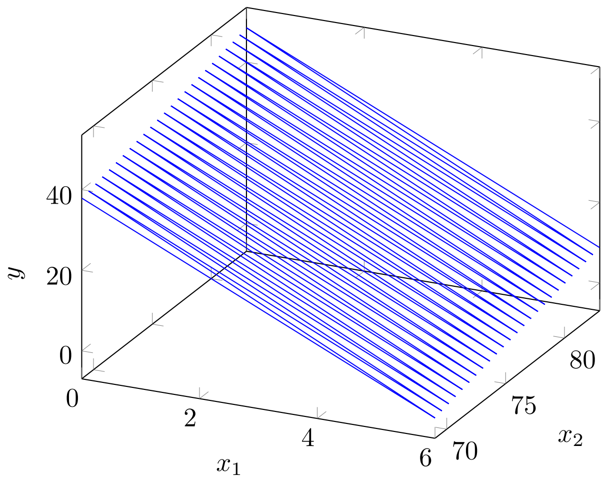


Figure 6.4: Hyperplane: $\hat{y} = -14.3473 - 6.63208x_1 + 0.757318x_2$

Make a plot of the hyperplane for the second best combination of features. Compare the QoF of these two models and explain how the feature combinations affect the response variable (`mpg`).

16. State and explain the conditions required for the Ordinary Least Squares (OLS) estimate of parameter vector **b** for multiple linear regression to be B.L.U.E. See the Gauss-Markov Theorem. B.L.U.E. stands for Best Linear Unbiased Estimator.

6.6.16 Further Reading

1. Introduction to Linear Regression Analysis, 5th Edition [127]

2. Regression: Linear Models in Statistics [18]

6.7 Ridge Regression

The `RidgeRegression` class supports multiple linear ridge regression. As with `Regression`, the predictor variables $\mathbf{x}$ are multi-dimensional $[x_1, \dots, x_k]$, as are the parameters $\mathbf{b} = [b_1, \dots, b_k]$. Ridge regression adds a penalty based on the ℓ^2 norm of the parameters $\mathbf{b}$ to reduce the chance of them taking on large values that may lead to less robust models.

The penalty holds down the values of the parameters and this may result in several advantages: (1) better out-of-sample (e.g., cross-validation) quality of fit, (2) reduced impact of multi-collinearity, (3) turn singular matrices, non-singular, and to a limited extent (4) eliminate features/predictor variables from the model.

The penalty is not to be included on the intercept parameter b_0, as this would shift predictions in a way that would adversely affect the quality of the model. See the exercise on scale invariance.

6.7.1 Model Equation

Centering of the data allows the intercept to be removed from the model. The combined centering on both the predictor variables and the response variable takes care of the intercept, so it is not included in the model. Thus, the goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_1 x_1 \,+\, \cdots + b_k x_k \,+\, \epsilon} \tag{6.63}$$

where ϵ represents the residuals (the part not explained by the model).

6.7.2 Training

Centering the dataset $(X, \mathbf{y})$ has the following effects: First, when the X matrix is centered, the intercept $b_0 = \mu_y$. Second, when $\mathbf{y}$ is centered, μ_y becomes zero, implying $b_0 = 0$. To rescale back to the original response values, μ_y can be added back during prediction. Therefore, both the data/input matrix X and the response/output vector $\mathbf{y}$ should be *centered* (zero mean).

$$X^{(c)} \;=\; X - \boldsymbol{\mu}_x \qquad \text{subtract predictor column means} \tag{6.64}$$

$$\mathbf{y}^{(c)} \;=\; \mathbf{y} - \mu_y \qquad \text{subtract response mean} \tag{6.65}$$

The regularization of the model adds an ℓ^2-penalty on the parameters $\mathbf{b}$. The objective function to minimize is now the loss function $\mathcal{L}(\mathbf{b}) = \frac{1}{2} sse$ plus the ℓ^2-penalty.

$$\boxed{f_{obj} \;=\; \mathcal{L}(\mathbf{b}) \,+\, \frac{1}{2}\lambda \,\|\mathbf{b}\|^2 \;=\; \frac{1}{2}\boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} \,+\, \frac{1}{2}\lambda \,\mathbf{b} \cdot \mathbf{b}} \tag{6.66}$$

where λ is the shrinkage parameter. A large value for λ will drive the parameters $\mathbf{b}$ toward zero, while a small value can help stabilize the model (e.g., for nearly singular matrices or high multi-collinearity).

$$f_{obj} \;=\; \frac{1}{2}(\mathbf{y} - X\mathbf{b}) \cdot (\mathbf{y} - X\mathbf{b}) \,+\, \frac{1}{2}\lambda \,\mathbf{b} \cdot \mathbf{b} \tag{6.67}$$

6.7.3 Optimization

Fortunately, the quadratic nature of the penalty function allows it to be combined easily with the quadratic error terms, so that matrix factorization can still be used for finding optimal values for parameters.

Taking the gradient of the objective function f_{obj} with respect to $\mathbf{b}$ and then setting it equal to zero yields

$$-X^{\intercal}(\mathbf{y} - X\mathbf{b}) + \lambda\mathbf{b} \;=\; \mathbf{0} \tag{6.68}$$

Recall the first term of the gradient was derived in the Regression section. See the exercises below for deriving the last term of the gradient. Multiplying out gives,

$$\begin{aligned} -X^{\intercal}\mathbf{y} + (X^{\intercal}X)\mathbf{b} + \lambda\mathbf{b} \;&=\; \mathbf{0} \\ (X^{\intercal}X)\mathbf{b} + \lambda\mathbf{b} \;&=\; X^{\intercal}\mathbf{y} \end{aligned}$$

Since $\lambda\mathbf{b} = \lambda I\mathbf{b}$ where I is the n-by-n identity matrix, we may write

$$\boxed{(X^{\intercal}X + \lambda I)\mathbf{b} \;=\; X^{\intercal}\mathbf{y}} \tag{6.69}$$

Matrix factorization may now be used to solve for the parameters $\mathbf{b}$ in the modified Normal Equations. For example, use of matrix inversion yields,

$$\mathbf{b} \;=\; (X^{\intercal}X + \lambda I)^{-1}X^{\intercal}\mathbf{y} \tag{6.70}$$

For Cholesky factorization, one may compute $X^{\intercal}X$ and simply add λ to each of the diagonal elements (i.e, along the ridge). QR and SVD factorizations require similar, but slightly more complicated, modifications. Note, use of SVD can improve the efficiency of searching for an optimal value for λ [71, 196].

6.7.4 Centering

Before creating a `RidgeRegression` model, the X data matrix and the $\mathbf{y}$ response vector should be centered. This is accomplished by subtracting the means (vector of column means for X and a mean value for $\mathbf{y}$).

```
    val mu_x = x.mean                                   // column-wise mean of x
    val mu_y = y.mean                                   // mean of y
    val x_c  = x - mu_x                                 // centered x (column-wise)
    val y_c  = y - mu_y                                 // centered y
```

The centered matrix `x_c` and center vector `y_c` are then passed into the `RidgeRegression` constructor.

```
    val mod = new RidgeRegression (x_c, y_c)
    mod.trainNtest ()
```

Now, when making predictions, the new data vector $\mathbf{z}$ needs to be centered by subtracting `mu_x`. Then the `predict` method is called, after which the mean of $\mathbf{y}$ is added.

```
    val z_c = z - mu_x                                  // center z first
    yp = mod.predict (z_c) + mu_y                       // predict z_c and add y's mean
    println (s"predict (z) = yp")
```

6.7.5 The λ Hyper-parameter

The value for λ can be user specified (typically a small value) or chosen by a method like `findLambda`. It finds a roughly optimal value for the shrinkage parameter λ based on the cross-validated sum of squared errors `sse_cv`. The search starts with the low default value for λ and then doubles it with each iteration, returning the minimizing λ and its corresponding cross-validated `sse`. A more precise search could be used to provide a better value for λ.

```
def findLambda: (Double, Double) =
    var l      = lambda                          // start with a small default value
    var l_best = l
    var sse    = Double.MaxValue
    for i <- 0 to 20 do
        RidgeRegression.hp("lambda") = l
        val mod = new RidgeRegression (x, y)
        val stats = mod.crossValidate ()
        val sse2 = stats(QoF.sse.ordinal).mean
        banner (s"RidgeRegession with lambda = $mod.lambda_} has sse = $sse2")
        if sse2 < sse then { sse = sse2; l_best = l }
        l *= 2
    end for
    (l_best, sse)                                // best lambda and its sse_cv
end findLambda
```

6.7.6 Comparing `RidgeRegression` with `Regression`

This subsection compares the results of `RidgeRegression` with those of `Regression` by examining the estimated parameter vectors, the quality of fit, predictions made, and comparing the summary information.

```
// 5 data points:          x_0     x_1
val x = MatrixD ((5, 2), 36.0,   66.0,                  // 5-by-2 matrix data matrix
                         37.0,   68.0,
                         47.0,   64.0,
                         32.0,   53.0,
                          1.0,  101.0)
val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0)      // 5-dim response vector
```

First, create a Regression model with an intercept and produce a summary.

```
banner ("Regression")
val ox = VectorD.one (y.dim) +^: x                      // prepend column of all 1's
val rg = new Regression (ox, y)                         // create a Regression model
rg.trainNtest ()()                                      // train and test the model
```

Second, create a RidgeRegression model using the centered data

```
banner ("RidgeRegression")
val mu_x = x.mean                                       // column-wise mean of x
val mu_y = y.mean                                       // mean of y
val x_c  = x - mu_x                                     // centered x (column-wise)
val y_c  = y - mu_y                                     // centered y
val mod  = new RidgeRegression (x_c, y_c)               // create a Ridge Regression
mod.trainNtest ()()                                     // train and test the model
```

Third, predict a value for new input vector z using each model.

```
    banner ("Make Predictions")
    val z    = VectorD (20.0, 80.0)                          // new instance to predict
    val _1z  = VectorD.++ (1.0, z)                           // prepend 1 to z
    val z_c  = z - mu_x                                      // center z
    println (s"rg.predict  (z) ={rg.predict (_1z)}")          // predict using _1z
    println (s"mod.predict (z) ={mod.predict (z_c) + mu_y}")  // predict using z_c and add
     y's mean
```

The summary information for `Regression` is shown below.

```
    println (rg.summary ())
```

```
SUMMARY
    Parameters/Coefficients:
    Var      Estimate      Std. Error         t value       Pr(>|t|)            VIF
----------------------------------------------------------------------------------
    x0    -281.426985     835.349154       -0.336897       0.768262              NA
    x1      -7.611030       8.722908       -0.872534       0.474922        3.653976
    x2      19.010291       8.423716        2.256758       0.152633        3.653976

    Residual standard error: 159.206002 on 2.0 degrees of freedom
    Multiple R-squared:  0.943907,     Adjusted R-squared:  0.887815
    F-statistic: 16.827632701228243 on 2.0 and 2.0 DF,  p-value: 0.05609269703717268
----------------------------------------------------------------------------------
```

The summary information for `RidgeRegression` is shown below.

```
    println (mod.summary ())
```

```
SUMMARY
    Parameters/Coefficients:
    Var      Estimate      Std. Error         t value       Pr(>|t|)            VIF
----------------------------------------------------------------------------------
    x0      -7.611271       8.722908       -0.872561       0.474910              NA
    x1      19.009947       8.423716        2.256717       0.152638        3.653976

    Residual standard error: 159.206002 on 2.0 degrees of freedom
    Multiple R-squared:  0.943907,     Adjusted R-squared:  0.887815
    F-statistic: 16.827632684676626 on 2.0 and 2.0 DF,  p-value: 0.056092697089250576
----------------------------------------------------------------------------------
```

Notice there is very little difference between the two models. Try increasing the value of the shrinkage hyper-parameter λ beyond its default value of 0.01. This example can be run as follows:

```
    $ sbt
    sbt> runMain scalation.modeling.ridgeRegressionTest
```

Automatic Centering

ScalaTion provides factory methods, `apply` and `center`, in the `RigdgeRgression` companion object that center the data for the user.

```scala
//  val mod = RidgeRegression (xy, fname)              // apply takes a combined matrix xy
    val mod = RidgeRegression.center (x, y, fname)     // center takes a matrix x and
    vector y
    mod.trainNTest ()()
    val yp = mod.predict (z - x.mean) + y.mean
```

The user must still center any vectors passed into the `predict` method and add back the response mean at the end, e.g., pass `z - x.mean` and add back `y.mean`.

Note, care should be taken regarding `x.mean` and `y.mean` when preforming `validation` or `crossValidation`. The means for the full, training and testing sets may differ.

6.7.7 `RidgeRegression` Class

Class Methods:

```scala
    @param x        the centered data/input m-by-n matrix NOT augmented with a column of 1s
    @param y        the centered response/output m-vector
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the shrinkage hyper-parameter, lambda (0 => OLS) in the penalty term
                    'lambda * b dot b'

    class RidgeRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                           hparam: HyperParameter = RidgeRegression.hp)
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2, df = x.dim - x.dim2 - 1):

    def lambda_ : Double = lambda
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    def findLambda: (Double, Double) =
    override def predict (x_ : MatrixD): VectorD = x_ * b
    override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                          b_ : VectorD = b, vifs: VectorD = vif ()): String =
    override def buildModel (x_cols: MatrixD): RidgeRegression =
```

6.7.8 Exercises

1. Based on the example given in this section, try increasing the value of the hyper-parameter λ and examine its effect on the parameter vector $\mathbf{b}$, the quality of fit and predictions made.

```scala
    import RidgeRegression.hp

    println (s"hp = $hp")
    val hp2 = hp.updateReturn ("lambda", 1.0)
    println (s"hp2 = $hp2")
```

Alternatively,

```scala
    hp("lambda") = 1.0
```

See the `HyperParameter` class in the `scalation.modeling` package for details.

2. For the AutoMPG dataset, use the `findLambda` method find a value for λ that roughly minimizes out-of-sample `sse_cv` based on using the `crossValidate` method. Plot `sse_cv` vs. λ.

3. Why is it important to center (zero mean) both the data matrix X and the response vector $\mathbf{y}$? What is *scale invariance* and how does it relate to centering the data?

4. The Degrees of Freedom (DoF) used in ScalaTion's `RidgeRegression` class is approximate. As the shrinkage parameter λ increases the effective DoF (eDoF) should be used instead. A general definition of effective DoF is the trace tr of the hat matrix $H = X(X^{\intercal}X + \lambda I)^{-1}X^{\intercal}$

$$df_m^{\text{eff}} \;=\; \text{tr}(H)$$

Read [86] and explain the difference between DoF and effective DoF (eDoF) for Ridge Regression.

5. A matrix that is close to singularity is said to be ill-conditioned. The condition number κ of a matrix A (e.g., $A = X^{\intercal}X$) is defined as follows:

$$\kappa \;=\; \|A\|\,\|A^{-1}\| \;\geq\; 1$$

When κ becomes large the matrix is considered to be ill-conditioned. In such cases, it is recommended to use QR or SVD Factorization for Least-Squares Regression [39]. Compute the condition number for of $X^{\intercal}X$ for various datasets.

6. For the last term of the gradient of the objective function, show that

$$\frac{\partial}{\partial b_j}\frac{\lambda}{2}\mathbf{b}\cdot\mathbf{b} \;=\; \frac{\lambda}{2}\sum_i b_i^2 \;=\; \lambda b_j$$

Put these together to show that $\nabla\frac{\lambda}{2}\mathbf{b}\cdot\mathbf{b} \;=\; \lambda\mathbf{b}$

7. For over-parameterized (or under-determined) regression where the $n > m$, (number of parameters > number of instances) it is common to seek a min-norm solution.

$$min_{\mathbf{b}}\{\|\mathbf{b}\|_2^2 \mid \mathbf{y} = X\mathbf{b}\}$$

Use `http://people.csail.mit.edu/bkph/articles/Pseudo_Inverse.pdf` to derive a solution for the parameters $\mathbf{b}$.

$$\mathbf{b} \;=\; X^{\intercal}(XX^{\intercal})^{-1}\mathbf{y}$$

Compare the use of regression and ridge regression of such problems.

8. Compare different algorithms for finding a suitable value for the shrinkage parameter λ.

Hint: see *Lecture Notes on Ridge Regression* - `https://arxiv.org/pdf/1509.09169.pdf` - [196].

6.8 Lasso Regression

The `LassoRegression` class supports multiple linear regression using the Least absolute shrinkage and selection operator (Lasso) that constrains the values of the $\mathbf{b}$ parameters and effectively sets those with low impact to zero (thereby deselecting such variables/features). Rather than using an ℓ^2-penalty (Euclidean norm) like `RidgeRegression`, it uses and an ℓ^1-penalty (Manhattan norm). In `RidgeRegression` when b_j approaches zero, b_j^2 becomes very small and has little effect on the penalty. For `LassoRegression`, the effect based on $|b_j|$ will be larger, so it is more likely to set parameters to zero. See section 6.2.2 in [85] for a more detailed explanation on how `LassoRegression` can eliminate a variable/feature by setting its parameter/coefficient to zero.

6.8.1 Model Equation

As with `Regression`, the goal is to fit the parameter vector $\mathbf{b} \in \mathbb{R}^n$ $(k = n - 1)$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... + b_k x_k + \epsilon} \tag{6.71}$$

where ϵ represents the residuals (the part not explained by the model). See the exercise that considers whether to include the intercept b_0 in the shrinkage.

6.8.2 Training

The regularization of the model adds an ℓ^1-penalty on the parameters $\mathbf{b}$. The objective function to minimize is now the loss function $\mathcal{L}(\mathbf{b}) = \frac{1}{2} sse$ plus the penalty.

$$\boxed{f_{obj} \;=\; \frac{1}{2}\, sse \,+\, \lambda\, \|\mathbf{b}\|_1 \;=\; \frac{1}{2}\, \|\boldsymbol{\epsilon}\|_2^2 \,+\, \lambda\, \|\mathbf{b}\|_1} \tag{6.72}$$

where λ is the shrinkage parameter. Substituting $\boldsymbol{\epsilon} = \mathbf{y} - X\mathbf{b}$ yields,

$$f_{obj} \;=\; \frac{1}{2}\, \|\mathbf{y} - X\mathbf{b}\|_2^2 \,+\, \lambda\, \|\mathbf{b}\|_1 \tag{6.73}$$

Replacing the norms with dot products gives,

$$f_{obj} \;=\; \frac{1}{2}\, (\mathbf{y} - X\mathbf{b}) \cdot (\mathbf{y} - X\mathbf{b}) \,+\, \lambda\, \mathbf{1} \cdot |\mathbf{b}| \tag{6.74}$$

Although similar to the ℓ^2-penalty used in Ridge Regression, it may often be more effective. Still, the ℓ^1-penalty for Lasso has a disadvantage that the absolute values in the ℓ^1 norm make the objective function non-differentiable.

$$\lambda\, \mathbf{1} \cdot |\mathbf{b}| \;=\; \lambda \sum_{j=0}^{k} |b_j| \tag{6.75}$$

Therefore, the straightforward strategy of setting the gradient equal to zero to develop appropriate modified Normal Equations that allow the parameters to be determined by matrix factorization will no longer works. Instead, the objective function needs to be minimized using a search based optimization algorithm.

6.8.3 Optimization Strategies

There are multiple optimization algorithms that can be applied for parameter estimation in Lasso Regression.

Coordinate Descent

Coordinate Descent attempts to optimize one variable/feature at a time (repeated one dimensional optimization). For normalized data the following algorithm has been shown to work: `https://xavierbourretsicotte.github.io/lasso_implementation.html`.

Alternating Direction Method of Multipliers

ScalaTion uses the Alternating Direction Method of Multipliers (ADMM) [22] algorithm to optimize the **b** parameter vector. The algorithm for using ADMM for Lasso Regression is outlined in section 6.4 of Boyd [22] (`https://stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf`). Optimization problems in ADMM form separate the objective function into two parts f and g.

$$\min\ f(\mathbf{b})\ +\ g(\mathbf{z})\ \ \text{subject to}\ \ \mathbf{b} - \mathbf{z} = \mathbf{0} \tag{6.76}$$

For Lasso Regression, the f function will capture the loss function ($\frac{1}{2}\,sse$), while the g function will capture the ℓ^1 regularization, i.e.,

$$f(\mathbf{b}) = \frac{1}{2}\,\|\mathbf{y} - X\mathbf{b}\|_2^2\ ,\quad g(\mathbf{z}) = \lambda\,\|\mathbf{z}\|_1 \tag{6.77}$$

Introducing **z** allows the functions to be separated, while the constraint keeps **z** and **b** close. Therefore, the iterative step in the ADMM optimization algorithm becomes

$$\begin{aligned}\mathbf{b} &= (X^{\intercal}X + \rho I)^{-1}(X^{\intercal}\mathbf{y} + \rho(\mathbf{z} - \mathbf{u}))\\ \mathbf{z} &= S_{\lambda/\rho}(\mathbf{b} + \mathbf{u})\\ \mathbf{u} &= \mathbf{u} + \mathbf{b} - \mathbf{z}\end{aligned}$$

where **u** is the vector of Lagrange multipliers and S_λ is the soft thresholding function.

$$S_\lambda(\zeta)\ =\ \text{sign}(\zeta)\,(|\zeta| - \lambda)_+ \tag{6.78}$$

Note $(a)_+ = \max(a, 0)$.

The details of ADMM for Lasso Regression are left as an exercise. In addtion to Boyd, see [73] and see `scalation.optimization.LassoAdmm` for coding details.

6.8.4 The λ Hyper-parameter

The shrinkage parameter λ can be tuned to control feature selection. The larger the value of λ, the more features (predictor variables) whose parameters/coefficients will be set to zero. The `findLambda` method may be used to find a value `lambda` that improves the cross-validated sum of squared errors `sse_cv`.

```
    def findLambda: (Double, Double) =
        var l      = lambda
        var l_best = l
        var sse    = Double.MaxValue
        for i <- 0 to 20 do
```

```
            LassoRegression.hp("lambda") = l
            val mod   = new LassoRegression (x, y)
            val stats = mod.crossValidate ()
            val sse2  = stats(QoF.sse.ordinal).mean
            banner (s"LassoRegession with lambda = $mod.lambda_} has sse = $sse2")
            if sse2 < sse then
                sse = sse2; l_best = l
            end if
            Fit.showQofStatTable (stats)
            l *= 2
        end for
        (l_best, sse)
    end findLambda
```

As the default value for the shrinkage/penalty parameter λ is very small, the optimal solution will be close to the Ordinary Least Squares (OLS) solution shown in green at $\mathbf{b} = [b_1, b_2] = [3, 1]$ in Figure 6.5. Increasing the penalty parameter will pull the optimal $\mathbf{b}$ towards the origin. At any given point in the plane, the objective function is the sum of the loss function $\mathcal{L}(\mathbf{b})$ and the penalty function $p(\mathbf{b})$. The contours in blue show points of equal height for the penalty function, while those in black show the same for the loss function. Suppose for some λ the point [2, 0] is this penalized optimum. This would mean that moving toward the origin would be non-productive, as the increase in the loss would exceed the drop in the penalty. On the other hand, moving toward $[3, 1]$ would be non-productive as the increase in the penalty would exceed the drop in the loss. Notice in this case, that the penalty has pulled the b_1 parameter to zero (an example of feature selection). Ridge regression will be less likely to pull a parameter to zero, as its contours are circles rather than diamonds. Lasso regression's contours have sharp points on the axis which thereby increase the chance of intersecting a loss contour on an axis.

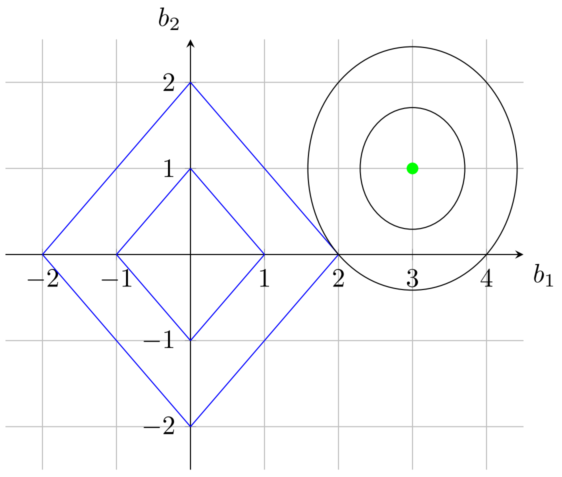


Figure 6.5: Contour Curves for Lasso Regression

6.8.5 Regularized and Robust Regression

Regularized and Robust Regression are useful in many cases including high-dimensional data, correlated data, non-normal data and data with outliers [70]. These techniques work by adding a ℓ^1 and/or ℓ^2-penalty terms to shrink the parameters and/or changing from an ℓ^2 to ℓ^1 loss function. Modeling techniques include Ridge, Lasso, Elastic Nets, Least Absolute Deviation (LAD) and Adaptive LAD [70].

6.8.6 `LassoRegression` Class

Class Methods:

```
    @param x        the data/input m-by-n matrix
    @param y        the response/output m-vector
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the shrinkage hyper-parameter, lambda (0 => OLS) in the penalty term
                    'lambda * b dot b'

    class LassoRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                           hparam: HyperParameter = LassoRegression.hp)
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def lambda_ : Double = lambda
    def findLambda: (Double, Double) =
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                          b_ : VectorD = b, vifs: VectorD = vif ()): String =
    override def buildModel (x_cols: MatrixD): LassoRegression =
```

6.8.7 Exercises

1. Compare the results of `LassoRegression` with those of `Regression` and `RidgeRegression`. Examine the parameter vectors, quality of fit and predictions made.

```
    // 5 data points:         one    x_0     x_1
    val x = MatrixD ((5, 3), 1.0, 36.0,   66.0,               // 5-by-3 matrix
                             1.0, 37.0,   68.0,
                             1.0, 47.0,   64.0,
                             1.0, 32.0,   53.0,
                             1.0,  1.0,  101.0)
    val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0)
    val z = VectorD (1.0, 20.0, 80.0)

// Create a LassoRegression model

    val lrg = new LassoRegression (x, y)

// Predict a value for new input vector z using each model.
```

2. Based on the last exercise, try increasing the value of the hyper-parameter λ and examine its effect on the parameter vector **b**, the quality of fit and predictions made.

```
    import LassoRegression.hp

    println (s"hp = $hp")
    val hp2 = hp.updateReturn ("lambda", 1.0)
    println (s"hp2 = $hp2")
```

3. Using the above dataset and the AutoMPG dataset, determine the effects of (a) **centering** the data ($\mu = 0$), (b) **standardizing** the data ($\mu = 0, \sigma = 1$).

```
    import MatrixTransforms._

    val x_n = normalize (x, (mu_x, sig_x))
    val y_n = y.standardize
```

4. Explain how the Coordinate Descent Optimization Algorithm works for Lasso Regression. See `https://xavierbourretsicotte.github.io/lasso_implementation.html`.

5. Explain how the ADMM Optimization Algorithm works for Lasso Regression. See `https://stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf`.

6. Compare `LassoRegression` the with `Regression` that uses forward selection or backward elimination for feature selection. What are the advantages and disadvantages of each for feature selection.

7. Compare `LassoRegression` the with `Regression` on the AutoMPG dataset. Specifically, compare the quality of fit measures as well as how well feature selection works.

8. Show that the contour curves for the Simple Regression loss function $\mathcal{L}(b_0, b_1)$ are ellipses. The general equation of an ellipse centered at (h, k) is

$$A(x-h)^2 + B(x-h)(y-k) + C(y-k)^2 \ = \ 1$$

   where $A, B > 0$ and $B^2 < 4AC$.

9. **Elastic Nets** combine both ℓ^2 and ℓ^1 penalties to try to combine the best features of both `RidgeRegression` and `LassoRegression`. Elastic Nets naturally includes two shrinkage parameters, λ_1 and λ_2. Is the additional complexity worth the benefits?

10. Regularization using Lasso has the nice property of being able to force parameters/coefficients to zero, but this may require a large shrinkage hyper-parameter λ that shrinks non-zero coefficients more than desired. Newer regularization techniques reduce the shrinkage effect compared to Lasso, by having a penalty profile that matches Lasso for small coefficients, but is below Lasso for large coefficient values. Make of plot of the penalty profiles for Lasso, Smoothly Clipped Absolute Deviations (SCAD) and Mimimax Concave Penalty (MCP).

6.8.8 Further Reading

1. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers [22]
2. The Application of Alternating Direction Method of Multipliers on ℓ^1-norms Problems [73]
3. Feature Selection Using LASSO [49]

6.9 Quadratic Regression

The `quadratic` method in the `SymbolicRegression` object adds quadratic terms into the model. It can often be the case that the response variable y will have a nonlinear relationship with one more of the predictor variable x_j. The simplest such nonlinear relationship is a quadratic relationship. Looking at a plot of y vs. x_j, it may be evident that a bending curve will fit the data much better than a straight line. For example, a particle under constant acceleration will have a position that changes quadratically with time.

When there is only one predictor variable x, the response y is modeled as a quadratic function of x (forming a parabola).

$$y \;=\; b_0 + b_1 x + b_2 x^2 + \epsilon \tag{6.79}$$

The `quadratic` method achieves this simply by expanding the data matrix. From the dataset (initial data matrix), all columns will have another column added that contains the values of the original column squared. It is important that the **initial data matrix has no intercept**. The expansion will optionally add an intercept column (column of all ones). Since $1^2 = 1$, the ones columns and its square will be perfectly collinear and make the matrix singular, if the user includes a ones column.

6.9.1 Model Equation

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the quadratic model/regression equation is the following:

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x}' + \epsilon \;=\; b_0 + b_1 x_1 + b_2 x_2 + b_3 x_1^2 + b_4 x_2^2 + \epsilon} \tag{6.80}$$

where $\mathbf{x}' = [1, x_1, x_2, x_1^2, x_2^2]$, $\mathbf{b} = [b_0, b_1, b_2, b_3, b_4]$, and ϵ represents the residuals (the part not explained by the model).

The number of terms (nt) in the model increases linearly with the dimensionality of the space (n) according to the following formula:

$$nt \;=\; 2n + 1 \qquad e.g., \quad nt \;=\; 5 \;\; \text{for} \;\; n = 2 \tag{6.81}$$

Each column in the initial data matrix is expanded into two in the expanded data matrix and an intercept column is optionally added.

6.9.2 Comparison of quadratic and Regression

This subsection compares results and Quality of Fit of the `quadratic` method in the `SymbolicRegression` object to the `Regression` class. Factors in choosing between the two include the accuracy of the model and information provided in by the `summary` method (e.g., p-values and VIF).

```
    // 8 data points:         x   y
    val xy = MatrixD ((8, 2), 1,  2,                  // 8-by-2 combined matrix
                              2,  5,
                              3, 10,
                              4, 15,
                              5, 20,
                              6, 30,
                              7, 50,
                              8, 60)
    val (x, y) = (xy.not(?, 1), xy(?, 1))              // x is first column, y is last column
    val ox = VectorD.one (xy.dim) +^: x                 // prepend a column of all ones

    val rg = new Regression (ox, y)                     // create a regression model
    rg.trainNtest ()()                                  // train and test the model
    println (rg.summary ())                             // show summary

    val qrg = SymbolicRegression.quadratic (x, y)       // create a quadratic regression model
    qrg.trainNtest ()()                                 // train and test the model
    println (qrg.summary ())                            // show summary
```

Now compare their summary results. The summary results for the `Regression` model are shown below:

```
SUMMARY
    Parameters/Coefficients:
    Var      Estimate     Std. Error         t value       Pr(>|t|)            VIF
----------------------------------------------------------------------------------
    x0     -13.285714       5.154583       -2.577457       0.041913             NA
    x1       8.285714       1.020760        8.117205       0.000188       1.000000

    Residual standard error: 6.615278 on 6.0 degrees of freedom
    Multiple R-squared:  0.916538,     Adjusted R-squared:  0.902628
    F-statistic: 65.88900979325355 on 1.0 and 6.0 DF,  p-value: 1.8767258045970792E-4
----------------------------------------------------------------------------------
```

The summary results for the `SymbolicRegression.quadratic` model are given here:

```
SUMMARY
    Parameters/Coefficients:
    Var      Estimate     Std. Error         t value       Pr(>|t|)            VIF
----------------------------------------------------------------------------------
    x0       4.035714       3.873763        1.041807       0.345231             NA
    x1      -2.107143       1.975007       -1.066904       0.334798      21.250000
    x2       1.154762       0.214220        5.390553       0.002965      21.250000

    Residual standard error: 2.776603 on 5.0 degrees of freedom
    Multiple R-squared:  0.987747,     Adjusted R-squared:  0.982846
    F-statistic: 201.53335392217406 on 2.0 and 5.0 DF,  p-value: 4.872837579850131E-5
----------------------------------------------------------------------------------
```

The summary results for the `SymbolicRegression.quadratic` model highlight a couple of important issues:

1. moderately high `Pr(>|t|)` (p-values) and
2. borderline high `VIF` (Variance Inflation Factor) values.

Try eliminating x_1 to see if these two improve without much of a drop in Adjusted R-squared $\bar{R}^2$. Note, eliminating x_1 makes the model non-hierarchical (see the exercises). Figure 6.6 shows the predictions (`yp`) of the `Regression` and `quadratic` models.

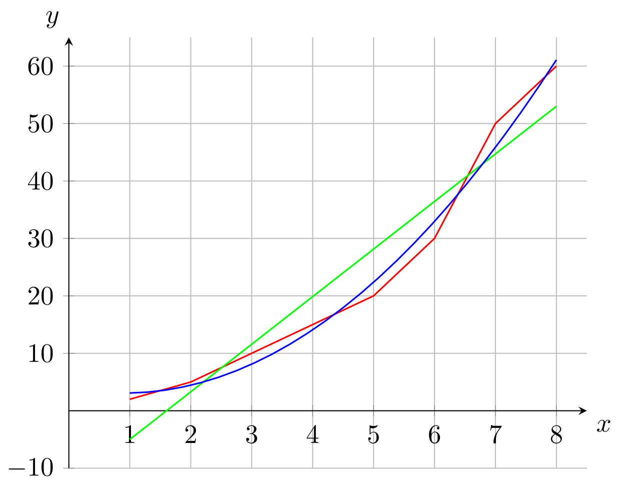


Figure 6.6: Actual y (red) vs. Regression yp (green) vs. quadratic yp (blue)

The `quadratic` method in the `SymbolicRegression` object creates a `Regression` object that uses multiple regression to fit a quadratic surface to the data.

6.9.3 `SymbolicRegression.quadratic` Method

Method:

```
    @param x          the initial data/input m-by-n matrix (before quadratic term expansion)
                          must not include an intercept column of all ones
    @param y          the response/output m-vector
    @param fname      the feature/variable names (defaults to null)
    @param intercept  whether to include the intercept term (column of ones) _1
                          (defaults to true)
    @param cross      whether to include 2-way cross/interaction terms x_i x_j
                          (defaults to false)
    @param hparam     the hyper-parameters (defaults to Regression.hp)

    def quadratic (x: MatrixD, y: VectorD, fname: Array [String] = null,
                   intercept: Boolean = true, cross: Boolean = false,
                   hparam: HyperParameter = Regression.hp): Regression =
        val mod       = apply (x, y, fname, Set (1, 2), intercept, cross, false, hparam)
        mod.modelName = "SymbolicRegression.quadratic" + (if cross then "X" else "")
        mod
    end quadratic
```

The `apply` method is defined in the `SymbolicRegression` object. The `Set (1, 2)` specifies that first (Linear) and second (Quadratic) order terms will be included in the model. The `intercept` flag indicates whether a column of ones will be added to the input/data matrix.

The next few modeling techniques described in subsequent sections support the development of low-order multi-dimensional polynomial regression models. Higher order polynomial regression models are typically restricted to one-dimensional problems (see the `PolyRegression` class).

6.9.4 Quadratic Regression with Cross Terms

The `quadratic` method provides the option of adding cross/interaction terms in addition to the quadratic terms. The `cross` flag indicates whether cross terms will be added to the model. A cross term such as the one based on the product x_1x_2 indicates a combined effect of two predictor variables on the response variable y.

Model Equation

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the quadratic cross model/regression equation is the following:

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x}' + \epsilon \;=\; b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_2^2 + b_5x_1x_2 + \epsilon} \tag{6.82}$$

where the components of the model equation are defined as follows:

$$\begin{aligned} \mathbf{x}' &= [1, x_1, x_2, x_1^2, x_2^2, x_1x_2] && \text{expanded input vector} \\ \mathbf{b} &= [b_0, b_1, b_2, b_3, b_4, b_5] && \text{parameter/coefficient vector} \\ \epsilon &= y - \mathbf{b} \cdot \mathbf{x}' && \text{error/residual} \end{aligned}$$

The number of terms (nt) in the model increases quadratically with the dimensionality of the space (n) according to the formula for triangular numbers shifted by ($n \to n + 1$).

$$nt \;=\; \binom{n+2}{2} \;=\; \frac{(n+2)(n+1)}{2} \qquad e.g., \quad nt \;=\; 6 \;\; \text{for} \;\; n = 2 \tag{6.83}$$

This result may derived by summing the number of constant terms (1), linear terms (n), quadratic terms (n), and cross terms $\binom{n}{2}$.

Such models generalize `quadratic` by introducing cross terms, e.g., x_1x_2. Adding cross terms makes the number of terms increase quadratically rather than linearly with the dimensionality. Consequently, multi-collinearity problems (check VIF scores) may be intensified and the need for feature selection, therefore, increases.

6.9.5 Response Surface

One may think of a `quadratic` model as well as more complex models as approximating a response surface in multiple dimensions.

$$y \;=\; f(x_1, x_2) + \epsilon \tag{6.84}$$

For example, a model with two predictor variables and one response variable may be displayed in three dimensions. Such a response surface can also be shown in two dimensions using contour plots where a contour/curve shows points of equal height. Figure 6.7 shows three types of contours that represent the types of terms in `quadratic` regression (1) linear terms, (2) quadratic terms, and (3) cross terms. In the figure, the first green line is for $x_1 + x_2 = 4$, the first blue curve is for $x_1^2 + x_2^2 = 16$, and the first red curve is for $x_1 x_2 = 4$.

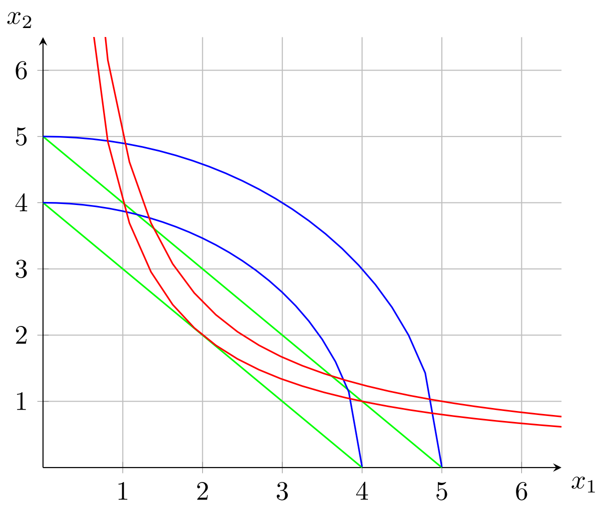


Figure 6.7: quadratic Contours: $x_1 + x_2$ (green), $x_1^2 + x_2^2$ (blue), $x_1 x_2$ (red)

A constant term simply moves the whole response surface up or down. The coefficients for each of terms can rotate and stretch these curves.

The response surface for Quadratic Regression on AutoMPG based on the best combination of features, `weight` and `modelyear`, is shown in 6.8.

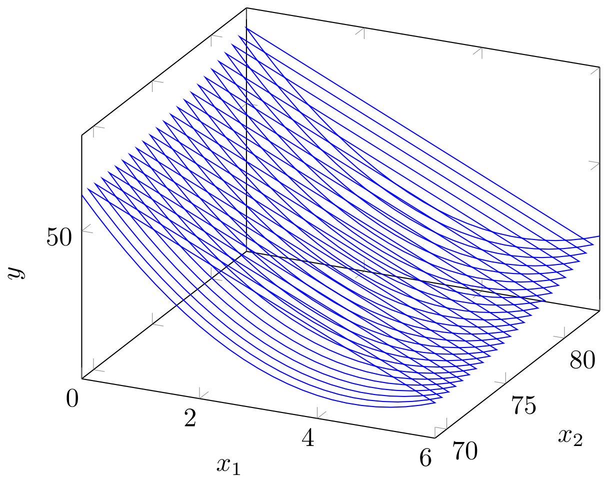


Figure 6.8: Response Surface: $\hat{y} = 355.139 - 21.1463x_1 - 8.50562x_2 + 2.29950x_1^2 + 0.0614339x_2^2$

6.9.6 Exercises

1. Enter the `x, y` dataset from the example given in this section and use it to create a `quadratic` model. Show the expanded input/data matrix and the response vector using the following two print statements.

```
    val qrg = new SymbolicRegression.quadratic (x, y)
    println (s"expanded x = ${qrg.getX}")
    println (s"y = ${qrg.getY}")
```

2. Perform Quadratic Regression on the `Example_BPressure` dataset using the first two columns of its data matrix `x`.

```
    import Example_BPressure.{x01 => x, y}
```

3. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response. Which feature selection approach (forward selection or backward elimination) finds a model with the highest $\bar{R}^2$?

4. Generate a dataset with data matrix `x` and response vector `y` using the following loop where `noise = new Normal (0, 10 * m * m)`.

```
    for i <- x.indices do
        x(i, 0) = i
        y(i) = i*i + i + noise.gen
    end for
```

   Compare the results of `Regression` vs. `quadratic`. Compare the Quality of Fit and the parameter values. What correspondence do the parameters have with the coefficients used to generate the data? Plot `y` vs. `x`, `yp` and `y` vs. `t` for both `Regression` and `quadratic`. Also plot the residuals `e` vs. `x` for both. Note, `t` is the index vector `VectorD.range (0, m)`.

5. Generate a dataset with data matrix `x` and response vector `y` using the following loop where `noise = new Normal (0, 10 * s * s)` and `grid = 1 to s`.

```
    var k = 0
    for i <- grid; j <- grid do
        x(k) = VectorD (i, j)
        y(k) = x(k, 0)~^2 + 2 * x(k, 1) + noise.gen
        k += 1
    end for
```

   Compare the results of `Regression` vs. `quadratic`. Try modifying the equation for the response and see how the Quality of Fit changes.

6. The `quadratic` model as well as its more complex cousin `cubic` may have issues with having **high multi-collinearity** or high VIF values. Although high VIF values may not be a problem for prediction accuracy, they can make interpretation and inferencing difficult. For the problem given in this section, rather than adding x^2 to the existing `Regression` model, find a second order polynomial that could be added without causing high VIF values. VIF values are the lowest when column vectors are **orthogonal**. See the section on Polynomial Regression for more details.

7. **Extrapolation** far from the training data can be risky for many types of models. Show how having higher order polynomial terms in the model can increase this risk.

8. A polynomial regression model is said to be **hierarchical** [143, 167, 127] if it contains all terms up to x^k, e.g., a model with x, x^2, x^3 is hierarchical, while a model with x, x^3 is not. Show that hierarchical models are invariant under linear transformations.

   Hint: Consider the following two models where x is the distance on I-70 West in miles from the center of Denver (junction with I-25) and y is the elevation in miles above sea level.

$$\begin{aligned}\hat{y} &= b_0 + b_1 x + b_2 x^2 \\ \hat{y} &= b_0 + b_2 x^2\end{aligned}$$

   The first model is hierarchical, while the second is not. A second study is conducted, but now the distance z is from the junction of I-70 and I-76. A linear transformation can be used to resolve the problem.

$$x = z + 7$$

   Putting z into the second model (assuming the first study indicated a linear term is not needed) gives,

$$\hat{y} = b_0 + b_2(z+7)^2 = (b_0 + 49b_2) + 14b_2 z + b_2 z^2$$

   but now the linear term is back in the model.

9. Perform quadratic and quadratic (with cross terms) regression on the `Example_BPressure` dataset using the first two columns of its data matrix `x`.

```
import Example_BPressure.{x01 => x, y}
```

10. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response. Which feature selection approach (forward selection or backward elimination) finds a model with the highest $\bar{R}^2$?

11. Generate a dataset with data matrix `x` and response vector `y` using the following loop where `noise = new Normal (0, 10 * s * s)` and `grid = 1 to s`.

```
var k = 0
for i <- grid; j <- grid do
    x(k) = VectorD (i, j)
    y(k) = x(k, 0)~^2 + 2 * x(k, 1) + x(k, 0) * x(k, 1) +  noise.gen
    k += 1
end for
```

   Compare the results of `Regression`, `quadratic` with `cross = false`, and `quadratic` with `cross = true`.

12. Prove that the number of terms for a quadratic function $f(\mathbf{x})$ in n dimensions is $\binom{n+2}{2}$, by decomposing the function into its quadratic (both squared and cross), linear and constant terms,

$$f(\mathbf{x}) = \mathbf{x}^\intercal A \mathbf{x} + \mathbf{b}^\intercal \mathbf{x} + c$$

where A in an n-by-n matrix, $\mathbf{b}$ is an n-dimensional column vector and c is a scalar. Hint: A is symmetric, but the main diagonal is not repeated, and we are looking for unique terms (e.g., x_1x_2 and x_2x_1 are treated as the same). Note, when $n = 1$, A and $\mathbf{b}$ become scalars, yielding the usual quadratic function $ax^2 + bx + c$.

6.10 Cubic Regression

The `cubic` method in the `SymbolicRegression` object adds cubic terms in addition to the quadratic terms added by the `quadratic` method. Linear terms in a model allow for slopes and quadratic terms allow for curvature. If the curvature changes substantially or there is an *inflection point* (curvature changes sign), then cubic terms may be useful. For example, before the inflection point the curve/surface may be concave upward, while after the point it may be concave downward, e.g., a car stops accelerating and starts decelerating.

When there is only one predictor variable x, the response y is modeled as a cubic function of x.

$$y = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \epsilon \tag{6.85}$$

6.10.1 Model Equation

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the cubic regression equation is the following:

$$y = \mathbf{b} \cdot \mathbf{x}' + \epsilon = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_1^2 + b_4 x_2^2 + b_5 x_1^3 + b_6 x_2^3 + \epsilon \tag{6.86}$$

where the components of the model equation are defined as follows:

$$\begin{aligned} \mathbf{x}' &= [1, x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3] && \text{expanded input vector} \\ \mathbf{b} &= [b_0, b_1, b_2, b_3, b_4, b_5, b_6] && \text{parameter/coefficient vector} \\ \epsilon &= y - \mathbf{b} \cdot \mathbf{x}' && \text{error/residual} \end{aligned}$$

The number of terms (nt) in the model still increases quadratically with the dimensionality of the space (n) according to the formula for triangular numbers shifted by ($n \to n + 1$) plus n for the cubic terms.

$$nt = \binom{n+2}{2} + n = \frac{(n+2)(n+1)}{2} + n \qquad \textit{e.g.,} \quad nt = 8 \text{ for } n = 2 \tag{6.87}$$

When $n = 10$, the number of terms and corresponding parameters $nt = 76$, whereas for `Regression`, `quadratic` and `quadratic` with cross terms and order 2, it would 11, 21 and 66, respectively. Issues related to negative Degrees of Freedom, over-fitting and multi-collinearity will need careful attention.

6.10.2 Comparison of `cubic`, `quadratic` and `Regression`

This subsection compares the `cubic` method to the `quadratic` method and the `Regression` class.

```
    // 8 data points:          x   y
    val xy = MatrixD ((8, 2), 1,   2,                  // 8-by-2 combined matrix
                              2,  11,
                              3,  25,
                              4,  28,
                              5,  30,
                              6,  26,
                              7,  42,
                              8,  60)
    val (x, y) = (xy.not(?, 1), xy(?, 1))              // x is first column, y is last column
    val ox = VectorD.one (x.dim) +^: x                 // prepend a column of all ones

    val rg  = new Regression (ox, y)                         // create a regression model
    rg.trainNtest ()()                                       // train and test the model
    println (rg.summary ())                                  // show summary

    val qrg = SymbolicRegression.quadratic (x, y)            // create a quadratic regression model
    qrg.trainNtest ()()                                      // train and test the model
    println (qrg.summary ())                                 // show summary

    val crg = SymbolicRegression.cubic (x, y)                // create a cubic regression model
    crg.trainNtest ()()                                      // train and test the model
    println (crg.summary ())                                 // show summary
```

Figure 6.9 shows the predictions (`yp`) of the `Regression`, `quadratic` and `cubic` models.

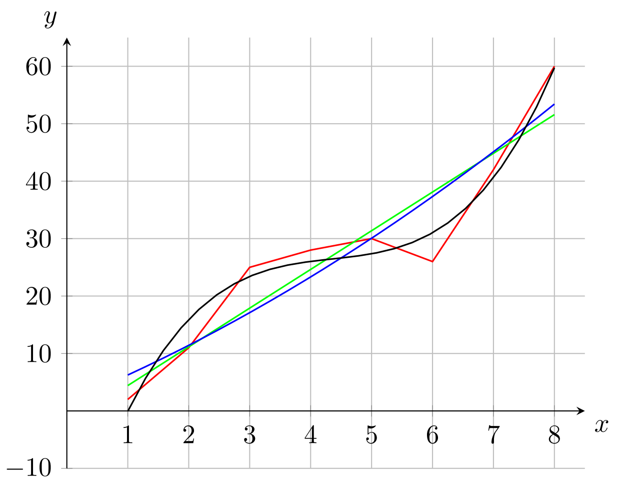


Figure 6.9: Actual y (red) vs. Regression (green) vs. quadratic (blue) vs. cubic (black)

Notice the quadratic curve follows the linear curve (line), while the cubic curve more closely follows the data.

6.10.3 `SymbolicRegression.cubic` Method

Class Methods:

```
    @param x           the initial data/input m-by-n matrix (before quadratic term expansion)
                           must not include an intercept column of all ones
    @param y           the response/output m-vector
    @param fname       the feature/variable names (defaults to null)
    @param intercept   whether to include the intercept term (column of ones) _1
                           (defaults to true)
    @param cross       whether to include 2-way cross/interaction terms x_i x_j
                           (defaults to false)
    @param cross3      whether to include 3-way cross/interaction terms x_i x_j x_k
                           (defaults to false)
    @param hparam      the hyper-parameters (defaults to Regression.hp)

    def cubic (x: MatrixD, y: VectorD, fname: Array [String] = null,
               intercept: Boolean = true, cross: Boolean = false, cross3: Boolean = false,
               hparam: HyperParameter = Regression.hp): Regression =
        val mod       = apply (x, y, fname, Set (1, 2, 3), intercept, cross, cross3, hparam)
        mod.modelName = "SymbolicRegression.cubic" + (if cross then "X" else "") +
                                                     (if cross3 then "X" else "")
        mod
    end cubic
```

The `Set (1, 2, 3)` specifies that first (Linear), second (Quadratic), and third (Cubic) order terms will be included in the model. The `intercept` flag indicates whether a column of ones will be added to the input/data matrix.

6.10.4 Cubic Regression with Cross Terms

The `cubic` method provides the option of adding 2-way cross/interaction terms (e.g., x_2x_1) controlled by the `cross` flag and/or 3-way cross/interaction terms (e.g., $x_1^2x_2$) controlled by the `cross3` flag.

Model Equation

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the cubic model/regression equation with `cross` terms is the following:

$$y \;=\; \mathbf{b}\cdot\mathbf{x}' + \epsilon \;=\; b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_2^2 + b_5x_1^3 + b_6x_2^3 + b_7x_1x_2 + \epsilon \tag{6.88}$$

and with `cross3` terms is

$$y \;=\; \mathbf{b}\cdot\mathbf{x}' + \epsilon \;=\; b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_2^2 + b_5x_1^3 + b_6x_2^3 + b_7x_1x_2 + b_8x_1^2x_2 + b_9x_1x_2^2 + \epsilon \tag{6.89}$$

where the components of the model equation are defined as follows:

$$\begin{aligned}
\mathbf{x}' &= [1, x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3, x_1x_2, x_1^2x_2, x_1x_2^2] && \text{expanded input vector}\\
\mathbf{b} &= [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9] && \text{parameter/coefficient vector}\\
\epsilon &= y - \mathbf{b}\cdot\mathbf{x}' && \text{error/residual}
\end{aligned}$$

Naturally, the number of terms in the model increases cubically with the dimensionality of the space (n) according to the formula for tetrahedral numbers shifted by ($n \to n+1$).

$$nt \;=\; \binom{n+3}{3} \;=\; \frac{(n+3)(n+2)(n+1)}{6} \qquad e.g.,\;\; nt \;=\; 10 \;\;\text{for}\;\; n = 2 \tag{6.90}$$

When $n = 10$, the number of terms and corresponding parameters $nt = 286$, whereas for `Regression`, `quadratic`, `quadratic` with cross and `cubic` with both crosses and order 2, it would 11, 21, 66 and 76, respectively. Issues related to negative Degrees of Freedom, over-fitting and multi-collinearity will need even more careful attention.

If polynomials of higher degree are needed, ScalaTion provides a couple of means to deal with it. First, when the data matrix consists of single column and **x** is one dimensional, the `PolyRegression` class may be used. If one or two variables need higher degree terms, the caller may add these columns themselves as additional columns in the data matrix input into the `Regression` class. The `SymbolicRegression` object described in the next section allows the user to try many function forms.

Categorical Variables and Collinearity

Quadratic and Cubic Regression may fail producing Not-a-Number (NaN) results when a dataset contains one or more categorical variables. For example, a variable like citizen "no", "yes" is likely to be encoded 0, 1. If such a column is squared or cubed, the new column will be identical to the original column, so that they will be perfectly collinear. One solution is not to expand such columns. If one must, then a different encoding may be used, e.g., 1, 2. See the section on `RegressionCat` for more details.

6.10.5 Exercises

1. Generate and compare the model summaries produced by the three models (`Regression`, `quadratic` and `cubic`) applied to the dataset given in this section.

2. An inflection point occurs when the second derivative changes sign. Find the inflection point in the following cubic equation:

$$y \;=\; f(x) \;=\; x^3 - 6x^2 + 12x - 5$$

   Plot the cubic function to illustrate. Explain why there are no inflection points for quadratic models.

3. Many laws in science involve quadratic and cubic terms as well as the inverses of these terms (e.g., inverse square laws). Find such a law and an open dataset to test the law.

4. Perform Cubic and Cubic with cross terms Regression on the `Example_BPressure` dataset using the first two columns of its data matrix `x`.

```
    import Example_BPressure.{x01 => x, y}
```

5. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response. Which feature selection approach (forward selection or backward elimination) finds a model with the highest $\bar{R}^2$?

6. Generate a dataset with data matrix `x` and response vector `y` using the following loop where `noise = new Normal (0, 10 * s * s)` and `grid = 1 to s`.

```
    var k = 0
    for i <- grid; j <- grid do
        x(k) = VectorD (i, j)
        y(k) = x(k, 0)~^2 + 2 * x(k, 1) + x(k, 0) * x(k, 1) +  noise.gen
        k += 1
    end for
```

Compare the results of `Regression`, `quadratic` with `cross = false`, `quadratic` with `cross = true`, `cubic` with `cross = false`, `cubic` with `cross = true`, `cubic` with `cross = true, cross3 = true`, Try modifying the equation for the response and see how the Quality of Fit changes.

6.11 Symbolic Regression

The last two sections covered Quadratic and Cubic Regression, but there are many possible functional forms. For example, in physics force often decreases with distance following a inverse square law. The *Newton's Law of Universal Gravitation* states that masses m_1 and m_2 with center of mass positions at $\mathbf{p}_1$ and $\mathbf{p}_2$ (with distance $r = \|\mathbf{p}_2 - \mathbf{p}_1\|$ will attract each other with force f,

$$f \;=\; G\frac{m_1 m_2}{r^2} \tag{6.91}$$

where the gravitational constant $G = 6.67408 \cdot 10^{-11}\mathrm{m}^3\mathrm{kg}^{-1}\mathrm{s}^{-2}$.

6.11.1 Sample Calculation

Let m_1 be the mass of a man (100 kg), m_2 be the mass of the Earth ($5.97219 \cdot 10^{24}$ kg), r be the distance to the center of the Earth (sea-level) ($6.371 \cdot 10^6$ m), then

$$f \;=\; 6.67408 \cdot 10^{-11}\frac{100 \cdot 5.97219 \cdot 10^{24}}{(6.371 \cdot 10^6)^2} \;=\; 982 \;\; \mathrm{kg \cdot m/s^2} \;\; \text{(or newtons)}$$

The `Calc` object in ScalaTion may be used to evaluate the following function, passing in 100 kilograms.

```
    def f(x: Double): Double = 6.67408E-11 * 5.97219E24 * x / 6.371E6~^2
```

The calculation is performed by the following: `runMain scalation.runCalc 100`.

6.11.2 As a Data Science Problem

This can be recast a symbolic regression problem using the following renaming ($m_1 \to x_0, m_2 \to x_1, r \to x_2, f \to y$).

$$y \;=\; b_0 x_0 x_1 x_2^{-2} + \epsilon \tag{6.92}$$

Given a four column dataset $[x_0, x_1, x_2, y]$ a Symbolic Regression could be run to estimate a more general model that includes all possible terms with powers $x_j^{-2}, x_j^{-1}, x_j^1, x_j^2$. It could also include cross (two-way interaction) terms between all these terms. In this case, it is necessary to add cross3 (three-way interaction) terms. An intercept would imply force with no masses involved, so it should be left out of the model.

It is easier to collect data where the Earth is used for mass 1 and mass 2 is for people at various distances from the center of the Earth ($m_1 \to x_0, r \to x_1, f \to y$).

$$y \;=\; b_0 x_0 x_1^{-2} + \epsilon \tag{6.93}$$

In this case the parameter b_0 will correspond to GM, where G is the Gravitational Constant and M is the Mass of the Earth. The following code provides simulated data and uses symbolic regression to determine the Gravitational Constant.

```
    val noise = Normal (0, 10)                          // random noise
    val rad   = Uniform (6370, 7000)                    // distance from center of Earth in km
    val mas   = Uniform (50, 150)                       // mass of person

    val M = 5.97219E24                                  // mass of Earth in kg
    val G = 6.67408E-11                                 // gravitational const. m^3 kg^-1 s^-2

    val xy = new MatrixD (100, 3)                         // simulated gravity data
    for i <- xy.indices do
        val m = mas.gen                                   // unit of kilogram (kg)
        val r = 1000 * rad.gen                            // unit of meter (m)
        xy(i, 0) = m                                      // mass of person
        xy(i, 1) = r                                      // radius/distance
        xy(i, 2) = G * M * m / r~^2 + noise.gen           // force of gravity GM m/r^2
    end for

    val fname = Array ("mass", "radius")

    println (s"xy = $xy")
    val (x, y) = (xy.not (?, 2), xy(?, 2))

    banner ("Newton's Universal Gravity Symbolic Regression")
    val mod = SymbolicRegression (x, y, fname, null, false, false,
              terms = Array ((0, 1.0), (1, -2.0)))        // add one custom term

    mod.trainNtest ()()                                   // train and test the model
    println (mod.summary ())                              // parameter/coefficient statistics
    println (s"b =~ GM = ${G * m1}")                      // Gravitational Constant * Earth Mass
```

The statement `val mod = SymbolicRegression (...)` invokes the factory method called `apply` in the `SymbolicRegression` object. The `SymbolicRegression` object provides methods for quadratic, cubic, and more general symbolic regression.

6.11.3 `SymbolicRegression` Object

Object Methods:

```
    object SymbolicRegression:

    def apply (x: MatrixD, y: VectorD, fname: Array [String] = null, ...
    def buildMatrix (x: MatrixD, fname: Array [String], ...
    def rescale (x: MatrixD, y: VectorD, fname: Array [String] = null, ...
    def crossNames (nm: Array [String]): Array [String] =
    def crossNames3 (nm: Array [String]): Array [String] =
    def quadratic (x: MatrixD, y: VectorD, fname: Array [String] = null, ...
    def cubic (x: MatrixD, y: VectorD, fname: Array [String] = null, ...

    end SymbolicRegression
```

The `apply` method is flexible enough to include many functional forms as terms in a model. Feature selection can be used to eliminate many of the terms to produce a meaningful and interpretable model. Note, unless measurements are precise and experiments are controlled, other terms besides the one given by Newton's of Universal Gravitation are likely to be selected.

```
    @param x          the initial data/input m-by-n matrix (before expansion)
                          must not include an intercept column of all ones
    @param y          the response/output m-vector
    @param fname      the feature/variable names (defaults to null)
    @param powers     the set of powers to raise matrix x to (defaults to null)
    @param intercept  whether to include the intercept term (column of ones) _1
                          (defaults to true)
    @param cross      whether to include 2-way cross/interaction terms x_i x_j
                          (defaults to true)
    @param cross3     whether to include 3-way cross/interaction terms x_i x_j x_k
                          (defaults to false)
    @param hparam     the hyper-parameters (defaults to Regression.hp)
    @param terms      custom terms to add into the model, e.g.,
                          Array ((0, 1.0), (1, -2.0)) adds x0 x1^(-2)

    def apply (x: MatrixD, y: VectorD, fname: Array [String] = null,
               powers: Set [Double] = null, intercept: Boolean = true,
               cross: Boolean = true, cross3: Boolean = false,
               hparam: HyperParameter = Regression.hp,
               terms: Array [Xj2p]*): Regression =
        val fname_ = if fname != null then fname
                     else x.indices2.map ("x" + _).toArray

        val (xx, f_name) = buildMatrix (x, fname_, powers, intercept, cross, cross3,
                                        terms :_*)
        val mod       = new Regression (xx, y, f_name, hparam)
        mod.modelName = "SymbolicRegression" + (if cross then "X" else "") +
                                               (if cross3 then "X" else "")
        mod
    end apply
```

where `type Xj2p = (Int, Double)` indicates raising column `Xj` to the `p`-th power.

6.11.4 Implementation of the `apply` Method

The `apply` method forms an expanded matrix and passes it to the `Regression` class. The following arguments control what terms are added to a model:

1. The `powers` set takes each column in matrix X and raises it to the p^{th} power for every $p \in$ `powers`. The expression X^p produces a matrix with all columns raised to the p^{th} power. For example, `Set (1, 2, 0.5)` will add the original columns, quadratic columns, and square root columns.

2. The `intercept` flag indicates whether an intercept (column of ones) is to be added to the model. Again, such a column must not be included in the original matrix.

3. The `cross` flag indicates whether two-way cross/interaction terms of the form $x_i x_j$ (for $i \neq j$) are to be added to the model.

4. The `cross3` flag indicates whether three-way cross/interaction terms of the form $x_i x_j x_k$ (for i, j, k not all the same) are to be added to the model.

5. The `terms` (repeated) array allows custom terms to add into the model. For example,

```
Array ((0, 1.0), (1, -2))
```

   adds the term $x_0 x_1^{-2}$ to the model. As this argument is repeated (`Array [Xj2p]*`) due to the star (`*`), additional custom terms may be added. The `*` makes the last argument a vararg.

Much of functionality to do this is supplied by the `MatrixD` class in the `mathstat` package. The operator `++^` concatenates two matrices column-wise, while operator `x~^p` returns a new matrix where each of the columns in the original matrix is raised to the p^{th} power. The `crossAll` method returns a new matrix consisting of columns that multiply each column by every other column. The `crossAll3` method returns a new matrix consisting of columns that multiply each column by all combinations of two other columns.

`buildMatrix` Method

The bulk of the work is done by the `buildMatrix` method that creates the input data matrix, column by column.

```
    def buildMatrix (x: MatrixD, fname: Array [String],
                     powers: Set [Double], intercept: Boolean,
                     cross: Boolean, cross3: Boolean,
                     terms: Array [Xj2p]*): (MatrixD, Array [String]) =
        val _1      = VectorD.one (x.dim)                            // one vector
        var xx      = new MatrixD (x.dim, 0)                         // start empty
        var fname_ = Array [String] ()

        if powers != null then
            if powers contains 1 then
                xx      = xx ++^ x                                   // add linear terms x
                fname_ = fname
            end if
            for p <- powers if p != 1 do
                xx       = xx ++^ x~^p                               // add other power x^p terms
                fname_ ++= fname.map ((n) => s"$n^$p.toInt}")
            end for
        end if

        if terms != null then
            debug ("buildMatrix", s"add customer terms = $stringOf (terms)}")
            var z = _1.copy
            var s = ""
            for t <- terms do
                for (j, p) <- t do                                   // x_j to the p-th power
                    z *= x(?, j)~^p
                    s = s + s"x$j^$p.toInt}"
                end for
                xx      = xx :^+ z                                   // add custom term/column t
                fname_ = fname_ :+ s
            end for
        end if

        if cross then
           xx       = xx ++^ x.crossAll                              // add 2-way cross x_i x_j
           fname_ ++= crossNames (fname)
        end if

        if cross3 then
           xx       = xx ++^ x.crossAll3                          // add 3-way cross x_i x_j x_k
           fname_ ++= crossNames3 (fname)
        end if

        if intercept then
            xx     = _1 +^: xx                                     // add intercept term (_1)
            fname_ = Array ("one") ++ fname_
        end if

        (xx, fname_)                                               // return expanded matrix
    end buildMatrix
```

6.11.5 Regularization

Due to fact that symbolic regression may introduce many terms into the model and have high multi-collinearity, regularization becomes even more important.

Symbolic Ridge Regression

Symbolic Ridge Regression can be beneficial in dealing with multi-collinearity. The `SymRidgeRegression` object supports the same methods that `SymbolicRegression` does, except `buildMatrix` that it reuses.

```
object SymRidgeRegression:

    @param x       the initial data/input m-by-n matrix (before expansion)
                       must not include an intercept column of all ones
    @param y       the response/output m-vector
    @param fname   the feature/variable names (defaults to null)
    @param powers  the set of powers to raise matrix x to (defaults to null)
    @param cross   whether to include 2-way cross/interaction terms x_i x_j
                       (defaults to true)
    @param cross3  whether to include 3-way cross/interaction terms x_i x_j x_k
                       (defaults to false)
    @param hparam  the hyper-parameters (defaults to RidgeRegression.hp)
    @param terms   custom terms to add into the model, e.g.,
                       Array ((0, 1.0), (1, -2.0)) adds x0 x1^(-2)

    def apply (x: MatrixD, y: VectorD, fname: Array [String] = null,
               powers: Set [Double] = null, cross: Boolean = true, cross3: Boolean = false,
               hparam: HyperParameter = RidgeRegression.hp,
               terms: Array [Xj2p]*): RidgeRegression =
        val fname_ = if fname != null then fname
                     else x.indices2.map ("x" + _).toArray                 // default names

        val (xx, f_name) = SymbolicRegression.buildMatrix (x, fname_, powers,
                                                           false, cross, cross3, terms :_*)
//      val mod        = new RidgeRegression (xx, y, f_name, hparam)       // user centers
        val mod        = RidgeRegression.center (xx, y, f_name, hparam)    // auto. centers
        mod.modelName = "SymRidgeRegression" + (if cross then "X" else "") +
                                               (if cross3 then "XX" else "")
        mod
    end apply
```

It requires the data to be centered and has no intercept (see exercises).

Symbolic Lasso Regression

Other forms of regularization can be useful as well. Symbolic Lasso Regression can be beneficial in dealing with multi-collinearity and more importantly by setting some parameters/coefficients b_j to zero, thereby eliminating the j^{th} term. This is particularly important for symbolic regression as the number of possible terms can become very large.

```
object SymLassoRegression:

    @param x           the initial data/input m-by-n matrix (before expansion)
                           must not include an intercept column of all ones
    @param y           the response/output m-vector
    @param fname       the feature/variable names (defaults to null)
    @param intercept   whether to include the intercept term (column of ones) _1 (defaults to
     true)
    @param powers      the set of powers to raise matrix x to (defaults to null)
    @param cross       whether to include 2-way cross/interaction terms x_i x_j (defaults to
    true)
    @param cross3      whether to include 3-way cross/interaction terms x_i x_j x_k (defaults
     to false)
    @param hparam      the hyper-parameters (defaults to LassoRegression.hp)
    @param terms       custom terms to add into the model, e.g., Array ((0, 1.0), (1, -2.0))
                           adds x0 x1^(-2)

    def apply (x: MatrixD, y: VectorD, fname: Array [String] = null,
               powers: Set [Double] = null, intercept: Boolean = true,
               cross: Boolean = true, cross3: Boolean = false,
               hparam: HyperParameter = LassoRegression.hp,
               terms: Array [Xj2p]*): LassoRegression =
        val fname_ = if fname != null then fname
                     else x.indices2.map ("x" + _).toArray                // default names

        val (xx, f_name) = SymbolicRegression.buildMatrix (x, fname_, powers, intercept,
                                                            cross, cross3, terms :_*)
        val mod        = new LassoRegression (xx, y, f_name, hparam)
        mod.modelName = "SymLassoRegression" + (if cross then "X" else "") +
                                               (if cross3 then "XX" else "")
        mod
    end apply
```

6.11.6 Exercises

1. **Exploratory Data Analysis Revisited**. For each predictor variable x_j in the `Example_AutoMPG` dataset, determine the best power to raise that column to. Plot `y` and `yp` versus `xj` for `SimpleRegression`. Compare this to the plot of `y` and `yp` versus `xj` for `SymbolicRegression` using the best power.

2. Combine all the best powers together to form a model matrix with the same number of columns as the original AutoMPG matrix and compare `SymbolicRegression` with `Regression` on the original matrix.

3. Use forward, backward and stepwise regression to look for a better (than the last exercise) combination of features for the AutoMPG dataset.

4. Redo the last exercise using `SymRidgeRegression`. Note any differences.
5. Redo the last exercise using `SymLassoRegression`. Note any differences.
6. When there are for example quadratic terms added to the expanded matrix, explain why it will not work to simply center (by subtracting the column means) the original data matrix X.
7. Compare the effectiveness of the following two search strategies that are used in Symbolic Regression: (a) Genetic Algorithms and (b) FFX Algorithm.
8. Present a review of a paper that discusses how Symbolic Regression has been used to reproduce a theory in a scientific discipline.

6.12 Transformed Regression

The `TranRegression` class supports transformed multiple linear regression and hence, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ...x_k]$. In certain cases, the relationship between the response scalar y and the predictor vector $\mathbf{x}$ is not linear. There are many possible functional relationships that could apply [144], but five obvious choices are the following:

1. The response grows exponentially versus a linear combination of the predictor variable.
2. The response grows quadratically versus a linear combination of the predictor variable.
3. The response grows as the square root of a linear combination of the predictor variable.
4. The response grows logarithmically versus a linear combination of the predictor variable.
5. The response grows inversely (as the reciprocal) versus a linear combination of the predictor variable.

The capability can be easily implemented by introducing a transform (transformation function) into `Regression`. The transformation function and its inverse are passed into the `TranRegression` class which extends the `Regression` class.

$$f_t : \mathbb{R} \to \mathbb{R} \qquad \text{transformation function} \tag{6.94}$$

$$f_a : \mathbb{R} \to \mathbb{R} \qquad \text{activation function} \tag{6.95}$$

The transform f_t (`tran`) and its inverse transform $f_a = f_t^{-1}$ (`itran`) for the five cases are as follows:

```
(log, exp), (log1p, expm1), (sqrt, sq), (sq, sqrt), (exp, log), (recip, recip)
```

The second pair is the first pair shifted by one ($\text{log1p}(x) = \log(1+x)$ and $\text{expm1}(x) = \exp(x) - 1$) to better handle cases where x is very small. These and other common functions can be found in the `scala.math` package or the `scalation` package defined in `CommonFunctions.scala`.

Transformed Regression models extend the reach of linear models while maintaining their simplicity and highly efficient parameter estimation techniques. Beyond these models lay Generalized Linear Models and Nonlinear Models.

6.12.1 Model Equation

The goal then is to fit the parameter vector $\mathbf{b}$ in the transformed model/regression equation

$$\boxed{f_t(y) \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... \, b_k x_k + \epsilon} \tag{6.96}$$

where ϵ represents the residuals (the part not explained by the model) and f_t (`tran`) is the function (defaults to log) used to transform the response y. For example, for a log transformation, the equation becomes the following:

$$\log(y) \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... \, b_k x_k + \epsilon \tag{6.97}$$

The transform f_t (`tran`) is implemented in the `TranRegression` class by transforming $\mathbf{y}$ and passing it to the `Regression` superclass (multiple linear regression).

```
    Regression (x, y.map (tran), fname_, hparam)
```

The inverse transform f_a (`itran`) is then applied in the overridden `predict` method.

```
    override def predict (z: VectorD): Double = itran (b dot z)
```

6.12.2 Training

Using several data samples as a training set $(X, \mathbf{y})$, a loss function $\mathcal{L}(\mathbf{b})$ can be minimized to find an optimal solution for the parameter vector $\mathbf{b}$.

The training diagram shown in Figure 6.10 illustrates how the i^{th} instance/row flows through the diagram computing the transformed response $z = f_t(y)$, the predicted transformed response $\hat{z} = \mathbf{b} \cdot \mathbf{x}$ and the transformed error $e = z - \hat{z}$.

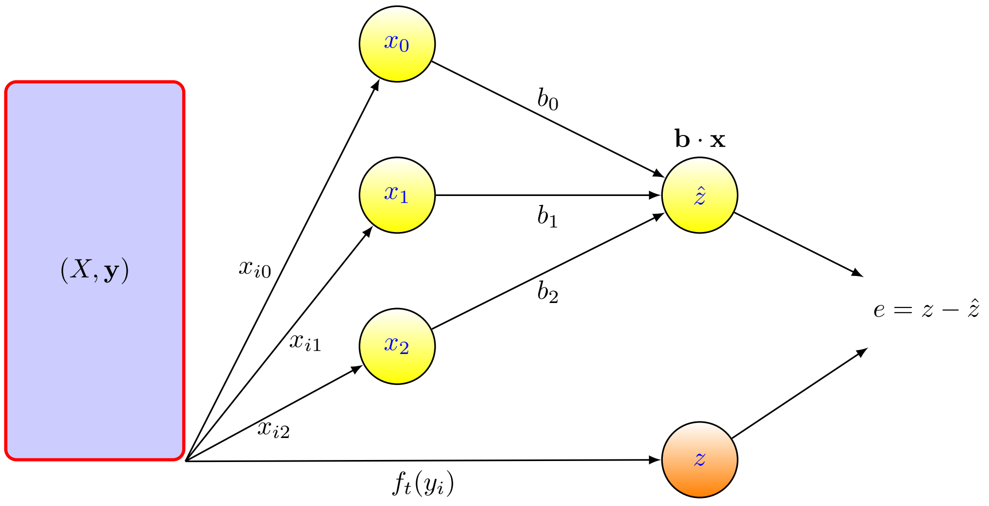


Figure 6.10: Training Diagram for Transformed Regression

Note, the actual (untransformed) error is $\epsilon = y - \hat{y} = f_a(z) - f_a(\hat{z})$.

The loss function is based on the transformed errors,

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\|\mathbf{e}\|^2 \;=\; \frac{1}{2}\|\mathbf{z} - \hat{\mathbf{z}}\|^2 \;=\; \frac{1}{2}\|\mathbf{f}_t(\mathbf{y}) - X\mathbf{b}\|^2 \tag{6.98}$$

where the transformed error vector $\mathbf{e} = \mathbf{z} - \hat{\mathbf{z}}$, $\mathbf{z} = \mathbf{f}_t(\mathbf{y})$, and $\hat{\mathbf{z}} = X\mathbf{b}$. Note, $\mathbf{f}_t : \mathbb{R}^m \to \mathbb{R}^m$ is the vectorized version of f_t. See the exercises for a loss function based the actual (untransformed) errors.

Taking the gradient of the loss function and setting it to zero,

$$\nabla\mathcal{L}(\mathbf{b}) \;=\; X^{\intercal}[\mathbf{f}_t(\mathbf{y}) - X\mathbf{b}] \;=\; \mathbf{0} \tag{6.99}$$

yields the Normal Equations for Transformed Regression.

$$(X^{\intercal}X)\mathbf{b} = X^{\intercal}\mathbf{f}_t(\mathbf{y}) \tag{6.100}$$

6.12.3 Square Root Transformation

The square root transformation takes the square root of the response variable and regresses it onto a linear model.

$$\boxed{f_t(y) \;=\; \sqrt{y} \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon} \tag{6.101}$$

As an example in 1D, the `TranRegression` class is compared to the `quadratic` method and the `Regression` class.

$$\begin{aligned} y &= b_0 + b_1 x + \epsilon && \text{regression} \\ y &= b_0 + b_1 x + b_2 x^2 + \epsilon && \text{quadratic regression} \\ \sqrt{y} &= b_0 + b_1 x + \epsilon && \text{sqrt transformed regression} \end{aligned}$$

The ScalaTion code shown below compares these three models. To run this code, be sure to add the appropriate `package` statement, depending on where this code is placed.

```
import scala.math.sqrt
import scalation._
import scalation.mathstat._
import scalation.modeling._

@main def tranRegressionTest (): Unit =

    // 8 data points:          x    y
    val xy = MatrixD ((8, 2), 1,   2,                        // 8-by-2 combined matrix
                              2,   5,
                              3,  10,
                              4,  15,
                              5,  20,
                              6,  30,
                              7,  50,
                              8,  60)
    val x_fname  = Array ("x")                               // names of features for x
    val ox_fname = Array ("_1", "x")                         // names of features for ox

    println ("model: y = b0 + b1*x1 + b2*x1^2")
    println (s"xy = $xy")

    val oxy     = VectorD.one (xy.dim) +^: xy                // combined matrix: ones column
    val (ox, y) = (oxy.not(?, 2), oxy(?, 2))                 // (data matrix, response column)
    val x       = xy.not(?, 1)                               // data matrix with no _1 column

    banner ("Regression")
    val reg = Regression (oxy, ox_fname)()                   // create a Regression model
    reg.trainNtest ()()                                      // train and test the model
    println (reg.summary ())                                 // parameter/coefficient stats
    val yp = reg.predict (ox)                                // y predicted for Regression

    banner ("Quadrastic Regression")
    val qrg = SymbolicRegression.quadratic (x, y, x_fname)  // create a Quadratic Regression
    qrg.trainNtest ()()                                      // train and test the model
    println (qrg.summary ())                                   // parameter/coefficient stats
    val yp2 = qrg.predict (qrg.getX)                           // y predicted for Quadratic

    banner ("Transformed Regression")
    val mod = new TranRegression (ox, y, ox_fname, Regression.hp,
                                  sqrt, sq)                    // sqrt Transformed Regression
    mod.trainNtest ()()                                        // train and test the model
    println (mod.summary ())                                   // parameter/coefficient stats
    val yp3 = mod.predict (ox)                                 // y predicted for Transformed

    val mat = MatrixD (y, yp, yp2, yp3)
    println (s"mat = $mat")
    new PlotM (null, mat, null, "y vs. yp vs. yp2 vs. yp3", true)

end tranRegressionTest
```

Figure 6.11 shows the predictions (`yp`) of the `Regression`, `quadratic` and `TranRegression` models.

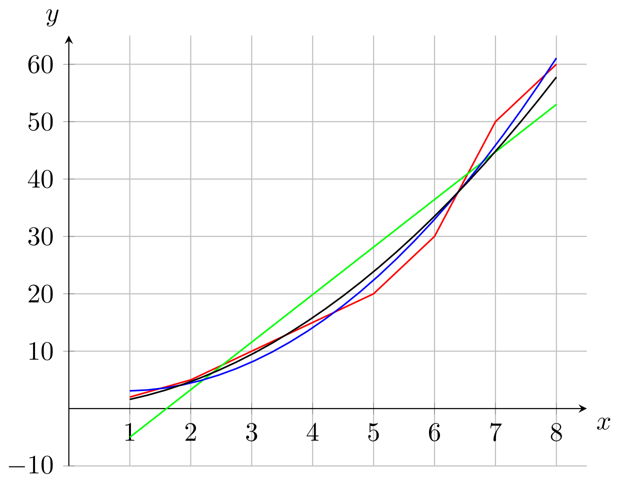


Figure 6.11: Actual y (red) vs. Regression (green) vs. quadratic (blue) vs. TranRegression (black)

Notice that the sqrt transformed regression model closely follows the quadratic regression model, yet has one fewer parameter. The square root transformation can model quadratic effects and stabilize error variance, but it makes interpretation of coefficients less direct. See the exercises for a comparison.

6.12.4 Log Transformation

The log transformation takes the logarithm (defaults to ln, the natural log) of the response variable and regresses it onto a linear model.

$$\boxed{f_t(y) \ = \ \log(y) \ = \ \mathbf{b} \cdot \mathbf{x} + \epsilon} \tag{6.102}$$

Imagine a system where the rate of change of the response variable y with the predictor variable x (e.g., time) is proportional to its current value y and is y_0 when $x = 0$.

$$\frac{dy}{dx} = gy$$

This *differential equation* can be solved by direct integration to obtain

$$\int \frac{dy}{y} = \int g\,dx$$

As the integral of $\frac{1}{y}$ is $\ln(y)$, integrating both sides gives

$$\ln(y) = gx + C$$

Solving for the constant gives $C = \ln(y_0)$, and then taking the exp function of both sides produces (ignoring noise/error)

$$\begin{aligned} \ln(y) &= gx + \ln(y_0) \\ y &= y_0 e^{gx} \end{aligned}$$

When the growth factor g is positive, the system exhibits exponential growth, while when it is negative, it exhibits exponential decay. So far we have ignored noise. For previous modeling techniques, we have assumed that noise is additive and typically normally distributed. For phenomena exhibiting exponential growth or decay, this may not be the case. When the error is multiplicative, we may collect it into the exponent.

$$y = y_0 e^{gx+\epsilon}$$

Now applying a log transformation, will yield

$$\log(y) = log(y_0) + gx + \epsilon = b_0 + b_1 x + \epsilon$$

An alternative to using `TranRegression` is to use Exponential Regression `ExpRegression`, a form of Nonlinear Regression Model (see the exercises for a comparison).

6.12.5 Reciprocal Transformation

The reciprocal (or inverse) transformation takes the reciprocal of the response variable and regresses it onto a linear model.

$$\boxed{f_t(y) = y^{-1} = \mathbf{b} \cdot \mathbf{x} + \epsilon} \qquad (6.103)$$

It may be the case that rates, such as Miles-Per-Gallon (MPG), are inversely related to other quantities. This can be tested using the AutoMPG dataset as follows:

```
    import scala.math._
    import scalation.mathstat._
    import scalation.modeling._
    import scalation.modeling.Example_AutoMPG._
    import scalation.modeling.TranRegression._

    val f = (recip, recip, "recip")
//  val f = (log,   exp,   "log")
//  val f = (sqrt,  sq,    "sqrt")
//  val f = (sq,    sqrt,  "sq")
//  val f = (exp,   log,   "exp")
//  TranRegression.setLambda (0.2); val f = (box_cox, cox_box, "box_cox")

    banner (s"TranRegression with $f._3} transform")
    val mod = TranRegression (ox, y, ox_fname, Regression.hp, f._1, f._2)
    mod.trainNTest ()()
    println (mod.summary ())

    val (cols, rSq) = mod.forwardSelAll ()                          // R^2, R^2 bar, R^2 cv
    val k = cols.size
    println (s"k = $k, n = $ox.dim2}")
    new PlotM (null, rSq.𝒯, Array ("R^2", "R^2 bar", "R^2 cv"),
               s"R^2 vs n for TranRegression $f._3}", lines = true)
    println (s"rSq = $rSq")
```

Be sure to try all the commented out transformations as well as various values for λ for the Box-Cox transformation (see the next subsection).

6.12.6 Box-Cox Transformation

As should be apparent from the last section, there are many transformations that could be applied. In order to provide a more systematic approach for transforming the response variable, Box-Cox transformations were developed. These transformations may be used to (1) improve prediction accuracy, (2) stabilize variance (reduce heteroscedasticity), and (3) improve normality.

Consider the following family of transformation functions.

$$f_t(y) \;=\; \frac{y^\lambda - 1}{\lambda} \tag{6.104}$$

where $\lambda \neq 0$ determines the power function on y, e.g., 0.5 for `sqrt` and 2.0 for `sq`. The following simplified version of the form given by Box and Cox may be used as well.

$$\boxed{f_t(y) \;=\; y^\lambda} \tag{6.105}$$

The inverse transform is $f_a(y) = y^{1/\lambda}$. When $\lambda = 0$, the `log` transformation is used. See the exercises for a more detailed treatment.

One way to find suitable values for λ (`lambda`) is to perform grid search over the interval $[-3, 3]$, possibly as large as $[-5, 5]$.

```
    TranRegression (x, y, lambda)
```

Box-Cox transformations are provided in the class' companion object.

Note, although the focus has been on transforming the response variable, the predictor variables could be transformed as well.

6.12.7 Quality of Fit

For a fair comparison with other modeling techniques, the Quality of Fit (QoF) or overall diagnostics are based on the original response values, as provided by the usual `test` method. It may be useful to study the Quality of Fit for the transformed response vector `y.map (tran)` as well via the `test0` method.

6.12.8 TranRegression Class

Class Methods:

```
    @param x        the data/input matrix
    @param y        the response/output vector
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to Regression.hp)
    @param tran     the transformation function (defaults to log)
    @param itran    the inverse transformation function to rescale predictions
                    to original y scale (defaults to exp)

    class TranRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                          hparam: HyperParameter = Regression.hp,
                          tran: FunctionS2S = log, itran: FunctionS2S = exp)
          extends Regression (x, y.map (tran), fname_, hparam):

    def test0 (x_ : MatrixD = x, y_ : VectorD = getY): (VectorD, VectorD) =
    override def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def trainNtest (x_ : MatrixD = x, y_ : VectorD = getY)
    override def predict (z: VectorD): Double = itran (b dot z)
    override def predict (x_ : MatrixD): VectorD = (x_ * b).map (itran)
    override def validate (rando: Boolean = true, ratio: Double = 0.2)
                          (idx : IndexedSeq [Int] =
                           testIndices ((ratio * y.dim).toInt, rando)): VectorD =
    override def buildModel (x_cols: MatrixD): Regression =
```

6.12.9 Exercises

1. Use the following code to generate a dataset. You will need to import from `scalation.math.sq` and `scalation.random`.

```
    val cap     = 30
    val rng     = 0 until cap
    val (m, n)  = (cap * cap, 3)
    val err     = Normal (0, cap)
    val x       = new MatrixD (m, n)
    val y       = new VectorD (m)
    for i <- rng; j <- rng do x(cap * i + j) = VectorD (1, i, j)
    for k <- y.indices do y(k) = sq (10 + 2 * x(k, 1) + err.gen)
```

As an alternative, try

```
    for k <- y.indices do y(k) = sq (10 + 2 * x(k, 1) + 0.3 * x(k, 2) + err.gen)
```

Notice that it uses a linear model inside and takes the square for the response variable `y`. Use `Regression` to create a predictive model. Ideally, the model should approximately recapture the equations used to generate the data. What correspondence do the parameters `b` have to these equations? Next, examine the relationship between the response `y` and predicted response `yp`, as well as the residuals (or remaining error) from the model.

```
val reg = new Regression (x, y)
reg.trainNtest ()()
println (reg.summary ())

val yp = reg.predict (x)
val e  = y - yp

new Plot (null, y, yp, "Original Regression y and yp vs. t")
new Plot (null, e, null, "Original e vs. t")
```

Are there discernible patterns in the residuals?

2. Transform the response `y` to a transformed response `y2` that is the square root of the former.

```
val y2 = y.map (sqrt)
```

Redo the regression as before, but now using the transformed response `y2`, i.e., `new Regression (x, y2)`. Compute and plot the corresponding `y2` versus `yp2` and then `e2` vectors. What do the residuals look like now? How can predictions be made on the original scale?

3. Now transform `yp2` to `yp3` in order the match the actual response `y`, by using the inverse transformation function `sq`. Now, compute and plot the corresponding `y` versus `yp3` and then `e3` vectors. How well does `yp3` predict the original response `y`? Compute the Coefficient of Determination R^2. What is the difference between the residuals `e2` and `e3`? Finally, use `PlotM` to compare `Regression` vs. Transformed `Regression`.

```
val ys2 = MatrixD (y2, yp2)
val ys3 = MatrixD (y, yp3, yp)
new PlotM (null, ys2.𝒯, null, "Transformed")
new PlotM (null, ys3.𝒯, null, "Tran-back")
```

4. The `TranRegression` class provides direct support for making transformations. Compare the quality of fit resulting from `Regression` versus `TranRegression`.

```
banner ("Regression")
val rg = new Regression (x, y)
rg.trainNTest ()()
println (rg.summary ())

banner ("TranRegression")
val trg = new TranRegression (x, y, null, null, sqrt, sq)
trg.trainNTest ()()
println (rg.summary ())
```

5. Extend the previous exercise to include `quadratic` regression in the comparison.

6. Compare `SimpleRegression`, `TranRegression` and `ExpRegression` on the beer foam dataset `www.tf.uni-kiel.de/matwis/amat/iss/kap_2/articles/beer_article.pdf`. The last two are similar, but `TranRegression` assumes multiplicative noise, while `ExpRegression` assumes additive noise, so they produce different predictions. Plot and compare the three predictions.

```
    val x1 = VectorD (0, 15, 30, 45, 60, 75, 90, 105, 120, 150, 180,
                      210, 240, 300, 360)
    val y  = VectorD (14.0, 12.1, 10.9, 10.0, 9.3, 8.6, 8.0, 7.5,
                       7.0,  6.2,  5.5, 4.5, 3.5, 2.0, 0.9)
    val _1 = VectorD.one (x1.dim)
    val x  = MatrixD (_1, x1)
```

7. Compare the following loss function based on actual (untransformed) errors $\boldsymbol{\epsilon}$ with the one based on transformed errors $\mathbf{e}$.

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\|\boldsymbol{\epsilon}\|^2 \;=\; \frac{1}{2}\|\mathbf{y}-\hat{\mathbf{y}}\|^2 \;=\; \frac{1}{2}\|\mathbf{y}-\mathbf{f}_a(X\mathbf{b})\|^2$$

8. A general form of transformation functions [164] was given by Tukey in 1957.

$$f_t(y) \;=\; y^\lambda \qquad \lambda \neq 0 \qquad (6.106)$$

$$f_t(y) \;=\; \log y \qquad \lambda = 0 \qquad (6.107)$$

and was refined by Box and Cox in 1964.

$$f_t(y) \;=\; \frac{y^\lambda - 1}{\lambda} \qquad \lambda \neq 0 \qquad (6.108)$$

$$f_t(y) \;=\; \log y \qquad \lambda = 0 \qquad (6.109)$$

Show the former equations have a discontinuity at $\lambda = 0$, while the latter do not. Hint: Take the limit as $\lambda \to 0$ and use L'Hospital's Rule.

9. Explain the advantage of the following transformation for $\lambda \neq 0$ proposed by Bickel and Doksum in 1981.

$$f_t(y) \;=\; \frac{|y|^\lambda \text{sign}(y) - 1}{\lambda} \qquad (6.110)$$

6.13 Regression with Categorical Variables

An ANalysis of COVAriance (ANCOVA) model may be developed using the `RegressionCat` class. This type of model comes into play when input variables are mixed, i.e., some are (i) *continuous/ordinal*, while others are (ii) *categorical/binary*. The main difference between the two types of variables is type (i) variables define the notion of less than (<), while variables of type (ii) do not. Also, the expected value means much less for type (ii) variables, e.g., what is the expected value of English, French and Spanish? If we encode a language variable x_j as 0, 1 or 2 for English, French and Spanish, respectively, and half of a group speaks English with the rest speaking Spanish, then the expected value would be French. Worse, if the encoding changes, so does the expected value.

6.13.1 Handling Categorical Variables

Binary Variables

In the *binary case*, when a variable x_j may take on only two distinct values, e.g., Red or Black, then it may simply be encoded as 0 for Red and 1 for Black. Therefore, a single zero-one, encoded/dummy variable x_j, can be used to distinguish the two cases. For example, when $x_j \in \{Red, Black\}$, it would be replaced by one encoded/dummy variable, x_{j0} as shown in Table 6.11.

Table 6.8: Encoding of a Binary Variable

x_j	**encoded** x_j	**dummy** x_{j0}
Red	0	0
Black	1	1

Categorical Variables

For the more general *categorical case*, when the number distinct values for a variable x_j is greater than two, simply encoding the j^{th} column may not be ideal. Instead multiple dummy variables should be used. The number of dummy variables required is one less than the number of distinct values n_{dv}. In one hot encoding, the number of dummy variables may be equal to the n_{dv}, however, this will produce a singular expanded data matrix X, i.e., perfect multi-collinearity (see the exercises).

First, the categorical variable x_j may be encoded using integer values as follows:

$$\text{encoded } \; x_j \; = \; 0, 1, \ldots, n_{dv} - 1$$

Next, for categorical variable x_j, create $n_{dv} - 1$ dummy variables $\{x_{jk} | k = 0, \ldots, n_{dv} - 2\}$ and use the following loop to set the value for each dummy variable.

```
   for k <- 0 until n_dv - 1 do x_jk = is (x_j == k+1)
```

where $\mathbb{1}_{\{c\}}$ is the indicator function that returns 1 when the condition c evaluates to true and 0 otherwise (the `is` function in ScalaTion). In this way, $x_j \in \{English, French, German, Spanish\}$ would be replaced by three dummy variables, x_{j0}, x_{j1} and x_{j2}, as shown in Table 6.11.

Table 6.9: Conventional Dummy Encoding of a Categorical Variable

x_j	**encoded** x_j	**dummy** x_{j0}, x_{j1}, x_{j2}
English	0	0, 0, 0
French	1	1, 0, 0
German	2	0, 1, 0
Spanish	3	0, 0, 1

Unfortunately, for the conventional encoding of a categorical variable, a dummy variable column will be identical to its square, which will result in singular matrix for `quadratic` regression. One solution is to exclude dummy variables in the column expansion done by `quadratic`. Alternatively, a more robust encoding such as the one given in Table 6.10 may be used.

Table 6.10: Robust Dummy Encoding of a Categorical Variable

x_j	**encoded** x_j	**dummy** x_{j0}, x_{j1}, x_{j2}
English	0	1, 1, 1
French	1	2, 1, 1
German	2	1, 2, 1
Spanish	3	1, 1, 2

`map2Int` Method

Conversion from strings to an integer encoding can be accomplished using the `map2Int` method in the `VectorS` class within the `scalation.mathstat` package. It converts a `VectorS` into a `VectorI` by mapping each distinct value in `VectorS` into a distinct numeric integer value, returning the new vector and the bidirectional mapping, e.g., `VectorS ("A", "B", "C", "A", "D")` will be mapped to `VectorI (0, 1, 2, 0, 3)`. Use the `from` method in `BiMap` to recover the original string.

```
    def map2Int: (VectorI, BiMap [String, Int]) =
        val map   = new BiMap [String, Int] ()
        var count = 0
        for i <- indices if ! (map contains (v(i))) do
            map   += v(i) -> count
            count += 1
        end for
        val vec = VectorI (for i <- indices yield map(v(i)))
        (vec, map)
    end map2Int
```

The vector of encoded integers `vec` can be made into a matrix using `MatrixI (vec)`. To produce the dummy variable columns the `dummyVars` function within the `RegressionCat` companion object may be called. See the first exercise for an example.

Multi-column expansion may done by the caller in cases where there are few categorical variables, by expanding the input data matrix before passing it to the Regression class. The expansion occurs automatically when the `RegressionCat` class is called. This class performs the expansion and then delegates to the work to the `Regression` class.

Before continuing the discussion of the `RegressionCat` class, a restricted form is briefly discussed.

6.13.2 ANOVA

An ANalysis Of VAriance (ANOVA) model may be developed using the `ANOVA1` class. This type of model comes into play when all input/predictor variables are categorical/binary. One-way Analysis of Variance allows only one binary/categorical treatment variable and is framed in ScalaTion using General Linear Model notation and supports the use of one binary/categorical treatment variable t. For example, the treatment variable t could indicate the type of fertilizer applied to a field.

The `ANOVA1` class in ScalaTion only supports one categorical variable, so in general, $\mathbf{x}$ consists of $n_{dv} - 1$ dummy variables d_k for $k \in \{1, n_{dv} - 1\}$

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 \,+\, b_1 d_1 \,+\, \ldots \,+\, b_l d_l \,+\, \epsilon \tag{6.111}$$

where $l = n_{dv} - 1$ and ϵ represents the residuals (the part not explained by the model). The dummy variables are binary and are used to determine the level/type of a categorical variable. See `http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf`.

In ScalaTion, the `ANOVA1` class is implemented using regular multiple linear regression. A data/input matrix X is built from columns corresponding to levels/types for the treatment vector $\mathbf{t}$. As with multiple linear regression, the $\mathbf{y}$ vector holds the response values. Multi-way Analysis of Variance may be performed using the more general `RegressionCat` class.

6.13.3 RegressionCat Implementation

When there is only one categorical/binary variable, $\mathbf{x}$ consists of the usual $k = n - 1$ continuous variables x_j. Assuming there is a single categorical variable, call it t, it will need to be expanded into $n_{dv} - 1$ additional dummy variables.

$$t \quad \text{expands to} \quad \mathbf{d} \;=\; [d_0, \ldots, d_l] \qquad \text{where} \;\; l = n_{dv} - 2$$

Therefore, the regression equation becomes the following:

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 \,+\, b_1 x_1 \,+\, \ldots \,+\, b_k x_k \,+\, b_{k+1} d_0 \,+\, \ldots \,+\, b_{k+l} d_l \,+\, \epsilon \tag{6.112}$$

The dummy variables are binary (or shifted binary) and are used to determine the level of a categorical variable. See `http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx`.

In general, there may be multiple categorical variables and an expansion will be done for each such variable. Then the data for continuous variable are collected into matrix $X_{\_}$ and the values for the categorical variables are collected into matrix T.

In ScalaTion, RegressionCat is implemented using regular multiple linear regression. An augmented data/input matrix X is build from $X_{\_}$ corresponding to the continuous variables with additional columns corresponding to the multiple levels for columns in the treatment matrix T. As with multiple linear regression, the $\mathbf{y}$ vector holds the response values.

6.13.4 RegressionCat Class

Class Methods:

```
@param x_      the data/input matrix of continuous variables
@param t       the treatment/categorical variable matrix
@param y       the response/output vector
@param fname_  the feature/variable names (defaults to null)
@param hparam  the hyper-parameters (defaults to Regression.hp)

class RegressionCat (x_ : MatrixD, t: MatrixI, y: VectorD,
                     fname_ : Array [String] = null,
                     hparam: HyperParameter = Regression.hp)
      extends Regression (x_ ++^ RegressionCat.dummyVars (t), y, fname_, hparam)
         with ExpandableVariable:

def expand (zt: VectorD, nCat: Int = t.dim2): VectorD =
def predict_ex (zt: VectorD): Double = predict (expand (zt))
```

6.13.5 Exercises

1. **Mapping Strings to Integers**. Use the map2Int method in the VectorS class within ScalaTion's scalation.mathstat package to convert the given strings into encoded integers. Turn this vector into a matrix and pass it into the dummyVars function to produce the dummy variable columns. Print out the values xe, xm and xd.

```
val x1 = VectorS ("English", "French", "German", "Spanish")
val (xe, map) = x1.map2Int                          // map strings to integers
val xm = MatrixI (xe)                               // form a matrix from vector
val xd = RegressionCat.dummyVars (xm)               // make dummy variable columns
```

   Add code to recover the string values from the encoded integers using the returned map.

2. Compare the results of using the RegressionCat class versus the Regression class for the following dataset.

```
// 6 data points:         one   x_1    x_2
val x = MatrixD ((6, 3), 1.0, 36.0,   66.0,                   // 6-by-3 matrix
                         1.0, 37.0,   68.0,
                         1.0, 47.0,   64.0,
                         1.0, 32.0,   53.0,
                         1.0, 42.0,   83.0,
                         1.0,  1.0, 101.0)
val t = MatrixI ((6, 1), 1, 1, 2, 2, 3, 3)                    // treatments levels
val y = VectorD (745.0, 895.0, 442.0, 440.0, 643.0, 1598.0)   // response vector
val z = VectorD (1.0, 20.0, 80.0, 2)                          // new instance

println (s"x = $x")
println (s"t = $t")
println (s"y = $y")
```

```

    val xt = x ++^ t                                                // combine x and t

    banner ("Regression Model")
    val reg = new Regression (xt, y)                                // treated as ordinal
    println (s"xt = $xt")
    reg.trainNtest ()()                                             // train and test the model
    println (reg.summary ())                                        // parameter statistics
    val yp = reg.predict (z)
    println (s"predict ($z) = $yp")

    banner ("RegressionCat Model")
    val mod = new RegressionCat (x, t, y)                           // treated as categorical
    println (s"xt = $mod.getX}")
    mod.trainNtest ()()                                             // train and test the model
    println (mod.summary ())                                        // parameter statistics
    val ze = VectorD (1.0, 20.0, 80.0, 2, 1)                        // expanded vector
    assert (ze == mod.expand (z))

    println (s"predict ($ze)    = $mod.predict (ze)}")
    println (s"predict_ex ($z) = $mod.predict_ex (z)}")
```

3. Test the `RegressionCat` class on the AutoMPG dataset using the `RegressionCat.apply` function.

```
    val mod = RegressionCat (oxr, y, 6, oxr_fname)
```

The `apply` method creates a `RegressionCat` object from a single data matrix, splitting it into regular and categorical matrices based on the value of `nCat`.

```
    @param x        the data/input matrix of continuous variables
    @param y        the response/output vector
    @param nCat     the index at which the categorical variables start
    @param fname    the feature/variable names
    @param hparam   the hyper-parameters

    def apply (xt: MatrixD, y: VectorD, nCat: Int, fname: Array [String] = null,
               hparam: HyperParameter = Regression.hp): RegressionCat =
        val x = xt(?, 0 until nCat)
        val t: MatrixI = xt(?, nCat until xt.dim2)
        new RegressionCat (x, t, y, fname, hparam)
    end apply
```

4. There is problem called, "too many dummy variables". What is this problem and when does it become significant?

5. To reduce the number of dummy variables and thereby the number of columns in a data matrix, depending on the application, it may make sense to combine or fuse similar levels. Suppose a dataset has `state` as categorical variable with 50 possible values. Rather than converting this into 49 new dummy variable columns, consider a way of grouping based on (1) proximity/geography or (2) similarity/common characteristics.

6. For each string (or word) **One-Hot Encoding** will introduce a column for each distinct string. The column that is hot (1) directly indicates what the string is, (e.g., $[0, 1, 0, 0]$ represents French).

Table 6.11: One-Hot Encoding of a Categorical Variable

x_j	**encoded** x_j	x_{j0}, x_{j1}, x_{j2}, x_{j3}
English	0	1, 0, 0 0
French	1	0, 1, 0 0
German	2	0, 0, 1 0
Spanish	3	0, 0, 1 1

Although this scheme is simple, consider the following equation.

$$x_{j0} + x_{j1} + x_{j2} + x_{j3} \ = \ 1$$

Is this a problem? Explain.

6.14 Weighted Least Squares Regression

The `RegressionWLS` class supports weighted multiple linear regression. In this case, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ...x_k]$.

6.14.1 Model Equation

As before the model/regression equation is

$$y \; = \; \mathbf{b} \cdot \mathbf{x} + \epsilon \; = \; b_0 \; + \; b_1 x_1 \; + \; \ldots \; + \; b_k x_k + \epsilon \tag{6.113}$$

where ϵ represents the residuals (the part not explained by the model). Under multiple linear regression, the parameter vector $\mathbf{b}$ is estimated using matrix factorization with the Normal Equations.

$$(X^{\intercal} X)\mathbf{b} \; = \; X^{\intercal}\mathbf{y}$$

Let us look at the error vector $\boldsymbol{\epsilon} = \mathbf{y} - X\mathbf{b}$ in more detail. A basic assumption is that $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 I)$. i.e., each error is Normally distributed with mean 0 and variance σ^2. If this is violated substantially, the estimate for the parameters $\mathbf{b}$ may be less accurate than desired. One way this can happen is that the variance changes $\epsilon_i \sim N(0, \sigma_i^2)$. This is called *heteroscedasticity* (or *heteroskedasticity*) and it would imply that certain instances (data points) would have greater influence $\mathbf{b}$ than they should. The problem can be corrected by weighting each instance by the inverse of its residual/error variance.

$$w_i \; = \; \frac{1}{\sigma_i^2} \tag{6.114}$$

This begs the question on how to estimate the residual/error variance σ_i^2.

6.14.2 Root Absolute Deviation

There are many ways to try estimate the residual/error variance. One way by performing unweighted regression of $\mathbf{y}$ onto X to obtain the error vector $\boldsymbol{\epsilon}$. It is used to compute a Root Absolute Deviation (RAD) vector $\mathbf{r}$.

$$\mathbf{r} \; = \; \sqrt{|\boldsymbol{\epsilon}|} \tag{6.115}$$

In ScalaTion, RAD is computed by the `rad` method.

```
    @param x   the input/data m-by-n matrix
    @param y   the response/output m-vector

    def rad (x: MatrixD, y: VectorD): VectorD =
        val ols_y = new Regression (x, y)                  // run OLS on original data
        ols_y.train ()                                     // train the model on y
        val e = y - ols_y.predict (x)                      // deviation/error vector
        e.map ((ei) => sqrt (abs (ei)))                    // root absolute deviations (RAD's)
    end rad
```

A simple approach would be to make the weight w_i inversely proportional to r_i.

$$w_i \; = \; \frac{1}{r_i}$$

More commonly, a second unweighted regression is performed, regressing $\mathbf{r}$ onto X to obtain the predictions $\hat{\mathbf{r}}$. The predicted RAD's may be more smooth and less likely to be zero. See Exercise 1 for a comparison or the two methods `setWeights0` (uses r_i) and `setWeights` (uses $\hat{r}_i$).

$$w_i \;=\; \frac{1}{\hat{r}_i} \tag{6.116}$$

See [127] for additional discussion concerning how to set weights. These weights can be used to build a diagonal weight matrix W that factors into the Normal Equations

$$X^{\intercal} W X \mathbf{b} \;=\; X^{\intercal} W \mathbf{y} \tag{6.117}$$

In ScalaTion, this is accomplished by computing a weight vector $\mathbf{w}$ and taking its square root $\boldsymbol{\omega} = \sqrt{\mathbf{w}}$. The data matrix X is then re-weighted by pre-multiplying it by $\boldsymbol{\omega}$ (`rtW` in the code), as if it is a diagonal matrix `rtW *~: x`. The response vector $\mathbf{y}$ is re-weighted using vector multiplication `rtW * y`. The re-weighted matrix and vector are passed into the `Regression` class, which solves for the parameter vector $\mathbf{b}$.

In summary, Weighted Least-Squares (WLS) is accomplished by re-weighting and then using Ordinary Least Squares (OLS). See `http://en.wikipedia.org/wiki/Least_squares#Weighted_least_squares`.

6.14.3 `RegressionWLS` Class

Class Methods:

```
    @param x       the data/input m-by-n matrix
                       (augment with a first column of ones to include intercept in model)
    @param y       the response/output m vector
    @param fname_  the feature/variable names (defaults to null)
    @param w       the weight vector (if null, compute in companion object)
    @param hparam  the hyper-parameters (defaults to Regression.hp)

    class RegressionWLS (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                         private var w: VectorD = null,
                         hparam: HyperParameter = Regression.hp)
          extends Regression ({ setWeights (x, y, w); reweightX (x, w) },
                              reweightY (y, w), fname_, hparam):
    def weights: VectorD = w
    override def diagnose (y: VectorD, yp_ : VectorD, w_ : VectorD = w): VectorD =
    override def train (x_ : MatrixD = getX, y_ : VectorD = getY): Unit = super.train (x_,
    y_)
    override def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def trainNtest (x_ : MatrixD = getX, y_ : VectorD = getY)
    override def validate (rando: Boolean = true, ratio: Double = 0.2)
                          (idx : IndexedSeq [Int] =
                           testIndices ((ratio * y.dim).toInt, rando)): VectorD =
```

6.14.4 Exercises

1. The `setWeights0` method used actual RAD's rather than predicted RAD's used by the `setWeights` method. Compare the two methods of setting the weights on the following dataset.

```
    // 5 data points:         one    x_1     x_2
    val x = MatrixD ((5, 3), 1.0, 36.0,   66.0,                 // 5-by-3 matrix
                             1.0, 37.0,   68.0,
                             1.0, 47.0,   64.0,
                             1.0, 32.0,   53.0,
                             1.0,  1.0, 101.0)
    val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0)
    val z = VectorD (1.0, 20.0, 80.0)
```

   Try the two methods on other datasets and discuss the advantages and disadvantages.

2. Explain why r_i or $\hat{r}_i$ can serve as replacements for the residual/error variance σ_i^2.

3. As Weighted Least Squares (WLS) reduces the contribution of instances with large (predicted) residuals, one might expect MSE, $RMSE$, and R^2 to be worse, but MAE to be better than for Ordinary Least Squares (OLS). Check this on multiple datasets.

4. Show that re-weighting the data matrix X and the response vector $\mathbf{y}$ and solving for the parameter vector $\mathbf{b}$ in the standard Normal Equations $(X^\intercal X)\mathbf{b} = X^\intercal \mathbf{y}$ gives the same result as not re-weighting and solving for the parameter vector $\mathbf{b}$ in the Weighted Normal Equations $X^\intercal W X \mathbf{b} = X^\intercal W \mathbf{y}$.

5. Given an error vector $\boldsymbol{\epsilon}$, what does its covariance matrix $\mathbb{C}[\boldsymbol{\epsilon}]$ represent? How can it be estimated? What are its diagonal elements?

6. When the non-diagonal elements are non-zero, it may be useful to consider using Generalized Least Squares (GLS). What are the trade-offs of using this more complex technique?

6.15 Polynomial Regression

The `PolyRegression` class supports polynomial regression. In this case, $\mathbf{x}$ is formed from powers of a single parameter t, $[1,\, t,\, t^2,\, \ldots,\, t^k]$.

6.15.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation

$$y \;=\; \mathbf{b}\cdot\mathbf{x} + \epsilon \;=\; b_0 \,+\, b_1 t \,+\, b_2 t^2 \,+\, \ldots \,+\, b_k t^k \,+\, \epsilon \tag{6.118}$$

where ϵ represents the residuals (the part not explained by the model). Such models are useful when there is a nonlinear relationship between a response and a predictor variable, e.g., y may vary quadratically with t.

A training set now consists of two vectors, one for the m-vector $\mathbf{t}$ and one for the m-vector $\mathbf{y}$. An easy way to implement polynomial regression is to expand each t value into an $\mathbf{x}$ vector to form a data/input matrix X and pass it to the `Regression` class (multiple linear regression). The columns of data matrix X represent powers of the vector $\mathbf{t}$.

$$X \;=\; \left[\mathbf{1},\, \mathbf{t},\, \mathbf{t}^2,\, \ldots,\, \mathbf{t}^k\right] \tag{6.119}$$

In ScalaTion the vector $\mathbf{t}$ is expanded into a matrix X before calling `Regression`. The number of columns in matrix X is the order k plus 1 for the intercept.

```scala
    val x = new MatrixD (t.dim, 1 + k)
    for i <- t.indices do x(i) = expand (t(i))
    val mod = new Regression (x, y)
```

The `expand` method in the `PolyRegression` class calls the `forms` function in the `PolyRegression` object that takes a 1-vector and computes the values for all of its polynomial forms/terms, returning them as a vector.

```scala
    @param v   the 1-vector (e.g., i-th row of t) for creating forms/terms
    @param k   number of features/predictor variables (not counting intercept) = 1
    @param nt  the number of terms

    def forms (v: VectorD, k: Int, nt: Int): VectorD =
        val t = v(0)
        VectorD (for j <- 0 until nt yield t~^j)
    end forms
```

The code `t~^j` takes t to the j^{th} power.

Although ScalaTion support using polynomials with high orders, there is a danger in coincidental fits and wild extrapolations. Figure 6.12 shows the curves for various polynomials: `quadratic`, `cubic` and `quartic` functions.

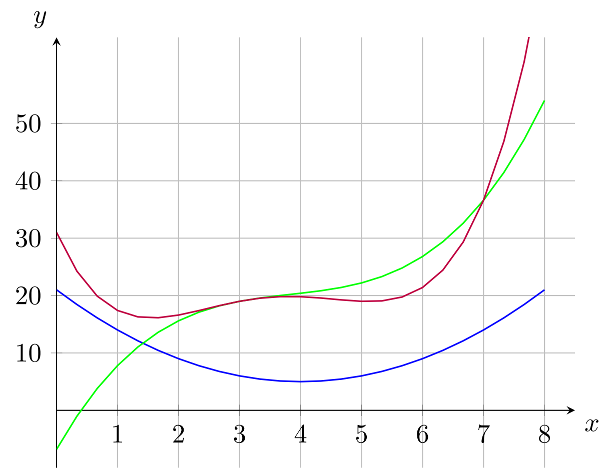


Figure 6.12: quadratic (blue), cubic (green), quartic (purple)

6.15.2 `PolyRegression` Class

Class Methods:

```
    @param t        the initial data/input m-by-1 matrix: t_i expands to
                        x_i = [1, t_i, t_i^2, ... t_i^k]
    @param y        the response/ouput vector
    @param ord      the order (k) of the polynomial (max degree)
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to PolyRegression.hp)

    class PolyRegression (t: MatrixD, y: VectorD, ord: Int, fname_ : Array [String] = null,
                          hparam: HyperParameter = PolyRegression.hp)
          extends Regression (PolyRegression.allForms (t, ord), y, fname_, hparam):

    def expand (z: VectorD): VectorD = PolyRegression.forms (z, n0, nt)
    def predict (z: Double): Double = predict_ex (VectorD (z))
    def predict_ex (z: VectorD): Double = predict (expand (z))
```

Unfortunately, when the order of the polynomial k get moderately large, the multi-collinearity problem can become severe. In such cases it is better to use *orthogonal polynomials* rather than regular polynomials [169]. This is done in ScalaTion by using the `PolyORegression` class.

6.15.3 `PolyORegression` Class

Class Methods:

```
    @param t        the initial data/input m-by-1 matrix: t_i expands to
                        x_i = [1, t_i, t_i^2, ... t_i^k]
    @param y        the response/ouput vector
    @param ord      the order (k) of the polynomial (max degree)
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameter (defaults to PolyRegression.hp

    class PolyORegression (t: MatrixD, y: VectorD, ord: Int, fname_ : Array [String] = null,
                           hparam: HyperParameter = PolyRegression.hp)
          extends Regression (PolyRegression.allForms (t, ord), y, fname_, hparam):

    def expand (z: VectorD): VectorD = PolyORegression.forms (z, n0, nt)
    def orthoVector (v: VectorD): VectorD =
    def predict (z: Double): Double = predict_ex (VectorD (z))
    def predict_ex (z: VectorD): Double = predict (orthoVector (expand (z)))
```

6.15.4 Exercises

1. Generate two vectors **t** and **y** as follows.

```
    val noise = Normal (0.0, 100.0)
    val t     = VectorD.range (0, 100)
    val y     = new VectorD (t.dim)
    for i <- 0 until 100 do y(i) = 10.0 - 10.0 * i + i~^2 + i * noise.gen
```

   Test `new PolyRegression (t, y, order)` for various orders and factorization techniques, e.g., reset hyper-parameter `hp`.

```
    val hp = new HyperParameter; hp += ("factorization", "Fac_Cholesky", "Fac_Cholesky"
    )
    hp("factorization") = "Fac_QR"
```

   Test for multi-collinearity using the correlation matrix and vif.

2. Test `new PolyORegression (t, y, order)` for various orders and factorization techniques. It will use orthogonal polynomials to be used instead of simple polynomials. Again, test for multi-collinearity using the correlation matrix and vif.

3. Work day traffic data has two peaks, one for the morning rush and one for the late afternoon rush. What polynomial function of time could match these characteristics. Collect traffic data and use `PolyRegression` to model the data. Use the lowest order polynomial that provides a reasonable fit.

6.16 Trigonometric Regression

The `TrigRegression` class supports trigonometric regression. In this case, $\mathbf{x}$ is formed from trigonometric functions of a single parameter t, $[1, \sin(\omega t), \cos(\omega t), \ldots, \sin(k\omega t), \cos(k\omega t)]$.

A periodic function can be expressed as linear combination of trigonometric functions (sine and cosine functions) of increasing frequencies. Consequently, if the data points have a periodic nature, a trigonometric regression model may be superior to alternatives.

6.16.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation

$$y \;=\; \mathbf{b}\cdot\mathbf{x} + \epsilon \;=\; b_0 + b_1\sin(\omega t) + b_2\cos(\omega t) + \ldots,\ b_{2k-1}\sin(k\omega t) + b_{2k}\cos(k\omega t) + \epsilon \tag{6.120}$$

where ω is the base angular displacement in radians (e.g., π) and ϵ represents the residuals (the part not explained by the model).

A training set now consists of two vectors, one for the m-vector $\mathbf{t}$ and one for the m-vector $\mathbf{y}$. As was done for polynomial regression, an easy way to implement trigonometric regression is to expand each t value into an $\mathbf{x}$ vector to form a data/input matrix X and pass it to the `Regression` class (multiple linear regression). The columns of data matrix X represent sines and cosines at at multiple harmonic frequencies of the vector t.

$$X \;=\; [\mathbf{1},\ \sin(\omega\mathbf{t}),\ \cos(\omega\mathbf{t}),\ \sin(2\omega\mathbf{t}),\ \cos(2\omega\mathbf{t}),\ \ldots,\ \sin(k\omega\mathbf{t}),\ \cos(k\omega\mathbf{t})] \tag{6.121}$$

For a model with k harmonics (maximum multiplier of ωt), the data matrix can be formed as follows:

```scala
val x = new MatrixD (t.dim, 1 + 2 * k)
for i <- t.indices do x(i) = expand (t(i))
val mod = new Regression (x, y)
```

6.16.2 `TrigRegression` Class

Class Methods:

```scala
@param t        the initial data/input m-by-1 matrix: t_i expands to x_i
@param y        the response/ouput vector
@param ord      the order (k), maximum multiplier in the trig function (kwt)
@param fname_   the feature/variable names (defaults to null)
@param hparam   the hyper-parameters (defaults to Regression.hp)

class TrigRegression (t: MatrixD, y: VectorD, ord: Int, fname_ : Array [String] = null,
                      hparam: HyperParameter = Regression.hp)
      extends Regression (TrigRegression.allForms (t, ord), y, fname_, hparam):

def expand (z: VectorD): VectorD = TrigRegression.forms (z, n0, nt, w)
def predict (z: Double): Double = predict_ex (VectorD (z))
def predict_ex (z: VectorD): Double = predict (expand (z))
```

6.16.3 Exercises

1. Create a noisy cubic function and test how well `TrigRegression` can fit the data for various values of k (harmonics) generated from this function.

```
    val noise = Normal (0.0, 10000.0)
    val t     = VectorD.range (0, 100)
    val y     = new VectorD (t.dim)
    for i <- 0 until 100 do
        val x = (i - 40)/2.0
        y(i) = 1000.0 + x + x*x + x*x*x + noise.gen
    end for
```

2. Make the noisy cubic function periodic and test how well `TrigRegression` can fit the data for various values of k (harmonics) generated from this function.

```
    val noise = Normal (0.0, 10.0)
    val t     = VectorD.range (0, 200)
    val y     = new VectorD (t.dim)
    for i <- 0 until 5 do
        for j <- 0 until 20 do
            val x = j - 4
            y(40*i+j) = 100.0 + x + x*x + x*x*x + noise.gen
        end for
        for j <- 0 until 20 do
            val x = 16 - j
            y(40*i+20+j) = 100.0 + x + x*x + x*x*x + noise.gen
        end for
    end for
```

3. Is the problem of multi-collinearity an issue for Trigonometric Regression?

4. How does Trigonometric Regression relate to Fourier Series?

Chapter 7

Classification

When the output/response y is defined on small domains (categorical response), e.g., $y \in \mathbb{B}$ or

$$y \;\in\; \mathbb{Z}_k \;=\; \{0, 1, \ldots, k-1\} \tag{7.1}$$

the problem shifts from prediction to classification. This facilitates giving the response meaningful class names, e.g., low-risk, medium-risk and high-risk. However, when the response is discrete, but unbounded (e.g, Poisson Regression), or ordinal (e.g., the number of states voting for a particular political party) the problem can be considered to be a prediction problem.

$$y \;=\; f(\mathbf{x};\; \mathbf{b}) + \epsilon \tag{7.2}$$

As with Regression in continuous domains, some of the modeling techniques in this chapter will focus on estimating the conditional expectation of y given $\mathbf{x}$.

$$y \;=\; \mathbb{E}\left[y|\mathbf{x}\right] + \epsilon \tag{7.3}$$

$$\hat{y} \;=\; \mathbb{E}\left[y|\mathbf{x}\right] \tag{7.4}$$

Others will focus on maximizing the conditional probability of y given $\mathbf{x}$, i.e., finding the conditional mode.

$$\hat{y} \;=\; \text{argmax}\; P(y|\mathbf{x}) \;=\; \mathbb{M}\left[y|\mathbf{x}\right] \tag{7.5}$$

Rather than find a real number that is the best predictor, one of a set of distinct given values (e.g., 0 (false), 1 (true); negative (-1), positive (1); or low (0), medium (1), high (2)) is chosen. Abstractly, we can label the classes $C_0, C_1, \ldots, C_{k-1}$. In the case of classification, the `train` and `test` methods are still used, but now the `classify` and `predictI` methods supplement the `predict` method.

Let us briefly contrast the two approaches based on the two equations (expectation vs. mode). For simplicity, a selection (not classification) problem is used. Suppose that the goal is to select one of three actors ($y \in \{0, 1, 2\}$) such that they have been successful in similar films, based on characteristics (features) of the films (captured in variables $\mathbf{x}$). From the data, the frequency of success for the actors in similar films has been 20, 0 and 30, respectively. Consequently, the expected value is 1.2 and one might be tempted to select actor 1 (the worst choice). Instead selecting the actor with maximum frequency (and therefore probability) will produce the best choice (actor 2).

7.1 Classifier

The `Classifier` trait provides a common framework for several classifiers such as `NaiveBayes`.

7.1.1 `Classifier` Trait

Trait Methods:

```
    @param x        the input/data m-by-n matrix
    @param y        the response/output m-vector (class values)
    @param fname    the feature/variable names (if null, use x_j s)
    @param k        the number of classes (categorical response values)
    @param cname    the names/labels for each class
    @param hparam   the hyper-parameters for the model

    trait Classifier (x: MatrixD, y: VectorI, protected var fname: Array [String],
                      k: Int = 2, protected var cname: Array [String] = null,
                      hparam: HyperParameter)
          extends Model:

    def shift2zero (z: MatrixD): Unit =
    def vc_fromData (z: MatrixD): VectorI =
    def getX: MatrixD = x
    def getY: VectorI = y
    def getFname: Array [String] = fname
    def numTerms: Int = getX.dim2
    def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def train (x_ : MatrixD, y_ : VectorD): Unit = train (x_, y_.toInt)
    def train2 (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD)
    def test (x_ : MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def trainNtest (x_ : MatrixD = x, y_ : VectorI = y)
    def predictI (z: VectorI): Int = p_y.argmax ()
    def predictI (z: VectorD): Int = p_y.argmax ()
    def predict (z: VectorD): Double = predictI (z.toInt)
    def predictI (x_ : MatrixD): VectorI =
    def lpredictI (z: VectorI): Int = ???               // only needed for some classifiers
    def lpredictI (z: VectorD): Int = ???               // only needed for certain
    classifiers
    def classify (z: VectorI): (Int, String, Double) =
    def classify (z: VectorD): (Int, String, Double) =
    def lclassify (z: VectorI): (Int, String, Double) =
    def hparameter: HyperParameter = hparam
    def parameter: VectorD = p_y
    def residual: VectorI = e
    override def report (ftVec: VectorD): String =
    def buildModel (x_cols: MatrixD): Classifier = null
    def selectFeatures (tech: SelectionTech, idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean
     = true):
    def forwardSel (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal): BestStep =
    def forwardSelAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
    def backwardElim (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal, first: Int
     = 1): BestStep =
    def backwardElimAll (idx_q: Int = QoF.rSqBar.ordinal, first: Int = 1, cross: Boolean =
    true):
    def stepRegressionAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
    def vif (skip: Int = 1): VectorD =
    inline def testIndices (n_test: Int, rando: Boolean): IndexedSeq [Int] =
    def validate (rando: Boolean = true, ratio: Double = 0.2)
                 (idx : IndexedSeq [Int] =
                  testIndices ((ratio * y.dim).toInt, rando)): VectorD =
    def crossValidate (k: Int = 5, rando: Boolean = true): Array [Statistic] =
```

For modeling, a user chooses one the of classes extending the trait `Classifier` (e.g., `DecisionTree_ID3`) to instantiate an object. Next the `train` method would be typically called.

```
    @param x_  the training/full data/input matrix (defaults to full x)
    @param y_  the training/full response/output vector (defaults to full y)

    def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
```

This implementation simply computes the class/prior frequencies (`nu_y`) and probabilities (`p_y`). This works for a simple model such as `NullModel`, but needs to be be overridden for most models. The `test` method is abstract and thus must be defined in all implementing classes.

```
    @param x_  the testing/full data/input matrix (defaults to full x)
    @param y_  the testing/full response/output vector (defaults to full y)

    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD)
```

While the modeling techniques in the last chapter focused on *minimizing errors*, the focus in this chapter will be on *minimizing incorrect classifications*. Generally, this is done by dividing a dataset up into a *training dataset* and *test dataset*. A technique for utilizing one dataset to produce single training and test datasets is called *validation*.

```
    def validate (rando: Boolean = true, ratio: Double = 0.2)
                 (idx: IndexedSeq [Int] =
                       testIndices (rando, (ratio * y.dim).toInt)): VectorD =
        val (x_e, x_, y_e, y_) = TnT_Split (x, y, idx)       // Test-n-Train Split

        train (x_, y_)                                       // train model on training set
        val qof = test (x_e, y_e)._2                         // test on test-set and get QoF
        if qof(QoF.sst.ordinal) <= 0.0 then                  // requires variation test-set
            flaw ("validate", "chosen testing set has no variability")
        end if
        qof
    end validate
```

Another technique for utilizing one dataset to produce multiple training and test datasets is called *cross-validation*. As discussed in the Model Validation section in the Prediction chapter, k-fold cross-validation is a useful general purpose strategy for examining the quality of a model. It performs k iterations of training (`train` method) and testing (`test` method).

```
    def crossValidate (k: Int = 5, rando: Boolean = true): Array [Statistic] =
        if k < MIN_FOLDS then flaw ("crossValidate", s"k = $k must be at least $MIN_FOLDS")
        val stats   = FitC.qofStatTable                      // create table - QoF measures
        val fullIdx = if rando then permGen.igen              // permuted indices
                      else VectorI.range (0, y.dim)           // ordered indices
        val sz      = y.dim / k                               // size of each fold
        val ratio   = 1.0 / k                                 // fraction used for testing

        for fold <- 0 until k do
            banner (s"crossValidate: fold $fold: train-test sizes = (${y.dim - sz}, $sz)")
            val idx = fullIdx (fold * sz until (fold+1) * sz).toMuIndexedSeq
            val qof = validate (rando, ratio)(idx)
            debug ("crossValidate", s"fold $fold: qof = $qof")
            if qof(QoF.sst.ordinal) > 0.0 then
                for q <- qof.indices do stats(q).tally (qof(q))
            end if
        end for
        stats
    end crossValidate
```

Setting `rando` to `true` is usually preferred, as it randomizes the instances selected for the test dataset, so that patterns coincidental to the index are broken up.

Once a model/classifier has been sufficiently trained and tested, it is ready to be put into practice on new data via the `classify` method.

```
    @param z  the data vector to classify

    def classify (z: VectorI): (Int, String, Double) =
        val best = predictI (z)                          // class with the highest probability
        val prob = if p_yz != null then p_yz(best)        // posterior probability
                   else if p_y != null then p_y(best)    // prior probability
                   else NO_DOUBLE                        // nothing applicable
        (best, cname(best), prob)                        // return best class, its name, prob
    end classify
```

It calls the model specific `predictI` method and returns its value as well the corresponding class label and its relative probability.

The `Classifier` trait also provides methods to determine the *value count* (`vc`) for the features/variables. A method to shift values in a vector toward zero by subtracting the minimum value. It has base implementations for `test` methods. Finally, several methods for features selection are provided. ScalaTion currently uses forward selection, backward elimination, and stepwise refinement algorithms for feature selection.

7.2 Quality of Fit for Classification

The `FitC` trait provides methods for computing Quality of Fit (QoF) measures for classifiers. Many are derived from the so-called *Confusion Matrix* that keeps track of correct and incorrect classifications. In ScalaTion when $k = 2$, the confusion matrix C is configured as follows:

$$\begin{bmatrix} c_{00} = tn & c_{01} = fp \\ c_{10} = fn & c_{11} = tp \end{bmatrix}$$

where tn and tp are true negatives and positives, respectively, and fn and fp are false negatives and positives, respectively. The selected row is determined by the actual value `y_`, while the selected column is determined by the predicted value `yp`. The confusion matrix is computed using the `confusion` method.

```
@param y_  the actual class values/labels for full (y) or test (y_e) dataset
@param yp  the precicted class values/labels

def confusion (y_ : VectorI, yp: VectorI): MatrixI =
    cmat.setAll (0)                                        // clear confusion matrix
    for i <- y_.indices    do cmat(y_(i), yp(i)) += 1      // increment counts
    for i <- cmat.indices  do rsum(i) = cmat(i).sum.toInt  // compute row sums
    for j <- cmat.indices2 do csum(j) = cmat(?, j).sum.toInt  // compute column sums
    tcmat += cmat
    p_r_s ()                                          // precision, recall and specificity
    cmat
end confusion
```

The first column indicates the prediction/classification is negative (no or 0), while the second column indicates it is positive (yes or 1). The first letter ('f' or 't') indicates whether the classification is correct (true) or not (false). After calling the `confusion` method, the `summary` should be called. To see values for the basic QoF measured from `FitM` the `diagnose` method may be called instead of tt confusion. The `FitC` trait includes several methods for directly computing QoF measures as well.

7.2.1 `FitC` Trait

Trait Methods:

```
@param y  the vector of actual class values/labels
@param k  the number distinct class values/labels

trait FitC (y: VectorI, k: Int = 2)
      extends FitM:

def clearConfusion (): Unit = tcmat.setAll (0)
def total_cmat (): MatrixI = { val t = tcmat.copy; tcmat.setAll (0); t }
override def diagnose (y_ : VectorD, yp: VectorD, w: VectorD = null): VectorD =
def diagnose (y_ : VectorI, yp: VectorI): VectorD =
def confusion (y_ : VectorI, yp: VectorI): MatrixI =
def contrast (yp: VectorI, y_ : VectorI = y): Unit =
def p_r_s (): Unit =
def pseudo_rSq: Double =  1.0 - sse / sst
def tn_fp_fn_tp (con: MatrixI = cmat): (Double, Double, Double, Double) =
def accuracy: Double = cmat.trace / cmat.sum.toDouble
```

```
    def f1_measure (p: Double, r: Double): Double = 2.0 * p * r / (p + r)
    def f1v: VectorD = (pv * rv * 2.0) / (pv + rv)
    def kappa: Double =
    def fit: VectorD =
    def fitMicroMap: Map [String, VectorD] =
    def help: String = FitC.help
    def fitLabel_v: Seq [String] = FitC.fitLabel_v
    def summary (x_ : MatrixD, fname: Array [String], b: VectorD, vifs: VectorD = null):
    String =
```

7.3 Null Model

The NullModel class implements a simple Classifier suitable for discrete input data. Corresponding to the Null Model in the Prediction chapter, one could imagine estimating probabilities for outcomes of a random variable y. Given an instance, this random variable indicates the classification or decision to be made. For example, it may be used for a decision on whether or not to grant a loan request. The model may be trained by collecting a training dataset. Probabilities may be estimated from data stored in an m-dimensional response/classification vector $\mathbf{y}$ within the training dataset. These probabilities are estimated based on the frequency ν (nu in the code) with which each class value occurs.

$$\nu(\mathbf{y} = c) \;=\; |\{i \,|\, y_i = c\}| \;=\; m_c \tag{7.6}$$

The right hand side is simply the size of the set containing the instance/row indices where $y_i = c$ for $c = 0, \ldots, k-1$. The probability that random variable y equals c can be estimated by the number of elements in the vector $\mathbf{y}$ where y_i equals c divided by the total number of elements.

$$P(y = c) \;=\; \frac{\nu(\mathbf{y} = c)}{m} \;=\; \frac{m_c}{m} \tag{7.7}$$

Exercise 1 below is the well-known toy classification problem on whether to play tennis ($y = 1$) or not ($y = 0$) based on weather conditions. Of the 14 days ($m = 14$), tennis was not played on 5 days and was played on 9 days, i.e.,

$$P(y = 0) = \frac{5}{14} \;\text{ and }\; P(y = 1) = \frac{9}{14}$$

This information, class frequencies and class probabilities, can be placed into a *Class Frequency Vector* (CFV) as shown in Table 7.1 and

Table 7.1: Class Fequency Vector

y	**0**	**1**
	5	9

a *Class Probability Vector* (CPV) as shown in Table 7.9.

Table 7.2: Class Probability Vector

y	**0**	**1**
	5/14	9/14

Picking the maximum probability case, one should always predict that tennis will be played, i.e., $\hat{y} = 1$.

This modeling technique should outperform purely random guessing, since it factors in the relative frequency with which tennis is played. As with the NullModel for prediction, more sophisticated modeling techniques should perform better than this NullModel for classification. If they are unable to provide higher accuracy, they are of questionable value.

7.3.1 NullModel Class

Class Methods:

```
    @param y         the response/output m-vector (class values)
    @param k         the number of distinct values/classes
    @param cname_    the names for all classes

    class NullModel (y: VectorI, k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"))
          extends Classifier (null, y, null, k, cname_, null)   // no x matrix, no hparam
             with FitC (y, k):

    def test (x_ : MatrixD = null, y_ : VectorI = y): (VectorI, VectorD) =
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = py, vifs: VectorD = null): String =
```

The `NullModel` is so simple that is just uses the `train` method from `Classifier`.

Typically, one dataset is divided into a training dataset and testing dataset. For example, 80% may be used for training (estimating probabilities) with the remaining 20% used for testing the accuracy of the model. Furthermore, this is often done repeatedly as part of a cross-validation procedure.

7.3.2 Exercises

1. The `NullModel` classifier can be used to solve problems such as the one below. Given the Outlook, Temperature, Humidity, and Wind determine whether it is more likely that someone will (1) or will not (0) play tennis. The data set is widely available on the Web. If is also available in `scalation.modeling.classifying.Example_PlayTennis`. Use the `NullModel` for classification and evaluate its effectiveness using cross-validation.

   The `Example_PlayTennis` object is used to test all integer based classifiers.

```
// The Example_PlayTennis object is the well-known classification problem on whether to
    play tennis
// based on given weather conditions.  Applications may need to slice xy.
//      val x = xy(?, 0 until 4)         // columns 0, 1, 2, 3
//      val y = xy(?, 4)                 // column 4
// @see euclid.nmu.edu/~mkowalcz/cs495f09/slides/lesson004.pdf

object Example_PlayTennis:

    // dataset ----------------------------------------------------------------
    // x0: Outlook:     Rain (0),   Overcast (1), Sunny (2)
    // x1: Temperature: Cold (0),   Mild (1),     Hot (2)
    // x2: Humidity:    Normal (0), High (1)
    // x3: Wind:        Weak (0),   Strong (1)
    // y:  the response/classification decision
    // variables/features:      x0     x1     x2     x3    y      // combined matrix
    val xy = MatrixI ((14, 5),  2,     2,     1,     0,    0,     // day  1
                                2,     2,     1,     1,    0,     // day  2
                                1,     2,     1,     0,    1,     // day  3
                                0,     1,     1,     0,    1,     // day  4
                                0,     0,     0,     0,    1,     // day  5
                                0,     0,     0,     1,    0,     // day  6
                                1,     0,     0,     1,    1,     // day  7
                                2,     1,     1,     0,    0,     // day  8
                                2,     0,     0,     0,    1,     // day  9
                                0,     1,     0,     0,    1,     // day 10
                                2,     1,     0,     1,    1,     // day 11
                                1,     1,     1,     1,    1,     // day 12
                                1,     2,     0,     0,    1,     // day 13
                                0,     1,     1,     1,    0)     // day 14

    val fn = Array ("Outlook", "Temp", "Humidity", "Wind")         // feature names
    val cn = Array ("No", "Yes")                                   // class names for y
    val k  = cn.size                                               // number of classes

    val x = xy.not(?, 4)                                           // columns 0, 1, 2, 3
    val y = xy(?, 4)                                               // column 4

end Example_PlayTennis
```

2. Build a `NullModel` classifier for the Breast Cancer problem (data in `breast-cancer.arff` file).

7.4 Naïve Bayes

The `NaiveBayes` class implements a Naïve Bayes (NB) Classifier suitable for discrete input data. A Bayesian Classifier is a special case of a Bayesian Network where one of the random variables is distinguished as the basis for making decisions, call it random variable y, the class variable. The `NullModel` ignores weather conditions which are the whole point of the `Example_PlayTennis` exercise. For Naïve Bayes, weather conditions (or other data relevant to decision making) are captured in an n-dimensional vector of random variables.

$$\mathbf{x} = [x_0, \dots, x_{n-1}], \tag{7.8}$$

For the `Example_PlayTennis` problem, $n = 4$ where x_0 is Outlook, x_1 is Temperature, x_2 is Humidity, and x_3 is Wind. The decision should be conditioned on the weather, i.e., rather than computing $P(y)$, we should compute $P(y|\mathbf{x})$. Bayesian classifiers are designed to find the class (value for random variable y) that maximizes the conditional probability of y given $\mathbf{x}$.

It may be complex and less robust to estimate $P(y|\mathbf{x})$ directly. Often it is easier to examine the conditional probability of $\mathbf{x}$ given y. This answers the question of how likely it is that the input data comes from a certain class y. Flipping the perspective can be done using Bayes Theorem.

$$P(y|\mathbf{x}) \;=\; \frac{P(\mathbf{x}|y)\,P(y)}{P(\mathbf{x})} \tag{7.9}$$

Since the denominator is the same for all y, it is sufficient to maximize the right hand side of the following proportionality statement.

$$P(y|\mathbf{x}) \;\propto\; P(\mathbf{x}|y)\,P(y) \tag{7.10}$$

Notice that the right hand side is the joint probability of all the random variables.

$$P(\mathbf{x}, y) \;=\; P(\mathbf{x}|y)\,P(y) \tag{7.11}$$

One could in principle represent the joint probability $P(\mathbf{x}, y)$ or the conditional probability $P(\mathbf{x}|y)$ in a matrix. Unfortunately, with 30 binary random variables, the matrix would have over one billion rows and exhibit issues with sparsity. Bayesian classifiers will factor the probability and use multiple matrices to represent the probabilities.

7.4.1 Factoring the Probability

A Bayesian classifier is said to be naïve, when it is assumed that the x_j's are sufficiently uncorrelated to factor $P(\mathbf{x}|\, y)$ into the product of their conditional probabilities (independence rule).

$$P(\mathbf{x}|\, y) \;=\; \prod_{j=0}^{n-1} P(x_j|y) \tag{7.12}$$

Research has shown that even though the assumption that given response/class variable y, the x-variables are independent is often violated by a dataset, Naïve Bayes still tends to perform well [212]. Substituting this factorization into the joint probability formula yields

$$\boxed{P(\mathbf{x}, y) \;=\; P(y) \prod_{j=0}^{n-1} P(x_j|y)} \tag{7.13}$$

The classification problem then is to find the class value for y that maximizes this probability, i.e., let $\hat{y}$ be the argmax of the product of the class probability $P(y)$ and all the conditional probabilities $P(x_j|y)$. The argmax is the value in the domain $D_y = \{0, \ldots, k-1\}$ that maximizes the probability.

$$\hat{y} \;=\; \underset{y \in \{0,\ldots,k-1\}}{\text{argmax}} \; P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{7.14}$$

7.4.2 Estimating Conditional Probabilities

For Integer-based classifiers $x_j \in \{0, 1, ..., vc_j - 1\}$ where vc_j is the value count for the j^{th} variable/feature (i.e., the number of distinct values). The Integer-based Naïve Bayes classifier is trained using an m-by-n data matrix X and an m-dimensional classification vector $\mathbf{y}$. Each data vector/row in the matrix is classified into one of k classes numbered $0, 1, \ldots, k-1$. The frequency or number of instances where column vector $\mathbf{x}_{:j} = h$ and vector $\mathbf{y} = c$ is as follows:

$$\nu(\mathbf{x}_{:j} = h, \, \mathbf{y} = c) \;=\; |\{i \,|\, x_{ij} = h, \, y_i = c\}| \tag{7.15}$$

The conditional probability for random variable x_j given random variable y can be estimated as the ratio of two frequencies.

$$P(x_j = h \,|\, y = c) \;=\; \frac{\nu(\mathbf{x}_{:j} = h, \, \mathbf{y} = c)}{\nu(\mathbf{y} = c)} \tag{7.16}$$

In other words, the conditional probability is the ratio of the joint frequency count for a given h and c divided by the class frequency count for a given c. These frequency counts can be collected into

- *Joint Frequency Matrices/Tables* (JFTs) and
- a *Class Frequency Vector* (CFV).

From these, it is straightforward to compute

- *Conditional Probability Matrices/Tables* (CPTs) and
- a *Class Probability Vector* (CPV).

`Example_PlayTennis` Problem

For the `Example_PlayTennis` problem, Figure 7.1 shows the Tree for a Naïve Bayes Classifier. The edges from classification variable y to the feature variables x_j are shown in black.

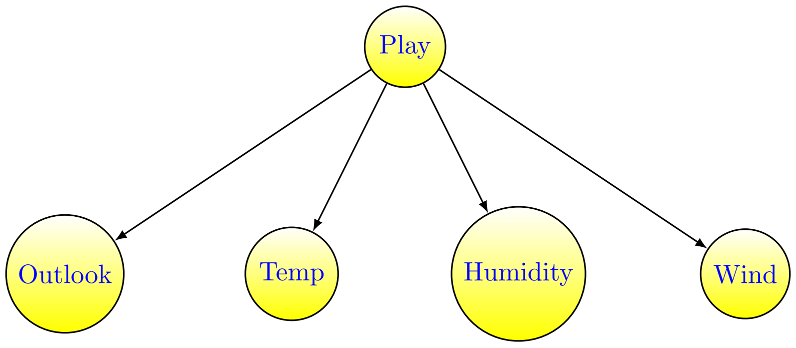


Figure 7.1: Naïve Bayes Classifier: y = Play, x_0 = Outlook, x_1 = Temp, x_2 = Humidity, x_3 = Wind

Table 7.3: JFT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash y$	**0**	**1**
0	2	3
1	0	4
2	3	2

For the this problem, the *Joint Frequency Matrix/Table* (JFT) for `Outlook` random variable x_0 is shown in Table 7.3.

$$\nu(\mathbf{x}_{\text{-}0} = h, \, \mathbf{y} = c) \quad \text{for} \quad h \in \{0, 1, 2\}, \; c \in \{0, 1\}$$

The column sums in the above matrix are 5 and 9, respectively. The corresponding *Conditional Probability Matrix/Table* (CPT) for random variable x_0, i.e., $P(x_0 = h \,|\, y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum.

Table 7.4: CPT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash y$	**0**	**1**
0	2/5	3/9
1	0	4/9
2	3/5	2/9

Continuing with the `Example_PlayTennis` problem, the *Joint Frequency Matrix/Table* for `Wind` random variable x_3 is shown in Table 7.5.

$$\nu(\mathbf{x}_{\text{-}3} = h, \, \mathbf{y} = c) \quad \text{for} \quad h \in \{0, 1\}, \; c \in \{0, 1\}$$

As expected, the column sums in the above matrix are again 5 and 9, respectively. The corresponding *Conditional Probability Matrix/Table* for random variable x_0, i.e., $P(x_0 = h \,|\, y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum as shown in table 7.6

Table 7.5: JFT for $\mathbf{x}_{\text{-}3}$

$x_3\backslash y$	**0**	**1**
0	2	6
1	3	3

Table 7.6: CPT for $\mathbf{x}_{\text{-}3}$

$x_3\backslash y$	**0**	**1**
0	2/5	6/9
1	3/5	3/9

Similar matrices/tables can be created for the other random variables: Temperature x_1 and Humidity x_2.

7.4.3 Laplace Smoothing

When there are several possible class values, a dataset may exhibit zero instances for a particular class. This will result in a zero in the CFV vector and cause a *divide-by-zero* error when computing CPTs. One way to avoid the divide-by-zero, is to add one ($m_e = 1$) fake instance for each class, guaranteeing no zeros in the CFV vector. If m-estimates are used, the conditional probability is adjusted slightly as follows:

$$P(x_j = h \,|\, y = c) \;=\; \frac{\nu(\mathbf{x}_{:j} = h,\, \mathbf{y} = c) \,+\, m_e/vc_j}{\nu(\mathbf{y} = c) \,+\, m_e} \tag{7.17}$$

where m_e is the parameter used for the m-estimate. The term added to the numerator, takes the one (or m_e) instance(s) and adds uniform probability for each possible values for x_j of which there are vc_j of them. Table 7.7 shows the result of adding 1/3 in the numerator and 1 in the denominator, (e.g., for h = 0 and c = 0, (2 + 1/3)/(5 + 1) = 7/18).

Table 7.7: CPT for $\mathbf{x}_{\text{-}0}$ with $m_e = 1$

$x_0\backslash y$	**0**	**1**
0	7/18	10/30
1	1/18	13/30
2	10/18	7/30

Or in decimal,

Another problem is when a conditional probability in a CPT is zero. If any CPT has a zero element, the corresponding product for the column (where the CPV and CPTs are multiplied) will be zero no matter how high the other probabilities may be. This happens when the frequency count is zero in the corresponding JFT (see element (1, 0) in Table 7.3). The question now is whether this is due to the combination of $x_0 = 1$

Table 7.8: CPT for $\mathbf{x}_{:0}$ with $m_e = 1$ in decimal

$x_0 \backslash y$	**0**	**1**
0	0.3889	0.3333
1	0.0556	0.4333
2	0.5556	0.2333

and $y = 0$ being highly unlikely, or that the dataset is not large enough to exhibit this combination. Laplace smoothing guards against this problem as well.

Other values (including fractional values) may be used for m_e as well. ScalaTion uses a small value for the default m_e to reduce the distortion of the CPTs.

7.4.4 Table Storage

The values within the class probability table and the conditional probability tables are assigned by the `train` method. The Conditional Probability Tables (CPTs) require three dimensional storage that can be accomplished with tensors/hyper-matrices, or arrays of matrices. In ScalaTion, ragged tensors `RTensorD` are used for storing CPTs (`p_Xy`).

```
    private var p_Xy: RTensorD = null        // Conditional Probability Tables (CPTs)
```

The dimensions are as follows: for each matrix, the number of rows is vc_j, the value count for feature x_j, and the number of columns is k, the number of class values; while the number of matrices is n, the number of x-random variables (features).

For the `Example_PlayTennis` problem, $\mathbf{vc} = [3, 3, 2, 2]$, $k = 2$, and $n = 4$.
Note that the alternative of storing the tables in a rectangular hyper-matrix or tensor would result in a dimensionality of 4-by-3-by-2, but this would in general be wasteful of space. Each variable only needs space for the values it allows, as indicated by the value counts $\mathbf{vc} = [3, 3, 2, 2]$. The user may specify the optional `vc` parameter in the constructor call. If the `vc` parameter is unspecified, then ScalaTion uses the `vc_fromData` method to determine the value counts from the training data. In some cases, the test data may include a value unseen in the training data. Currently, ScalaTion requires the user to pass `vc` into the constructor in such cases.

7.4.5 The `train` Method

The `train` method first calculates frequencies: The class frequency for each class value, e.g., (0 or 1) or label ("no", "yes") and a Joint Frequency Table (JFT) for each feature x_j.

```
    @param x_  the training/full data/input matrix (defaults to full x)
    @param y_  the training/full response/output vector (defaults to full y)

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
        super.train (x_, y_)                                  // set class freq and prob
        val nu_Xy = RTensorD.freq (x_, vc, y, k)              // JFT for each xj
        p_Xy      = cProb_Xy (x_, y_, nu_Xy)                  // CPT for each xj
    end train
```

The `freq` method for computing Joint Frequency Tables (JFTs) is defined in the `RTensorD` object. The `cprob_Xy` method defined in the `NaiveBayes` class divides each JFT by class frequencies `nu_y` to obtain the corresponding CPT. Laplace smoothing adds `me_v(j)` to the numerator and `me` to denominator.

```
    @param x_     the integer-valued data vectors stored as rows of a matrix
    @param y_     the class vector, where y(i) = class for row i of the matrix x, x(i)
    @param nu_Xy  the joint frequency of X and y for each feature xj and class value

    def cProb_Xy (x_ : MatrixD, y_ : VectorI, nu_Xy: RTensorD): RTensorD =
        val pXy = new RTensorD (x_.dim2, vc, k)
        for j <- x_.indices2; xj <- 0 until vc(j) do
            pXy(j, xj) = (nu_Xy(j, xj) + me_v(j)) / (nu_y + me)     // CPTs
        end for
        pXy
    end cProb_Xy
```

7.4.6 The `test` Method

The `test` method calls `predictI` to produce a predicted integer value for each data point in `x_`. These predicted values `yp` are then passed into the `diagnose` method along with actual values `y_`.

```
    @param x_  the testing/full data/input matrix (defaults to full x)
    @param y_  the testing/full response/output vector (defaults to full y)

    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
        val yp  = predictI (x_)                               // predicted classes
        val qof = diagnose (y_.toDouble, yp.toDouble)         // diagnose from y vs yp
        (yp, qof)
    end test
```

7.4.7 The `predictI` Method

A new instance can now be classified by simply matching its values with with those in the class probability table and conditional probability tables and multiplying all the entries. This is done for all k class values and the class with the highest product is chosen.

```
    @param z  the new vector to predict

    override def predictI (z: VectorI): Int =
        p_yz = p_y.copy                                 // start with class probabilities
        for j <- z.indices do p_yz *= p_Xy(j, z(j))     // multiply P(X_j = z_j | y = c)
        p_yz.argmax ()                                  // return class with highest prob
    end predictI
```

Note that the `classify` method defined in the `Classifier` trait calls `predictI` returning its value as well as its class label and relative probability.

7.4.8 The `lpredictI` Method

In situations where there are many variables/features the product calculation may underflow, due to multiplying several small probabilities together. An alternative calculation would be to add the log of the probabilities.

$$\log P(\mathbf{z}, y) \;=\; \log P(y) + \sum_{j=0}^{n-1} \log P(z_j | y) \tag{7.18}$$

The `lpredict` may be used in this situation.

```
@param z  the new vector to predict

override def lpredictI (z: VectorI): Int =
    p_yz = plog (p_y)                                 // start with class plogs
    for j <- z.indices do p_yz += plog (p_Xy(j, z(j))) // add plog P(X_j = z_j | y = c)
    p_yz.argmin ()                                    // return class with lowest plog
end lpredictI
```

The `plog` function computes the positive log of the probability.

7.4.9 Feature Selection

When there are many possible variables or features available for inclusion into a model, support for feature selection becomes useful. Suppose that x_1 and x_2 are not considered useful for classifying a day as to its suitability for playing tennis. For $z = [2, 1]$, i.e,. $z_0 = 2$ and $z_3 = 1$, the two relative probabilities are the following:

Table 7.9: Joint Data-Class Probability

P	**0**	**1**
y	5/14	9/14
z_0	3/5	2/9
z_3	3/5	3/9
$\mathbf{z}, y$	9/70	1/21

The two probabilities are approximately 0.129 for c = 0 (Do not Play) and 0.0476 for c = 1 (Play). The higher probability is for $c = 0$.

To perform feature selection in a systematic way ScalaTion provides several methods for feature selection including forward selection, backward elimination and stepwise refinement. (see the `Classifier` trait).

7.4.10 `NaiveBayes` Class

Class Methods:

```
@param x       the input/data m-by-n matrix
@param y       the class vector, where y(i) = class for row i of matrix x
@param fname_  the names of the features/variables (defaults to null)
@param k       the number of classes (defaults to 2)
@param cname_  the names of the classes
@param vc      the value count (number of distinct values) for each feature
```

```
    @param hparam  the hyper-parameters

class NaiveBayes (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                  k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                  protected var vc: VectorI = null,
                  hparam: HyperParameter = NaiveBayes.hp)
      extends Classifier (x, y, fname_, k, cname_, hparam)
         with FitC (y, k):

def getCPTs: RTensorD = p_Xy
override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
def cProb_Xy (x_ : MatrixD, y_ : VectorI, nu_Xy: RTensorD): RTensorD =
def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
override def predictI (z: VectorI): Int =
override def lpredictI (z: VectorI): Int =
override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                      b_ : VectorD = p_y, vifs: VectorD = null): String =
```

7.4.11 Exercises

1. With Laplace smoothing set at 1 fake instance, complete the *Example_PlayTennis* problem given in this section by creating CPTs for random variables x_1 and x_2 and then computing the relative probabilities for $z = [2, 2, 1, 1]$. Hint: The relative probabilities are [0.03376, 0.004288]. What decision would the classifier make?

2. Use SCALATION's integer-based `NaiveBayes` class to build a classifier for the `Example_PlayTennis` problem.

```
import scalation.modeling.classifying.Example_PlayTennis._
banner ("Play Tennis Example")
println (s"xy = $xy")                           // combined data matrix [x | y]

val mod = NaiveBayes (xy, fname)()              // create a classifier
mod.trainNtest ()()                             // train and test the classifier
println ("CPTs = " + mod.getCPTs)               // print conditional prob tables
println (mod.summary ())                        // summary statistics

val z = VectorI (2, 2, 1, 1)                    // new data vector to classify
banner (s"Classify $z")
println (s"Use mod to classify ($z) = ${mod.classify (z)}")
```

3. Compare the confusion matrix, accuracy, precision and recall of `NaiveBayes` on the full dataset to that of `NullModel`.

4. For the `Example_PlayTennis` problem, compare the accuracy of `NaiveBayes` to that of `NullModel` using

   (a) 80-20% train-test split validation

   (b) 5-fold cross-validation (cv).

```
    println ("mod test accu = " + mod.validate ()())   // out-of-sample validation
    FitM.showQofStatTable (mod.crossValidate ())       // 5-fold cross-validattion
```

Note: `validate` and `crossValidate` are better suited to larger datasets (the `Example_PlayTennis` toy dataset only has 14 instances).

5. Compare the confusion matrix, accuracy, precision and recall of `RoundRegression` on the full dataset to that of `NullModel`.
6. Perform feature selection on the `Example_PlayTennis` problem. Which feature/variable is removed from the model, first, second and third. Explain the basis for the `featureSelection` method's decision to remove a feature.
7. Use the integer-based `NaiveBayes` class to build a classifier for the Breast Cancer problem (data in `breast-cancer.arff` file). Compare its accuracy to that of `NullModel`.

7.5 Bayes Classifier

The `BayesClassifier` trait provides methods for more advanced Bayesian Classifiers, including calculations of joint probabilities and Conditional Mutual Information (CMI). For data with small value counts, CMI tends to be more useful than correlation for examining dependencies between random variables. More information will be provided in the next section on TAN Bayes.

7.5.1 `BayesClassifier` Trait

Class Methods:

```
    @param k  the number of classes

    trait BayesClassifier (k: Int = 2):

    def jProbXY (x: VectorI, vcx: Int, y: VectorI): MatrixD =
    def jProbXZY (x: VectorI, z: VectorI, vcxz: VectorI, y: VectorI): RTensorD =
    def cmi (x: VectorI, z: VectorI, vcxz: VectorI, y: VectorI): Double =
    def cmiMatrix (x: MatrixD, vc: VectorI, y: VectorI): MatrixD =
```

7.6 Tree Augmented Naïve Bayes

The `TANBayes` class implements a Tree Augmented Naïve (TAN) Bayes Classifier suitable for discrete input data. Unlike Naïve Bayes, a TAN Bayes model can capture more, yet limited dependencies between variables/features. In general, x_j can be dependent on the class y as well as one other variable x_{p_j}. Representing the dependency pattern graphically, y becomes a root node of a Directed Acyclic Graph (DAG), where each node/variable has at most two parents.

Starting with the joint probability defined in the section on Naïve Bayes,

$$P(\mathbf{x}, y) \;=\; P(\mathbf{x}|y)\, P(y)$$

we can obtain a better factored approximation (better than Naïve Bayes) by keeping the most important dependencies amongst the random variables. Each feature x_j, except a selected x-root, x_r, will have one x-parent x_{p_j} in addition to its y-parent, i.e.,

$$x\text{−parent}(x_j) \;=\; x_{p_j} \tag{7.19}$$

The dependency pattern among the x random variables forms a *tree* and this tree augments the Naïve Bayes structure where each x random variable has y as its only parent.

$$P(\mathbf{x}, y) \;=\; P(y) \prod_{j=0}^{n-1} P(x_j | x_{p_j}, y) \tag{7.20}$$

Now, each feature x_j is conditioned on its x-parent x_{p_j} and the class variable y. More precisely, since the root x_r, has no x-parent, it can be factored out as special case.

$$\boxed{P(\mathbf{x}, y) \;=\; P(y) P(x_r|y) \prod_{j\neq r} P(x_j | x_{p_j}, y)} \tag{7.21}$$

Figure 7.2 shows the DAG for a TAN Bayes Classifier. The edges from classification variable y to the feature variables x_j are shown in black, while edges between the feature variables forming the tree are shown in blue.

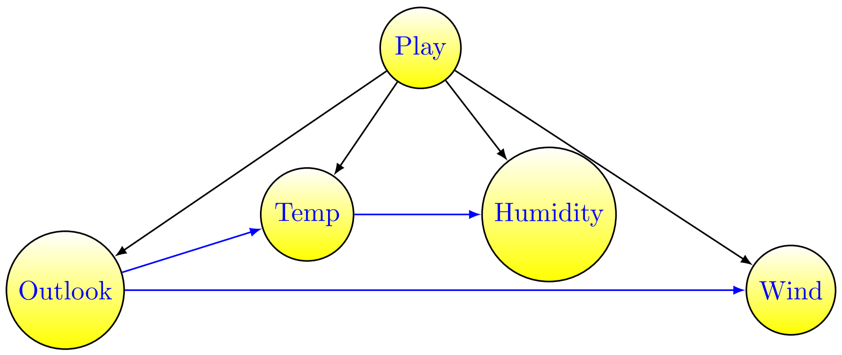


Figure 7.2: TAN Bayes Classifier: y = Play, x_0 = Outlook, x_1 = Temp, x_2 = Humidity, x_3 = Wind

As with Naïve Bayes, the goal is to find an optimal value for the random variable y that maximizes the probability.

$$\hat{y} = \operatorname*{argmax}_{y \in D_y} P(y) P(x_r|y) \prod_{j=0}^{n-1} P(x_j|x_{p_j}, y) \tag{7.22}$$

7.6.1 Structure Learning

Naïve Bayes has a very simple structure that does not require any structural learning. TAN Bayes, on the other hand, requires the tree structure among the x random variables/nodes to be learned. Various algorithms can be used to select the best parent x_{p_j} for each x_j. ScalaTion does this by constructing a maximum spanning tree where the edge weights are Conditional Mutual Information (alternatively correlation).

The Mutual Information (MI) between two random variables x (e.g., x_j) and z (e.g., x_l) is

$$I(x; z) = \sum_x \sum_z p(x, z) \log \frac{p(x, z)}{p(x)p(z)} \tag{7.23}$$

The Conditional Mutual Information (CMI) between two random variables x (e.g., x_j and z (e.g., x_l) given a third random variable y is

$$I(x; z|y) = \sum_y p(y) \sum_x \sum_z p(x, z|y) \log \frac{p(x, z|y)}{p(x|y)p(z|y)} \tag{7.24}$$

It may also be expressed in terms of joint probabilities.

$$I(x; z|y) = \sum_y \sum_x \sum_z p(x, z, y) \log \frac{p(y)p(x, z, y)}{p(x, y)p(z, y)} \tag{7.25}$$

The steps involved in the structure learning algorithm for TAN Bayes are the following:

1. Compute the CMI $I(x_j; x_l|y)$ for all combinations of random variables, $j \neq l$.

2. Build a complete undirected graph with a node for each x_j random variable. The weight on undirected edge $\{x_j, x_l\}$ is its CMI value.

3. Apply a *Maximum Spanning Tree* algorithm (e.g., Prim or Kruskal) to the undirected graphs to create a maximum spanning tree (those $n-1$ edges that (a) connect all the nodes, (b) form a tree, and (c) have maximum cumulative edge weights). Note, ScalaTion's `MinSpanningTree` in the `scalation.graph_db` package can be used with parameter `min = false`.

4. Pick one of the random variables to be the root node x_r.

5. To build the directed tree, start with root node x_r and traverse from there giving each edge directionality as you go outward from the root.

7.6.2 Conditional Probability Tables

For the `Example_PlayTennis` problem limited to two variables, x_0 and x_3, suppose that structure learning algorithm found the x-parents as shown in Table 7.10.

Table 7.10: Parent Table

x_j	x_{p_j}
x_0	x_3
x_3	null

Table 7.11: Extended JFT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash x_3, y$	**0, 0**	**0, 1**	**1, 0**	**1, 1**
0	0	3	2	0
1	0	2	0	2
2	2	1	1	1

In this case, the only modification to the CPV and CPTs from the Naïve Bayes solution, is that the JFT and CPT for x_0 are extended. The extended Joint Frequency Table (JFT) for x_0 is shown in Table 7.11. The column sums are 2, 6, 3, 3, respectively. Again they must add up to same total of 14. Dividing each element in the JFT by its column sum yields the extended Conditional Probability Table (CPT) shown in Table 7.12

Table 7.12: Extended CPT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash x_3, y$	**0, 0**	**0, 1**	**1, 0**	**1, 1**
0	0	1/2	2/3	0
1	0	1/3	0	2/3
2	1	1/6	1/3	1/3

In general for `TANBayes`, the x-root will have a regular CPT, while all other x-variables will have an extended CPT, i.e., the extended CPT for x_j is calculated as follows:

$$P(x_j = h \,|\, x_p = l, y = c) \quad = \quad \frac{\nu(\mathbf{x}_{:j} = h,\, \mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c)}{\nu(\mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c)} \tag{7.26}$$

7.6.3 Smoothing

The analog of Laplace smoothing used in Naïve Bayes is the following.

$$P(x_j = h \,|\, x_p = l, y = c) \quad = \quad \frac{\nu(\mathbf{x}_{:j} = h,\, \mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c) \, + \, m_e/vc_j}{\nu(\mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c) \, + \, m_e} \tag{7.27}$$

In Friedman's paper [50], he suggests using the marginal distribution rather than uniform (as shown above), which results in the following formula.

$$P(x_j = h \,|\, x_p = l, y = c) \;=\; \frac{\nu(\mathbf{x}_{:j} = h,\, \mathbf{x}_{:p} = l,\, \mathbf{y} = c) \,+\, m_e * mp_j}{\nu(\mathbf{x}_{:p} = l,\, \mathbf{y} = c) \,+\, m_e} \tag{7.28}$$

where

$$mp_j \;=\; \frac{\nu(\mathbf{x}_{:j})}{m} \tag{7.29}$$

7.6.4 The train Method

The **train** method computes class probabilities (CPV) and the extended Conditional Probability Tables (CPTs) that are used as the basis for classification.

```
    @param x_  the training/full data/input matrix (defaults to full x)
    @param y_  the training/full response/output vector (defaults to full y)

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
        super.train (x_, y_)                                    // set class freq and prob
        findParent (x_, y_)
        val nu_Xy  = freq_Xy (x_, y_, vc)                       // JFTs
        val nu_Xyp = freq_Xyp (x_, y_, nu_y, nu_Xy)             // extended JFTs
        p_Xyp      = cprob_Xyp (x_, y_, nu_y, nu_Xy, nu_Xyp)   // extended CPTs
    end train
```

7.6.5 The predictI Method

As with NaiveBayes, the predictI method (and thus classify) simply multiplies entries in the CPV and CPTs (all except the root are extended). Again the class with the highest product is chosen.

```
    @param z  the new vector to predict

    override def predictI (z: VectorI): Int =
        p_yz = p_y.copy                                 // start with class probabilities
        for j <- z.indices do                           // P(X_j = z_j | X_p = z_p, y = c)
            val p    = parent(j)                        // parent of xj
            val ecpt = p_Xyp(j)                         // get j-th extended CPT
            if p > -1 then                              // xj has a parent
                p_yz *= ecpt (z(j))(z(p))               // multiply in its (z(j), z(p)) row
            else                                        // xj does not have a parent
                p_yz *= ecpt (z(j))(0)                  // multiply in its z(j) row
            end if
        end for
        p_yz.argmax ()                                  // return class with highest prob
    end predictI
```

7.6.6 TANBayes Class

Class Methods:

```
    @param x       the input/data m-by-n matrix
    @param y       the class vector, where y(i) = class for row i of matrix x
    @param fname_  the names of the features/variables (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the names of the classes
    @param vc       the value count (number of distinct values) for each feature
    @param hparam   the hyper-parameters

    class TANBayes (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                    k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                    private var vc: VectorI = null, hparam: HyperParameter = BayesClassifier
    .hp)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with BayesClassifier (k, hparam)
             with FitC (y, k):

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def findParent (x_ : MatrixD, y_ : VectorI): Unit =
    def freq_Xyp (x_ : MatrixD, y_ : VectorI, nu_y: VectorD, nu_Xy: Array [MatrixD]):
    def cprob_Xyp (x_ : MatrixD, y_ : VectorI, nu_y: VectorD, nu_Xy: Array [MatrixD],
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorI): Int =
    override def lpredictI (z: VectorI): Int =
    def printCPTs (): Unit =
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

7.6.7 Exercises

1. Use the Integer-based `TANBayes` to build classifiers for (a) the `Example_PlayTennis` problem and (b) the Breast Cancer problem (data in `breast-cancer.arff` file). Compare its accuracy to that of `NullModel` and `NaiveBayes`.

2. Compare the correlation matrix on X with the corresponding Conditional Mutual Information (CMI) matrix. How well do they capture dependencies. Use multiple datasets.

3. Show that the two formulas (the one using conditional probability and the other using joint probability) for Conditional Mutual Information (CMI) give the samem results.

4. Re-engineer `TANBayes` to use correlation instead of Conditional Mutual Information (CMI). Compare the results with the current `TANBayes` implementation.

5. The `FANBayes` class implements a Forest Augmented Naïve (FAN) Bayes Classifier suitable for discrete input data. It competes with `TANBayes` by allowing multiple trees and is thus more flexible. Compare `FANBayes` and `TANBayes` on multiple datasets.

7.7 Bayesian Network Classifier

A Bayesian Network Classifier [17] is used to classify a discrete input data vector $\mathbf{x}$ by determining which of k classes has the highest Joint Probability of $\mathbf{x}$ and the response/outcome y (i.e., one of the k classes) of occurring.

$$P(y, x_0, x_1, \ldots, x_{n-1}) \tag{7.30}$$

Using the Chain Rule of Probability, the Joint Probability calculation can factored into multiple calculations of conditional probabilities as well as the class probability of the response. For example, given three variables, the joint probability may be factored as follows:

$$P(x_0, x_1, x_2) \;=\; P(x_0)P(x_1|x_0)P(x_2|x_0, x_1) \tag{7.31}$$

Conditional dependencies are specified using a Directed Acyclic Graph (DAG). A feature/variable represented by a node in the network is conditionally dependent on its parents only,

$$\hat{y} \;=\; \underset{y \in D_y}{\text{argmax}}\; P(y) \prod_{j=0}^{n-1} P(x_j|\mathbf{x}_{\mathbf{p}(j)}, y) \tag{7.32}$$

where $\mathbf{x}_{\mathbf{p}(j)}$ is the vector of features/variables that x_j is dependent on, i.e., its parents. In our model, each variable has dependency with the response variable y (a defacto parent). Note, some more general BN formulations do not distinguish one of the variables to be the response y as we do.

Conditional probabilities are recorded in tables referred to as Conditional Probability Tables (CPTs). Each variable will have a CPT and the number of columns in the table is governed by the number of other variables it is dependent upon. If this number is large, the CPT may become prohibitively large.

7.7.1 Network Augmented Naïve Bayes

The `TwoNANBayes` class implements a Network Augmented Naïve (NAN) Bayes Classifier suitable for discrete input data, that is restricted to at most two x-parents. It is a special case of a general Network Augmented Naïve (NAN) Bayes Classifier, also know as a Bayesian Network Classifier.

For `TwoNANBayes` the parents of variable x_j are recoded in a vector $\mathbf{x}_{\mathbf{p}(j)}$ of length 0, 1 or 2. Although the restriction to at most 2 parents might seem limiting, the problem of finding the optimal structure is still NP-hard [28].

7.8 Markov Network

A Markov Network is a probabilistic graphical model where directionality/causality between random variables is not considered, only their bidirectional relationships. In general, let $\mathbf{x}$ be an n-dimensional vector of random variables.

$$\mathbf{x} \;=\; [x_0, \ldots x_{n-1}]$$

Given a data instance $\mathbf{x}$, its likelihood of occurrence is given by the joint probability.

$$P(\mathbf{x} = \mathbf{x})$$

In order to compute the joint probability, it needs to be factored based on *conditional independencies*. These conditional independencies may be illustrated graphically, by creating a vertex for each random variable x_i and letting the structure of the graph reflect the conditional independencies,

$$x_i \perp x_k \,|\, \{x_j\}$$

such that removal of the vertices in the set $\{x_j\}$ will disconnect x_i and x_k in the graph. These conditional independencies may be exploited to factor the joint probability, e.g.,

$$P(x_i, x_k \,|\, \{x_j\}) = P(x_i \,|\, \{x_j\})\, P(x_k \,|\, \{x_j\})$$

When two random variables are directly connected by an undirected edge (denoted $x_i - x_j$) they cannot to separated by removal of other vertices. Together they form an Undirected Graph $G(\mathbf{x}, E)$ where the vertex-set is the set of random variables $\mathbf{x}$ and the edge-set is defined as follows:

$$E \;=\; \{x_i - x_j | \, x_i \text{ and } x_j \text{ are not conditionally independent}\}$$

When the random variables are distributed in space, the Markov Network may from a grid, in which case the network is often referred to as a Markov Random Field (MRF).

7.8.1 Markov Blanket

A vertex in the graph x_i will be conditionally independent of all other vertices, except those in its Markov Blanket. The Markov Blanket for random variable x_i is simply the immediate neighbors of x_i in G:

$$B(x_i) \;=\; \{x_j | \, x_i - x_j \in E\} \tag{7.33}$$

The edges E are selected so that random variable x_i will be conditionally independent of any other $(k \neq i)$ random variable x_k that is not in its Markov blanket.

$$x_i \perp x_k \,|\, B(x_i) \tag{7.34}$$

7.8.2 Factoring the Joint Probability

Factorization of the joint probability is based on the graphical structure of G that reflects the conditional independencies. It has been shown (see the Hammersley-Clifford Theorem) that $P(\mathbf{x})$ may be factored according the set of maximal cliques[1] Cl in graph G.

$$P(\mathbf{x}) \;=\; \frac{1}{Z} \prod_{c \in Cl} \phi_c(\mathbf{x}_c) \tag{7.35}$$

For each clique c in the set Cl, a potential function $\phi_c(\mathbf{x}_c)$ is defined. (Potential functions are non-negative functions that are used in place of marginal/conditional probabilities and need not sum to one; hence the normalizing constant Z).

Suppose a graph $G([x_0, x_1, x_2, x_3, x_4], E)$ has two maximal cliques, $Cl = \{[x_0, x_1, x_2], [x_2, x_3, x_4]\}$ then

$$P(\mathbf{x}) \;=\; \frac{1}{Z}\, \phi_0(x_0, x_1, x_2)\, \phi_1(x_2, x_3, x_4)$$

7.8.3 Exercises

1. Consider the random vector $\mathbf{x} \;=\; [x_0, x_1, x_2]$ with conditional independency

   $$x_0 \perp\; x_1 \,|\, x_2$$

   show that

   $$P(x_0, x_1, x_2) \;=\; P(x_2)\, P(x_0 \,|\, x_2)\, P(x_1 \,|\, x_2)$$

[1] a clique is a set of vertices that are fully connected

7.9 Decision Tree ID3

A Decision Tree (or Classification Tree) classifier [162, 158] will take an input vector $\mathbf{x}$ and classify it, i.e., give one of k class values to y by applying a set of decision rules configured into a tree. Abstractly, the decision rules may be viewed as a function f.

$$y = f(\mathbf{x}) = f(x_0, x_1, \dots, x_{n-1}) \tag{7.36}$$

The `DecisionTreeI_D3` [147] class implements a Decision Tree classifier using the Iterative Dichotomiser 3 (ID3) algorithm. The classifier is trained using an m-by-n data matrix X and a classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, 1, \dots, k-1$. Each column in the matrix represents a x-variable/feature (e.g., Humidity). The value count $\mathbf{vc}$ vector gives the number of distinct values per feature (e.g., 2 for Humidity).

7.9.1 Entropy

In decision trees, the goal is to reduce the disorder in decision making. Assume the decision is of the yes(1)/no(0) variety and consider the following decision/classification vectors: $\mathbf{y} = (1, 1, \dots, 1, 1)$ or $\mathbf{y}' = (1, 0, \dots, 1, 0)$. In the first case all the decisions are yes, while in the second, three are an equal number of yes and no decisions. One way to measure the level of disorder is Shannon entropy. To compute the entropy, first convert the m-dimensional decision/classification vector $\mathbf{y}$ into a k-dimensional probability vector $\mathbf{p}$. The `frequency` and `toProbability` functions in the `Probability` object may be used for this task (see `NullModel` from the last chapter).

For the two cases, $\mathbf{p} = (1, 0)$ and $\mathbf{p}' = (.5, .5)$, so computing the Shannon *entropy* $H(\mathbf{p})$ (see the Probability Chapter), we obtain $H(\mathbf{p}) = 0$ and $H(\mathbf{p}') = 1$. These indicate that there is no disorder in the first case and maximum disorder in the second case.
Entropy is used as measure of the *impurity* of a node (e.g., to what degree is it a mixture of '-' and '+'). For a discussion of additional measures see [158].

7.9.2 Example Problem

Let us consider the Tennis example from `NullModel` and `NaiveBayes` and compute the entropy level for the decision of whether to play tennis. There are 14 days worth of training data see Table 7.13, which indicate that for 9 of the days the decision was yes (play tennis) and for 5 it was no (do not play). Therefore, the entropy (if no features/variables are considered) is

$$H(\mathbf{p}) = H(\frac{5}{14}, \frac{9}{14}) = -\frac{5}{14} log_2(\frac{5}{14}) - \frac{9}{14} log_2(\frac{9}{14}) = 0.9403$$

Recall that the features are Outlook x_0, Temp x_1, Humidity x_2, and Wind x_3. To reduce entropy, find the feature/variable that has the greatest impact on reducing disorder. If feature/variable j is factored into the decision making, entropy is now calculated as follows:

$$\sum_{v=0}^{vc_j - 1} \frac{\nu(\mathbf{x}_{:j} = v\}}{m} H(\mathbf{p}_{\mathbf{x}_{:j}=v}) \tag{7.37}$$

where $\nu(\mathbf{x}_{:j} = v\}$ is the frequency count of value v for column vector $\mathbf{x}_{:j}$ in matrix X. The sum is the weighted average of the entropy over all possible vc_j values for variable j.

Table 7.13: Tennis Example

Day	$\mathbf{x}_{:0}$	$\mathbf{x}_{:1}$	$\mathbf{x}_{:2}$	$\mathbf{x}_{:3}$	$\mathbf{y}$
1	2	2	1	0	0
2	2	2	1	1	0
3	1	2	1	0	1
4	0	1	1	0	1
5	0	0	0	0	1
6	0	0	0	1	0
7	1	0	0	1	1
8	2	1	1	0	0
9	2	0	0	0	1
10	0	1	0	0	1
11	2	1	0	1	1
12	1	1	1	1	1
13	1	2	0	0	1
14	0	1	1	1	0

To see how this works, let us compute new entropy values assuming each feature/variable is used, in turn, as the principal feature for decision making. Starting with feature $j = 0$ (Outlook) with values of Rain (0), Overcast (1) and Sunny (2), compute the probability vector and entropy for each value and weight them by how often that value occurs.

$$\sum_{v=0}^{2} \frac{\nu(\mathbf{x}_{:0} = v)}{m} H(\mathbf{p}_{\mathbf{x}_{:0}=v}) \tag{7.38}$$

For $v = 0$, we have 2 no (0) cases and 3 yes (1) cases $(\mathbf{2}-, \mathbf{3}+)$, for $v = 1$, we have $(\mathbf{0}-, \mathbf{4}+)$ and for $v = 2$, we have $(\mathbf{3}-, \mathbf{2}+)$.

$$\frac{\nu(\mathbf{x}_{:0} = 0)}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=0}) + \frac{\nu(\mathbf{x}_{:0} = 1)}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=1}) + \frac{\nu(\mathbf{x}_{:0} = 2)}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=2})$$

$$\frac{5}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=0}) + \frac{4}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=1}) + \frac{5}{14} H(\mathbf{p}_{\mathbf{x}_{:0}=2})$$

We are left with computing three entropy values:

$$\begin{aligned}
H(\mathbf{p}_{\mathbf{x}_{:0}=0}) &= H(\frac{2}{5}, \frac{3}{5}) = -\frac{2}{5}\, log_2(\frac{2}{5}) - \frac{3}{5}\, log_2(\frac{3}{5}) &= 0.9710 \\
H(\mathbf{p}_{\mathbf{x}_{:0}=1}) &= H(\frac{0}{4}, \frac{4}{4}) = -\frac{0}{4}\, log_2(\frac{0}{4}) - \frac{4}{4}\, log_2(\frac{4}{4}) &= 0.0000 \\
H(\mathbf{p}_{\mathbf{x}_{:0}=2}) &= H(\frac{3}{5}, \frac{2}{5}) = -\frac{3}{5}\, log_2(\frac{3}{5}) - \frac{2}{5}\, log_2(\frac{2}{5}) &= 0.9710
\end{aligned}$$

The weighted average is then 0.6936, so that the drop in entropy (also called information gain) is 0.9403 - 0.6936 = 0.2467. As shown in Table 7.14, the other entropy drops are 0.0292 for Temperature (1), 0.1518 for Humidity (2) and 0.0481 for Wind (3).

Table 7.14: Choices for Principal Feature

j	**Variable/Feature**	**Entropy**	**Entropy Drop**
0	Outlook	0.6936	0.2467
1	Temperature	0.9111	0.0292
2	Humidity	0.7885	0.1518
3	Wind	0.8922	0.0481

Hence, Outlook ($j = 0$) should be chosen as the principal feature for decision making. As the entropy is too high, make a tree with Outlook (0) as the root and make a branch for each value of Outlook: Rain (0), Overcast (1), Sunny (2). Each branch defines a sub-problem.

The resulting tree is shown in Figure 7.3 where the node is designated by the variable (in this case x_0). The edges indicate the values that this variable can take on, while the two numbers $n^- p^+$ indicate the number of negative and positive cases.

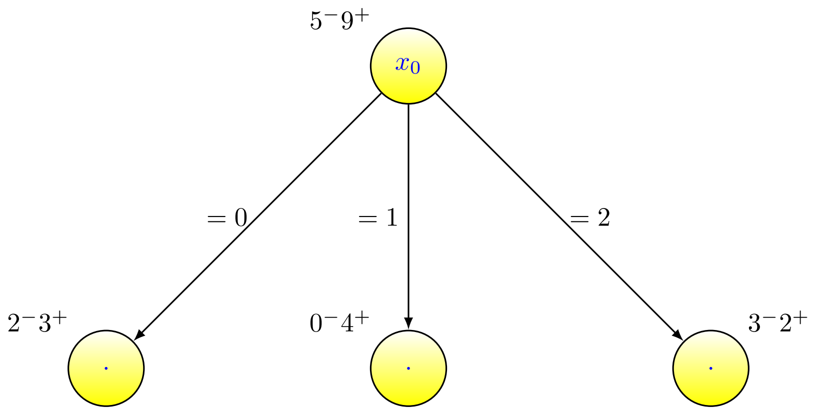


Figure 7.3: Decision Tree for "Play Tennis?" Example

Sub-problem $x_0 = 0$

The sub-problem for Outlook: Rain (0) see Table 7.15 is defined as follows: Take all five cases/rows in the data matrix X for which $\mathbf{x}_{:0} = 0$.
If we select Wind ($j = 3$) as the next variable, we obtain the following cases: For $v = 0$, we have $(\mathbf{0}-, \mathbf{3}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x}_{:3}=0} = (\frac{0}{5}, \frac{3}{5}) \quad H(\mathbf{p}_{\mathbf{x}_{:3}=0}) = 0$$

Table 7.15: Sub-problem for node x_0 and branch 0

Day	$\mathbf{x}_{:1}$	$\mathbf{x}_{:2}$	$\mathbf{x}_{:3}$	$\mathbf{y}$
4	1	1	0	1
5	0	0	0	1
6	0	0	1	0
10	1	0	0	1
14	1	1	1	0

For $v = 1$, we have $(\mathbf{2}-, \mathbf{0}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x}_{:3}=1} \;=\; (\frac{2}{5}, \frac{0}{5}) \qquad H(\mathbf{p}_{\mathbf{x}_{:3}=1}) = 0$$

If we stop expanding the tree at this point, we have the following rules.

```
    if x0 == 0 then
        if x3 == 0 then yes
        if x3 == 1 then no
    if x0 == 1 then yes
    if x0 == 2 then no
```

The overall entropy can be calculated as the weighted average of all the leaf nodes.

$$\frac{3}{14} \cdot 0 + \frac{2}{14} \cdot 0 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot .9710 \;=\; .3468$$

Sub-problem $x_0 = 2$

Note that if $x_0 = 1$, the entropy for this case is already zero, so this node need not be split and remains as a leaf node. There is still some uncertainty left when $x_0 = 2$, so this node may be split. The sub-problem for Outlook: Rain (2) see Table 8.1 is defined as follows: Take all five cases/rows in the data matrix X for which $x_{-0} = 2$.

Table 7.16: Sub-problem for node x_0 and branch 2

Day	$\mathbf{x}_{:1}$	$\mathbf{x}_{:2}$	$\mathbf{x}_{:3}$	$\mathbf{y}$
1	2	1	0	0
2	2	1	1	0
8	1	1	0	0
9	0	0	0	1
11	1	0	1	1

It should be obvious that $\mathbf{y} = \mathbf{1} - \mathbf{x}_{:2}$. For $v = 0$, we have $(\mathbf{0}-, \mathbf{2}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x}_{:2}=0} \;=\; (\frac{0}{5}, \frac{2}{5}) \quad H(\mathbf{p}_{\mathbf{x}_{:3}=0}) = 0$$

For $v = 1$, we have $(\mathbf{3}-, \mathbf{0}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x}_{:2}=1} \;=\; (\frac{3}{5}, \frac{0}{5}) \quad H(\mathbf{p}_{\mathbf{x}_{:3}=0}) = 0$$

At this point, the overall entropy is zero and the decision tree is the following (shown as a pre-order traversal from ScalaTion).

```
Decision Tree:
[ -1 ->  Node (j = 0, nu = VectorI(5, 9), y = 1, leaf = false)
    [ 0 ->  Node (j = 3, nu = VectorI(2, 3), y = 1, leaf = false)
        [ 0 ->  Node (j = 1, nu = VectorI(0, 3), y = 1, leaf = true) ]
        [ 1 ->  Node (j = 1, nu = VectorI(2, 0), y = 0, leaf = true) ]
    ]
    [ 1 ->  Node (j = 1, nu = VectorI(0, 4), y = 1, leaf = true) ]
    [ 2 ->  Node (j = 2, nu = VectorI(3, 2), y = 0, leaf = false)
        [ 0 -> Node (j = 1, nu = VectorI(0, 2), y = 1, leaf = true) ]
        [ 1 -> Node (j = 1, nu = VectorI(3, 0), y = 0, leaf = true) ]
    ]
]
```

The above process of creating the decision tree is done by a recursive, greedy algorithm. As with many greedy algorithms, it does not guarantee an optimal solution.

7.9.3 Early Termination

Producing a complex decision tree with zero entropy may suggest over-fitting, so that a simpler tree may be more robust. One approach would be terminate once entropy decreases to a certain level. One problem with this is that expanding a different branch could have led to a lower entropy with a tree of no greater complexity. Another approach is simply to limit the depth of the tree. Simple decision trees with limited depth are commonly used in Random Forests, a more advanced technique discussed later.

The `DecisionTree` trait provides methods for building decision trees, calculating entropy and recursive methods to support making predictions.

7.9.4 `DecisionTree` Trait

```scala
trait DecisionTree:

def addRoot (r: Node): Unit = root = r
def add (n: Node, v: Int, c: Node): Unit =
def add (n: Node, vc: (Int, Node)*): Unit =
def makeLeaf (n: Node): Unit =
def leafChildren (n: Node): Boolean = n.branch.values.forall (_.leaf)
def candidates: Set [Node] =
def bestCandidate (can: Set [Node]): (Node, Double) =
def calcEntropy (nodes: ArrayBuffer [Node] = leaves): Double =
def predictIrec (z: VectorI, n: Node = root): Int =
def predictIrecD (z: VectorD, n: Node = root): Int =
def printTree (): Unit =
```

The `DecisionTree_ID3` class extends this trait implementing the ID3 algorithm with methods for training, testing, making predictions and produing summary statistics. The `train` method calls the private `buildTree` method that recursively builds the decision tree by expanding nodes until entropy drops to the cutoff threshold or the tree depth is at the specified tree height.

7.9.5 `DecisionTree_ID3` Class

Class Methods:

```
    @param x        the input/data m-by-n matrix with instances stored in rows
    @param y        the response/classification m-vector, where y_i = class for row i
    @param fname_   the name for each feature/variable xj (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the name for each class
    @param hparam   the hyper-parameters for the Decision Tree classifier

    class DecisionTree_ID3 (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                            k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                            hparam: HyperParameter = DecisionTree.hp)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with FitC (y, k)
             with DecisionTree:

    override def parameter: VectorD = VectorD (param)
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorI): Int = predictIrec (z)
    override def predictI (z: VectorD): Int = predictIrecD (z)
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

7.9.6 Pruning

An alternative to early termination is to build a complex tree and then *prune* the tree. Pruning involves selecting a node whose children are all leaves and undoing the split that created the children. Compared to early termination, pruning will take more time to come up with the solution. For the tennis example, pruning could be used to turn node 5 into a leaf node (pruning away two nodes) where the decision would be the majority decision $y = 1$. The entropy for this has already been calculated to be .3468. Instead node 1 could be turned into a leaf (pruning away two nodes). This case is symmetric to the other one, so the entropy would be .3468, but the decision would be $y = 0$. The original ID3 algorithm did not use pruning, but its follow on algorithm C4.5 does (see the next Chapter). The ScalaTion implementation of ID3 does support pruning. The `DecisionTree_ID3wp` class extends `DecisionTree_ID3` with methods for finding candidates for pruning and doing the actual pruning.

7.9.7 `DecisionTree_ID3wp` Class

Class Methods:

```
@param x        the input/data m-by-n matrix with instances stored in rows
@param y        the response/classification m-vector, where y_i = class for row i of
matrix x
@param fname_   the name for each feature/variable xj (defaults to null)
@param k        the number of classes (defaults to 2)
@param cname_   the name for each class
@param hparam   the hyper-parameters for the Decision Tree classifier

class DecisionTree_ID3wp (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                          k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                          hparam: HyperParameter = DecisionTree.hp)
      extends DecisionTree_ID3 (x, y, fname_, k, cname_, hparam):

def prune (nPrune: Int = 1, threshold: Double = 0.98): Unit =
```

7.9.8 Exercises

1. The Play Tennis example (see `NaiveBayes`) can also be analyzed using decisions trees.

```
import Example_PlayTennis._

DecisionTree.hp("height") = 2
val mod = DecisionTree_ID3 (xy, fname)()              // create a classifier
mod.trainNtest ()()                                   // train and test classifier
mod.printTree ()                                      // print the decision tree
println (mod.summary ())                              // summary statistics

val z = VectorI (2, 2, 1, 1)                          // new data vector to
classify
banner (s"Classify $z")
println (s"classify ($z) = ${mod.classify (z)}")

banner ("Validation")
println ("mod test accu = " + mod.validate ()())      // out-of-sample testing
```

   Use `DecisionTree_ID3` to build classifiers for the `Example_PlayTennis` problem. Compare its accuracy to that of `NullModel`, `NaiveBayes` and `TANBayes`.

2. Do the same for the Breast Cancer problem (data in breast-cancer.arff file).

3. For the Breast Cancer problem, evaluate the effectiveness of the `prune` method.

4. Again for the Breast Cancer problem, explore the results for various limitations to the maximum height/depth of tree via the `height` hyper-parameter.

7.10 Hidden Markov Model

A Hidden Markov Model (HMM) provides a natural way to study a system with an internal state and external observations. One could image looking at a flame and judging the temperature (internal state) of the flame by its color (external observation). When this is treated as a discrete problem, an HMM may be used; whereas, as a continuous problem, a Kalman Filter may be used (see the chapter on State Space Models). For HMMs, we assume that the internal state is unknown (hidden), but may be predicted by from the observations.

Consider two discrete-valued, discrete-time stochastic processes. The first process represents the internal state of a system

$$\{x_t : t \in \{0, \dots T-1\}\}$$

while the second process represents corresponding observations of the system

$$\{y_t : t \in \{0, \dots T-1\}\}$$

The internal state influences the observations. In a deterministic setting, one might imagine

$$y_t = f(x_t)$$

Unfortunately, since both x_t and y_t are both stochastic processes, their trajectories need to be described probabilistically. For tractability and because it often suffices, the assumption is made that the state x_t is only significantly influenced by its previous state x_{t-1}.

$$P(x_t | x_{t-1}, x_{t-2}, x_0) = P(x_t | x_{t-1})$$

In other words, the transitions from state to state are governed by a *discrete-time Markov chain* and characterized by *state-transition probability matrix* $A = [a_{ij}]$, where

$$a_{ij} = P(x_t = j | x_{t-1} = i)$$

The influence of the state upon the observation is also characterized by *emission probability matrix* $B = [b_{kj}]$, where

$$b_{jk} = P(y_t = k | x_t = j)$$

is the conditional probability of the observation being k when the state is j. This represents a second simplifying assumption that the observation is effectively independent of prior states or observations. To predict the evolution of the system, it is necessary to characterize the initial state of the system x_0.

$$\pi_j = P(x_t = j)$$

The dynamics of an HMM model is thus represented by two matrices A, B and an *intial state probability vector* $\boldsymbol{\pi}$.

7.10.1 Example Problem

Let the system under study be a lane of road with a sensor to count traffic flow (number of vehicles passing the sensor in a five minute period). As a simple example, let the state of the road be whether or not there is an accident ahead. In other words, the state of road is either 0 (**N**o-accident) or 1 (**A**ccident). The only information available is the traffic counts and of course historical information for training an HMM model. Suppose the chance of an accident ahead is 10%.

$$\boldsymbol{\pi} \;=\; [0.9, 0.1]$$

From historical information, two transition probabilities are estimated: the first is for the transition from no accident to accident which is 20%; the second from accident to no-accident state (i.e., the accident has been cleared) which is 50% (i.e., probability of one half that the accident will be cleared by the next time increment). The number of states $n = 2$. Therefore, the state-transition probability matrix A is

$$\begin{bmatrix} 0.8 & 0.2 \\ 0.5 & 0.5 \end{bmatrix}$$

As A maps states to state, A is an n-by-n matrix.

Clearly, the state will influence the traffic flow (tens of cars per 5 minutes) with possible values of 0, 1, 2, 3. The number of observed values $m = 4$. Again from historical data the emission probability matrix B is estimated to be

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.2 & 0.2 & 0.1 \end{bmatrix}$$

As B maps states to observed values, B is an n-by-m matrix.

One question to address is, given a time series (observations sequence), what corresponding sequence of states gives the highest probability of occurrence to the observed sequences.

$$\mathbf{y} \;=\; [3, 3, 0]$$

This may be done by computing the joint probability $P(\mathbf{x}, \mathbf{y})$

$$\begin{aligned}
P(NNN, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{00} \cdot b_{00} &&= 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.8 \cdot 0.1 &&= 0.009216 \\
P(NNA, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{01} \cdot b_{10} &&= 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.2 \cdot 0.5 &&= 0.011520 \\
P(NAN, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{01} \cdot b_{13} \cdot a_{10} \cdot b_{00} &&= 0.9 \cdot 0.4 \cdot 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.1 &&= 0.000360 \\
P(NAA, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{01} \cdot b_{13} \cdot a_{11} \cdot b_{10} &&= 0.9 \cdot 0.4 \cdot 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.5 &&= 0.001800 \\
P(ANN, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{00} \cdot b_{00} &&= 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.8 \cdot 0.1 &&= 0.000160 \\
P(ANA, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{01} \cdot b_{10} &&= 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.2 \cdot 0.5 &&= 0.000200 \\
P(AAN, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{10} \cdot b_{00} &&= 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.1 &&= 0.000025 \\
P(AAA, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{11} \cdot b_{10} &&= 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.5 &&= 0.000125
\end{aligned}$$

The state giving the highest probability is $\mathbf{x} = NNA$. The marginal probability of the observed sequence $P(\mathbf{y})$ can be computed by summing over all eight states.

$$P(\mathbf{y}) \;=\; \sum_{\mathbf{x}} P(\mathbf{x}, \mathbf{y}) \;=\; 0.023406$$

The algorithms given in the subsections below are adapted from [181]. For these algorithms, we divide the time series/sequence of observations into two parts (past and future).

$$\begin{aligned} \mathbf{y^{t-}} &= [y_0, y_1, y_t] \\ \mathbf{y^{t+}} &= [y_{t+1}, y_{t+2}, y_{T-1}] \end{aligned}$$

They allow one to calculate (1) the probability of arriving in a state at time t with observations $\mathbf{y^{t-}}$, (2) the conditional probability of seeing future observations $\mathbf{y^{t+}}$ from a given state at time t, and (3) the conditional probability of being in a state at time t given all the observations $\mathbf{y}$.

7.10.2 Forward Algorithm

For longer observation sequences/time series, the approach of summing over all possible state vectors (of which there are n^T) will become intractable. Much of the computation is repetitive anyway. New matrices A, B and Γ are defined to save such intermediate calculations.

The forward algorithm (α-pass) computes the A matrix. The probability of being in state j at time t having observations up to time t is given by

$$\alpha_{tj} \;=\; P(x_t = j, \mathbf{y} = \mathbf{y^{t-}})$$

Computation of α_{tj} may be done efficiently using the following recurrence.

$$\alpha_{tj} \;=\; b_{j,y_t} \sum_{i=0}^{n-1} \alpha_{t-1,i}\, a_{ij} \;=\; b_{j,y_t} \left[\boldsymbol{\alpha}_{t-1} \cdot \boldsymbol{a}_{:j}\right]$$

To get to state j at time t, the system must transition from some state i at time $t-1$ and at time t emit the value y_t. These values may be saved in a T-by-n matrix A $= [\alpha_{tj}]$ and efficiently computed by moving forward in time.

```scala
    def forwardEval0 (): MatrixD =
        for j <- rstate do alp(0, j) = pi(j) * b(j, y(0))     // compute alpha_0 (at t = 0)
        for t <- 1 until tt; j <- rstate do                   // iterate over time and states
            alp(t, j) = b(j, y(t)) * (alp(t-1) dot a(?, j))
        end for
        alp
    end forwardEval0
```

The marginal probability is now simply the sum of the elements in the last row of the α matrix

$$P(\mathbf{y}) \;=\; \sum_{j=0}^{n} \alpha_{T-1,j}$$

ScalaTion also provides a `forwardEval` method that uses scaling to avoid underflow.

7.10.3 Backward Algorithm

The backward algorithm (β-pass) computes the B matrix. The conditional probability of having future observations after time t ($\mathbf{y} = \mathbf{y^{t+}}$) given the current state $x_t = i$ is

$$\beta_{ti} \;=\; P(\mathbf{y} = \mathbf{y^{t+}} | x_t = i)$$

Computation of β_{tj} may be done efficiently using the following recurrence.

$$\beta_{ti} \;=\; \sum_{j=0}^{n-1} a_{ij}\, b_{j,y_{t+1}} \beta_{t+1,j}$$

From state i at time t, the system must transition to some state j at time $t+1$ and at time $t+1$ emit the value y_{t+1}. These values may be saved in a T-by-n matrix $\text{B} = [\beta_{ti}]$ and efficiently computed by moving backward in time.

```
    def backwardEval0 (): MatrixD =
        for i <- rstate do bet(tt-1, i) = 1.0          // initialize beta_{tt-1} to 1
        for t <- tt-2 to 0 by -1; i <- rstate do       // iterate backover time, over states
            bet(t, i) = 0.0
            for j <- rstate do bet(t, i) += a(i, j) * b(j, y(t+1)) * bet(t+1, j)
        end for
        bet
    end backwardEval0
```

ScalaTion also provides a `backwardEval` method that uses scaling to avoid underflow.

7.10.4 Viterbi Algorithm

The Viterbi algorithm (γ-pass) computes the Γ matrix. The conditional probability of the state at time t being i, given all observations ($\mathbf{y} = \mathbf{y}$) is

$$\gamma_{ti} \;=\; P(x_t = i | \mathbf{y} = \mathbf{y})$$

As α_{ti} captures the probability up to time t and β_{ti} captures the probability after time t, the conditional probability may be calculated as follows:

$$\gamma_{ti} \;=\; \frac{\alpha_{ti}\beta_{ti}}{P(\mathbf{y})}$$

In ScalaTion, the $\Gamma = [\gamma_{ti}]$ matrix is calculated using the Hadamard product.

```
    def gamma (alp: MatrixD, bet: MatrixD): MatrixD = (alp *~ bet) / probY (alp)
```

The conditional probability of being in state i at time t and transitioning to state j at time $t+1$ given all observations ($\mathbf{y} = \mathbf{y}$) is

$$\gamma_{tij} \;=\; P(x_t = i, x_{t+1} = j | \mathbf{y} = \mathbf{y})$$

Note, this equation is not defined for the last time point $T-1$, since there is no next state. Getting to state i at time t is characterized by α_{ti}, the probability of the state transitioning to from i to j is characterized a_{ij}, the probability of emitting y_{t+1} from state j at time $t+1$ is $b_{j,y_{t+1}}$, and finally going from state j to the end is characterized by $\beta_{t+1,j}$.

$$\gamma_{tij} \;=\; \frac{\alpha_{ti} a_{ij} b_{j,y_{t+1}} \beta_{t+1,j}}{P(\mathbf{y})}$$

The Viterbi Algorithm `viterbiDecode` computes the Γ matrix (`gam` in code) from scaled versions of `alp` and `bet`. It also computes the $\Gamma = [\gamma_{tij}]$ tensor (`gat` in code).

7.10.5 Training

The `train` method will call `forwardEval`, `backwardEval` and `viterbiDecode` to calculate updated values for the A, B and Γ matrices as well as for the Γ tensor. These values are used to `re-estimate` the π, A and B parameters.

```
    @param x_  the training/full data/input matrix (ignored)
    @param y_  the training/full response/output vector (defaults to full y)

    override def train (x_ : MatrixD = null, y_ : VectorI = y): Unit =
        var oldLogPr = 0.0
        breakable {
            for it <- 1 to MIT do                         // up to Maximum ITerations
                val logPr = logProbY (true)               // compute the new log probability
                if logPr > oldLogPr then
                    oldLogPr = logPr                      // improvement => continue
                    forwardEval ()                        // alpha-pass
                    backwardEval ()                       // beta-pass
                    viterbiDecode ()                      // gamma-pass
                    reestimate ()                         // re-estimate the model (pi, a, b)
                else
                    println (s"train: HMM model converged after $it iterations")
                    break ()
                end if
            end for
        } // breakable
        println (s"train: HMM model did not converged after $MIT iterations")
    end train
```

The training loop will terminate early when there is no improvement to $P(\mathbf{y})$. To avoid underflow $-\log(P(\mathbf{y})$ is used.

7.10.6 Reestimation of Parameters

The parameters for an HMM model are π, A and B and may be adjusted to maximize the probability of seeing the observation vector $P(\mathbf{y})$. Since $\alpha_{0i} = \pi_i b_{i,y(0)}$, we can re-estimation π as follows:

$$\pi_i \;=\; \frac{\alpha_{0i}}{b_{i,y_0}} \;=\; \gamma_{0i}$$

The A matrix can re-estimated as follows:

$$a_{ij} \;=\; \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{ti}}$$

Similarly, the B matrix can re-estimated as follows:

$$b_{ik} \;=\; \frac{\sum_{t=0}^{T-1} I_{y_t=k}\, \gamma_{ti}}{\sum_{t=0}^{T-1} \gamma_{ti}}$$

The detailed derivations are left to the exercises.

7.10.7 HiddenMarkov Class

Class Methods:

```
    @param y        the observation vector/observed discrete-valued time series
    @param m        the number of observation symbols/values {0, 1, ... m-1}
    @param n        the number of (hidden) states in the model
    @param cname_   the class names for the states, e.g., ("Hot", "Cold")
    @param pi       the probabilty vector for the initial state
    @param a        the state transition probability matrix (n-by-n)
    @param b        the observation probability matrix (n-by-m)
    @param hparam   the hyper-parameters

    class HiddenMarkov (y: VectorI, m: Int, n: Int, cname_ : Array [String] = null,
                        private var pi: VectorD = null,
                        private var a:  MatrixD = null,
                        private var b:  MatrixD = null,
                        hparam: HyperParameter = null)
          extends Classifier (null, y, null, n, cname_, hparam)   // hidden x = null
             with FitC (y, n):

    def size: Int = n
    override def parameter: VectorD = pi
    def parameters: (MatrixD, MatrixD) = (a, b)
    def jointProb (x: VectorI): Double =
    def forwardEval0 (): MatrixD =
    def backwardEval0 (): MatrixD =
    def gamma (): MatrixD = (alp *~ bet) / probY ()
    def forwardEval (): MatrixD =
    def getC: VectorD = c
    def backwardEval (): MatrixD =
    def viterbiDecode (): MatrixD =
    def reestimate (): Unit =
    override def train (x_ : MatrixD = null, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = null, y_ : VectorI = y): (VectorI, VectorD) =
    override def report: String =
    override def predictI (z: VectorI): Int =
```

7.10.8 Exercises

1. Show that for $t \in \{0, \ldots T-2\}$,

$$\gamma_{ti} \;=\; \sum_{j=0}^{n-1} \gamma_{tij}$$

2. Show that

$$\pi_i = \frac{\alpha_{0i}}{b_{i,y_0}} = \gamma_{0i}$$

3. Show that

$$a_{ij} = \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{ti}}$$

4. Show that

$$b_{ik} = \frac{\sum_{t=0}^{T-1} I_{y_t=k}\, \gamma_{ti}}{\sum_{t=0}^{T-1} \gamma_{ti}}$$

7.11 Further Reading

1. Data Classification Algorithms and Applications [2]
2. Data Mining Practical Machine Learning Tools and Techniques, Fourth Edition [205]

Chapter 8

Classification: Continuous Variables

For the problems in this chapter, the response/classification variable is still discrete, but some/all of the feature variables are now continuous. Technically, classification problems fit in this category, if it is infeasible or nonproductive to compute frequency counts for all values of a variable (e.g., for x_j, the value count $vc_j = \infty$). If a classification problem almost fits in the previous chapter, one may consider the use of binning to convert numerical variables into categorical variables (e.g, convert weight into weight classes). Care should be taken since binning represents hidden parameters in the model and arbitrary choices may influence results.

8.1 Gaussian Naïve Bayes

The `NaïveBayesR` class implements a Gaussian Naïve Bayes Classifier, which is the most commonly used such classifier for continuous input data. The classifier is trained using a data matrix X and a classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, 1, \dots, k-1$.

Class probabilities are calculated based on the population of each class in the training-set. Relative probabilities are computed by multiplying these by values computed using conditional density functions based on the Normal (Gaussian) distribution. The classifier is naïve, because it assumes feature independence and therefore simply multiplies the conditional densities.

Starting with main results from the section on Naïve Bayes (equation 4.5),

$$\hat{y} \;=\; \underset{y \in \{0,\dots,k-1\}}{\operatorname{argmax}} \; P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{8.1}$$

if all the variables x_j are continuous, we may switch from conditional probabilities $P(x_j|y)$ to conditional densities $f(x_j|y)$. The best prediction for class y is the value $\hat{y}$ that maximizes the product of the conditional densities multiplied by the class probability.

$$\hat{y} \;=\; \underset{y \in \{0,\dots,k-1\}}{\operatorname{argmax}} \; P(y) \prod_{j=0}^{n-1} f(x_j|y) \tag{8.2}$$

Although the formula assumes the conditional independence of x_js, the technique can be applied as long as correlations are not too high.

Using the Gaussian assumption, the conditional density of x_j given y, is approximated by estimating the two parameters of the `Normal` distribution,

$$x_j|y \;\sim\; Normal(\mu_c, \sigma_c^2) \tag{8.3}$$

where class $c \in \{0, 1, \dots, k-1\}$, $\mu_c = \mathbb{E}[x|y = c]$ and $\sigma_c^2 = \mathbb{V}[x|y = c]$). Thus, the conditional density function is

$$f(x_j|y = c) \;=\; \frac{1}{\sqrt{2\pi}\sigma_c} e^{-\frac{(x-\mu_c)^2}{2\sigma_c^2}} \tag{8.4}$$

Class probabilities $P(y = c)$ may be estimated as $\frac{m_c}{m}$, where m_c is the frequency count of the number of occurrences of c in the class vector $\mathbf{y}$. Conditional densities are needed for each of the k class values, for each of the n variables (each x_j) (i.e., kn are needed). Corresponding means and variances may be estimated as follows:

$$\hat{\mu}_{cj} \;=\; \frac{1}{m_c} \sum_{i=0}^{m-1} (x_{ij}|y_i = c) \tag{8.5}$$

$$\hat{\sigma}_{cj}^2 \;=\; \frac{1}{m_c - 1} \sum_{i=0}^{m-1} ((x_{ij} - \hat{\mu}_{cj})^2 | y_i = c) \tag{8.6}$$

Using conditional density (cd) functions estimated in the `train` method (see code for details), an input vector $\mathbf{z}$ can be classified using the `predictI` or `classify` method.

```
    override def predictI (z: VectorD): Int =
        for c <- 0 until k; j <- x.indices2 do p_yz(c) *= cd(c)(j)(z(j))
        p_yz.argmax ()                          // return class with highest probability
    end predictI
```

8.1.1 NaiveBayesR Class

Class Methods:

```
    @param x        the real-valued data vectors stored as rows of a matrix
    @param y        the class vector, where y_i = class for row i of the matrix x, x(i)
    @param fname_   the names for all features/variables (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the names for all classes
    @param hparam   the hyper-parameters

    class NaiveBayesR (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                       k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                       hparam: HyperParameter = NaiveBayes.hp)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with FitC (y, k):

    def calcStats (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def calcHistogram (x_j: VectorD, intervals: Int): VectorD =
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int =
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

8.1.2 Exercises

1. Use `NaiveBayesR` to classify manufactured parts according whether they should pass quality control based on *curvature* and `diameter` tolerances. See `people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html` for details.

```
    // features/variable:
    // x1: curvature
    // x2: diameter
    // y:  classification: pass (0), fail (1)
    //                          x1    x2    y
    val xy = MatrixD ((7, 3), 2.95, 6.63, 0,            // joint data matrix
                              2.53, 7.79, 0,
                              3.57, 5.65, 0,
                              3.16, 5.47, 0,
                              2.58, 4.46, 1,
                              2.16, 6.22, 1,
                              3.27, 3.52, 1)

    val fname = Array ("curvature", "diameter")          // feature names
    val cname = Array ("pass", "fail")                  // class names
    val nbr   = NaiveBayesR (xy, fname, 2, cname)()     // create NaiveBayesR nbr
```

8.2 Simple Logistic Regression

The `SimpleLogisticRegression` class supports simple logistic regression. In this case, the predictor vector $\mathbf{x}$ is two-dimensional $[1, x_1]$. Again, the goal is to fit the parameter vector $\mathbf{b}$ in the regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + \epsilon \tag{8.7}$$

where ϵ represents the residuals (the part not explained by the model). This looks like simple linear regression, with the difference being that the response variable y is binary ($y \in \{0, 1\}$). Since y is binary, minimizing the distance, as was done before, may not work well. First, instead of focusing on $y \in \{0, 1\}$, we focus on the conditional probability of success $p_y(\mathbf{x}) \in [0, 1]$, i.e.,

$$p_y(\mathbf{x}) \;=\; P(y = 1|\mathbf{x}) \tag{8.8}$$

For example, the random variable y could be used to indicate whether a customer will pay back a loan (1) or not (0). The predictor variable x_1 could be the customer's FICA score.

8.2.1 `mtcars` Example

Another example is from the Motor Trends Cars (`mtcars`) dataset (see `https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html`), `gist.github.com/seankross/a412dfbd88b3db70b74b`). Try using `mpg` to predict/classify the car's engine as either `V`-shaped(0) or `S`traight(1), as in V-6 or S-4. First, use SimpleRegression to predict $p_y(\mathbf{x})$ where y is `V/S` and x_1 is `mpg`, ($\mathbf{x} = [1, x_1]$). Plot y versus x_1 and then add a vector to the plot for the predicted values for p_y. Utilizing simple linear regression to predict $p_y(\mathbf{x})$ would correspond to the following equation.

$$p_y(\mathbf{x}) \;=\; b_0 + b_1 x_1$$

8.2.2 Logistic Function

The linear relationship between y and x_1 may be problematic, in the sense that there is likely to be a range of rapid transition before which loan default is likely and after which loan repayment is likely. Similarly, there is rapid transition from `S`(1) to `V`(0) as `mpg` increases. This suggests that some "S-curve" function such as the *logistic* function may be more useful. The standard logistic function (sigmoid function) is

$$\text{logistic}(z) \;=\; \frac{1}{1 + e^{-z}} \;=\; \frac{e^z}{1 + e^z} \tag{8.9}$$

Letting $z = b_0 + b_1 x_1$, we obtain

$$p_y(\mathbf{x}) \;=\; \text{logistic}(b_0 + b_1 x_1) \;=\; \frac{e^{b_0 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}} \tag{8.10}$$

8.2.3 Logit Function

The goal now is to transform the right hand side into the usual linear form (i.e., $\mathbf{b} \cdot \mathbf{x}$).

$$p_y(\mathbf{x}) \;=\; \frac{e^{\mathbf{b} \cdot \mathbf{x}}}{1 + e^{\mathbf{b} \cdot \mathbf{x}}} \tag{8.11}$$

Multiplying through by $1 + e^{\mathbf{b}\cdot\mathbf{x}}$ gives

$$p_y(\mathbf{x}) + e^{\mathbf{b}\cdot\mathbf{x}} p_y(\mathbf{x}) \;=\; e^{\mathbf{b}\cdot\mathbf{x}} \tag{8.12}$$

Solving for $e^{\mathbf{b}\cdot\mathbf{x}}$ yields

$$e^{\mathbf{b}\cdot\mathbf{x}} \;=\; \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} \tag{8.13}$$

Taking the natural logarithm of both sides gives

$$\ln \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} \;=\; \mathbf{b}\cdot\mathbf{x} \;=\; b_0 + b_1 x_1 \tag{8.14}$$

where the function on the left hand side is called the *logit* function.

$$\text{logit}(p_y(\mathbf{x})) \;=\; \mathbf{b}\cdot\mathbf{x} \;=\; b_0 + b_1 x_1 \tag{8.15}$$

Putting the model in this form shows it is a special case of a Generalized Linear Model (see the Chapter on Generalized Linear Models) and will be useful in the estimation procedure.

8.2.4 Maximum Likelihood Estimation

Imagine you wish to create a model that is able to generate data that looks like the observed data (i.e., the data in the dataset). The choice of values for the parameters $\mathbf{b}$ (treated as a random vector) will impact the quality of the model. Define a function of $\mathbf{b}$ that will be maximized when the parameters are ideally set to generate the observed data.

8.2.5 Likelihood Function

We can think of this function as the likelihood of $\mathbf{b}$ given the predictor vector $\mathbf{x}$ and the response variable y.

$$L(\mathbf{b}|\mathbf{x}, y) \tag{8.16}$$

In this case, $y \in \{0, 1\}$, so if we estimate the likelihood for a single data instance (or row), we have

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; p_y(\mathbf{x})^y \, (1 - p_y(\mathbf{x}))^{1-y} \tag{8.17}$$

If $y = 1$, then $L = p_y(\mathbf{x})$ and otherwise $L = 1 - p_y(\mathbf{x})$. These are the probabilities for the two outcomes for a Bernoulli random variable (and equation 6.5 concisely captures both).

For each instance $i \in \{0, \ldots, m-1\}$, a similar factor is created. These are multiplied together for all the instances (in the dataset, or training or testing). The likelihood of $\mathbf{b}$ given the predictor matrix X and the response vector $\mathbf{y}$ is then

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; \prod_{i=0}^{m-1} p_y(\mathbf{x}_i)^{y_i} \, (1 - p_y(\mathbf{x}_i))^{1-y_i} \tag{8.18}$$

8.2.6 Log-likelihood Function

To reduce round-off errors, a log (e.g., natural log, ln) is taken

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \ln(p_y(\mathbf{x}_i)) + (1 - y_i)\ln(1 - p_y(\mathbf{x}_i)) \tag{8.19}$$

This is referred as the log-likelihood function. Collecting y_i terms give

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \ln \frac{p_y(\mathbf{x}_i)}{1 - p_y(\mathbf{x}_i)} + \ln(1 - p_y(\mathbf{x}_i)) \tag{8.20}$$

Substituting $\mathbf{b} \cdot \mathbf{x}_i$ for logit$(p_y(\mathbf{x}_i))$ gives

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i\, \mathbf{b} \cdot \mathbf{x}_i + \ln(1 - p_y(\mathbf{x}_i)) \tag{8.21}$$

Now substituting $\dfrac{e^{\mathbf{b}\cdot\mathbf{x_i}}}{1 + e^{\mathbf{b}\cdot\mathbf{x_i}}}$ for $p_y(\mathbf{x}_i)$ gives

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i\, \mathbf{b} \cdot \mathbf{x}_i - \ln(1 + e^{\mathbf{b}\cdot\mathbf{x}_i}) \tag{8.22}$$

Multiplying the log-likelihood by -2 makes the distribution approximately Chi-square (see Wilks Theorem[203]).

$$-2l \;=\; -2 \sum_{i=0}^{m-1} y_i\, \mathbf{b} \cdot \mathbf{x}_i - \ln(1 + e^{\mathbf{b}\cdot\mathbf{x}_i}) \tag{8.23}$$

Or since $\mathbf{b} = [b_0, b_1]$,

$$-2l \;=\; -2 \sum_{i=0}^{m-1} y_i(b_0 + b_1 x_{i1}) - \ln(1 + e^{b_0 + x_{i1}}) \tag{8.24}$$

Letting $\beta_i = b_0 + b_1 x_{i1}$ gives

$$-2l \;=\; -2 \sum_{i=0}^{m-1} y_i \beta_i - \ln(1 + e^{\beta_i}) \tag{8.25}$$

It is more numerically stable to perform a negative rather than positive e^z function.

$$-2l \;=\; -2 \sum_{i=0}^{m-1} y_i \beta_i - \beta_i - \ln(e^{-\beta_i} + 1) \tag{8.26}$$

8.2.7 Computation in ScalaTion

The computation of $-2l$ is carried out in ScalaTion via the `ll` method. It loops through all instances computing β_i (`bx` in the code) and summing all the terms given in equation 6.9.

```
    def ll (b: VectorD): Double =
        var sum = 0.0
        var bx  = 0.0                                   // beta
        for i <- y.indices do
```

```
            bx = b(0) + b(1) * x(i, 1)
            sum += y(i) * bx - bx - log (exp (-bx) + 1.0)
        end for
        -2.0 * sum
    end ll
```

8.2.8 Making a Decision

So far, **SimpleLogisticRegression** is a model for predicting $p_y(\mathbf{x})$. In order to use this for binary classification a decision needs to be made: deciding on either 0 (no) or 1 (yes). A natural way to do this is to choose 1 when $p_y(\mathbf{x})$ exceeds 0.5. The **predicI** pr **classify** methods may be use.

```
    override def predictI (z: VectorD): Int =
        val py = sigmoid (b dot z)              // P(y = 1|x)
        is (py > cThresh)                       // compare to classification threshold
    end predictI
```

Note, the **is** method in the **scalation** package is defined as follows:

```
    inline def is (p: Boolean): Int = if p then 1 else 0
```

In some cases, this may results in imbalance between false positives and false negatives. Quality of Fit (QoF) measures may improve by tuning the classification/decision threshold **cThresh**. Decreasing the threshold pushes false negatives to false positives. Increasing the threshold does the opposite. Ideally, the tuning of the threshold will also push more cases into the diagonal of the confusion matrix and minimize errors. Finally, in some cases it may be more important to reduce one more than the other, false negatives vs. false positives (see the exercises).

8.2.9 SimpleLogisticRegression Class

Class Methods:

```
    @param x        the input/design matrix augmented with a first column of ones
    @param y        the binary response vector, y_i in {0, 1}
    @param fname_   the names for all features/variables
    @param cname_   the names for both classes
    @param hparam   the hyper-parameters

    class SimpleLogisticRegression (x: MatrixD, y: VectorI,
                                    fname_ : Array [String] = Array ("one", "x1"),
                                    cname_ : Array [String] = Array ("No", "Yes"),
                                    hparam: HyperParameter = Classifier.hp)
          extends Classifier (x, y, fname_, 2, cname_, hparam)
             with FitC (y, 2):

    override def pseudo_rSq: Double = 1.0 - r_dev / n_dev
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def train_null (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def parameter: VectorD = b
    override def fit: VectorD =
    def fitLabel: Seq [String] = QoFC.values.map (_.toString).toIndexedSeq ++
                                                  Seq ("n_dev", "r_dev", "aic")
    override def predictI (z: VectorD): Int =
    override def predictI (z: VectorI): Int = predictI (z.toDouble)
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = b, vifs: VectorD = null): String =
```

8.2.10 Exercises

1. Plot the standard logistic function (`sigmoid` for scalars, `sigmoid_` for vectors).

```
    import scalation.modeling.ActivationFun.sigmoid_
    val z  = VectorD.range (0, 160) / 10.0 - 8.0
    val fz = sigmoid_ (z)
    new Plot (z, fz)
```

2. For the `mtcars` dataset, determine the model parameters b_0 and b_1 directly (i.e., do not call `train`). Rather perform a grid search for a minimal value of the `ll` function. Use the `x` matrix (`one`, `mpg`) and `y` vector (`V/S`) from `SimpleLogisticRegressionTest`.

```
    // 32 data points:         One    Mpg
    val x = MatrixD ((32, 2), 1.0,  21.0,          //  1 - Mazda RX4
                              1.0,  21.0,          //  2 - Mazda RX4 Wa
                              1.0,  22.8,          //  3 - Datsun 710
                              1.0,  21.4,          //  4 - Hornet 4 Drive
                              1.0,  18.7,          //  5 - Hornet Sportabout
                              1.0,  18.1,          //  6 - Valiant
                              1.0,  14.3,          //  7 - Duster 360
                              1.0,  24.4,          //  8 - Merc 240D
                              1.0,  22.8,          //  9 - Merc 230
                              1.0,  19.2,          // 10 - Merc 280
                              1.0,  17.8,          // 11 - Merc 280C
                              1.0,  16.4,          // 12 - Merc 450S
                              1.0,  17.3,          // 13 - Merc 450SL
                              1.0,  15.2,          // 14 - Merc 450SLC
                              1.0,  10.4,          // 15 - Cadillac Fleetwood
                              1.0,  10.4,          // 16 - Lincoln Continental
                              1.0,  14.7,          // 17 - Chrysler Imperial
                              1.0,  32.4,          // 18 - Fiat 128
                              1.0,  30.4,          // 19 - Honda Civic
                              1.0,  33.9,          // 20 - Toyota Corolla
                              1.0,  21.5,          // 21 - Toyota Corona
                              1.0,  15.5,          // 22 - Dodge Challenger
                              1.0,  15.2,          // 23 - AMC Javelin
                              1.0,  13.3,          // 24 - Camaro Z28
                              1.0,  19.2,          // 25 - Pontiac Firebird
                              1.0,  27.3,          // 26 - Fiat X1-9
                              1.0,  26.0,          // 27 - Porsche 914-2
                              1.0,  30.4,          // 28 - Lotus Europa
                              1.0,  15.8,          // 29 - Ford Pantera L
                              1.0,  19.7,          // 30 - Ferrari Dino
                              1.0,  15.0,          // 31 - Maserati Bora
```

```
                                1.0,   21.4)        // 32 - Volvo 142E

    // V/S (e.g., V-6 vs. I-4)
    val y = VectorI (0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0,
                     0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1)
```

3. Compare the effectiveness of `SimpleLogisticRegression` versus `SimpleRegression` on the mtcars dataset. See `simpleLogisticRegressionTest`.

4. If the treatment for a disease is risky and consequences of having the disease are minimal, would you prefer to focus on reducing false positives or false negatives?

5. If the treatment for a disease is safe and consequences of having the disease may be severe, would you prefer to focus on reducing false positives or false negatives?

8.3 Logistic Regression

The `LogisticRegression` class supports logistic regression. In this case, $\mathbf{x}$ may be multi-dimensional $[1, x_1, \ldots, x_k]$. Again, the goal is to fit the parameter vector $\mathbf{b}$ in the regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 \,+\, b_1 x_1 \,+\, \ldots \,+\, b_k x_k \,+\, \epsilon \tag{8.27}$$

where ϵ represents the residuals (the part not explained by the model). This looks like multiple linear regression. The difference being that the response variable y is binary ($y \in \{0, 1\}$). Since y is binary, minimizing the distance, as was done before may not work well. First, instead of focusing on $y \in \{0, 1\}$, we focus on the conditional probability of success $p_y(\mathbf{x}) \in [0, 1]$, i.e.,

$$p_y(\mathbf{x}) \;=\; P(y = 1|\mathbf{x}) \tag{8.28}$$

Still, $p_y(\mathbf{x})$ is bounded, while $\mathbf{b} \cdot \mathbf{x}$ is not. We therefore, need a transformation, such as the logit transformation, and fit $\mathbf{b} \cdot \mathbf{x}$ to this function. Treating this as a Generalized Linear Model problem,

$$y \;=\; \mu(\mathbf{x}) + \epsilon \tag{8.29}$$

$$g(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x} \tag{8.30}$$

we let the link function $g \;=\;$ logit.

$$\text{logit}(\mu(\mathbf{x})) \;=\; \ln \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} \;=\; \mathbf{b} \cdot \mathbf{x} \tag{8.31}$$

This is the logit regression equation. Second, instead of minimizing the sum of squared errors, we wish to maximize the likelihood of predicting correct outcomes. For the ith training case $\mathbf{x}_i$ with outcome y_i, the likelihood function is based on the Bernoulli distribution.

$$p_y(\mathbf{x}_i)^{y_i} (1 - p_y(\mathbf{x}_i))^{1-y_i} \tag{8.32}$$

The overall likelihood function is the product over all m cases. The equation is the same as 6.6 from the last section.

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; \prod_{i=0}^{m-1} p_y(\mathbf{x}_i)^{y_i}\, (1 - p_y(\mathbf{x}_i))^{1-y_i} \tag{8.33}$$

Following the same derivation steps, will give the same log-likelihood that is in equation 6.7.

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i\, \mathbf{b} \cdot \mathbf{x}_i - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x}_i}) \tag{8.34}$$

Again, multiplying the log-likelihood function by -2 makes the distribution approximately Chi-square.

$$-\,2l \;=\; -\,2 \sum_{i=0}^{m-1} y_i\, \mathbf{b} \cdot \mathbf{x}_i - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x}_i}) \tag{8.35}$$

The likelihood can be maximized by minimizing $-2l$, which is a nonlinear function of the parameter vector $\mathbf{b}$. Various optimization techniques may be used to search for optimal values for $\mathbf{b}$. Currently, ScalaTion uses BFGS, a popular general-purpose QuasiNewton NLP solver. Other possible optimizers include LBFGS and IRWLS. For a more detailed derivation, see `http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf`.

8.3.1 `LogisticRegression` Class

Class Methods:

```
    @param x        the input/design matrix augmented with a first column of ones
    @param y        the binary response vector, y_i in {0, 1}
    @param fname_   the names for all features/variables (defaults to null)
    @param cname_   the names for both classes
    @param hparam   the hyper-parameters

    class LogisticRegression (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                              cname_ : Array [String] = Array ("No", "Yes"),
                              hparam: HyperParameter = Classifier.hp)
          extends SimpleLogisticRegression (x, y, fname_, cname_, hparam):

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def vif: VectorD = ???
```

8.3.2 Exercises

1. Use `Logistic Regression` to classify whether stock market will be increasing or not. The `Smarket` dataset is in the ISLR library, see [85] section 4.6.2.

2. Use `Logistic Regression` to classify whether a customer will purchase caraavan insurance. The `Caravan` dataset is in the ISLR library, see [85] section 4.6.6.

8.4 Simple Linear Discriminant Analysis

The `SimpleLDA` class support Linear Discriminant Analysis which is useful for multiway classification of continuously valued data. The response/classification variable can take on k possible values, $y \in \{0, 1, \dots, k-1\}$. The feature variable x is one dimensional for `SimpleLDA`, but can be multi-dimensional for `LDA` discussed in the next section. Given the data about an instance stored in variable x, pick the best (most probable) classification $y = c$.

As was done for Naïve Bayes classifiers, we are interested in the probability of y given x.

$$P(y|x) \;=\; \frac{P(x|y)\,P(y)}{P(x)} \tag{8.36}$$

Since x is now continuous, we need to work with conditional densities as is done Gaussian Naïve Bayes classifiers,

$$P(y|x) \;=\; \frac{f(x|y)\,P(y)}{f(x)} \tag{8.37}$$

where

$$f(x) \;=\; \sum_{c=0}^{k-1} f(x|y=c)P(y=c) \tag{8.38}$$

Now let us assume the conditional probabilities are normally distributed with a common variance.

$$x|y \;\sim\; Normal(\mu_c, \sigma^2) \tag{8.39}$$

where class $c \in \{0, 1, \dots, k-1\}$, $\mu_c = \mathbb{E}[x|y=c]$ and σ^2 is the pooled variance (weighted average of $\mathbb{V}[x|y=c]$). Thus, the conditional density function is

$$f(x|y=c) \;=\; \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} \tag{8.40}$$

Substituting into eqaution 6.10 gives

$$P(y|x) \;=\; \frac{\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}\, P(y)}{f(x)} \tag{8.41}$$

where

$$f(x) \;=\; \sum_{c=0}^{k-1} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}\, P(y=c) \tag{8.42}$$

Because of differing means, each conditional density will be shifted resulting in a mountain range appearance when plotted together. Given a data point x, the question becomes, which mountain is it closest to in the sense of maximizing the conditional probability expressed in equation 6.11.

$$P(y|x) \;\propto\; \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}\, P(y) \tag{8.43}$$

Since the term $\frac{1}{\sqrt{2\pi}\sigma}$ is same for all values of y, it may be ignored. Taking the natural logarithm yields

$$\ln(P(y|x)) \;\propto\; \frac{-(x-\mu_c)^2}{2\sigma^2} + ln(P(y)) \tag{8.44}$$

Expanding $-(x-\mu_c)^2$ gives $-x^2 + 2x\mu_c - \mu_c^2$ and the first term may be ignored (same for all y).

$$\ln(P(y|x)) \;\propto\; \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + ln(P(y)) \tag{8.45}$$

The right hand side functions in 4.12 are linear in x and are called discriminant functions $\delta_c(x)$.

Given training data vectors $\mathbf{x}$ and $\mathbf{y}$, define $\mathbf{x}_c$ (or `xc` in the code) to be the vector of all x_i values where $y_i = c$ and let its length be denoted by m_c. Now the k means may be estimated as follows:

$$\hat{\mu}_c \;=\; \frac{\mathbf{1}\cdot\mathbf{x}_c}{m_c} \tag{8.46}$$

The common variance my be estimated using a pooled variance estimator.

$$\hat{\sigma}^2 \;=\; \frac{1}{m-k}\sum_{c=0}^{k-1}\|\mathbf{x}_c - \mu_c\|^2 \tag{8.47}$$

Finally, $\frac{m_c}{m}$ can be used to estimate $P(y)$.

These can easily be translated into ScalaTion code. Most of the calculations are done in the **train** method. It estimates the class probability vector `p_y`, the group means vector `mu` and the pooled variance. The vectors `term1` and `term2` capture the x-term (μ_c/σ^2) and the constant term $(\mu_c^2/2\sigma^2 - ln(P(y)))$ in equation 6.12.

```
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
        val xc = for c <- 0 until k yield                          // groups for x
            VectorD (for i <- y_.indices if y(i) == c yield x_(0, i))  // group c
        p_y = VectorD (xc.map (_.dim / y.dim.toDouble))            // probability y = c
        mu  = VectorD (xc.map (_.mean))                            // group means
        var sum = 0.0
        for c <- 0 until k do sum += (xc(c) - mu(c)).normSq
        sig2  = sum / (m - k).toDouble                             // pooled variance
        term1 = mu / sig2
        term2 = mu~^2 / (2.0 * sig2) - p_y.map (log (_))
    end train
```

Given the two precomputed terms, the `classify` method simply multiplies the first by `z(0)` and subtracts the second. Then it finds the `argmax` of the `delta` vector to return the class with the maximum `delta`, which corresponds the most probable classification.

$$\hat{y} \;=\; \text{argmax}_c \frac{z\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + ln(P(y)) \tag{8.48}$$

```
    override def predictI (z: VectorD): Int =
        val delta = term1 * z(0) - term2
        delta.argmax ()
    end predictI
```

8.4.1 SimpleLDA Class

Class Methods:

```
 *  @param x        the input/design matrix with only one column
 *  @param y        the response/classification vector, y_i in {0, 1}
 *  @param fname_   the name for the feature/variable
 *  @param k        the number of possible values for y (0, 1, ... k-1)
 *  @param cname_   the names for all classes
 *  @param hparam   the hyper-parameters
 */
class SimpleLDA (x: MatrixD, y: VectorI, fname_ : Array [String] = Array ("x1"),
                 k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                 hparam: HyperParameter = Classifier.hp)
      extends Classifier (x, y, fname_, k, cname_, hparam)
         with FitC (y, 2):

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int =
    override def predictI (z: VectorI): Int = predictI (z.toDouble)
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
```

8.4.2 Exercises

1. Generate two samples using `Normal (98.6, 1.0)` and `Normal (101.0, 1.0)` with 100 in each sample. Put the data instances into a single `x` vector. Let the `y` vector be 0 for the first sample and 1 for the second. Use `SimpleLDA` to classify all 200 data points and determine the values for `tp, tn, fn and fp`. See `scalation.modeling.classifying.simpleLDATest2`.

8.5 Linear Discriminant Analysis

Like `SimpleLDA`, the `LDA` class support Linear Discriminant Analysis that is used for multiway classification of continuously valued data. Similarly, the response/classification variable can take on k possible values, $y \in \{0, 1, \dots, k-1\}$. Unlike `SimpleLDA`, this class is intended for cases where the feature vector $\mathbf{x}$ is multi-dimensional. The classification $y = c$ is chosen to maximize the conditional probability of class y given the n-dimensional data/feature vector $\mathbf{x}$.

$$P(y|\mathbf{x}) \;=\; \frac{f(\mathbf{x}|y)\,P(y)}{f(\mathbf{x})} \tag{8.49}$$

where

$$f(\mathbf{x}) \;=\; \sum_{c=0}^{k-1} f(\mathbf{x}|y=c)P(y=c)$$

In the multi-dimensional case, $\mathbf{x}|y$ has a multivariate Gaussian distribution, $Normal(\boldsymbol{\mu}_c, \Sigma)$, where $\boldsymbol{\mu}_c$ are the mean vectors $\mathbb{E}\,[\mathbf{x}|y=c]$ and Σ is the common covariance matrix (weighted average of $\mathbb{C}\,[\mathbf{x}|y=c]$. The conditional density function is given by

$$f(\mathbf{x}|y=c) \;=\; \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^{\intercal}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c)}$$

Dropping factors independent of c and multiplying by $P(y=c)$ gives

$$f(\mathbf{x}|y=c)P(y=c) \;\propto\; e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^{\intercal}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c)}P(y=c)$$

Taking the natural logarithm

$$\ln(P(y|x)) \;\propto\; -\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^{\intercal}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c) + \ln(P(y=c))$$

The discriminant functions are obtained by multiplying out and again dropping terms independing of c.

$$\delta_c(\mathbf{x}) \;=\; \mathbf{x}^{\intercal}\Sigma^{-1}\boldsymbol{\mu}_c - \frac{\boldsymbol{\mu}_c{}^{\intercal}\Sigma^{-1}\boldsymbol{\mu}_c}{2} + \ln(P(y=c)) \tag{8.50}$$

As in the last section, the means for each class c ($\boldsymbol{\mu}_c$), the common covariance matrix (Σ), and the class probabilities ($P(y)$) must be estimated.

8.5.1 `LDA` Class

Class Methods:

```
    @param x        the real-valued training/test data vectors stored as rows of a matrix
    @param y        the training/test classification vector, where y_i = class for row i
    @param fname_   the names for all features/variables (defaults to null)
    @param k        the number of classes (k in {0, 1, ...k-1}
    @param cname_   the names for all classes
    @param hparam   the hyper-parameters

    class LDA (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                 k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
               hparam: HyperParameter = Classifier.hp)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with FitC (y, k):

    def corrected_cov (xc: MatrixD): MatrixD = (xc.𝒯 * xc) / xc.dim
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int =
    override def predictI (z: VectorI): Int = predictI (z.toDouble)
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = parameter, vifs: VectorD = null): String =
```

8.5.2 Exercises

1. Use `LDA` to classify manufactured parts according whether they should pass quality control based on *curvature* and `diameter` tolerances. See `people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html` for details.

8.6 K-Nearest Neighbors Classifier

The `KNN_Classifier` class is used to classify a new vector $\mathbf{z}$ into one of k classes $y \in \{0, 1, \ldots, k-1\}$. It works by finding its κ-nearest neighbors to the point $\mathbf{z}$. These neighbors essentially vote according to their classification. The class with the most votes is selected as the classification of vector $\mathbf{z}$. Using a distance metric, the κ vectors nearest to $\mathbf{z}$ are found in the training data, which are stored row-wise in data matrix X. The corresponding classifications are given in vector $\mathbf{y}$, such that the classification for vector $\mathbf{x}_i$ is given by y_i.

In ScalaTion to avoid the overhead of calling `sqrt`, the square of the Euclidean distance is used (although other metrics can easily be swapped in). The squared distance from vector $\mathbf{x}$ to vector $\mathbf{z}$ is then

$$d(\mathbf{x}) \;=\; d(\mathbf{x}, \mathbf{z}) \;=\; \|\mathbf{x} - \mathbf{z}\|^2$$

The distance metric is used to collect the κ nearest vectors into set $top_\kappa(\mathbf{z})$, such that there does not exists any vector $\mathbf{x}_j \notin top_\kappa(\mathbf{z})$ that is closer to $\mathbf{z}$.

$$top_\kappa(\mathbf{z}) \;=\; \{\mathbf{x}_i | i \in \{0, \ldots, \kappa - 1\} \text{ and } \nexists(\mathbf{x}_j \notin top_\kappa(\mathbf{z}) \text{ and } d(\mathbf{x}_j) < d(\mathbf{x}_i)\}$$

In case of ties for the most distant point to include in $top_\kappa(\mathbf{z})$ one could pick the first point encountered or the last point. A less biased approach would be to randomly break the tie.

Now $y(top_\kappa(\mathbf{z}))$ can be defined to be the vector of votes from the members of the set, e.g., $y(top_3(\mathbf{z})) = [1, 0, 1]$. The ultimate classification is then simply the mode (most frequent value) of this vector (e.g., 1 in this case).

$$\hat{y} \;=\; \text{mode } \mathbf{y}(top_\kappa(\mathbf{z}))$$

8.6.1 Lazy Learning

Training in the `KNN_Classifier` class is lazy, i.e., the work is done in the `predictI/classify` methods, rather than the `train` method.

```
   override def predictI (z: VectorD): Int =
        nearest (z)                                         // set top-kappa to kappa nearest
        for i <- 0 until kappa do count(y(topK(i)._1)) += 1    // tally votes per class
        val best = count.argmax ()                             // class with maximal count
        reset ()                                               // reset topK and counters
        best                                                   // return the best
    end predictI
```

The `kNearest` method finds the κ $\mathbf{x}$ vectors closest to the given vector $\mathbf{z}$. This method updates `topK` by replacing the most distant $\mathbf{x}$ vector in `topK` with a new one if it is closer. Each element in the `topK` array is a tuple `(j, d(j))` indicating which vector and its distance from $\mathbf{z}$. Each of these selected vectors will have their vote taken, voting for the class for which it is labeled. These votes are tallied in the `count` vector. The class with the highest count will be selected as the best class.

8.6.2 `KNN_Classifier` Class

Class Methods:

```
    @param x        the input/data matrix
    @param y        the classification of each vector in x
    @param fname_   the names for all features/variables (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the names for all classes
    @param kappa    the number of nearest neighbors to consider
    @param hparam   the hyper-parameters

    class KNN_Classifier (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                          k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                          kappa: Int = 3, hparam: HyperParameter = null)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with FitC (y, k):

    def distance (x: VectorD, z: VectorD): Double = (x - z).normSq
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int =
    override def predictI (z: VectorI): Int = predictI (z.toDouble)
    def reset (): Unit =
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

8.6.3 Exercises

1. Create a `KNN Classifier` for the joint data matrix given below and determine its tp, tn, fn, fp values upon re-classification of the data matrix. Let $k = 3$. Use Leave-One-Out validation for computing tp, tn, fn, fp.

```
    //                          x1  x2  y
    val xy = MatrixD ((10, 3),  1,  5,  1,     // joint data matrix
                                2,  4,  1,
                                3,  4,  1,
                                4,  4,  1,
                                5,  3,  0,
                                6,  3,  1,
                                7,  2,  0,
                                8,  2,  0,
                                9,  1,  0,
                               10,  1,  0)
```

2. Under what circumstances would one expect a `KNN_Classifier` to perform better than LogisticRegression?

3. How could `KNN_Classifier` be adpated to work for prediction problems?

8.7 Decision Tree C45

The DecisionTree_C45 class implements a Decision Tree classifier that uses the C4.5 algorithm. The classifier is trained using an m-by-n data matrix X and an n-dimensional classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, \ldots, k-1$. Each column in the matrix represents a feature (e.g., Humidity). The value count **vc** vector gives the number of distinct values per feature (e.g., 2 for Humidity).

Depending on the data type of a column, ScalaTion's implementation of C4.5 works like ID3 unless the column is continuous. A column is flagged isCont if it is continuous or relatively large ordinal. For a column that isCont, values for the feature are split into a left group and a right group based upon whether they are $\leq$ or $>$ an optimal threshold, respectively.

Candidate thresholds/split points are all the mid points between all column values that have been sorted. The threshold giving the maximum entropy drop (or *gain*) is the one that is chosen.

8.7.1 Example Problem

Consider the following continuous version of the play tennis example. The x_1 and x_2 columns (Temperature and Humidity) are now listed as continuous measurements rather than as categories as was the case for ID3.

The Example_PlayTennis_Cont object is used to test integer/continuous classifiers.

```scala
// The Example_PlayTennis_Cont is the well-known classification problem on whether to play
// tennis based on given weather conditions.
// The 'Cont' version uses continuous values for Temperature and Humidity,
// @see sefiks.com/2018/05/13/a-step-by-step-c4-5-decision-tree-example

object Example_PlayTennis_Cont:

    // combined data matrix [ x | y ]
    // dataset ----------------------------------------------------------------
    // x0: Outlook:     Rain (0),   Overcast (1), Sunny (2)
    // x1: Temperature: Continuous
    // x2: Humidity:    Continuous
    // x3: Wind:        Weak (0),   Strong (1)
    // y:  the response/classification decision
    // variables/features:      x0    x1     x2      x3    y
    val xy = MatrixD ((14, 5),  2,    85,    85,     0,    0,       // day  1
                                2,    80,    90,     1,    0,       // day  2
                                1,    83,    78,     0,    1,       // day  3
                                0,    70,    96,     0,    1,       // day  4
                                0,    68,    80,     0,    1,       // day  5
                                0,    65,    70,     1,    0,       // day  6
                                1,    64,    65,     1,    1,       // day  7
                                2,    72,    95,     0,    0,       // day  8
                                2,    69,    70,     0,    1,       // day  9
                                0,    75,    80,     0,    1,       // day 10
                                2,    75,    70,     1,    1,       // day 11
                                1,    72,    90,     1,    1,       // day 12
                                1,    81,    75,     0,    1,       // day 13
                                0,    71,    80,     1,    0)       // day 14

    val fname = Array ("Outlook", "Temp", "Humidity", "Wind")       // feature/variable names
    val conts = Set (1, 2)                                    // set of continuous
    features
    val cname = Array ("No", "Yes")                           // class names for y
    val k     = cname.size                                    // number of classes

    val x = xy.not (?, 4)                                     // columns 0, 1, 2, 3
    val y = xy(?, 4).toInt                                    // column 4

end Example_PlayTennis_Cont
```

As with the ID3 algorithm, the C4.5 algorithm picks x_0 as the root node. This feature is not continuous and has three branches. Branch `b0` will lead to a node where as before x_3 is chosen. Branch `b1` will lead to a leaf node. Finally, branch `b2` will lead to a node where continuous feature x_2 is chosen.

Sub-problem $x_0 = 2$

Note that if $x_0 = 0$ or 1, the algorithm works like ID3. However, there is still some uncertainty left when $x_0 = 2$, so this node may be split and it turn out the split will involve continuous feature x_2. The sub-problem for Outlook: Rain (2) see Table 8.1 is defined as follows: Take all five cases/rows in the data matrix X for which $x_{-0} = 2$.

Table 8.1: Sub-problem for node x_0 and branch 2

Day	$\mathbf{x}_{:1}$	$\mathbf{x}_{:2}$	$\mathbf{x}_{:3}$	**y**
1	85	85	0	0
2	80	90	1	0
8	72	95	0	0
9	69	70	0	1
11	75	70	1	1

The distinct values for feature x_2 in sorted order are the following: [70.0 ,85.0, 90.0, 95.0]. Therefore, the candidate threshold/split points for continuous feature x_2 are their midpoints: [77.5, 87.5, 92.5]. Threshold 77.5 yields (0-, 2+) on the left and (3-, 0+) on the right, 87.5 yields (1-, 2+) on the left and (2-, 0+) on the right, and 92.5 yields (2-, 2+) on the left and (1-, 0+) on the right. Clearly, the best threshold value is 77.5. Since a continuous feature splits elements into low (left) and high (right) groups, rather than branching on all possible values, the same continuous feature may be chosen again by a descendant node.

8.7.2 `DecisionTree_C45` Class

Class Methods:

```
    @param x        the input/data matrix with instances stored in rows
    @param y        the response/classification vector, where y_i = class for row i
    @param fname_   the names for all features/variables (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the names for all classes
    @param conts   the set of feature indices for variables that are treated as continuous
    @param hparam  the hyper-parameters for the Decision Tree classifier

    class DecisionTree_C45 (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                            k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                            conts: Set [Int] = Set [Int] (),
                            hparam: HyperParameter = DecisionTree.hp)
          extends Classifier (x, y, fname_, k, cname_, hparam)
             with FitC (y, k)
             with DecisionTree:

    override def parameter: VectorD = VectorD (param)
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorI): Int = predictIrec (z)
    override def predictI (z: VectorD): Int = predictIrecD (z)
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

8.7.3 Pruning

8.7.4 DecisionTree_C45wp Class

Class Methods:

```
    @param x       the input/data matrix with instances stored in rows
    @param y       the response/classification vector, where y_i = class for row i
    @param fname_  the names for all features/variables (defaults to null)
    @param k       the number of classes (defaults to 2)
    @param cname_  the names for all classes
    @param conts   the set of feature indices for variables that are treated as continuous
    @param hparam  the hyper-parameters for the decision tree

    class DecisionTree_C45wp (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                              k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                              conts: Set [Int] = Set [Int] (),
                              hparam: HyperParameter = DecisionTree.hp)
          extends DecisionTree_C45 (x, y, fname_, k, cname_, conts, hparam):

    def prune (nPrune: Int = 1, threshold: Double = 0.98): Unit =
```

8.7.5 Exercises

1. Run `DecisionTree_C45` on the `Example_PlayTennis` dataset and verify that it produces the same answer as `DecisionTree_ID3`.

2. Complete the C45 Decision Tree for the `Example_PlayTennis_Cont` problem.

3. Run `DecisionTree_C45` on the `winequality-white` dataset. Plot the accuracy and F_1 measure versus the maximum tree height/depth (`height`).

8.8 Bagging Trees

Bootstrap does sampling with replacement, thus allowing many large sub-samples of a dataset or training set to be created. Bootstrap Aggregation (Bagging) allows a modeling technique to be applied on several sub-samples. As a decision tree is at risk of overfitting, it is a good candidate for bagging. A decision tree is created for each sub-sample. Given a new data point is to be classified, each tree makes its prediction and the majority vote is taken as the overall classification. When k is larger than 2, the plurality will be taken. Bagging works on the notion of "wisdom of the crowd": The consensus of several trees is more likely to be correct than the prediction of a single tree. Experimentation has borne this out.

8.8.1 Creating Subsample

The creation of a sub-sample is done by the `subSample` method in the `modeling.Sampling` class. It creates a random sub-sample of rows from data matrix `x` and elements from classification vector `y`, returning the sub-sample matrix and vector, as well as the indices selected `irows`.

```
    @param x        the original input/data matrix
    @param y        the original integer-valued output/response vector
    @param nSamp    the desired sample size (number of rows in matrix)
    @param stream   the random number stream to use

    def subSample (x: MatrixD, y: VectorI, nSamp: Int, stream: Int):
                  (MatrixD, VectorI, Array [Int]) =
        if nSamp >= x.dim then
            (x, y, null)
        else
            val rsg   = RandomVecSample (x.dim, nSamp, stream)
            val irows = rsg.igen
            (x(irows), y(irows), irows)
        end if
    end subSample
```

8.8.2 Training

Training involves creating a `DecisionTree_C45` classifier for each sub-sample and calling `train` for each tree.

```
    @param x_  the training/full data/input matrix (defaults to full x)
    @param y_  the training/full response/output vector (defaults to full y)

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
        for l <- 0 until nTrees do
            val (sub_x, sub_y, irows) = subSample (x_, y_, sampleSize, l)

            trees(l) = new DecisionTree_C45 (sub_x, sub_y, fname, k, cname, conts, hparam)
            trees(l).train ()
        end for
    end train
```

8.8.3 Hyper-parameters

The following hyper-parameters can be adjusted to improve the model: The number of trees (`nTrees`) to create has a major effect. The bagging ratio (`bRatio`) is the ratio of the size of the sub-sample to the size of dataset (or training set). Finally, the height/depth limit (`height`) puts a limit of the height of each decision tree.

```
    protected val nTrees = hparam ("nTrees").toInt
    private   val bRatio = hparam ("bRatio").toDouble
    private   val height = hparam ("height").toInt
```

Note, many (more than 50) trees may be needed to get good results for `BaggingTrees` and `RandomForest`.

8.8.4 `BaggingTrees` Class

Class Methods:

```
    @param x        the data matrix (instances by features)
    @param y        the response/class labels of the instances
    @param fname_   the names of the variables/features (defaults to null)
    @param k        the number of classes (defaults to 2)
    @param cname_   the names of the classes
    @param conts    the set of feature indices for variables that are treated as continuous
    @param hparam   the hyper-parameters for the bagging trees

    class BaggingTrees (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                        k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                        conts: Set [Int] = Set [Int] (),
                        hparam: HyperParameter = DecisionTree.hp)
          extends Classifier (x, y, fname_, k , cname_, hparam)
             with FitC (y, k):

    override def parameter: VectorD = null
    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int =
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
```

8.9 Random Forest

The `BaggingTrees` class provides diversity of opinion via bootstrap sampling and Random Forests increase the diversity by having each tree work potentially on different, yet closely related problems. Each tree selects a subset of the columns in the original data matrix.

The `RandomForest` class builds multiple decision trees for a given problem. Each decision tree is built using a sub-sample (rows) of the data matrix `x` as well as a subset of the features/variables (columns) of `x`.

As with `BaggingTrees`, the fraction of rows used is given by the bagging ratio `bRatio`, while the new `RandomForest` hyper-parameter `fbRatio` specifies the ratio between the number of columns selected and total number of columns `fbRatio` Reasonable values for these ratios are around 0.7 (70%).

Given a new instance vector `z`, each of the trees will classify it and the class with the most number of votes (one from each tree), will be the overall response of the random forest.

8.9.1 Extracting Sub-features

Starting with a sub-sample of the data matrix `x`, a subset of the features/columns may be selected by randomly generating the column positions to be included. The projected matrix and the column index positions selected are returned by the `selectSubFeatures` method.

```
@param sub_x  the sub-sample of data matrix x to select features/columns from

def selectSubFeatures (sub_x: MatrixD): (MatrixD, VectorI) =
    val columns = rvg.igen.sorted                       // column indices selected
    val x_sub_f = sub_x(?, columns)                     // extract selected columns
    (x_sub_f, columns)
end selectSubFeatures
```

8.9.2 Training

Training involves creating a `DecisionTree_C45` classifier for each sub-sample on selected features and calling `train` for each tree. The current selected features held in the `columns` variable and saved for the l^{th} tree in `jcols`. In addition, it is needed to extract to relevant feature names (`fname2`) and continuous column indicators (`conts2`).

```
@param x_  the training/full data/input matrix (defaults to full x)
@param y_  the training/full response/output vector (defaults to full y)

override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    for l <- 0 until nTrees do
        val (sub_x, sub_y, irows) = subSample (x_, y_, sampleSize, l)
        val (xf, columns) = selectSubFeatures (sub_x)
        val fname2 = columns.map (fname(_)).toArray
        val conts2 = conts.filter (columns contains _)
        jcols(l)   = columns

        trees(l) = new DecisionTree_C45 (xf, sub_y, fname2, k, cname, conts2, hparam)
        trees(l).train ()
    end for
end train
```

8.9.3 `RandomForest` Class

Class Methods:

```
@param x        the data matrix (instances by features)
@param y        the response class labels of the instances
@param fname_   the feature/variable names (defaults to null)
@param k        the number of classes  (defaults to 2)
@param cname_   the names of the classes
@param conts    the set of feature indices for variables that are treated as continuous
@param hparam   the hyper-parameters

class RandomForest (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                    k: Int = 2, cname_ : Array [String] = Array ("No", "Yes"),
                    conts: Set [Int] = Set [Int] (),
                    hparam: HyperParameter = DecisionTree.hp)
      extends BaggingTrees (x, y, fname_, k , cname_, conts, hparam):

def selectSubFeatures (xx: MatrixD, rStream: Int): (MatrixD, VectorI) =
override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
override def predictI (z: VectorD): Int =
```

8.9.4 Exercises

1. Compare `DecisionTree_C45`, `BaggingTrees` and `Random Forest` classifiers on the White-Wine, Breast Cancer, and Diabetes datasets. Use the default setting for the hyper-parameters.

2. Compare `BaggingTrees` and `Random Forest` classifiers on the White-Wine, Breast Cancer, and Diabetes datasets. Increase the number of trees from low default setting (Supreme Court 9) to 51, incrementing by 2. Plot the accuracy and F_1 measure versus the number of trees.

3. Compare `BaggingTrees` and `Random Forest` classifiers on the White-Wine, Breast Cancer, and Diabetes datasets. Increase the maximum height/depth of the trees from 3 to 20. Plot the accuracy and F_1 measure versus the maximum tree height.

8.10 Support Vector Machine

The idea behind support vector machines is actually quite simple and can be easy to follow when $x \in \mathbb{R}^2$ and $y \in \{-1, +1\}$. The negative responses can be indicated in a plot by "-" (or "o"), while positive responses can be indicted by "+" (or "*"). Consider the following Eight Point Example depicted in Figure 8.1 where the goal is to separate the two sets of points.

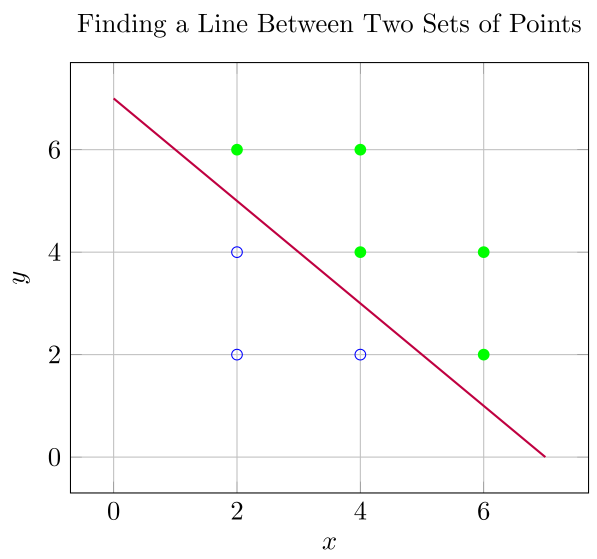


Figure 8.1: SVM: Eight Point Example

Suppose the points represent tank regimes for the blue (o) and green (*) armies. Your mission as a peace keeping force is split the two armies. Naturally, one should try to maximize the distance to any tank regime. At this point, the peace keeping force is to be placed in a line, although this can relaxed later on.

The first question is determine the equation of line. This can be done by lining a ruler up on the front of one army and moving it to toward the other army and stopping in the middle. Clearly that line is $x + y = 7$. This can verified to computing the distance to each of points and observing the minimum distance orthogonal to the line for each army is $\frac{1}{\sqrt{2}} = 0.7071$. This distance is referred to as the half margin and the goal is to maximize it. From a data science perspective it makes sense to maximum the margin as any point below the split line will be classified as -1, while any point above the split line will be classified as $+1$.

Notice that only five of eight points are relevant in determining the position/equation of the middle separting line (see the points on the red lines in Figure 8.2). These may be viewed as front-line points or more generally as *support vectors*.

8.10.1 Separating Hyperplane

In higher dimensions, the line separating the two sets of points becomes a hyperplane, i.e., the set points $\mathbf{x} \in \mathbb{R}^n$ satisfying the following equation,

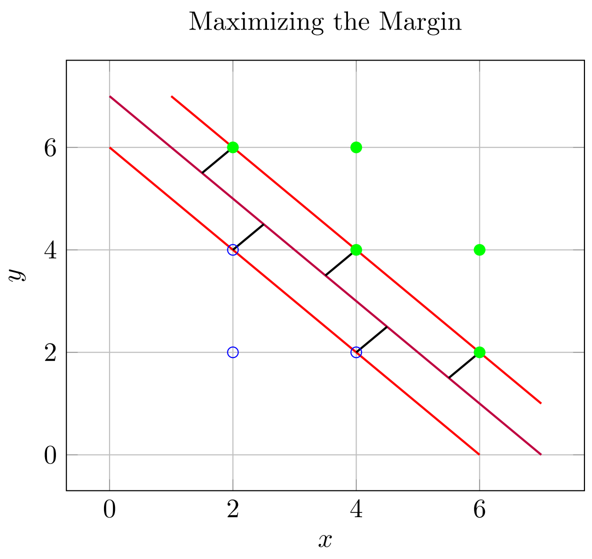


Figure 8.2: SVM: Support Vectors in Eight Point Example

$$\mathbf{w} \cdot \mathbf{x} - b \;=\; 0 \tag{8.51}$$

where $\mathbf{w}$ is a vector normal to the hyperplane and b is an offset. For the example, $\mathbf{w} = [1, 1]$ and $b = 7$. Note that $\mathbf{w}$ is the gradient of $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b$ w.r.t. $\mathbf{x}$.

The minimum directional distance from the hyperplane to the origin $\mathbf{0}$ is given by

$$\frac{b}{\|\mathbf{w}\|} \tag{8.52}$$

and equals $\frac{7}{\sqrt{2}} = 4.9497$ for the example problem. Subtracting this value would move the hyperplane to the origin.

The equations for the two red lines/hyperplanes are given by

$$\mathbf{w} \cdot \mathbf{x} - b \;=\; \text{-1} \tag{8.53}$$

$$\mathbf{w} \cdot \mathbf{x} - b \;=\; 1 \tag{8.54}$$

Note, making these two equations valid in general may require rescaling of $\mathbf{w}$. In this case, the equations can be verified as follows:

$$[1, 1] \cdot [2, 4] - 7 \;=\; \text{-1} \tag{8.55}$$

$$[1, 1] \cdot [2, 6] - 7 \;=\; 1 \tag{8.56}$$

The full margin is the distance between these two red lines/hyperplanes and is given by

$$\frac{2}{\|\mathbf{w}\|} \tag{8.57}$$

which equals $\frac{2}{\sqrt{2}} = \sqrt{2}$ for the example problem. Therefore, maximizing the margin is equivalent to minimizing the norm of the $\mathbf{w}$, $\|\mathbf{w}\|$.

8.10.2 Optimization Problem

Given a dataset $(X, \mathbf{y})$ with $\mathbf{x}_i$ being the i^{th} row of matrix $X \in \mathbb{R}^{m \times n}$ and y_i being the i^{th} element of vector $\mathbf{y} \in \mathbb{R}^m$, the goal of minimizing $\|\mathbf{w}\|$ can be cast as a constrained optimization problem:

$$\min \frac{1}{2}\|\mathbf{w}\|^2 \;=\; \min \frac{1}{2}\sum_{j=0}^{n-1} w_i^2 \tag{8.58}$$

subject to

$$\mathbf{w} \cdot \mathbf{x}_i - b \;\leq\; \text{-1} \quad \text{if } y_i = \text{-1} \tag{8.59}$$

$$\mathbf{w} \cdot \mathbf{x}_i - b \;\geq\; 1 \quad \text{if } y_i = 1 \tag{8.60}$$

The constraints can be written more concisely as follows:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \;\geq\; 1 \quad \text{for } i = 0, \ldots, m-1 \tag{8.61}$$

or in the form of standard inequality constraints,

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1 \;\geq\; 0 \quad \text{for } i = 0, \ldots, m-1 \tag{8.62}$$

This is a quadratic optimization problem with linear inequality constraints involving $n+1$ unknowns ($\mathbf{w}$ and b) and m constraints. Such optimization problems can be solved by introducing a Lagrange multiplier $\alpha_i \geq 0$ for each inequality constraint `https://aa.ssdi.di.fct.unl.pt/files/AA-09_notes.pdf`. Therefore, the Lagrangian (see the Appendix) is given by

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) \;=\; \frac{1}{2}\|\mathbf{w}\|^2 \;-\; \sum_{i=0}^{m-1} \alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \;-\; 1\right] \tag{8.63}$$

The problem is now to minimize the Lagrangian by finding optimal values for the parameters $\mathbf{w}$ and b. Taking the gradient of L w.r.t. $\mathbf{w}$ and b, and setting it equal to zero yields,

$$\frac{\partial L}{\partial \mathbf{w}} \;=\; \mathbf{w} - \sum_{i=0}^{m-1} \alpha_i y_i \mathbf{x}_i \;=\; \mathbf{0} \tag{8.64}$$

$$\frac{\partial L}{\partial b} \;=\; -\sum_{i=0}^{m-1} \alpha_i y_i \;=\; 0 \tag{8.65}$$

Dual Formulation

The dual form will reformulate the problem in terms of the Lagrange multipliers. As an exercise, show that

$$\|\mathbf{w}\|^2 \;=\; \mathbf{w} \cdot \mathbf{w} \;=\; \sum_i \sum_j \alpha_i y_i \mathbf{x}_i \cdot \alpha_j y_j \mathbf{x}_j \;=\; \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{8.66}$$

The rest of the formula can be rearranged as follows:

$$\sum_{i=0}^{m-1} \alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \;-\; 1\right] \;=\; \sum_i \alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i \;+\; \sum_i \alpha_i y_i b \;+\; \sum_i \alpha_i$$

The first term becomes

$$\sum_i \alpha_i y_i \left[\sum_j \alpha_j y_j \mathbf{x}_j \right] \cdot \mathbf{x}_i$$

The second term is zero due to the constraint $\sum_i \alpha_i y_i = 0$. Combining yields

$$L_D(\boldsymbol{\alpha}) = -\frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_i \alpha_i \tag{8.67}$$

The dual problem is formulated as a maximization problem (see the Appendix).

$$\max L_D(\boldsymbol{\alpha}) = -\frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_i \alpha_i \tag{8.68}$$

subject to

$$\sum_i \alpha_i y_i = 0 \qquad \text{and} \quad \alpha_i \geq 0 \tag{8.69}$$

See the exercise for algorithms to solve this optimization problem.

8.10.3 Running the Example Problem

```
    val xy = MatrixD ((8, 3), 2, 2, -1,                          // 8 data points
                              4, 2, -1,
                              2, 4, -1,
                              2, 6,  1,
                              4, 4,  1,
                              6, 2,  1,
                              6, 4,  1,
                              4, 6,  1)

    val (x, y) = (xy.not(?, 2), xy(?, 2).toInt)

    val mod = new SupportVectorMachine (x, y)                    // create optimizer
    mod.trainNtest ()()
    println (mod.summary ())
```

8.10.4 `SupportVectorMachine` Class

The `SupportVectorMachine` class implements linear support vector machines (SVM). A set of vectors stored in a matrix are divided into positive(1) and negative(-1) cases. The algorithm finds a hyperplane that best divides the positive from the negative cases. Each vector $\mathbf{x}_i$ is stored as a row in the X matrix.

Class Methods:

```
    @param x        the input/data matrix with points stored as rows
    @param y        the classification of the data points stored in a vector
    @param fname_   the feature/variable names (defaults to null)
    @param cname_   the names of the classes
    @param hparam   the hyper-parameters

```

```
class SupportVectorMachine (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                            cname_ : Array [String] = Array ("-", "+"),
                            hparam: HyperParameter = null)
      extends Classifier (x, y, fname_, 2, cname_, hparam)
         with FitC (y, k = 2):

override def parameter: VectorD = w :+ b
override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
override def predictI (z: VectorD): Int = if (w dot z) >= b then 1 else -1
override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                      b_ : VectorD = p_y, vifs: VectorD = null): String =
override def toString: String = "(w, b) = " + (w, b)
```

8.10.5 Exercises

1. Develop a `SupportVectorMachine` model for the Eight Point Example given in this section. What are **w** and *b*. What is the size of the margin?

2. Compare DecisionTree C45, BaggingTrees, Random Forest, and Support Vector Machine classifiers on the White-Wine, Breast Cancer, and Diabetes datasets.

3. Give pseudocode for Platt's Sequential Minimal Optimization (SMO) algorithm for SVM [146].

4. Explain the improvements to Platt's SMO algorithm suggested by Keerthi et al. [93].

8.11 Neural Network Classifiers

It is easy to use Neural Networks as classifiers. The last layer can be set up to provide a probability of outcome. For example, for two-way classification ($k = 2$) a single output node's value can indicate the probability of a positive outcome ($y = 1$). One minus this value is used as the indicator for the corresponding negative outcome ($y = 0$). Whichever is higher is then the predicted class. Of course for better balance of false positives and false negatives a threshold may be used.

8.11.1 Model Equation

As it produces values between 0 (negative outcome) and 1 (positive outcome), the sigmoid activation function $S(u)$ is often used for the last layer of a Neural Network classifier.

$$S(u) = \frac{1}{1 + e^{-u}} \tag{8.70}$$

The `NeuralNet_Class_3L` class supports single-output, 3-layer (input, hidden and output) Neural-Network classifiers. The model equation is a specialization of the one given for `NeuralNet_3L` (see the Chapter on Nonlinear Models and Neural Networks). Given an input vector $\mathbf{x} \in \mathbb{R}^n$ the model predicts a value which is considered off by ϵ.

$$\boxed{y \;=\; S(B \cdot \mathbf{f}(A \cdot \mathbf{x})) + \epsilon \;=\; S(B^{\intercal}\mathbf{f}(A^{\intercal}\mathbf{x})) + \epsilon} \tag{8.71}$$

where

- $\mathbf{f} : \mathbb{R}^{n_z} \to \mathbb{R}^{n_z}$ is the hidden layer activation function (user specified)
- $S : \mathbb{R} \to \mathbb{R}$ is the output layer activation function (sigmoid)
- $A \in \mathbb{R}^{n \times n_z}$ is the input-hidden parameter matrix with bundled $\boldsymbol{\alpha}$ vector
- $B \in \mathbb{R}^{n_z \times 1}$ is the hidden-output parameter matrix with bundled $\boldsymbol{\beta}$ vector
- $\epsilon \in \mathbb{R}$ is the residual/error

As discussed in the Chapter on Nonlinear Models and Neural Networks the parameter weight matrix and bias vector are bundled into a `NetParam` object.

The corresponding network diagram is shown in Figure 8.3. Its input layer has $n = 2$ nodes, hidden layer has $n_z = 3$ nodes, output layer 1 node.

8.11.2 Training Equation

Given a dataset $(X, \mathbf{y})$ consisting of a input/data matrix $X \in \mathbb{R}^{m \times n}$ and an output/response vector $\mathbf{y} \in \mathbb{R}^m$ (training data), the goal is to fit the parameters $A \in \mathbb{R}^{n \times n_z}$ and $B \in \mathbb{R}^{n_z \times 1}$ connecting the layers to minimize a loss function (e.g., sse or cross-entropy).

$$\boxed{\mathbf{y} \;=\; \mathbf{S}(\mathbf{f}(XA)B) + \boldsymbol{\epsilon}} \tag{8.72}$$

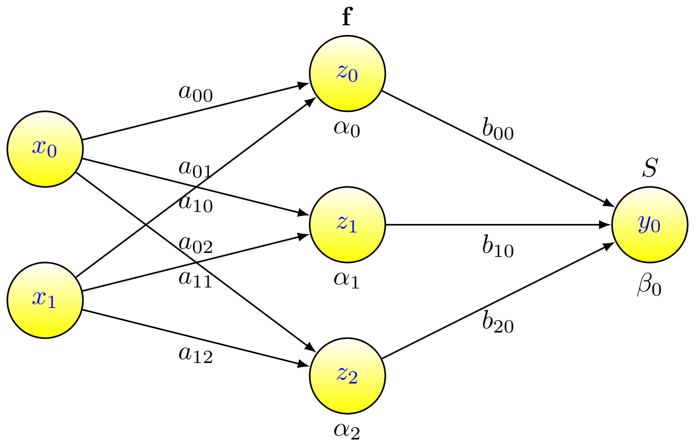


Figure 8.3: A Simple Three-Layer (input, hidden, output) Neural Network Classifier

8.11.3 Prediction Equation

Once trained, the network can classify the output value for a new input vector $\mathbf{z}$,

$$\hat{y} \;=\; S(B \cdot \mathbf{f}(A \cdot \mathbf{z})) \;=\; S(B^{\intercal}\mathbf{f}(A^{\intercal}\mathbf{z})) \tag{8.73}$$

If $\hat{y} > 0.5$ then return 1 (positive classification); otherwise return 0 (negative classification). More generally, the test can involve a threshold τ, i.e., $\hat{y} > \tau$.

8.11.4 Optimization

One may utilize the same loss that used for regression type problems (sse or mse) in which case the mse loss function is

$$\mathcal{L}(A, B) \;=\; \frac{1}{2m}\|\mathbf{y} - \mathbf{S}(\mathbf{f}(XA)B)\|^2 \tag{8.74}$$

Or using summation notation,

$$\mathcal{L}(A, B) \;=\; \frac{1}{2m}\sum_{i=0}^{m-1} y_i - S(B^{\intercal}\mathbf{f}(A^{\intercal}\mathbf{x}_i)) \tag{8.75}$$

However, a typically better alternative is to use *cross-entropy* for the loss function [135] (section 3.1).

$$\mathcal{L}(A, B) \;=\; -\frac{1}{m}\sum_{i=0}^{m-1} y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i) \tag{8.76}$$

Substituting for $\hat{y}_i$ yields,

$$\mathcal{L}(A, B) \;=\; -\frac{1}{m}\sum_{i=0}^{m-1} y_i \ln S(B^{\intercal}\mathbf{f}(A^{\intercal}\mathbf{x}_i)) + (1 - y_i)\ln\left(1 - S(B^{\intercal}\mathbf{f}(A^{\intercal}\mathbf{x}_i))\right) \tag{8.77}$$

Partial Derivatives

The partial derivative w.r.t, weight b_h connecting hidden node h with the output node (there is only one) is

$$\frac{\partial \mathcal{L}}{\partial b_h} \;=\; \frac{1}{m}\mathbf{z}_{:h} \cdot (\hat{\mathbf{y}} - \mathbf{y}) \tag{8.78}$$

The partial derivative w.r.t, weight a_{jh} connecting input node j with hidden node h is

$$\frac{\partial \mathcal{L}}{\partial a_{jh}} \;=\; \frac{1}{m}... \tag{8.79}$$

See the Probability Chapter for more information about cross-entropy and the Nonlinear Models and Neural Network Chapter for computing gradients (and their partial derivatives) of loss functions. See the exercises for partial derivatives for the biases $\boldsymbol{\alpha}$ and β.

8.11.5 NeuralNet_Class_3L Class

Class Methods:

```
    @param x        the m-by-n input matrix (training data consisting of m input vectors)
    @param y        the m output vector (training data consisting of m output integer values)
    @param fname_   the feature/variable names (defaults to null)
    @param cname_   the names for all classes
    @param nz       the number of nodes in hidden layer (-1 => use default formula)
    @param hparam   the hyper-parameters for the model/network
    @param f        the activation function family for layers 1->2 (input to hidden)
                    the activation function family for layers 2->3 (hidden to output) is
    sigmoid

    class NeuralNet_Class_3L (x: MatrixD, y: VectorI, fname_ : Array [String] = null,
                              cname_ : Array [String] = Array ("No", "Yes"),
                              nz: Int = -1, hparam: HyperParameter = NeuralNet_Class_3L.hp,
                              f: AFF = f_tanh)
          extends Classifier (x, y, fname_, 2, cname_, hparam)
             with FitC (y, 2):

    override def train (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    override def train2 (x_ : MatrixD = x, y_ : VectorI = y): Unit =
    def trainNtest (x_ : MatrixD = x, y_ : VectorI = y)
    def trainNtest2 (x_ : MatrixD = x, y_ : VectorI = y)
    def test (x_ : MatrixD = x, y_ : VectorI = y): (VectorI, VectorD) =
    override def predictI (z: VectorD): Int = (nn3.predict (z)(0) + cThresh).toInt
    override def summary (x_ : MatrixD = null, fname_ : Array [String] = null,
                          b_ : VectorD = p_y, vifs: VectorD = null): String =
```

8.11.6 Exercises

1. Compare the QoF measures for Neural_Class_3L using sse vs. cross-entropy for the loss function. Do this for the Breast Cancer and Diabetes datasets.

2. Suppose model 1 predicts 0.1 when the actual value is 1 and 0.9 when its 0. Consider the following two vectors: $\mathbf{y} = [1, 0]$ and $\hat{\mathbf{y}} = [0.1, 0.9]$. Compute the cross-entropy loss function

$$\mathcal{L} \ = \ -\frac{1}{2}(1 \ln 0.1 + 0 \ln 0.9)$$

Now suppose for model 2 the predictions are in closer agreement with the actual values $\hat{\mathbf{y}} = [0.1, 0.9]$. Recompute the cross-entropy loss function

$$\mathcal{L} \ = \ -\frac{1}{2}(1 \ln 0.9 + 0 \ln 0.1)$$

Which model is better? How do these two examples help explain why cross-entropy works as a loss function.

3. Derive the results for the partial derivative w.r.t. weight b_h, $\dfrac{\partial \mathcal{L}}{\partial b_h}$.

4. Derive the results for the partial derivative w.r.t. weight a_{jh}, $\dfrac{\partial \mathcal{L}}{\partial a_{jh}}$.

5. Derive the results for the partial derivative w.r.t. bias α_h, $\dfrac{\partial \mathcal{L}}{\partial \alpha_h}$.

6. Derive the results for the partial derivative w.r.t. bias β, $\dfrac{\partial \mathcal{L}}{\partial \beta}$.

Chapter 9

Generalized Linear Models and Regression Trees

General Linear Models presented in Prediction Chapter cover a wide range of simple models that are easy to use and explain and can be rapidly trained typically using Ordinary Least Squares (OLS) that boils down to matrix factorization.

Complex nonlinear models can provide improved Quality of Fit and handle cases where General Linear Models are not sufficiently predictive. However, they often result in the following disadvantage: harder to use and explain, substantially longer training times, more likely to overfit and require tuning of hyper-parameters.

This chapter examines two categories of modeling techniques that fall between the two extreme categories of models just discussed. The two categories of models of intermediate complexity are Generalized Linear Models and Regression Trees.

9.1 Generalized Linear Model

A Linear Model may be viewed as fitting a distribution to a continuous random variable y.

$$y \sim \text{Normal}(\mathbf{b} \cdot \mathbf{x}, \sigma^2) \tag{9.1}$$

By subtracting the mean $\mathbf{b} \cdot \mathbf{x}$, y becomes the sum of two terms.

$$y = \mathbf{b} \cdot \mathbf{x} + \epsilon \tag{9.2}$$

where $\epsilon \sim \text{Normal}(0, \sigma^2)$.

Now generalize the first term $\mathbf{b} \cdot \mathbf{x}$ into an optionally transformed linear function of $\mathbf{x}$ called the mean function.

$$y = \mu_y(\mathbf{x}) + \epsilon \tag{9.3}$$

The mean function is restricted to be of the following form.

$$g(\mu_y(\mathbf{x})) = \mathbf{b} \cdot \mathbf{x} \tag{9.4}$$

where g is an invertible function. One may think of g is as a link function, or its inverse g^{-1} as an activation function. The idea of a *link* function is that it uncovers an underlying linear model $\mathbf{b} \cdot \mathbf{x}$. In other words, g is a function that links y's mean to a linear combination of the predictor variables. The idea of an *activation* function is that is allows a transformation function to be applied to the linear combination $\mathbf{b} \cdot \mathbf{x}$, as is done in Perceptrons.

$$\mu_y(\mathbf{x}) \;=\; g^{-1}(\mathbf{b} \cdot \mathbf{x}) \tag{9.5}$$

For example, a commonly used activation function in Perceptrons is the sigmoid function.

$$\mu_y(\mathbf{x}) \;=\; \text{sigmoid}(\mathbf{b} \cdot \mathbf{x}) \tag{9.6}$$

where the *sigmoid* function (a simple form of the more general logistic function) is

$$\text{sigmoid}(t) \;=\; \frac{1}{1 + e^{-t}} \tag{9.7}$$

The inverse of the sigmoid function is the logit function.

$$\text{logit}(u) \;=\; \ln \frac{u}{1 - u} \tag{9.8}$$

Applying the logit function to both sides of equation yields

$$\text{logit}(\mu_y(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x} \tag{9.9}$$

The form of the response random variable y is also generalized, e.g., it may be a discrete random variable.

So far, we have been focusing on the first term, the mean function, but what about the second term, the residual/error term? The treatment of residuals/errors in Generalized Linear Models, is generalized as well. For one, they need not follow the Normal distribution. Also, they are not required to be additive, e.g., they may be multiplicative.

To produce this level of generality/flexibility the methodology for Generalized Linear Models is to separate the model into a *systematic component* to determine the mean function, and a *random component* to handle the residuals/errors. Continuing with the mean function above, let y have domain $D_y = \{0, 1\}$ and the errors ϵ be additive and distributed as Bernoulli(p). Writing the two components together yields

$$\begin{aligned} y \;&=\; \mu_y(\mathbf{x}) + \epsilon \\ \text{logit}(\mu_y(\mathbf{x})) \;&=\; \mathbf{b} \cdot \mathbf{x} \end{aligned}$$

These are the model equations of Logistic Regression. When the link function is the identity function, the response random variable is continuous and the errors ϵ distributed as Normal$(0, \sigma^2)$, we are back to a Linear Model, written as a Generalized Linear Model.

$$\begin{aligned} y \;&=\; \mu_y(\mathbf{x}) + \epsilon \\ \mu_y(\mathbf{x}) \;&=\; \mathbf{b} \cdot \mathbf{x} \end{aligned}$$

Several additional combinations of link functions and residual distributions are commonly used as shown in the table below.

Model Type	**Response Type (y)**	**Link Function**	**Residual Distribution**
Logistic Regression	binary $\{0, 1\}$	logit	Bernoulli Distribution
Poisson Regression	integer $\{0, \ldots, \infty\}$	ln	Poisson Distribution
Exponential Regression	continuous $[0, \infty)$	ln or reciprocal	Exponential Distribution
General Linear Model (GLM)	continuous $(-\infty, \infty)$	identity	Normal Distribution

Table 9.1: Types of Generalized Linear Models

See `http://idiom.ucsd.edu/~rlevy/lign251/fall2007/lecture_13.pdf` and `http://link.springer.com/article/10.1023%2FA%3A1022436007242#page-1`. for additional details.

Since the response variable for Logistic Regression is defined on finite domains, it has been placed under Classification (see the last chapter).

9.2 Maximum Likelihood Estimation

In this section, rather than estimating parameters using *Least Squares Estimation* (LSE), *Maximum Likelihood Estimation* (MLE) will be used. For a deeper view of estimation concepts see [15]. Given a dataset with m instances, the model will produce an error for each instance. When the error is large, the model is in disagreement with the data. When errors are normally distributed, the probability density will be low for a large error, meaning this is an unlikely case. If this is true for many of the instances, the problem is not the data, it is the values given for the parameters. The parameter vector $\mathbf{b}$ should be set to maximize the likelihood of seeing instances in the dataset. This notion is captured in the likelihood function $L(\mathbf{b})$. Note, for the Simpler Regression model there is only a single parameter, the slope b.

Given an m instance dataset $(\mathbf{x}, \mathbf{y})$ where both are m-dimensional vectors and a Simpler Regression model

$$y \ = \ bx + \epsilon$$

where $\epsilon \sim N(0, \sigma^2)$, let us consider how to estimate the parameter b. While LSE is based upon the distance of errors from zero, MLE transforms this distance based upon for example the pdf of the error distribution. Notice that for small errors, the Normal distribution is more tolerant than the Exponential distribution, while for larger errors it is less tolerant. See the exercises for how to visualize this.

For this model and dataset, the likelihood function $L(b)$ is the product of m Normal density functions (making the assumption that the instances are independent).

$$L(b) \ = \ \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi}\sigma} e^{-\epsilon_i^2/2\sigma^2}$$

Since $\epsilon_i = y_i - bx_i$, we may rewrite $L(b)$ as

$$L(b) \ = \ \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi}\sigma} e^{-(y_i - bx_i)^2/2\sigma^2}$$

Taking the natural logarithm gives the log-likelihood function $l(b)$

$$l(b) \ = \ \sum_{i=0}^{m-1} -\ln(\sqrt{2\pi}\sigma) - (y_i - bx_i)^2/2\sigma^2$$

The derivative of $l(b)$ w.r.t. b is

$$\frac{dl}{db} \ = \ \sum_{i=0}^{m-1} -2x_i(y_i - bx_i)/2\sigma^2$$

For optimization, the derivative may be set to zero

$$\sum_{i=0}^{m-1} x_i(y_i - bx_i) \ = \ 0$$

Solving for b gives

$$b \ = \ \frac{\sum x_i y_i}{\sum x_i^2} \ = \ \frac{\mathbf{x} \cdot \mathbf{y}}{\mathbf{x} \cdot \mathbf{x}}$$

9.2.1 Akaike Information Criterion

The Akaike Information Criterion (AIC) serves as an alternative to measures like R^2-Adjusted, $\bar{R}^2$.

$$\text{AIC} \;=\; 2(n+1) - 2l(\mathbf{b}, \sigma^2)$$

It is twice the number of parameters $n+1$ to be estimated minus twice the optimal log-likelihood. Note, the n comes from estimating the coefficients $\dim(\mathbf{b})$ and 1 comes from estimating the error/redidual variance σ^2. The smaller the AIC value, the better the model.

For linear regression, the above formula is equivalent to

$$\text{AIC} \;=\; 2(n+1) - m \ln\left(\frac{sse}{m}\right) + constant$$

and the contant may be ignored as all models for the given dataset will have the same constant.

9.2.2 MLE for Generalized Linear Models

Maximum Likelihood Estimation (MLE) is general purpose and applies widely to statistical estimation problems.

Let us consider how it can be applied to Generalized Linear Models. A Generalized Linear Model for a response random variable y is of the following form.

$$\begin{aligned} y &= \mu_y(\mathbf{x}) \circledast \epsilon \\ g(\mu_y(\mathbf{x})) &= \mathbf{b} \cdot \mathbf{x} \end{aligned}$$

The $\circledast$ is used to indicate either + (for additive errors) or $\times$ (for multiplicative errors).

Now given an m-instance dataset $(X, \mathbf{y})$, we need to determine the link function g, the residual/error distribution $\epsilon \sim \text{Dist}(\mathbf{b})$, and finally the parameters $\mathbf{b}$ using MLE.

When y is continuous, the pdf conditioned on the parameters may be used to capture the error distribution.

$$f_\epsilon(\epsilon|\mathbf{b})$$

The error for each instance may be calculated from the dataset.

$$\begin{aligned} \epsilon_i &= y_i - g^{-1}(\mathbf{b} \cdot \mathbf{x}_i) && \text{for additive errors} \\ \epsilon_i &= y_i / g^{-1}(\mathbf{b} \cdot \mathbf{x}_i) && \text{for multiplicative errors} \end{aligned}$$

When errors are highly correlated, Generalized Linear Models are not ideal. Thus, we assume the errors are independent and identically distributed (iid).

Because of the independence assumption, the joint error density is the product of the density for each instance ϵ_i, Therefore, the likelihood function w.r.t. the parameter vector $\mathbf{b}$ is

$$L(\mathbf{b}) \;=\; \prod_{i=0}^{m-1} f_\epsilon(\epsilon_i|\mathbf{b})$$

Taking the natural logarithm yields

$$l(\mathbf{b}) \ = \ \sum_{i=0}^{m-1} \ln f_{\epsilon}(\epsilon_i|\mathbf{b}) \tag{9.10}$$

Analagously, for discrete random variables, the pdf is replaced with pmf

$$l(\mathbf{b}) \ = \ \sum_{i=0}^{m-1} \ln p_{\epsilon}(\epsilon_i|\mathbf{b}) \tag{9.11}$$

For Maximum Likelihood Estimation, the above two equations may serve as a starting point.

9.3 Poisson Regression

The PoissonRegression class can be used for developing Poisson Regression models. In this case, a response y may be thought of as a count that may take on a non-negative integer value. The probability density function (pdf) for the Poisson distribution with mean λ may be defined as follows:

$$f(y; \lambda) \;=\; \frac{\lambda^y}{y!} e^{-\lambda}$$

Again, treating this as a Generalized Linear Model problem,

$$y \;=\; \mu(\mathbf{x}) + \epsilon$$

$$g(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x}$$

The link function g for Poisson Regression is the ln (natural logarithm) function.

$$\ln(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x}$$

The residuals ϵ_i are distributed according to the Poisson distribution.

$$\frac{\mu(\mathbf{x}_i)^{y_i}}{y_i!} e^{-\mu(\mathbf{x}_i)}$$

Therefore, the likelihood function for Poisson Regression is as follows:

$$L \;=\; \prod_{i=0}^{m-1} \frac{\mu(\mathbf{x}_i)^{y_i}}{y_i!} e^{-\mu(\mathbf{x}_i)}$$

Taking the natural logarithm gives the log-likelihood function.

$$l \;=\; \sum_{i=0}^{m-1} y_i \ln(\mu(\mathbf{x}_i) - \mu(\mathbf{x}_i) - \ln(y_i!)$$

Substituting $\mu(\mathbf{x}_i) \;=\; e^{\mathbf{b} \cdot \mathbf{x}_i}$ yields the following:

$$l \;=\; \sum_{i=0}^{m-1} y_i \mathbf{b} \cdot \mathbf{x}_i - e^{\mathbf{b} \cdot \mathbf{x}_i} - \ln(y_i!)$$

Since the last term is independent of the parameters, removing it will not affect the optimization.

$$l \;=\; \sum_{i=0}^{m-1} y_i \mathbf{b} \cdot \mathbf{x}_i - e^{\mathbf{b} \cdot \mathbf{x}_i}$$

See http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf for more details.

Example Problem:

9.3.1 PoissonRegression Class

Class Methods:

```
    @param x        the data/input matrix augmented with a first column of ones
    @param y        the integer response/output vector, y_i in {0, 1, ... }
    @param fname_   the names of the features/variables (defaults to null)
    @param hparam   the hyper-parameters (currently has none)

    class PoissonRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                          hparam: HyperParameter = null)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def ll (b: VectorD): Double =
    def ll_null (b: VectorD): Double =
    def train (x_ : MatrixD = x, y_ : VectorD = y.toDouble): Unit =
    def train_null (): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def fit: VectorD =
    override def predict (z: VectorD): Double = round (exp (b dot z)).toDouble
    def buildModel (x_cols: MatrixD): PoissonRegression =
```

9.4 Regression Trees

As with Decision (or Classification) Trees, Regression Trees make predictions based upon what range each variable/feature is in. If the tree is binary, there are two ranges for each feature split: low (below a threshold) and high (above a threshold). Building a Regression Tree essentially then requires finding thresholds for splitting variables/features. A threshold will split a dataset into two groups. Letting θ_k be a threshold for splitting variable x_j, we may split the rows in the X matrix into left and right groups.

$$\text{left}_k(X) = \{\mathbf{x}_i | x_{ij} \leq \theta_k\} \tag{9.12}$$

$$\text{right}_k(X) = \{\mathbf{x}_i | x_{ij} > \theta_k\} \tag{9.13}$$

For splitting variable x_j, the threshold θ_k should be chosen to minimize the weighted sum of the Mean Squared Error (MSE) of the left and right sides. Alternatively, one can minimize the Sum of Squared Errors (SSE). This variable becomes the root node of the regression tree. The dataset for the root node's left branch consists of $\text{left}_k(X)$, while the right branch consists of $\text{right}_k(X)$. If the maximum tree depth is limited to one, the root's left child and right child will be leaf nodes. For a leaf node, the prediction value that minimizes MSE is the mean $\mu(y)$, see exercises.

9.4.1 Example Problem

Consider the following small dataset with just one predictor variable x_0.

```
val x = MatrixD ((10, 1), 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val y = VectorD (5.23, 5.7, 5.91, 6.4, 6.8, 7.05, 8.9, 8.7, 9.0, 9.05)
```

Given a limit on the *depth* of the tree, nodes are split recursively, starting with the root. The process terminates when the limit is reached or improvement is deemed inadequate. Each split adds a constraint on a variable.

Depth = 1

In this case, $\theta_0 = 6.5$ divides the dataset into

$$\text{left}_0(X) = \{1, 2, 3, 4, 5, 6\}$$

$$\text{right}_0(X) = \{7, 8, 9, 10\}$$

```
Root (-Inf, Inf]
    Leaf x0 in (-Inf, 6.5]
    Leaf x0 in (6.5, Inf]
```

with means $\mu_0(y) = 6.18$ (left) and $\mu_1(y) = 8.91$ (right).

Figure 9.1 shows a regression tree of depth 1. The constraints are shown on edges and the MSE and number of qualifying instances are shown by the node.

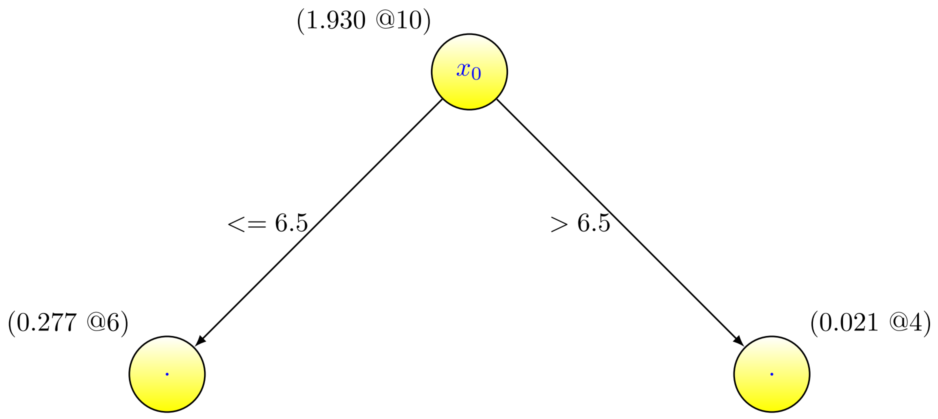


Figure 9.1: Regression Tree Example: Depth = 1

Depth = 2

Further splitting may occur on x_0 (or x_j for multidimensional examples). If we let the maximum tree depth be two, we obtain the following four regions, corresponding to the four leaf nodes,

```
Root (-Inf, Inf]
    Node x0 in (-Inf, 6.5]
        Leaf x0 in (-Inf, 3.5]
        Leaf x0 in (3.5, 6.5]
    Node x0 in (6.5, Inf]
        Leaf x0 in (6.5, 8.5]
        Leaf x0 in (8.5, Inf]
```

with means $\mu_0(y) = 5.61$, $\mu_1(y) = 6.75$, $\mu_2(y) = 8.80$ and $\mu_3(y) = 9.03$. Each internal (non-leaf) node will have a threshold. They are $\theta_0 = 6.5$, $\theta_1 = 3.5$ and $\theta_2 = 8.5$.

9.4.2 Regions

The number of regions (or leaf nodes) is always one greater than the number of thresholds. The *region* for leaf node l, $R_l = (x_j, (a_l, b_l])$, defines the feature/variable being split and the interval of inclusion. Corresponding to each region R_l is an *indicator function*,

$$\mathbb{1}_l(\mathbf{x}) \ = \ x_j \in (a_l, b_l] \tag{9.14}$$

which simply indicates {false, true} or {0, 1} whether variable x_j is in the interval $(a_l, b_l]$. Now define $\mathbb{1}^*_l(\mathbf{x})$ as the product of the indicator functions from leaf l until (not including) the root of the tree,

$$\mathbb{1}^*_l(\mathbf{x}) \ = \ \prod_{h \in \text{anc}(l)} \mathbb{1}_h(\mathbf{x}) \tag{9.15}$$

where anc(l) is the set of ancestors of leaf node l (inclusive of l, exclusive of root). Since only one of these $\mathbb{1}^*$ indicator functions can be true for any given $\mathbf{x}$ vector, we may concisely express the regression tree model as follows:

$$y \;=\; \sum_{l\,\in\, leaves} \mathbb{1}_l^*(\mathbf{x})\,\mu_l(y) \;+\; \epsilon \tag{9.16}$$

Thus, given a predictor vector $\mathbf{x}$, predicting a value for the response variable y corresponds to taking the mean y-value of the vectors in $\mathbf{x}$'s composite region (the intersection of regions from the leaf until the root). The prediction function to compute $\hat{y}$ is thus piecewise constant.

$$\hat{y} \;=\; \sum_{l\,\in\, leaves} \mathbb{1}_l^*(\mathbf{x})\,\mu_l(y) \tag{9.17}$$

As locality determines the prediction for Regression Trees, they are similar to KNN Regression.

9.4.3 Determining Thresholds

For the k^{th} split, a simple way to determine the best threshold is to take each feature/variable x_j and find a value θ_k that minimizes the weighted sum of the MSEs.

$$\min_{\theta_k}\; w_l\, \text{mse}(\text{left}_k(X)) + w_r\, \text{mse}(\text{right}_k(X)) \tag{9.18}$$

where w_l and w_r are the weights for the left and right sides, respectively. The overall score for a tree is then just the weighted sum of the MSEs for all leaf nodes, where the weight is the fraction of the instances that qualify for that leaf node.

Possible values for θ_k are the values between any two consecutive values in vector $\mathbf{x}_{:j}$ sorted. This will allow any possible split of $\mathbf{x}_{:j}$ to be considered. For example, $\{1, 10, 11, 12\}$ should not be split in the middle, e.g., into $\{1, 10\}$ and $\{11, 12\}$, but rather into $\{1\}$ and $\{10, 11, 12\}$. Possible thresholds (split points) are the averages of any two consecutive values, i.e., 5.5, 10.5 and 11.5. A straitforward way to implement determining the next variable x_j and its threshold θ_k would be to iterate over all features/variables and split points. Calculating the weighted sum of left and right mse from scratch for each candidate split point is inefficient. These values may be computed incrementally using the fast thresholding algorithm [40]. See [195] for derivations of efficient algorithms.

9.4.4 `RegressionTree` Class

Class Methods:

```
    @param x              the m-by-n input/data matrix
    @param y              the response m-vector
    @param fname_         the names of the features/variables (defaults to null)
    @param hparam         the hyper-parameters for the model
    @param curDepth       current depth
    @param branchValue    the branch value for the tree node
    @param feature        the feature for the tree's parent node
    @param leaves         the leaf counter

    class RegressionTree (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                      hparam: HyperParameter = RegressionTree.hp, curDepth: Int = 0,
                      branchValue: Int = -1, feature: Int = -1, leaves: Counter = Counter ()
    )
```

```
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def numLeaves: Int = leaves.get
    def train (x_ : MatrixD, y_ : VectorD): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    def printTree (): Unit =
    def printTree2 (): Unit =
    override def predict (z: VectorD): Double =
    override def predict (z: MatrixD = x): VectorD =
    override def buildModel (x_cols: MatrixD): RegressionTree =
```

9.5 Linear Model Trees

The regression trees in the last section have prediction functions that are piecewise constant, while those in this section are piecewise linear. Each region in the previous section is covered by a flat surface, while the regions in this section are covered by hyperplanes. These types of regression trees are also called Model Trees (e.g., M5 [148]).

At each leaf node, rather than taking the average of all the points (like using a Null Model), a multiple linear regression model is used. As such more data points are needed in each leaf, implying the need to have a sufficiently large dataset and typically have less splitting (smaller tree depth).

As the degrees of freedom in leaves may become small, it is useful to use Stepwise Refinement to reduce the number of parameters.

9.5.1 Splitting

A node should only be split if its multiple regression model is significantly worse than the combination of the two regression models of its children.

9.5.2 Pruning

9.5.3 Smoothing

The response surface may be made more smooth by taking a weighted average of the predictions of all models from the root to the leaf.

9.5.4 `RegressionTreeMT` class

```
@param x            the m-by-n input/data matrix
@param y            the response m-vector
@param fname_       the names of the features/variables (defaults to null)
@param hparam       the hyper-parameters for the model
@param curDepth     current depth
@param branchValue  the branch value for the tree node
@param feature      the feature for the tree's parent node
@param leaves       the leaf counter

class RegressionTreeMT (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                        hparam: HyperParameter = RegressionTree.hp, curDepth: Int = 0,
                        branchValue: Int = -1, feature: Int = -1, leaves: Counter =
Counter ())
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def numLeaves = leaves.get
def train (x_ : MatrixD, y_ : VectorD): Unit =
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
def printTree (): Unit =
def printTree2 (): Unit =
override def predict (z: VectorD): Double =
override def predict (z: MatrixD = x): VectorD =
override def buildModel (x_cols: MatrixD): RegressionTreeMT =
```

9.6 Random Forest Regression

Random Forest Regression takes multiple regreession trees and averages their predictions. The trees are used in parallel.

9.6.1 RegressionTreeRF Class

```
    @param x        the data matrix (instances by features)
    @param y        the response/class labels of the instances
    @param fname_   the names of the variables/features (defaults to null)
    @param hparam   the hyper-parameters to the random forest

    class RegressionTreeRF (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                            hparam: HyperParameter = hp)
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def train (x_ : MatrixD, y_ : VectorD): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def predict (z: VectorD): Double =
    override def buildModel (x_cols: MatrixD): RegressionTreeGB =
```

9.7 Gradient Boosting Regression

Gradient Boosting Regression tries to correct the predictions f_m of a regression tree by using a subsequent tree (with prediction function f_{m+1}) to predict the stage m residuals/errors, $\boldsymbol{\epsilon} = \mathbf{y} - f_m(X)$. The trees are used sequencially in M stages and the corrections are moderated by a learning rate.

9.7.1 `RegressionTreeGB` Class

```
@param x        the input/data matrix
@param y        the output/response vector
@param fname_   the feature/variable names (defaults to null)
@param hparam   the hyper-parameters for the model

class RegressionTreeGB (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                        hparam: HyperParameter = RegressionTree.hp)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def train (x_ : MatrixD, y_ : VectorD): Unit =
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
override def predict (z: VectorD): Double =
override def predict (z: MatrixD = x): VectorD =
override def buildModel (x_cols: MatrixD): RegressionTreeGB =
```

9.8 Exercises

1. Explain the difference between AIC and (Bayesian Information Criterion) BIC.

2. Let $x \sim$ Exponential(λ) and $y \sim$ Normal(μ, σ^2) with $\lambda = 1$, $\mu = 0$ and $\sigma = 1$, the corresponding pdfs are as follows:

$$\begin{aligned} f_x(x) &= e^{-x} \\ f_y(x) &= \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \end{aligned}$$

   Plot and compare the pdfs f_x and f_y vs. x over the interval $[0, 4]$.

3. For Regression Trees, show that for each leaf node that the optimal value for the constant is the mean.

4. Consider the following two-dimensional Regression Tree problem. FIX.

5. Create Regression Tree models for the `Example_AutoMPG` dataset. Compare the results for multiple depths.

6. How can bagging and boosting be used to improve upon Regression Trees.

7. Contrast KNN Regression with Regression Trees in terms of the shape of and how regions are formed.

9.9 Further Reading

1. Inductive Learning of Tree-based Regression Models [195] `https://www.dcc.fc.up.pt/~ltorgo/PhD/`
2. Generalized Linear Models, Second Edition [121]
3. Generalized Linear Models (GLM) [191]
4. Optimal Classification and Regression Trees with Hyperplanes Are as Powerful as Classification and Regression Neural Networks,
`https://dbertsim.mit.edu/pdfs/papers/2018-sobiesk-optimal-classification-and-regression-trees.pdf`

Chapter 10

Nonlinear Models and Neural Networks

10.1 Nonlinear Regression

The `NonlinearRegression` class supports Nonlinear Regression (NLR). In this case, the vector of input/predictor variable $\mathbf{x} \in \mathbb{R}^n$ can be multi-dimensional $[1, x_1, ...x_k]$ and the function f is nonlinear in the parameters $\mathbf{b} \in \mathbb{R}^p$.

10.1.1 Model Equation

As before, the goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; f(\mathbf{x}; \mathbf{b}) + \epsilon} \tag{10.1}$$

where ϵ represents the residuals (the part not explained by the model). The function $f : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}$ is called the *mean response function*.

Note that $y \;=\; b_0 + b_1 x_1 + b_2 x_1^2 + \epsilon$ is still *linear in the parameters*. The example below is not, as there is no transformation that will make the formula linear in the parameters.

$$y \;=\; (b_0 + b_1 x_1)/(b_2 + x_1) + \epsilon$$

Nonlinear Regression can more precisely model phenomena compared to using the linear approximations of multiple linear regression. Although quadratic and cubic regressions allow the lines to be bent to better fit the data, they do not provide the flexibility of nonlinear regression, see Chapter 13 of [102]. The functional forms in nonlinear regressions can be selected by the physics driving the phenomena. At the other extreme, simple nonlinear functions of linear combinations of several input variables, allow many diverse phenomena to be modeled, for example, using Neural Networks. Neural Networks trade interpretability for a universal modeling framework.

10.1.2 Training

A training dataset consisting of m input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector $\mathbf{b}$. Given an input matrix X consisting of m input vectors and an output vector $\mathbf{y}$ consisting of m output values, minimize the distance between the target output vector $\mathbf{y}$ and the predicted output vector $\mathbf{f}(X; \mathbf{b})$.

The model training process depicted in Figure 10.1 involves

- a Dataset $(X, \mathbf{y})$ where input/data matrix $X \in \mathbb{R}^{m \times n}$ and output/response vector $\mathbf{y} \in \mathbb{R}^m$,
- a Model with mean response function f or vectorized mean response function $\mathbf{f} : \mathbb{R}^{m \times n} \times \mathbb{R}^p \to \mathbb{R}^m$, and
- a Trainer with loss function $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$.

In Figure 10.1, the i^{th} row/instance of the Dataset is taken as the input vector $\mathbf{x} \in \mathbb{R}^n$ to the model to produce a predicted response,

$$\begin{aligned} \hat{y} \;&=\; f(\mathbf{x}; \mathbf{b}) && \text{instance level} \\ \hat{\mathbf{y}} \;&=\; \mathbf{f}(X; \mathbf{b}) && \text{dataset level} \end{aligned}$$

where f ($\mathbf{f}$) is the (vectorized) mean response function and the parameter vector $\mathbf{b} \in \mathbb{R}^p$.

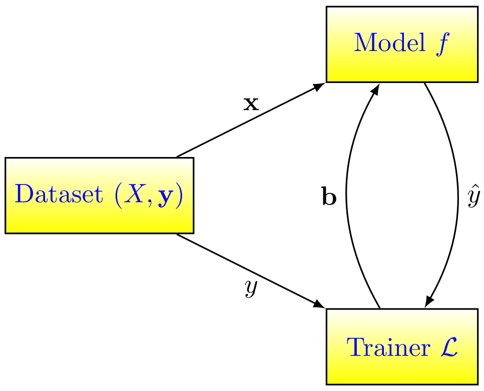


Figure 10.1: Nonlinear Regression: Dataset - Model - Trainer View

Note that for linear models typically $p = n$, but for nonlinear models, it is not uncommon for $p > n$, i.e., the number of parameters to be greater than the number of input/predictor variables.

Loss Function

Optimization techniques can be used to determine a "good" value for the parameter vector $\mathbf{b}$, by minimizing a loss function.

$$\mathcal{L}(\mathbf{b}; \mathbf{y}, \hat{\mathbf{y}})$$

Common forms of loss functions are based on minimizing an ℓ^1 or ℓ^2 norm of the error vector $\boldsymbol{\epsilon} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{f}(X; \mathbf{b})$,

$$min_{\mathbf{b}} \, \|\mathbf{y} - \mathbf{f}(X; \mathbf{b})\| \tag{10.2}$$

Again for an ℓ^2 norm, it is convenient to use its square and minimize the dot product of the error with itself (or rather half of that $\frac{1}{2}\boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}$).

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}(\mathbf{y} - \mathbf{f}(X; \mathbf{b})) \cdot (\mathbf{y} - \mathbf{f}(X; \mathbf{b})) \tag{10.3}$$

An alternative equation for the loss function uses summation notation to add up the square of each error $\epsilon_i = y_i - f(\mathbf{x}_i; \mathbf{b})$ over all m instances.

$$\boxed{\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i; \mathbf{b})]^2} \tag{10.4}$$

Special Case of a Linear Model

For a linear model, f is simply a linear combination of the input variables, i.e., $\mathbf{f}(X;\mathbf{b}) = X\mathbf{b}$, so

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}(\mathbf{y} - X\mathbf{b}) \cdot (\mathbf{y} - X\mathbf{b})$$

Recall that taking the gradient with respect to $\mathbf{b}$ and setting it equal to $\mathbf{0}$, yields the Normal Equations,

$$(X^{\intercal} X)\mathbf{b} \;=\; X^{\intercal}\mathbf{y}$$

Then matrix factorization can be used to relatively quickly find a globally optimal value for the parameter vector $\mathbf{b}$.

10.1.3 Optimization

For nonlinear regression, a Least-Squares (minimizing the errors/residuals) method can be used to fit the parameter vector $\mathbf{b}$. Finding optimal values for the parameters $\mathbf{b}$ may be found by either solving the Normal Equations or using an iterative optimization algorithm. In either case, gradients of the loss function are used to find these values.

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i;\mathbf{b})]^2$$

The j^{th} partial derivative of this loss function $\mathcal{L}(\mathbf{b})$ may be determined using the chain rule.

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i;\mathbf{b})]\, \frac{\partial f(\mathbf{x}_i;\mathbf{b})}{\partial b_j}} \tag{10.5}$$

Setting each partial derivative ($j = 0 \dots p-1$) to zero yields,

$$\sum_{i=0}^{m-1} f(\mathbf{x}_i;\mathbf{b})\, \frac{\partial f(\mathbf{x}_i;\mathbf{b})}{\partial b_j} \;=\; \sum_{i=0}^{m-1} y_i\, \frac{\partial f(\mathbf{x}_i;\mathbf{b})}{\partial b_j}$$

Collecting these equations for all values of j leads to the Normal Equations for nonlinear models. Unfortunately, they form a system of nonlinear equations requiring numerical solution. Instead of solving the Normal Equations for nonlinear models, one may use an optimization algorithm to minimize the loss function $\mathcal{L}(\mathbf{b})$.

First-Order Optimization Algorithms

First-order optimization algorithms utilize first derivatives, typically moving in the opposite direction to the gradient.

$$\nabla \mathcal{L}(\mathbf{b}) \;=\; \left[\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_0}, \dots, \frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_{p-1}}\right] \tag{10.6}$$

Noting that $\epsilon_i = y_i - f(\mathbf{x}_i;\mathbf{b})$, the j^{th} partial derivative can be written more concisely.

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, \frac{\partial f(\mathbf{x}_i;\mathbf{b})}{\partial b_j}} \tag{10.7}$$

Therefore, an optimization algorithm essentially operates off of the gradient of the mean response function f. For greater accuracy and efficiency for a given function f, utilization of formulas for its partial derivatives is preferred over their numerical calculation.

Second-Order Optimization Algorithms

Second-order optimization algorithms utilize both first derivatives (gradient) and second derivatives (Hessian). A user defined mean response function f that takes a vector of inputs $\mathbf{x}$ and a vector of parameters $\mathbf{b}$,

```
    f: (VectorD, VectorD) => Double
```

is passed as a parameter to the `NonlinearRegression` class. This function is used to create a predicted output value yp_i for each input vector $\mathbf{x}_i$. The `sseF` method (as a loss function) applies function f to all m input vectors to compute predicted output values. These are then subtracted from the actual output to create an error vector, whose squared Euclidean norm is returned. The `sseF` is embedded in the `train` method.

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output vector

    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =

        //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
        /** Function to compute sse for the given values for the parameter vector b.
         *  @param b  the parameter vector
         */
        def sseF (b: VectorD): Double =
            val yp = x_.map (f(_, b))               // predicted response vector
            (y_ - yp).normSq                        // sum of squared errors
        end sseF

        val bfgs   = new BFGS (sseF)                // minimize sseF using nonlinear optimizer
        val result = bfgs.solve (b_init)            // result from optimizer
        val sse    = result._1                      // optimal function value
        b          = result._2                      // optimal parameter vector
    end train
```

ScalaTion's `optimization` package provide several solvers for linear, quadratic, integer and nonlinear programming/optimization. Currently, the `BFGS` class (a Quasi-Newton optimizer) is used for finding an optimal $\mathbf{b}$ by minimizing `sseF`. It uses the second-order Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm that can improve convergence over first-order algorithms such as gradient/steepest descent. The BFGS algorithm determines a search direction by deflecting the gradient/steepest descent direction vector (opposite the gradient) by multiplying it by a matrix that approximates the inverse Hessian. This optimizer requires an initial guess `b_init` for the parameter vector $\mathbf{b}$.

For more information see `http://www.bjsos.umd.edu/socy/alan/stats/socy602_handouts/kut86916_ch13.pdf`.

10.1.4 Use of the Chain Rule

The *Chain Rule* for Differential Calculus can be used to obtain derivatives, partial derivatives and gradients needed in Nonlinear Regression and Neural Networks. Consider the following function of the variable x; the rule works as follows:

$$f(x) \;=\; (7 - 3x^2)^2$$

To obtain the derivative of the function f w.r.t. x, $\dfrac{df}{dx}$, f can be expressed as the composition of two functions $f = \mathsf{f} \circ u$,

$$f(x) \;=\; \mathsf{f}(u(x)) \qquad \text{where } \; u(x) = (7 - 3x^2) \;\text{ and }\; \mathsf{f}(u) = u^2$$

so the chain rule can be applied.

$$\boxed{\frac{df}{dx} \;=\; \frac{d\mathsf{f}}{du}\frac{du}{dx}} \tag{10.8}$$

where

$$\frac{d\mathsf{f}}{du} \;=\; 2u$$

$$\frac{du}{dx} \;=\; -\,6x$$

Multiplying and substituting for u yields

$$\frac{df}{dx} \;=\; [2(7 - 3x^2)][-6x] \;=\; -\,12x(7 - 3x^2)$$

This basic rule carries over to partial derivatives.

$$\frac{\partial \mathcal{L}}{\partial b_j} \;=\; \frac{d\mathcal{L}}{du}\frac{\partial u}{\partial b_j} \tag{10.9}$$

For example, consider the function $f(x, y) = (7 - 3xy)^2$. Let $u(x, y) = 7 - 3xy$ and $\mathsf{f}(u) = u^2$; applying the chain rule gives

$$\boxed{\frac{\partial f}{\partial x} \;=\; \frac{d\mathsf{f}}{du}\frac{\partial u}{\partial x}} \tag{10.10}$$

$$\frac{\partial f}{\partial x} \;=\; [2u][-3y] \;=\; -\,6y(7 - 3xy)$$

Of course, there are additional chain rules in multivariate calculus, see `https://www.whitman.edu/mathematics/multivariable/multivariable.pdf`.

10.1.5 `NonlinearRegression` Class

Class Methods:

```
@param x        the data/input matrix optionally augmented with a first column of ones
@param y        the response/output vector
@param f        the nonlinear function f(x, b) to fit
@param b_init   the initial guess for the parameter vector b
@param fname_   the feature/variable names (defaults to null)
@param hparam   the hyper-parameters (currently has none)

class NonlinearRegression (x: MatrixD, y: VectorD, f: FunctionP2S,
                           b_init: VectorD, fname_ : Array [String] = null,
                           hparam: HyperParameter = null)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def sseF (b: VectorD): Double =
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
override def predict (z: VectorD): Double = f(z, b)
```

10.1.6 Exercises

1. Given the Normal Equations for Nonlinear Regression models,

$$\sum_{i=0}^{m-1} f(\mathbf{x}_i; \mathbf{b}) \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j} \; = \; \sum_{i=0}^{m-1} y_i \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

simplify them assuming f is linear in the parameters $\mathbf{b}$, i.e.,

$$f(\mathbf{x}_i; \mathbf{b}) \; = \; \mathbf{x}_i \cdot \mathbf{b}$$

Hint:

$$\frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j} \; = \; x_{ij}$$

2. List and compare several **nonlinear optimization algorithms** from the following classes:

   (a) derivative-free optimization algorithms,

   (b) first-order optimization algorithms, and

   (c) second-order optimization algorithms.

3. Enzyme catalyzed biochemical reactions may be described by a Michaelis-Menten nonlinear regression model [119]. The reaction rate/velocity $y = v$ is a function of the substrate concentration x,

$$y = \frac{b_0 x}{x + b_1} + \epsilon$$

where parameter $b_0 = V_{max}$ is the maximum reaction velocity (occurs at high substrate concentration) and parameter $b_1 = K_m$ is the Michaelis constant that measures the strength of the enzyme-substrate interaction. Use the dataset given in Table 2 of [119] to train the model.

4. The j^{th} partial derivative of the loss function $\mathcal{L}(\mathbf{b})$ is copied below.

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} = -\sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i; \mathbf{b})] \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} = -\sum_{i=0}^{m-1} [\epsilon_i] \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

The first-order optimization algorithms use gradients/partial derivatives to iteratively improve the parameters. Gradient Descent (GD) uses the whole training set, while Stochastic Gradient Descent (SGD) uses a (typically random) subset of the training set (called a mini-batch). For pure Stochastic Gradient Descent (SGD), the parameters are updated for every data instance i, so the above equation becomes the following:

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} = -[\epsilon_i] \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

Now, suppose the error $\epsilon_i > 0$ (positive). This means that the predicted value $\hat{y}_i = f(\mathbf{x}_i; \mathbf{b})$ is too low. This begs the question of how to change the parameter b_j to reduce the loss function, make the error smaller or equivalently increase the mean response function f.

Clearly, the answer depends on the slope of the mean response function.

$$\frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

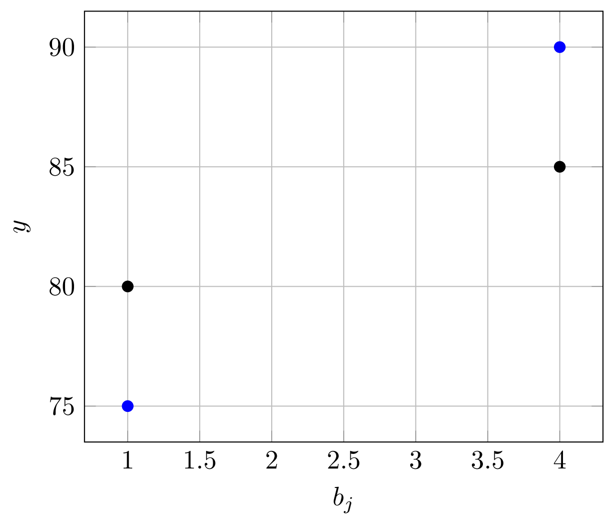
For the case where the slope of the mean response function is positive, determine whether to increase or decrease the parameter b_j. Also, explain what this does to the loss function.

5. In the figure below, determine the value of ϵ_i for

(a) $b_j = 1$

(b) $b_j = 4$

How to Update Parameter b_j, y(black), $\hat{y}$(blue)



Assuming the slope of the mean response function f is positive at both values of b_j, how should the j^{th} parameter b_j be changed (increased or decreased) for (a) and (b)?

10.2 Simple Exponential Regression

The `SimpleExpRegression` class can be used for developing Simple Exponential Regression models. These are useful when data exhibit exponential growth or decay.

10.2.1 Model Equation

Simple Exponential Regression models have a single predictor variable $\mathbf{x} = [x]$. They are a type of Nonlinear Model for a response random variable y having a model equation of the following form,

$$\boxed{y \;=\; b_0 e^{b_1 x} \,+\, \epsilon} \tag{10.11}$$

The mean response function has two parameters, a multiplier b_0 and a growth rate b_1.

$$f(\mathbf{x}, \mathbf{b}) \;=\; b_0 e^{b_1 x} \tag{10.12}$$

Note, for `SimpleExpRegression` the number of parameters $p = 2$, while the number of predictor variables $n = 1$.

10.2.2 Training

Given a dataset $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^m$, the parameters $\mathbf{b} = [b_0, b_1]$ may be determined using Least Squares Estimation (LSE). The loss function may be deduced from the general NLR loss function in summation form.

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i; \mathbf{b})]^2$$

Replacing the general mean response function $f(\mathbf{x}_i; \mathbf{b})$ with $b_0 e^{b_1 x_i}$ yields,

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2} \sum_{i=0}^{m-1} [y_i - b_0 e^{b_1 x_i}]^2 \tag{10.13}$$

When $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 I)$, i.e., each error has mean 0 and variance σ^2, the same parameter estimates can be obtained using the Maximum Likelihood Estimation (MLE), see the Chapter on Generalized Linear Models. See the Exercises, to show the equivalence of the two estimation techniques.

10.2.3 Optimization

Optimization for Simple Exponential Regression involves determining the gradient of the loss function with respect to the parameters $\mathbf{b}$. Starting with the general formula for the j^{th} partial derivative of the loss function $\mathcal{L}(\mathbf{b})$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i] \, \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

the partial derivatives of the mean response function are needed. In this case, there are two parameters $\mathbf{b} = [b_0, b_1]$ and two formulas:

Partial Derivatives of the Mean Response Function

$$\frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_0} \;=\; e^{b_1 x_i} \tag{10.14}$$

$$\frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_1} \;=\; x_i b_0 e^{b_1 x_i} \tag{10.15}$$

Partial Derivatives of the Loss Function

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_0} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, e^{b_1 x_i} \tag{10.16}$$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_1} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, x_i b_0 e^{b_1 x_i} \tag{10.17}$$

Since the ith predicted response $\hat{y}_i = b_0 e^{b_1 x_i}$, these two equations may be simplied as follows:

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_0} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, \hat{y}_i / b_0 \;=\; -\,\boldsymbol{\epsilon} \cdot (\hat{\mathbf{y}}/b_0) \tag{10.18}$$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_1} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, x_i \hat{y}_i \;=\; -\,\boldsymbol{\epsilon} \cdot (\mathbf{x} * \hat{\mathbf{y}}) \tag{10.19}$$

where $*$ is the element-wise vector product (e.g., [2, 4, 6] $*$ [5, 3, 1] = [10, 12, 6]). The gradient of the loss function is the vector formed from the two partial derivatives given above.

$$\nabla \mathcal{L}(\mathbf{b}) \;=\; \left[\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_0}, \frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_1}\right] \tag{10.20}$$

$$\nabla \mathcal{L}(\mathbf{b}) \;=\; -\,[\boldsymbol{\epsilon} \cdot (\hat{\mathbf{y}}/b_0),\, \boldsymbol{\epsilon} \cdot (\mathbf{x} * \hat{\mathbf{y}})] \tag{10.21}$$

A Gradient Descent Optimizer is a simple type of first-order optimization algorithm. It starts at a random (or guessed) point in parameter space and iteratively moves in the direction opposite to the gradient. To control how rapidly the algorithm moves in that direction, a learning rate η is introduced as a hyper-parameter to tune. With each iteration the parameter $\mathbf{b}$ is updated as follows:

$$\mathbf{b} \;=\; \mathbf{b} - \eta\, \nabla \mathcal{L}(\mathbf{b})$$

The pseudocode for the Gradient Descent Algorithm for Simple Exponential Regression is as follows:

```
b = b_init                                  // initial random/guessed parameter vector
while loss_decreasing do                    // stopping rule
    val yp  = f(x, b)                       // y-predicted vector
    val ϵ   = y - yp                        // error vector
    val δ0  = ϵ dot yp / b0                 // delta 0 - partial of loss w.r.t. b0
    val δ1  = ϵ dot yp * x                  // delta 1 - partial of loss w.r.t. b1
    b0 += η * δ0                            // update to first parameter
    b1 += η * δ1                            // update to second parameter
end while
```

10.2.4 Linearization

In this subsection, we examine two versions of simple exponential regression to see if the model can be linearized, by transforming the model so that it becomes linear in the parameters. The mean response function for the simple exponential regression model is

$$f(x, \mathbf{b}) \;=\; b_0\, e^{b_1 x}$$

Multiplicative Errors

Consider this mean response function with multiplicative errors,

$$y \;=\; b_0\, e^{b_1 x}\, \epsilon$$

Taking the natural logarithm yields

$$\ln(y) \;=\; \ln(b_0) \,+\, b_1 x \,+\, \ln(\epsilon)$$

This linearized form can be used to fit the transformed response as was done with `TranRegression`.

$$\ln(y) \;=\; \beta_0 \,+\, b_1 x \,+\, e$$

Additive Errors

Now consider this mean response function with additive errors,

$$y \;=\; b_0\, e^{b_1 x} \,+\, \epsilon$$

Taking the natural logarithm yields

$$\ln(y) \;=\; \ln(b_0\, e^{b_1 x} \,+\, \epsilon)$$

Notice that the error model is different (multiplicative vs. additive) so the transformed linear regression model is different from the original nonlinear exponential regression model. They may be viewed as competitors, see the exercises.

Some nonlinear models are intrinsically linear in that the data can be transformed into a linear form, although the transformation may transform the errors inappropriately. The following nonlinear model, however, cannot be transformed to a linear form, regardless of the error model. It adds an additional parameter to the simple exponential model.

$$y \;=\; b_0 \,+\, b_1 e^{b_2 x} \,+\, \epsilon \tag{10.22}$$

10.2.5 Exercises

1. Consider the following hospitalization dataset from Chapter 13 of [102].

```
    val xy = MatrixD ((15, 2), 2, 54,
                               5, 50,
                               7, 45,
                              10, 37,
                              14, 35,
                              19, 25,
                              26, 20,
                              31, 16,
                              34, 18,
                              38, 13,
                              45, 8,
                              52, 11,
                              53, 8,
                              60, 4,
                              65, 6)
```

Plot `y = xy(?, 1)` patient prognostic index versus `x = xy(?, 0)` days hospitalized.

2. Model the hospitalization dataset using Simple Linear Regression, plot the regression line and show the Quality of Fit (QoF) measures.

3. Model the hospitalization dataset using Simple Exponential Regression, plot the regression curve and show the Quality of Fit (QoF) measures. How do the QoF measures compare with those produced by Simple Linear Regression?

4. Model the hospitalization dataset using Log Transformed Linear Regression, plot the regression curve and show the Quality of Fit (QoF) measures. How do the QoF measures compare with those produced by Simple Linear Regression and Simple Exponential Regression?

5. For the following exponential regression model

$$y \;=\; b_0 e^{b_1 x} \,+\, \epsilon$$

show that when error vector $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 I)$, the LSE and MLE estimation techniques will produce the same parameter estimates.

6. Use the pseudo-code given in this section to write a Gradient Descent Optimizer for Simple Exponential Regression using ScalaTion.

7. Pass the loss function into a second-order Quasi-Newton Optimizer and compare to the first-order algorithm in term of the parameter solution, the value of the loss function, and the number of steps needed for convergence.

10.3 Exponential Regression

The simple exponential regression model can be extended to have multiple predictor variables, e.g., $\mathbf{x} = [x_1, x_2]$ and $\mathbf{b} = [b_0, b_1, b_2]$.

$$y \;=\; b_0\, e^{\mathbf{b}_{1-2}\cdot\mathbf{x}} + \epsilon \tag{10.23}$$

Note, $\mathbf{b}_{1-2} = [b_1, b_2]$. The mean response function then is

$$f(\mathbf{x}, \mathbf{b}) \;=\; b_0\, e^{\mathbf{b}_{1-2}\cdot\mathbf{x}}$$

By introducing a new variable, $x_0 = 1$, the equation may be rewritten.

$$f(\mathbf{x}, \mathbf{b}) \;=\; e^{\mathbf{b}\cdot\mathbf{x}} \tag{10.24}$$

The ℓ^2 loss function for Exponential Regression is

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\sum_{i=0}^{m-1} [y_i - e^{\mathbf{b}\cdot\mathbf{x}}]^2$$

Again, starting with the general formula for the j^{th} partial derivative of the loss function $\mathcal{L}(\mathbf{b})$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i]\, \frac{\partial f(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

the problem reduces to finding the j^{th} partial derivative of the mean response function.

$$\frac{\partial f(\mathbf{x}_i, \mathbf{b})}{\partial b_j} \;=\; x_{ij} e^{\mathbf{b}\cdot\mathbf{x}_i}$$

Substituting this result into the general formula gives

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; \sum_{i=0}^{m-1} [\epsilon_i] x_{ij} e^{\mathbf{b}\cdot\mathbf{x}_i} \tag{10.25}$$

This mean response function can be linearized using a log transformation. Alternatively, one can also look into using a Generalized Linear Model (GLM) for such datasets to provide more options for dealing with the error distribution. See `http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf` for more details. Also, see the exercises.

10.3.1 `ExpRegression` Class

Class Methods:

```
    @param x        the data/input matrix
    @param y        the response/output vector
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (currently none)
    @param nonneg   whether to check that responses are nonnegative

    class ExpRegression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                            hparam: HyperParameter = null, nonneg: Boolean = true)
           extends Predictor (x, y, fname_, hparam)
              with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def ll (b: VectorD): Double =
    def ll_null (b: VectorD): Double =
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def train_null (): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                             b_ : VectorD = b, vifs: VectorD = vif ()): String =
    override def predict (z: VectorD): Double = exp (b dot z)
    override def buildModel (x_cols: MatrixD): ExpRegression =
```

10.3.2 Exercises

1. A Generalized Linear Model (GLM) that utilized Maximum Likelihood Estimation (MLE) may be used for such data. The response variable y is modeled as the product of a mean response function $f(\mathbf{x};\mathbf{b}) = \mu_y(\mathbf{x})$ and exponentially distributed residuals/errors ϵ.

$$\begin{aligned} y &= \mu_y(\mathbf{x}) \times \epsilon \\ \ln(\mu_y(\mathbf{x})) &= \mathbf{b}\cdot\mathbf{x} \end{aligned}$$

   where the link function $g = \ln$ with inverse ($g^{-1} = \exp$).

$$g(\mu_y(\mathbf{x})) \;=\; \ln(\mu_y(\mathbf{x})) \;=\; \mathbf{b}\cdot\mathbf{x}$$

   Explain why the errors/residuals $\epsilon_i = y_i/\mu_y(\mathbf{x}_i)$ follow and Exponential distribution.

$$f(y_i/\mu_y(\mathbf{x}_i)) \;=\; \frac{1}{\mu_y(\mathbf{x}_i)}\, e^{-y_i/\mu_y(\mathbf{x}_i)}$$

2. Show that the likelihood function for Exponential Regression is the following:

$$L \;=\; \prod_{i=0}^{m-1} \frac{1}{\mu_y(\mathbf{x}_i)}\, e^{-y_i/\mu_y(\mathbf{x}_i)}$$

   Substituting for $\mu_y(\mathbf{x}_i) = e^{\mathbf{b}\cdot\mathbf{x}_i}$ gives

$$L \;=\; \prod_{i=0}^{m-1} e^{-\mathbf{b}\cdot\mathbf{x}_i}\, e^{-y_i/e^{\mathbf{b}\cdot\mathbf{x}_i}}$$

   Taking the natural logarithm gives the log-likelihood function.

$$l = \sum_{i=0}^{m-1} -\mathbf{b} \cdot \mathbf{x}_i - \frac{y_i}{e^{\mathbf{b} \cdot \mathbf{x}_i}}$$

3. Write a function to compute the negative log-likelihood ($-l$) and pass it into a nonlinear optimization algorithm for minimization.

10.4 Perceptron

The `Perceptron` class supports single-valued 2-layer (input and output) Neural Networks. The inputs into a Neural Net are given by the input vector $\mathbf{x}$, while the outputs are given by the output value y. As depicted in Figure 10.2, each component of the input x_j is associated with an input node in the network, while the output y is associated with the single output node. The input layer consists of n input nodes, while the output layer consists of 1 output node.

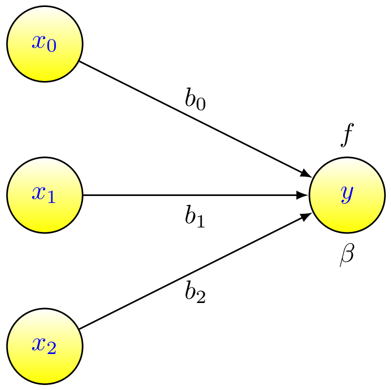


Figure 10.2: Network Diagram for a Perceptron

An edge connects each input node with the output node, i.e., there are n edges in the network. To include an intercept in the model (also referred to as bias) one of the inputs (say x_0) must always be set to 1. Alternatively, a bias offset β can be associated with the output node and added to the weighted sum (see below).

10.4.1 Model Equation

The weights on the edges are analogous to the parameter vector $\mathbf{b}$ in regression. The output y has an associated parameter vector $\mathbf{b}$, where parameter value b_j is the edge weight connecting input node x_j with output node y.

Recall the model equation for multiple linear regression.

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... b_{n-1} x_{n-1} + \epsilon$$

We now take the linear combination of the inputs, $\mathbf{b} \cdot \mathbf{x}$, and apply an activation function f_a (or simply f).

$$\boxed{y \;=\; f(\mathbf{b} \cdot \mathbf{x}) + \epsilon \;=\; f(\sum_{j=0}^{n-1} b_j x_j) + \epsilon} \tag{10.26}$$

The general mean response function $f_r(\mathbf{x}; \mathbf{b})$ of nonlinear regression, which takes two vectors as input, is now replaced by first taking a linear combination of $\mathbf{x}$ and then applying a simpler activation function that takes a single scalar as input, i.e., $f(\mathbf{b} \cdot \mathbf{x})$.

10.4.2 Ridge Functions

Restricting the multi-dimensional function $f_r(\mathbf{x}; \mathbf{b})$ to $f(\mathbf{b} \cdot \mathbf{x})$ means that the response surface is simply a *ridge function*. As an example, let $\mathbf{b} = [.5, 2, 1]$ and $\mathbf{x} = [1, x_1, x_2]$, then

$$\hat{y} \;=\; f([.5, 2, 1] \cdot [1, x_1, x_2]) \;=\; f(2x_1 + x_2 + .5)$$

Choosing the activation function f to be the sigmoid function results in the ridge function shown in Figure 10.3. Compared to the hyperplane shown in the Regression section, its more flexible shape provides provides more potential for matching a variety of response surfaces.

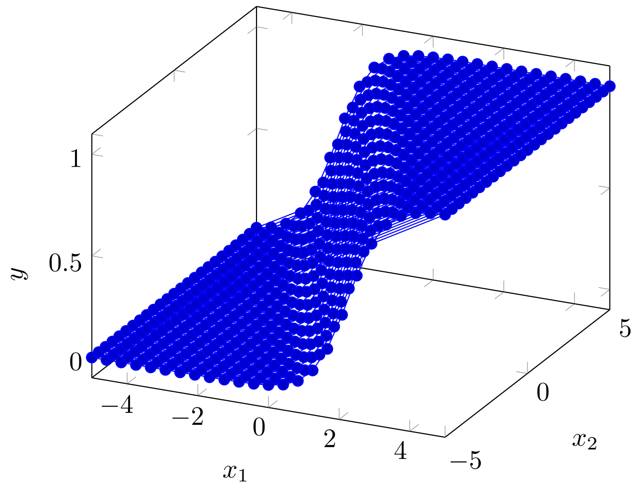


Figure 10.3: Ridge Function: $\hat{y} = \text{sigmoid}(2x_1 + x_2 + .5)$

10.4.3 Training

Given several input vectors and output values (e.g., in a training set), optimize/fit the weights $\mathbf{b}$ connecting the layers. After training, given an input vector $\mathbf{x}$, the net can be used to predict the corresponding output value $\hat{y}$.

A training dataset consisting of m input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight vector $\mathbf{b} \in \mathbb{R}^n$. Given an input/data matrix $X \in \mathbb{R}^{m \times n}$ consisting of m input vectors and an output vector $\mathbf{y} \in \mathbb{R}^{\mathbf{m}}$ consisting of m output values, minimize the distance between the actual/target output vector $\mathbf{y}$ and the predicted output vector $\hat{\mathbf{y}}$,

$$\boxed{\hat{\mathbf{y}} \;=\; \mathbf{f}(X\mathbf{b})} \tag{10.27}$$

where $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$ is the vectorized version of the activation function f. The vectorization may occur over the entire training set or more likely, an iterative algorithm may work with a group/batch of instances at a time. In other words, the goal is to minimize some norm of the error vector.

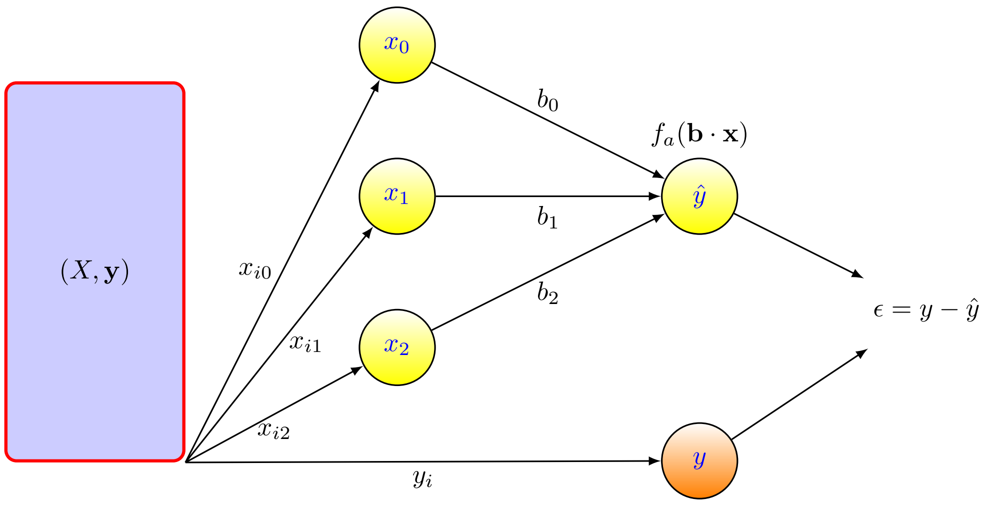


Figure 10.4: Training Diagram for Perceptron

$$\boxed{\boldsymbol{\epsilon} \;=\; \mathbf{y} - \hat{\mathbf{y}} \;=\; \mathbf{y} - \mathbf{f}(X\mathbf{b})} \tag{10.28}$$

Using the Euclidean (ℓ^2) norm, we have

$$\min_{\mathbf{b}} \|\mathbf{y} - \mathbf{f}(X\mathbf{b})\|$$

As was the case with regression, it is convenient to minimize the dot product of the error with itself ($\|\boldsymbol{\epsilon}\|^2 = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}$). In particular, we aim to minimize half of this value, half *sse* (loss function $\mathcal{L}$).

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\,[\mathbf{y} - \mathbf{f}(X\mathbf{b})] \cdot [\mathbf{y} - \mathbf{f}(X\mathbf{b})] \tag{10.29}$$

Using summation notation gives

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i \cdot \mathbf{b})]^2 \tag{10.30}$$

10.4.4 Optimization

Optimization for Perceptrons and Neural Networks is typically done using an iterative optimization algorithm that utilizes gradients. Popular optimizers include Gradient Descent (GD), Stochastic Gradient Descent (SGD), Stochastic Gradient Descent with Momentum (SGDM), Root Mean Square Propagation (RMSProp) and Adaptive Moment Estimation (Adam) (see Appendix on Optimization for details).

The gradient of the loss function $\mathcal{L}$ is calculated by computing all of the partial derivatives with respect to the parameters/weights.

$$\nabla\mathcal{L}(\mathbf{b}) \;=\; \left[\frac{\partial \mathcal{L}}{\partial b_0}, \ldots, \frac{\partial \mathcal{L}}{\partial b_{n-1}}\right]$$

Partial Derivative for b_j

Starting with the nonlinear regression general formula for the j^{th} partial derivative of the loss function $\mathcal{L}(\mathbf{b})$

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i] \, \frac{\partial f_r(\mathbf{x}_i; \mathbf{b})}{\partial b_j}$$

and specializing the mean response function $f_r(\mathbf{x}_i; \mathbf{b})$ to $f(\mathbf{x}_i \cdot \mathbf{b})$, i.e., the function of two vectors is replaced with an activation function applied to the dot product of the two vectors, yields,

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i] \, \frac{\partial f(\mathbf{x}_i \cdot \mathbf{b})}{\partial b_j}$$

Now, the partial derivative of $f(\mathbf{x}_i \cdot \mathbf{b})$ with respect to b_j can be determined via the *chain rule* using the scalar pre-activation response $u_i = \mathbf{x}_i \cdot \mathbf{b}$.

$$\frac{\partial f(\mathbf{x}_i \cdot \mathbf{b})}{\partial b_j} \;=\; \frac{\partial f(u_i)}{\partial u_i} \, \frac{\partial u_i}{\partial b_j} \tag{10.31}$$

Derivative of the Activation Function

Since $\frac{\partial f(u_i)}{\partial u_i}$ is a regular derivative as it is one dimensional, it can be replaced with $\frac{df(u_i)}{du_i}$ which may be denoted as $f'(u_i)$.

$$\frac{\partial f(u_i)}{\partial u_i} \;=\; \frac{df(u_i)}{du_i} \;=\; f'(u_i) \tag{10.32}$$

Derivative of the Pre-Activation Response

$$\frac{\partial u_i}{\partial b_j} \;=\; \frac{\partial}{\partial b_j}(b_0 x_{i0} + \cdots + b_j x_{ij} + \cdots + b_k x_{ik}) \;=\; x_{ij} \tag{10.33}$$

Combining Results

$$\frac{\partial f(\mathbf{x}_i \cdot \mathbf{b})}{\partial b_j} \;=\; f'(u_i) \, x_{ij}$$

Therefore, the j^{th} partial derivative of the loss function $\mathcal{L}$ becomes

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\sum_{i=0}^{m-1} [\epsilon_i] \, x_{ij} \, f'(u_i)} \tag{10.34}$$

The sum can be replaced with a dot product (sum of products),

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\,\mathbf{x}_{:j} \cdot (\boldsymbol{\epsilon} * \mathbf{f}'(\mathbf{u}))} \tag{10.35}$$

where $\mathbf{x}_{:j}$ is the j^{th} column of data matrix X, $*$ is the element-wise vector product, $\mathbf{u} = X\mathbf{b}$, and $\mathbf{f}'(\mathbf{u})$ is the vector extension of the derivative of the activation function $\frac{df(u_i)}{du_i} = f'(u_i)$ over all m instances.

See the derivation below that expresses the loss function as the dot product of the error vectors. It provides another way to obtain the same results.

Alternative Derivation at the Vector Level

The same basic derivation can be carried out at the vector level as well. Taking the partial derivative with respect to the j^{th} parameter/weight, b_j, is a bit complicated since we need to use the chain rule and the product rule. First, letting $\mathbf{u} = X\mathbf{b}$ (the pre-activation response) allows the loss function to be simplified to

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\,[\mathbf{y} - \mathbf{f}(\mathbf{u})] \cdot [\mathbf{y} - \mathbf{f}(\mathbf{u})] \tag{10.36}$$

The chain rule from vector calculus to be applied is

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; \frac{\partial \mathcal{L}(\mathbf{b})}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial b_j} \tag{10.37}$$

The first partial derivative is

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial \mathbf{u}} \;=\; -\,[\mathbf{y} - \mathbf{f}(\mathbf{u})] * \mathbf{f}'(\mathbf{u}) \tag{10.38}$$

where the first part of the r.h.s. is $\mathbf{f}'(\mathbf{u})$ which is the derivative of $\mathbf{f}$ with respect to vector $\mathbf{u}$ and the second part is the difference between the actual and predicted output/response vectors. The two vectors are multiplied together, element-wise.

The second partial derivative is

$$\frac{\partial \mathbf{u}}{\partial b_j} \;=\; \mathbf{x}_{:j} \tag{10.39}$$

where $\mathbf{x}_{:j}$ is the j^{th} column of matrix X (see Exercises 4, 5 and 6 for details).

The dot product of the two partial derivatives gives

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\,\mathbf{x}_{:j} \cdot [\mathbf{y} - \mathbf{f}(\mathbf{u})]\,\mathbf{f}'(\mathbf{u})$$

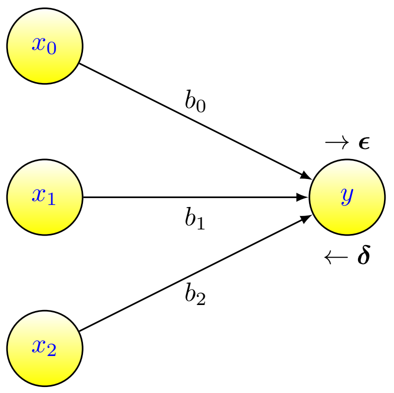
Since the error vector $\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{f}(\mathbf{u})$, we may simplify the expression.

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; -\,\mathbf{x}_{:j} \cdot (\boldsymbol{\epsilon} * \mathbf{f}'(\mathbf{u})) \tag{10.40}$$

The j^{th} partial derivative (or j^{th} element of the gradient) indicates the relative amount to move (change b_j) in the j^{th} dimension to reduce $\mathcal{L}$. Notice that equations 10.33 and 10.38 are the same.

The δ Vector

The goal of training is to minimize the errors. The errors are calculated using forward propagation through the network. The parameters (weights and bias) are updated by back-propagation of the errors, or more precisely, the slope-adjusted errors as illustrated in Figure 10.5.



x_0
x_1
x_2
b_0
b_1
b_2
y
$\rightarrow \boldsymbol{\epsilon}$
$\leftarrow \boldsymbol{\delta}$

Figure 10.5: Perceptron: the $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$ Vectors

Therefore, it is helpful especially for multi-layer neural networks to define the delta vector $\boldsymbol{\delta}$ as follows:

$$\boxed{\boldsymbol{\delta} \;=\; \frac{\partial \mathcal{L}(\mathbf{b})}{\partial \mathbf{u}} \;=\; -\,\boldsymbol{\epsilon} * \mathbf{f}'(\mathbf{u})} \tag{10.41}$$

It multiplies the derivative of the vectorized activation function $\mathbf{f}$ by the negative error vector, element-wise. If the error is small or the derivative is small, the adjustment to the parameter should be small. The partial derivative of $\mathcal{L}$ with respect to b_j now simplifies to

$$\frac{\partial \mathcal{L}(\mathbf{b})}{\partial b_j} \;=\; \mathbf{x}_{:j} \cdot \boldsymbol{\delta} \tag{10.42}$$

10.4.5 Example Calculation for $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$

Assume that the activation function is the sigmoid function. Starting with the parameter/weight vector

$$\mathbf{b} \;=\; [.1, .2, .1],$$

compute the m-dimensional vectors, $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$, for Exercise 7. With these parameters, the predicted output/response vector $\hat{\mathbf{y}}$ may be computed in two steps: The first step computes the response, pre-activation $\mathbf{u} = X\mathbf{b}$. The second step takes this vector and applies the activation function to each of its elements. This requires looking ahead to the subsection on *activation functions*. The sigmoid function (abbreviated sigmoid) is defined as follows:

$$\text{sigmoid}(u) \;=\; [1 + e^{-u}]^{-1} \tag{10.43}$$

Figure 10.6 show the response surface for the example problem where the response y is color-coded.

Pre-activation Vector u

From Figure 10.6 create the input/data matrix $X \in \mathbb{R}^{9\times 3}$ and multiply it by the current parameter vector $\mathbf{b} \in \mathbb{R}^3$. The pre-activation vector is the aggregated signal before activation.

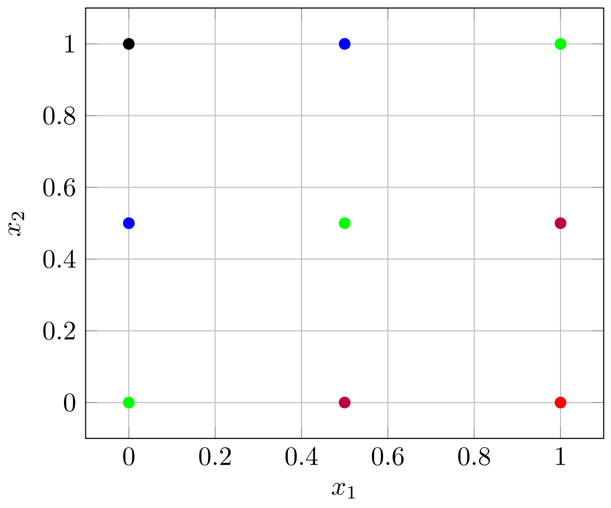


Figure 10.6: Example Perceptron Problem (Exercise 7): response values, black (.2), blue (.3), green (.5), purple/crimson (.8), red (1), form a diagonal terrace pattern

$$\mathbf{u} \;=\; X\mathbf{b} \;=\; [.1, .15, .2, .2, .25, .3, .3, .35, .4]$$

Predicted Response Vector $\hat{\mathbf{y}}$

The predicted response vector is determined by applying the activation function to each element of the pre-activation vector.

$$\begin{aligned}\hat{\mathbf{y}} \;&=\; \text{sigmoid}(\mathbf{u}) \;=\; \text{sigmoid}([.1, .15, .2, .2, .25, .3, .3, .35, .4]) \\ &=\; [.5249, .5374, .5498, .5498, .5621, .5744, .5744, .5866, .5986]\end{aligned}$$

Error Vector $\boldsymbol{\epsilon}$

The error vector $\boldsymbol{\epsilon}$ is simply the difference between the actual and predicted output/response vectors.

$$\begin{aligned}\boldsymbol{\epsilon} \;&=\; \mathbf{y} - \hat{\mathbf{y}} \\ &[.5000, .3000, .2000, .8000, .5000, .3000, 1.0000, .8000, .5000] \;- \\ &[.5249, .5374, .5498, .5498, .5621, .5744, .5744, .5866, .5986] \;= \\ &[-.0249, -.2374, -.3498, .2501, -.0621, -.2744, .4255, .2133, -.0986]\end{aligned}$$

Delta Vector $\boldsymbol{\delta}$

To compute the delta vector $\boldsymbol{\delta}$, we must look ahead to get the derivative of the activation function (see exercise 8).

$$\text{sigmoid}'(\text{u}) \;=\; \text{sigmoid}(\text{u})\,[1 - \text{sigmoid}(\text{u})]$$

Therefore, since $\text{sigmoid}(\mathbf{u}) = \hat{\mathbf{y}}$

$$\begin{aligned}\boldsymbol{\delta} &= -\boldsymbol{\epsilon} * [\hat{\mathbf{y}} * (\mathbf{1} - \hat{\mathbf{y}})] \\ &[.0062, .0590, .0865, -.0619, .0153, .0670, -.1040, -.0517, .0237]\end{aligned}$$

Forming the Gradient

Combining the partial derivatives $\frac{\partial \bar{h}}{\partial b_j}$ for the loss function into an n-dimensional vector (i.e., the gradient) yields

$$\nabla\mathcal{L}(\mathbf{b}) = \frac{\partial\mathcal{L}(\mathbf{b})}{\partial\mathbf{b}} = -X^{\intercal}[\boldsymbol{\epsilon} * \mathbf{f}'(X\mathbf{b})] = X^{\intercal}\boldsymbol{\delta} \tag{10.44}$$

Parameter Update

Since many optimizers such as gradient-descent, move in the direction opposite to the gradient by a distance governed by the learning rate η (alternatively step size), the following term should be subtracted from the weight/parameter vector $\mathbf{b}$.

$$\nabla\mathcal{L}(\mathbf{b})\,\eta = X^{\intercal}\boldsymbol{\delta}\,\eta \tag{10.45}$$

The right hand side is an n-by-m matrix, m vector product yielding an n vector result. Since gradient-based optimizers move in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate η, the parameter/weight vector $\mathbf{b}$ is updated as follows:

$$\boxed{\mathbf{b} = \mathbf{b} - X^{\intercal}\boldsymbol{\delta}\,\eta} \tag{10.46}$$

10.4.6 Initializing Weights/Parameters

The weight/parameter vector **b** should be randomly set at the start of the optimization. The `weightVec` function in the `Initializer` object generates a random weight/parameter vector with elements values in (0, limit). The `stream` may be changed to generate different random numbers.

```
@param rows    the number of rows
@param limit   the maximum value for any weight
@param stream  the random number stream to use

def weightVec (rows: Int, stream: Int = 0, limit: Double = -1.0): VectorD =
    val lim = if limit <= 0.0 then limitF (rows) else limit
    val rvg = new RandomVecD (rows, lim, 0.0, stream = stream)
    rvg.gen
end weightVec

private inline def limitF (rows: Int): Double = 1.0 / sqrt (rows)
```

For testing or learning purposes, the weights may also be set manually.

```
@param w0  the initial weights for parameter b

def setWeights (w0: VectorD): Unit = b = w0
```

10.4.7 Activation Functions

An activation function f_a (or simple f) takes an aggregated signal and transforms it. These activation functions typically introduce smooth non-linearities either between lowers and upper bound (e.g., $[0,1]$ or $[-1,1]$) or follow a rectified linear unit (zero for negative signals and linear for positive signals).

The simplest activation function is the `id` or identity function where the aggregated signal is passed through unmodified. In this case, `Perceptron` is in alignment with `Regression` (see Exercise 9). This activation function is usually not intended for neural nets with more layers, since theoretically they can be reduced to a two-layer network (although it may be applied in the last layer).

More generally useful activation functions include `reLU`, `lreLU`, `eLU`, `seLU`, `geLU`, `sigmoid`, `tanh` and `gaussian`. Several activation functions are compared in [118, 137]. For these activation functions the outputs in the **y** vector need to be transformed into the range specified for the activation function, see Table 10.1. It may be also useful to transform/standardize the inputs to hit the sweet spot for the particular activation function being used.

The curves of three of the activation functions are shown in Figure 10.7

Table 10.1: Activation Functions: Identity `id`, Rectified Linear Unit `reLU`, Leaky Rectified Linear Unit `lreLU`, Exponential Linear Unit `eLU`, Scaled Exponential Unit `seLU`, Gaussian Error Linear Unit `geLU`, Sigmoid `sigmoid`, Hyperbolic Tangent `tanh`, and Gaussian `gaussian`.

Name	**Function** $y = f(u)$	**Domain**	**Range**	**Derivative** $f'(u)$	**Inverse** $u = f^{-1}(y)$
id	u	$\mathbb{R}$	$\mathbb{R}$	1	y
reLU	$\max(0,u)$	$\mathbb{R}$	$\mathbb{R}^+$	$\mathbb{1}_{u>0}$	y for $y>0$
lreLU	$\max(\alpha u, u),\ \alpha<1$	$\mathbb{R}$	$\mathbb{R}$	$\text{if}_{u<0}(\alpha, 1)$	$\min(\frac{y}{\alpha}, y)$
eLU	$\text{if}_{u<0}(\alpha(e^u-1), u)$	$\mathbb{R}$	$\mathbb{R}$	$\text{if}_{u<0}(f(u)+\alpha, 1)$	$\text{if}_{y<0}(\ln(\frac{y}{\alpha}+1), y)$
seLU	$\lambda[\text{if}_{u<0}(\alpha(e^u-1), u)]$	$\mathbb{R}$	$\mathbb{R}$	$\lambda[\text{if}_{u<0}(f(u)+\alpha, 1)]$	$\frac{1}{\lambda}[\text{if}_{y<0}(\ln(\frac{y}{\alpha}+1), y)]$
geLU	$u\Phi(u)$	$\mathbb{R}$	$\mathbb{R}$	.	.
sigmoid	$[1+e^{-u}]^{-1}$	$\mathbb{R}$	$(0,1)$	$f(u)[1-f(u)]$	$-\ln(\frac{1}{y}-1)$
tanh	$\tanh(u)$	$\mathbb{R}$	$(-1,1)$	$1-f(u)^2$	$.5\ln\left(\frac{1+y}{1-y}\right)$
gaussian	e^{-u^2}	$\mathbb{R}$	$(0,1]$	$-2ue^{-u^2}$	$\sqrt{-\ln(y)}$

The Gaussian Error Linear Unit (`geLU`) activation function is $u\Phi(u)$ where $\Phi(u)$ is the CDF for the standard Normal (or Gaussian) distribution (see the Probability Chapter). As this function is computed numerically, the following approximation may be used instead.

$$u\Phi(u) \;\approx\; f(u) \;=\; .5u\left[1 + \tanh\left[\sqrt{2/\pi}(u + .044715u^3)\right]\right] \tag{10.47}$$

The derivative of the `geLU` activation function $f'(u)$ may be computed using product and chain rules [133].

$$f'(u) \;=\; .5\tanh(.0356774u^3 + .797885u) \;+\; .5 + \tag{10.48}$$

$$(.0535161u^3 + .398942u)\cosh^{-2}(.0356774u^3 + .797885u) \tag{10.49}$$

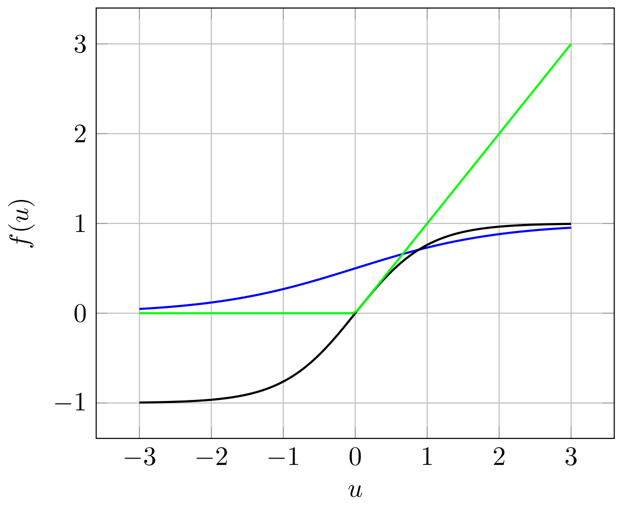


Figure 10.7: Activation Functions: sigmoid (blue), tanh (black), reLU (green)

The `sigmoid` function has an 'S' shape, which facilitates its use as a smooth and differentiable version of a step function, with larger negative values tending to zero and larger positive values tending to one. In the case of using sigmoid for the activation function, $f'(u) = f(u)[1 - f(u)]$, so

$$\boldsymbol{\delta} \;=\; -\boldsymbol{\epsilon} * [\mathbf{f'}(\mathbf{u})] \;=\; -\boldsymbol{\epsilon} * [\mathbf{f}(\mathbf{u}) * [\mathbf{1} - \mathbf{f}(\mathbf{u})]$$

Gradient-descent algorithms iteratively move in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate η, so the parameter/weight vector **b** is adjusted as follows:

$$\mathbf{b} \;=\; \mathbf{b} \,-\, X^{\intercal}\, \boldsymbol{\delta}\, \eta$$

Continuation of Example Calculation

Assuming the learning rate $\eta = 1$ and taking the $\boldsymbol{\delta}$ vector from the example, the update to parameter/weight vector **b** is

$$X^{\intercal}\, \boldsymbol{\delta}\, \eta \;=\; [.0402, -.1218, .1886]$$

Consequently, the updated value for the parameter/weight vector **b** is

$$\mathbf{b} \;=\; [.1, .2, .1] \;-\; [.0402, -.1218, .1886] \;=\; [.0597, .3218, -.0886]$$

Check to see if the new values for **b** have improved/decreased the loss function $\mathcal{L}$.

10.4.8 Basic Gradient Descent Algorithm

A basic gradient descent algorithm is an iterative process that typically terminates when the drop in the loss function $\mathcal{L}$ is small or a maximum number of iterations is exceeded. The parameters η and `max_epochs` need careful adjustments to obtain nearly (locally) optimal values for $\mathcal{L}$. Gradient-descent works by iteratively moving in the opposite direction as the gradient until a *stopping rule* evaluates to true (e.g., stop after q^{th} increase in $\mathcal{L}$ and return best solution so far). The rate of convergence can be adjusted using the learning rate η which multiplies the gradient. Setting it too low slows convergence, while setting it too high can cause oscillation or divergence. In ScalaTion, the learning rate η (`eta` in the code) is a hyper-parameter that defaults to 0.1, but is easily adjusted, e.g.,

```
    Optimizer.hp ("eta") = 0.05
```

`train` Method

The `train` method contains the main training loop that is shown below. Inside the loop, new values `yp` are predicted, from which an error vector `e` is determined. This is used to calculate the delta vector `d`, which along `x.T` and `eta` are used to update the parameter/weight vector `b`. Note, `x_, y_` which default to `x, y` constitute the training set.

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output vector

    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
        if b == null then b = weightVec (n)                  // initialize parameters/weights
        var sse0 = Double.MaxValue

        var (go, epoch) = (true, 1)
        cfor (go && epoch <= maxEpochs, epoch += 1) {       // epoch learning phase
            val yp = f.f_ (x_ * b)                           // predicted output vector f(Xb)
            e      = y_ - yp                                 // error vector for y
            val δ  = -f.d (yp) * e                           // delta vector for y
            b     -= x_.T * δ * η                            // update the parameters/weights

            val sse = (y_ - f.f_ (x_ * b)).normSq            // recompute sum of errors^2
            collectLoss (sse)                                // collect loss per epoch
            if sse >= sse0 then go = false                   // stop when sse increases
            else sse0 = sse                                  // save prior sse
        } // cfor
    end train
```

The vector function `f.f_` is the vectorization of the activation function `f.f`, and is created in ScalaTion using the `vectorize` high-order function defined in the `mathstat` package, e.g., given a scalar function f, it can produce the corresponding vector function **f**.

```
    def vectorize (f: FunctionS2S): FunctionV2V = (x: VectorD) => x.map (f(_))
    val f_ = vectorize (f)
```

The function `f.d` is the derivative of the vector activation function. The `collectLoss` method is defined in the `MonitorLoss` trait.

The core of the algorithm is the first four lines in the loop. Table 10.2 show the correspondence between these lines of code and the main/boxed equations derived in this section. Note, all the equations in the table are vector assignments.

The third line of code appears to be different from the mathematical equation, in terms of passing the pre-activation versus the post-activation response. It turns out that all derivatives for the activation functions

Table 10.2: Correspondence between Code and Boxed Equations

Code	Equation	Equation Number
`yp = f.f_ (x_ * b)`	$\hat{\mathbf{y}} = \mathbf{f}(X\mathbf{b})$	10.25
`e = y_ - yp`	$\boldsymbol{\epsilon} = \mathbf{y} - \hat{\mathbf{y}}$	10.26
`δ = -f.d (yp) * e`	$\boldsymbol{\delta} = -\boldsymbol{\epsilon} * \mathbf{f}'(X\mathbf{b})$	10.39
`b -= x_.𝒯 * δ * η`	$\mathbf{b} = \mathbf{b} - X^{\intercal}\boldsymbol{\delta}\eta$	10.44

(except Gaussian) are either formulas involving constants or simple functions of the activation function itself, so for efficiency, the `yp` vector is passed in.

Warning: This implementation is minimal to illustrate the basic required mechanisms, and as such is likely to be less accurate. For example, the code does not create mini-batches, has a stopping rule that is too simple, uses a fixed learning rate, does not utilize a modern optimization algorithm. For a useful perceptron, `NeuarlNet_2L` in the `neuralnet` package may be used with one node in the output layer.

A perceptron can be considered to be a special type of nonlinear or transformed regression, see Exercise 10.

The `ActivationFun` Object

The `Perceptron` class defaults to the `f_sigmoid` Activation Function Family (`AFF`), which is defined with the `ActivationFun` object.

```
    @param name     the name of the activation function
    @param f        the activation function itself (scalar version)
    @param f_       the vector version of the activation function
    @param d        the vector version of the activation function derivative
    @param bounds   the (lower, upper) bounds on the range of the activation function
                        e.g., (0, 1) for sigmoid, defaults to null => no limit
    @param arange   the (lower, upper) bounds on the input (active) range of the function
                        e.g., (-2, 2) for sigmoid, defaults to null => no limit

    case class AFF (name: String, f: FunctionS2S, f_ : FunctionV2V, d: FunctionV2V,
                     bounds: (Double, Double) = null, arange: (Double, Double) = null):

        val fM = matrixize (f_)       // the matrix version of activation function
        val dM = matrixize (d)        // the matrix version of activation function derivative

    end AFF
```

The `sigmoid` and `tanh` famalies are defined as follows:

```
    val f_sigmoid = AFF ("sigmoid", sigmoid, sigmoid_, sigmoidD, (0, 1), (-2, 2))
    val f_tanh    = AFF ("tanh", tanh, tanh_, tanhD, (-1, 1), (-2, 2))
```

In general, the `AFF` for family `f` contains the following:

```
    val f = AFF (n, f, f_, d, (lb, ub), (a1, a2))
```

Rescaling

Depending on the activation function, rescaling of outputs and/or inputs may be necessary:

1. **Output**: If the actual response vector $\mathbf{y}$ is outside the bounds/range of the activation function, it will be impossible for the predicted response vector $\hat{\mathbf{y}}$ to approximate it, so rescaling will be necessary. The `bounds` in `AFF` is used for rescaling vector $\mathbf{y}$.

2. **Input**: If the linear combination of inputs takes the pre-activation value beyond the effective domain (active range) of the activation function, training will become slow (or even ineffective), e.g., when u is large, $f(u)$ will be nearly 1 for `sigmoid`, even when there are substantial changes to u induced by updates to the parameters. The `arange` in `AFF` is used for rescaling the columns of matrix X.

As a convenience, all the modeling techniques in ScalaTion have a factory method for rescaling the inputs and outputs.

```
    @param x        the data/input matrix
    @param y        the response/output vector
    @param fname    the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to hp)
    @param f        the activation function family for layers 1->2 (input to output)

    def rescale (x: MatrixD, y: VectorD, fname: Array [String] = null,
                 hparam: HyperParameter = hp, f: AFF = f_sigmoid): Perceptron =
        var itran: FunctionV2V = null                 // inverse transform -> original scale

        val x_s = if scale then rescaleX (x, f)
                  else x
        val y_s = if f.bounds != null then { val y_i = rescaleY (y, f);
                                             itran = y_i._2; y_i._1 }
                  else y

        new Perceptron (x_s, y_s, fname, hparam, f, itran)
    end rescale
```

Other activation functions should be experimented with, as one may produce better results. All the activation functions shown in Table 10.1 are available in the `ActivationFun` object.

Essentially, parameter optimization in perceptrons involves using/calculating several vectors as summarized in Table 10.3 where n is the number of parameters and m is the number of instances used at a particular point in the iterative optimization algorithm, for example, corresponding to the total number of instances in a training set for Gradient Descent (GD).

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) utilizes a fraction of the m instances to make updates to the parameters, corresponding to just 1 for pure SGD or to the size of a *mini-batch* (e.g., 30) for the common form of SGD. Instead of single loop of the `Perceptron`, there is a nested loop, the outer loop is over training epochs, while the inner loop is over mini-batches.

```
     cfor (go && epoch <= maxEpochs, epoch += 1) {               // iterate over each epoch
         val batches = permGen.igen.chop (nB)                    // permute indices & chop
         for ib <- batches do b -= updateWeight (x(ib), y(ib))   // iterate: update param b
```

This has the obvious benefit of reducing the amount of computation required to make parameter updates. Somewhat surprisingly, it also makes it easier for the algorithm to escape local minima and to generally be a more robust algorithm. The mini-batch size is another hyper-parameter that can be tuned.

Table 10.3: Vectors Used in Perceptrons

Vector	Space	Formula	Description
$\mathbf{x}_i$	$\mathbb{R}^n$	given	the i^{th} row of the input/data matrix
$\mathbf{x}_{:j}$	$\mathbb{R}^m$	given	the j^{th} column of the input/data matrix
$\mathbf{b}$	$\mathbb{R}^n$	given	the parameter vector (updated per iteration)
$\mathbf{u}$	$\mathbb{R}^m$	$X\mathbf{b}$	the pre-activation vector
$\mathbf{y}$	$\mathbb{R}^m$	given	the actual output/response vector
$\hat{\mathbf{y}}$	$\mathbb{R}^m$	$\mathbf{f}(\mathbf{u})$	the predicted output/response vector
$\boldsymbol{\epsilon}$	$\mathbb{R}^m$	$\mathbf{y} - \hat{\mathbf{y}}$	the error/residual vector
$\boldsymbol{\delta}$	$\mathbb{R}^m$	$-\boldsymbol{\epsilon} * \mathbf{f}'(\mathbf{u})$	the negative-slope-weighted error vector
$\beta \oplus \mathbf{w}$	$\mathbb{R}^n$	$\mathbf{b}$	concatenation of bias scalar and weight vector

10.4.9 `Perceptron` Class

Class Methods:

```
    @param x        the data/input m-by-n matrix (data consisting of m input vectors)
    @param y        the response/output m-vector (data consisting of m output values)
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters for the model/network (defaults to Perceptron.hp)
    @param f        the activation function family for layers 1->2 (input to output)
    @param itran    the inverse transformation function returns responses to original scale

    class Perceptron (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                      hparam: HyperParameter = Perceptron.hp,
                      f: AFF = f_sigmoid, val itran: FunctionV2V = null)
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2)
             with MonitorLoss:

    def setWeights (w0: VectorD): Unit = b = w0
    def reset (eta_ : Double): Unit = eta = eta_
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def predict (z: VectorD): Double = f.f (b dot z)
    override def predict (z: MatrixD = x): VectorD = f.f_ (z * b)
    override def buildModel (x_cols: MatrixD): Perceptron =
```

The `train` method uses Gradient Descent with a simple stopping rule and a non-adaptive learning rate.

A better optimizer is used by the `train` method in the `NeuralNet_2L` class that uses Stochastic Gradient Descent, a better stopping rule (see `StoppingRule` class), and an adaptive learning rate. The work is delegated to the `Optimizer_SGD` object and can easily be changes to use Stochastic Gradient Descent with Momentum using the `Optimizer_SGDM` object. A `NeuralNet_2L` may be thought of as multiple `Perceptron`s.

10.4.10 Exercises

1. Plot the rest of activation functions in Table 10.1 in the same plot and compare them.

2. The Texas Temperature regression problem can also be analyzed using a perceptron.

```
    val fname = Array ("one", "Lat", "Elev", "Long")

    // 16 data points:         one       x1       x2        x3
    //                                   Lat     Elev      Long         County
    val x = MatrixD ((16, 4), 1.0, 29.767,   41.0,  95.367,     // Harris
                              1.0, 32.850,  440.0,  96.850,     // Dallas
                              1.0, 26.933,   25.0,  97.800,     // Kennedy
                              1.0, 31.950, 2851.0, 102.183,     // Midland
                              1.0, 34.800, 3840.0, 102.467,     // Deaf Smith
                              1.0, 33.450, 1461.0,  99.633,     // Knox
                              1.0, 28.700,  815.0, 100.483,     // Maverick
                              1.0, 32.450, 2380.0, 100.533,     // Nolan
                              1.0, 31.800, 3918.0, 106.400,     // El Paso
                              1.0, 34.850, 2040.0, 100.217,     // Collington
                              1.0, 30.867, 3000.0, 102.900,     // Pecos
                              1.0, 36.350, 3693.0, 102.083,     // Sherman
                              1.0, 30.300,  597.0,  97.700,     // Travis
                              1.0, 26.900,  315.0,  99.283,     // Zapata
                              1.0, 28.450,  459.0,  99.217,     // Lasalle
                              1.0, 25.900,   19.0,  97.433)     // Cameron

    val y = VectorD (56.0, 48.0, 60.0, 46.0, 38.0, 46.0, 53.0, 46.0,
                     44.0, 41.0, 47.0, 36.0, 52.0, 60.0, 56.0, 62.0)

    banner ("Perceptron with scaled y values")
    val mod = Perceptron.rescale (x, y, fname)      // factory method auto rescales
//  val mod = new Perceptron (x, y, fname)          // constructor does not auto
    rescale

    mod.reset (eta_ = 0.5)                          // try several learning rates
    mod.trainNtest ()()                             // train and test the model

    banner ("scaled prediction")
    val yp = mod.predict ()                         // scaled predicted output values
    println ("target output:    y  = " + y)
    println ("predicted output: yp = " + yp)

    banner ("unscaled prediction")
    val (ymin, ymax) = (y.min, y.max)
    val ypu = unscaleV ((ymin, ymax), (0, 1)) (yp)  // unscaled predicted output values
    println ("target output:   y   = " + y)
    println ("unscaled output: ypu = " + ypu)
```

3. Analyze the `Example_Concrete` dataset in the `neuralnet` package, which has three output variables y_0, y_1 and y_2. Create a perceptron for each output variable.

4. Use the following formula for matrix-vector multiplication

$$\mathbf{u} \;=\; X\mathbf{b} \;=\; \sum_j b_j \mathbf{x}_{:j}$$

to derive the formula for the following partial derivative

$$\frac{\partial \mathbf{u}}{\partial b_j} \;=\; \mathbf{x}_{:j}$$

5. Given the formula for the loss function $\mathcal{L} : \mathbb{R}^m \to \mathbb{R}$ expressed in terms of the pre-activation vector $\mathbf{u} = X\mathbf{b}$ and the vectorized activation function $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$,

$$\mathcal{L}(\mathbf{u}) \;=\; \frac{1}{2}\,(\mathbf{y} - \mathbf{f}(\mathbf{u}))\,\cdot\,(\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

derive the formula for the gradient of $\mathcal{L}$ with respect to $\mathbf{u}$.

$$\nabla\mathcal{L} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{u}} \;=\; -\,\mathbf{f}'(\mathbf{u})(\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

Hint: Take the gradient, $\frac{\partial \mathcal{L}}{\partial \mathbf{u}}$, using the product rule $(d_1 \cdot f_2 + f_1 \cdot d_2)$.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{u}} \;=\; -\,\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}} \cdot (\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

where $f_1 = f_2 = \mathbf{y} - \mathbf{f}(\mathbf{u})$ and $d_1 = d_2 = -\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$. Next, assuming $\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$ is a diagonal matrix, show that the above equation can be rewritten as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{u}} \;=\; -\,\mathbf{f}'(\mathbf{u})\,(\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

where $\mathbf{f}'(\mathbf{u}) = [f'(u_0), \dots, f'(u_{m-1})]$ and the two vectors, $\mathbf{f}'(\mathbf{u})$ and $\mathbf{y} - \mathbf{f}(\mathbf{u})$, are multiplied, element-wise.

6. Show that the m-by-m Jacobian matrix, $\mathbb{J}_\mathbf{f}(\mathbf{u}) = \frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$, is a diagonal matrix, i.e.,

$$\mathbb{J}_\mathbf{f}(\mathbf{u}) \;=\; \left[\frac{\partial f_i(\mathbf{u})}{\partial u_j}\right] \;=\; 0 \text{ if } i \neq j$$

where $f_i = f$ the scalar activation function. Each diagonal element is the derivative of the activation function applied to the i^{th} input, $f'(u_i)$. See the section on Vector Calculus in Chapter 2 that discusses Gradient Vectors, Jacobian Matrices and Hessian Matrices.

7. Show the first 10 iterations that update the parameter/weight matrix $\mathbf{b}$ that is initialized to $[.1, .2, .1]$. Use the following combined input-output matrix. Let the perceptron use the default sigmoid function.

```
    // 9 data points:          one     x1     x2      y
    val xy = MatrixD ((9, 4), 1.0,   0.0,   0.0,   0.5,
                              1.0,   0.0,   0.5,   0.3,
                              1.0,   0.0,   1.0,   0.2,
                              1.0,   0.5,   0.0,   0.8,
                              1.0,   0.5,   0.5,   0.5,
                              1.0,   0.5,   1.0,   0.3,
                              1.0,   1.0,   0.0,   1.0,
                              1.0,   1.0,   0.5,   0.8,
                              1.0,   1.0,   1.0,   0.5)

    Preceptron.hp("eta") = 1.0                       // try several values for eta
    val nn = new Perceptron (x, y, null, hp)         // create a perceptron, user control
//  val nn = Perceptron (xy, null, hp)               // create a perceptron, automatic
    scaling
```

For each iteration, do the following: Print the weight/parameter update vector $X^\intercal \boldsymbol{\delta}\eta$ and the new value for weight/parameter vector $\mathbf{b}$, Make a table with m rows showing values for

$$x_1, x_2, y;\ u, \hat{y}, \epsilon, \epsilon^2, \hat{y}(1-\hat{y}), \text{ and } \delta$$

Try letting $\eta = 1$ then 2. Also, compute sse and R^2.

8. Show that for `sigmoid` function

$$\text{sigmoid}(u) \;=\; [1+e^{-u}]^{-1} \;=\; \frac{1}{1+e^{-u}}$$

its derivative is

$$\text{sigmoid}'(u) \;=\; \frac{d\,\text{sigmoid}(u)}{du} \;=\; \text{sigmoid}(u)[1-\text{sigmoid}(u)]$$

9. Show that when the activation function f is the id function, that $f'(\mathbf{u})$ is the one vector, $\mathbf{1}$. Plug this into the equation for the gradient of the loss function to obtain the following result.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} \;=\; -\,X^\intercal[\mathbf{1}\,\boldsymbol{\epsilon}] \;=\; -\,X^\intercal(\mathbf{y}-X\mathbf{b})$$

Setting the gradient equal to zero, now yields $(X^\intercal X)\mathbf{b} = X^\intercal \mathbf{y}$, the Normal Equations.

10. Show that a `Perceptron` with an invertible activation function f is similar to `TranRegression` with tranform f^{-1}. Explain any differences in the parameter/weight vector $\mathbf{b}$ and the sum of squared errors sse. Use the sigmoid activation function and the AutoMPG dataset and make the following two plots (using `PlotM`): `y`, `ypr`, `ypt` vs. `t` and `y`, `ypr`, `ypp` vs. `t`, where `y` is the actual response/output, `ypr` is the prediction from `Regression`, `ypt` is the prediction from `TranRegression` and `ypp` is the prediction from `Perceptron`.

10.5 Multi-Output Prediction

The `PredictorMV` trait (Predictor Multi-Variate) provides the basic structure and API for a variety of modeling techniques that produce multiple responses/outputs, e.g., Neural Networks and Multi-Variate Regression. It serves the same role that `Predictor` does for the regression modeling techniques that have a single response/output variable.

10.5.1 Model Equation

For modeling techniques extending this trait, the model equation takes an input vector $\mathbf{x}$, pre-multiplies it by the transpose of the parameter matrix B, applies a function $\mathbf{f}$ to the resulting vector and adds an error vector $\boldsymbol{\epsilon}$,

$$\mathbf{y} \;=\; \mathbf{f}(B \cdot \mathbf{x}) + \boldsymbol{\epsilon} \;=\; \mathbf{f}(B^{\intercal}\mathbf{x}) + \boldsymbol{\epsilon} \tag{10.50}$$

where

- $\mathbf{y}$ is an n_y-dimensional output/response random vector,
- $\mathbf{x}$ is an n-dimensional input/data vector,
- B is an n-by-n_y parameter matrix,
- $\mathbf{f} : \mathbb{R}^{n_y} \to \mathbb{R}^{n_y}$ is a function mapping vectors to vectors, and
- $\boldsymbol{\epsilon}$ is an n_y-dimensional residual/error random vector.

For Multi-Variate Regression (`RegressionMV`), $\mathbf{f}$ is the identity function.

10.5.2 Training

The training equation takes the model equation and several instances in a dataset to provide estimates for the values in parameter matrix B. Compared to the single response/output variable case, the main difference is that the response/output vector, the parameter vector, and the error vector now all become matrices.

$$Y \;=\; \mathbf{f}(XB) + E \tag{10.51}$$

where X is an m-by-n data/input matrix, Y is an m-by-n_y response/output matrix, B is an n-by-n_y parameter matrix, $\mathbf{f}$ is a function mapping one m-by-n_y matrix to another, and E is an m-by-n_y residual/error matrix. Note, a bold function symbol $\mathbf{f}$ is used is used to denote a function mapping either vectors to vectors (as was the case in the Model Equation subsection) or matrices to matrices (as is the case here).

$$\mathbf{f} : \mathbb{R}^{m \times n_y} \to \mathbb{R}^{m \times n_y}$$

If one is interested in referring to the k^{th} component (or column) of the output, the model equation decomposes into

$$\mathbf{y}_{:k} \;=\; \mathbf{f}(X\mathbf{b}_{:k}) + \boldsymbol{\epsilon}_{:\mathbf{k}} \tag{10.52}$$

Recall that $\mathbf{y}_k$ indicates the k^{th} row of matrix Y, while $\mathbf{y}_{:k}$ indicates the k^{th} column.

Analogous to other predictive modeling techniques, `PredictorMV` takes four arguments: the data/input matrix `x`, the response/output matrix `y`, the feature/variable names `fname`, and the hyper-parameters for the model/network `hparam`.

10.5.3 `PredictorMV` Trait

Class Methods:

```
    @param x        the input/data m-by-n matrix
                        (augment with a first column of ones to include intercept in model)
    @param y        the response/output m-by-ny matrix
    @param fname    the feature/variable names (if null, use x_j s)
    @param hparam   the hyper-parameters for the model/network

    trait PredictorMV (x: MatrixD, y: MatrixD, protected var fname: Array [String],
                       hparam: HyperParameter)
          extends Model:

    def getX: MatrixD = x
    def getY: MatrixD = y
    def getFname: Array [String] = fname
    def numTerms: Int = getX.dim2
    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit
    def train (x_ : MatrixD, y_ : VectorD): Unit =                        // first column only
    def train2 (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD)
    def test (x_ : MatrixD, y_ : VectorD): (VectorD, VectorD) =           // first column only
    def trainNtest (x_ : MatrixD = x, y_ : MatrixD = y)
                   (xx: MatrixD = x, yy: MatrixD = y): (MatrixD, MatrixD) =
    def trainNtest2 (x_ : MatrixD = x, y_ : MatrixD = y)
                    (xx: MatrixD = x, yy: MatrixD = y): (MatrixD, MatrixD) =
    def makePlots (yy: MatrixD, yp: MatrixD): Unit =
    override def report (ftMat: MatrixD): String =
    def orderByY (y_ : VectorD, yp_ : VectorD): (VectorD, VectorD) =
    def predict (z: VectorD): VectorD                                     // = b dot z
    def predict (x_ : MatrixD): MatrixD =
    def hparameter: HyperParameter = hparam
    def parameter: MatrixD =
    def parameters: NetParams = bb
    def residual: MatrixD = e

    def buildModel (x_cols: MatrixD): PredictorMV
    def selectFeatures (tech: SelectionTech, idx_q: Int = QoF.rSqBar.ordinal,
                        cross: Boolean = true): (LinkedHashSet [Int], MatrixD) =
    def forwardSel (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal): BestStep =
    def forwardSelAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
                      (LinkedHashSet [Int], MatrixD) =
    def backwardElim (cols: LinkedHashSet [Int], idx_q: Int = QoF.rSqBar.ordinal,
                      first: Int = 1): BestStep =
    def backwardElimAll (idx_q: Int = QoF.rSqBar.ordinal, first: Int = 1,
                         cross: Boolean = true): (LinkedHashSet [Int], MatrixD) =
    def stepRegressionAll (idx_q: Int = QoF.rSqBar.ordinal, cross: Boolean = true):
                          (LinkedHashSet [Int], MatrixD) =
```

```scala

    def vif (skip: Int = 1): VectorD =
    inline def testIndices (n_test: Int, rando: Boolean): IndexedSeq [Int] =
    def validate (rando: Boolean = true, ratio: Double = 0.2)
                 (idx : IndexedSeq [Int] =
                  testIndices ((ratio * y.dim).toInt, rando)): MatrixD =
    def crossValidate (k: Int = 5, rando: Boolean = true): Array [Statistic] =
```

The methods provided by PredictorMV are similar to those in Predictor, but extensions to handle response matrices are added.

The RegressionMV class extends PredictorMV and shares the factorization but individually solves for each output/response variable and is hence faster than performing Regression individually.

10.5.4 RegressionMV Class

Class Methods:

```scala
    @param x        the data/input m-by-n matrix
                        (augment with a first column of ones to include intercept in model)
    @param y        the response/output m-by-ny matrix
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to Regression.hp)

    class RegressionMV (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                        hparam: HyperParameter = Regression.hp)
          extends PredictorMV (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD) =
    override def summary (x_ : MatrixD = getX, fname_ : Array [String] = fname,
    def predict (z: VectorD): VectorD = b.asInstanceOf [MatrixD] dot z
    override def predict (x_ : MatrixD): MatrixD = x_ * b.asInstanceOf [MatrixD]
    def buildModel (x_cols: MatrixD): RegressionMV =
```

10.5.5 Optimizer Object and Trait

The default hyper-parameters for all neural networks are defined in the Optimizer object. Typically, some of these hyper-parameters will need to be tuned. The constants are more baked-in than the hyper-parameters and require recompilation to change.

```scala
object Optimizer:

    val hp = new HyperParameter
    hp += ("eta", 0.1, 0.1)                   // learning/convergence rate
    hp += ("bSize", 20, 20)                   // mini-batch size, common range 10 to 40
    hp += ("maxEpochs", 400, 400)             // maximum number of epochs/iterations
    hp += ("lambda", 0.01, 0.01)              // regularization/shrinkage hyper-parameter
    hp += ("upLimit", 4, 4)                   // up-limit hyper-parameter for stopping rule
    hp += ("beta", 0.9, 0.9)                  // momentum decay hyper-parameter
    hp += ("nu", 0.9, 0.9)                    // importance of momentum in parameter est.

    val ADJUST_PERIOD  = 100                  // number of epochs before adjusting eta
    val ADJUST_FACTOR  = 1.1                  // learning rate adjustment factor (1+)
    val NSTEPS         = 16                   // steps in eta grid/line search
    val estat = new Statistic ("epochs")

end Optimizer
```

The Optimizer trait provides an abstract method called optimize that must be implemented in extending optimization classes such Optimizer_SGD. It also provides a method for automatic optimization that includes built-in grid search. In addition, it provides a helper method (permGenerator) that facilitates the creation of random mini-batches.

```
trait Optimizer extends MonitorLoss with StoppingRule:

    def permGenerator (m: Int, rando: Boolean = true): PermutedVecI =
    def optimize (x: MatrixD, y: MatrixD, b: NetParams, eta_ : Double, f: Array [AFF]):
                 (Double, Int)
    def auto_optimize (x: MatrixD, y: MatrixD, b: NetParams, etaI: (Double, Double),
                       f: Array [AFF], opti: (MatrixD, MatrixD, NetParams, Double,
                       Array [AFF]) => (Double, Int)): (Double, Int) =
end Optimizer
```

10.5.6 NetParam Class

A model producing multiple output variables will have parameters as weight matrices. They may also have bias vectors. To unify these cases, ScalaTion utilizes the NetParam case class for holding a weight matrix along with an optional bias vector. Linear algebra like operators are provided for convenience, e.g., the *: allows one to write x *: p, corresponding to the mathematical expression XP where X is the input/data matrix and P holds the parameters. If the bias b is null, this is just matrix multiplication.

```
    def *: (x: MatrixD): MatrixD = x * w + b
```

Inside, the x is multiplied by the weight matrix w and the bias vector b is added. Note, the *: is right associative since the NetParam object is on right (see NeuralNet_2L for an example of its usage).

Class Methods:

```
    @param w  the weight matrix
    @param b  the optional bias/intercept vector (null => not used)

    case class NetParam (w: MatrixD, var b: VectorD = null)

    def copy: NetParam = NetParam (w.copy, if b != null then b.copy else null)
    def trim (dim: Int, dim2: Int): NetParam =
    def update (c: NetParam): Unit = { w = c.w; b = c.b }
    def set (c: NetParam): Unit = { w = c.w; b = c.b }
    def update (cw: MatrixD, cb: VectorD = null): Unit = { w = cw; b = cb }
    def set (cw: MatrixD, cb: VectorD = null): Unit = { w = cw; b = cb }
    def += (c: NetParam): Unit =
    def += (cw: MatrixD, cb: VectorD): Unit =
    def -= (c: NetParam): Unit =
    def -= (cw: MatrixD, cb: VectorD = null): Unit =
    def * (x: MatrixD): MatrixD =
    def *: (x: MatrixD): MatrixD = x * w + b
    def dot (x: VectorD): VectorD = (w dot x) + b
    def toMatrixD: MatrixD = if b == null then w else b +: w
    override def toString: String = s"b.w = $w \\n b.b = $b"
```

10.6 Two-Layer Neural Networks

The `NeuralNet_2L` class supports multi-valued 2-layer (input and output) Neural Networks. The inputs into a Neural Network are given by the input vector $\mathbf{x}$, while the outputs are given by the output vector $\mathbf{y}$. Each input x_j is associated with an input node in the network, while each output y_k is associated with an output node in the network, as shown in Figure 10.8. The input layer consists of n input nodes, while the output layer consists of n_y output nodes.

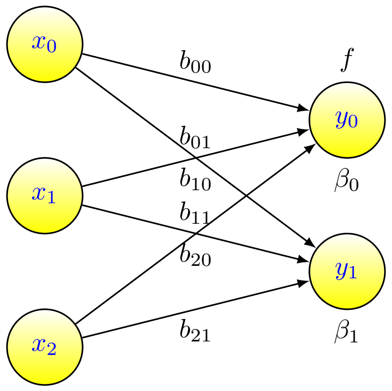


Figure 10.8: Two-Layer (input, output) Neural Network

An edge connects each input node with each output node, i.e., there are nn_y edges in the network. The edge connecting input node j to output node k has weight b_{jk}. In addition, below each output node is a bias β_k. An alternative to having explicit bias offsets, is to include an intercept in the model (as an implicit bias) by adding a special input node (say x_0) having its input always set to 1.

10.6.1 Model Equation

The weights on the edges are analogous to the parameter vector $\mathbf{b}$ in regression. Each output variable y_k, has its own parameter vector $\mathbf{b}_{:k}$. These are collected as column vectors into a parameter/weight matrix B, where parameter value b_{jk} is the edge weight connecting input node x_j with output node y_k.

After training, given an input vector $\mathbf{x}$, the network can be used to predict the corresponding output vector $\mathbf{y}$. The network predicts an output/response value for y_k by taking the weighted sum of its inputs and passing this sum through activation function f.

$$y_k \;=\; f(\mathbf{b}_{:k} \cdot \mathbf{x}) + \epsilon_k \;=\; f\big(\sum_{j=0}^{n-1} b_{jk} x_j\big) + \epsilon_k$$

The model equation for `NeuralNet_2L` can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f}(B \cdot \mathbf{x}) + \boldsymbol{\epsilon} \;=\; \mathbf{f}(B^{\intercal}\mathbf{x}) + \boldsymbol{\epsilon}} \tag{10.53}$$

10.6.2 Training

Given several input vectors and output vectors in a training dataset ($i = 0, \ldots, m-1$), the goal is to optimize/fit the parameters/weights B. The training dataset consisting of m input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight matrix B. Given an input matrix $X \in \mathbb{R}^{m \times n}$ consisting of m input vectors and an output matrix $Y \in \mathbb{R}^{m \times n_y}$ consisting of m output vectors, minimize the distance between the actual/target output matrix Y and the predicted output matrix $\hat{Y}$,

$$\hat{Y} \;=\; \mathbf{f}(XB) \tag{10.54}$$

This will minimize the error matrix $E = Y - \hat{Y}$

$$\min_B \|Y - \mathbf{f}(XB)\|_F \tag{10.55}$$

where $\|\cdot\|_F$ is the *Frobenius norm*, X is a m-by-n matrix, Y is a m-by-n_y matrix, and B is a n-by-n_y matrix. Other norms may be used as well, but the square of the Frobenius norm will give the overall sum of squared errors sse.

$$\|E\|_F^2 \;=\; \sum_{i=0}^{m-1} \sum_{j=0}^{n_y-1} \epsilon_{ij}^2 \tag{10.56}$$

10.6.3 Optimization

As was the case with regression, it is convenient to minimize the dot product of the error with itself. We do this for each of the columns of the Y matrix to get the sse for each y_k and sum them up. The goal then is to simply minimize the loss function $\mathcal{L}(B) = \frac{1}{2} sse(B)$. As in the Perceptron section, we work with half of the sum of squared errors sse. Summing the error over each column vector $\mathbf{y}_{:k}$ in matrix Y gives

$$\mathcal{L}(B) \;=\; \frac{1}{2} \sum_{k=0}^{n_y-1} (\mathbf{y}_{:k} - \mathbf{f}(X\mathbf{b}_{:k})) \cdot (\mathbf{y}_{:k} - \mathbf{f}(X\mathbf{b}_{:k})) \tag{10.57}$$

This nonlinear optimization problem may be solved by a variety of optimization techniques, including Gradient-Descent, Stochastic Gradient Descent or Stochastic Gradient Descent with Momentum.

Most optimizers require a derivative and ideally these should be provided in functional form (otherwise the optimizer will need to numerically approximate them). Again, for the `sigmoid` activation function,

$$\text{sigmoid}(u) \;=\; \frac{1}{1 + e^{-u}}$$

the derivative is

$$\text{sigmoid}(u)[1 - \text{sigmoid}(u)]$$

To minimize the loss function, we decompose it into n_y functions.

$$\mathcal{L}(\mathbf{b}_{:k}) \;=\; \tfrac{1}{2} (\mathbf{y}_{:k} - \mathbf{f}(X\mathbf{b}_{:k})) \cdot (\mathbf{y}_{:k} - \mathbf{f}(X\mathbf{b}_{:k}))$$

Notice that this is the same as the Perceptron loss function, just with subscripts on $\mathbf{y}$ and $\mathbf{b}$.

In Regression, we took the gradient and set it equal to zero. Here, gradients will need to be computed by the optimizer. The equations will be the same as given in the Perceptron section, again just with subscripts added. The boxed equations from the Perceptron section become the following: The prediction vector for the k^{th} response/output is

$$\boxed{\hat{\mathbf{y}}_{:\mathbf{k}} \;=\; \mathbf{f}(X\mathbf{b}_{:k})} \tag{10.58}$$

The error vector for the k^{th} response/output is

$$\boxed{\boldsymbol{\epsilon}_{:\mathbf{k}} \;=\; \mathbf{y}_{:k} - \hat{\mathbf{y}}_{:\mathbf{k}} \;=\; \mathbf{y}_{:k} - \mathbf{f}(X\mathbf{b}_{:k})} \tag{10.59}$$

The delta vector for the k^{th} response/output is

$$\boxed{\boldsymbol{\delta}_{:\mathbf{k}} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{u}_k} \;=\; -\,\boldsymbol{\epsilon}_{:\mathbf{k}} * \mathbf{f}'(X\mathbf{b}_{:k})} \tag{10.60}$$

where $\mathbf{u}_k = X\mathbf{b}_{:k}$. The gradient with respect to the k^{th} parameter vector is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_{:k}} \;=\; -\,X^{\intercal}[\boldsymbol{\epsilon}_{:\mathbf{k}} * \mathbf{f}'(X\mathbf{b}_{:k})] \;=\; X^{\intercal}\boldsymbol{\delta}_{:\mathbf{k}}$$

Finally, the update for the k^{th} parameter vector is

$$\boxed{\mathbf{b}_{:k} \;=\; \mathbf{b}_{:k} \;-\; X^{\intercal}\,\boldsymbol{\delta}_{:\mathbf{k}}\,\eta} \tag{10.61}$$

Sigmoid Case

For the sigmoid function, $\mathbf{f}'(X\mathbf{b}_{:k}) = \mathbf{f}(X\mathbf{b}_{:k}) * [1 - \mathbf{f}(X\mathbf{b}_{:k})]$, so

$$\boldsymbol{\delta} \;=\; -\,\boldsymbol{\epsilon}_{:\mathbf{k}} * \mathbf{f}(X\mathbf{b}_{:k}) * [1 - \mathbf{f}(X\mathbf{b}_{:k})]$$

10.6.4 Matrix Version

Of course the boxed equations may be rewritten in matrix form. The m-by-n_y prediction matrix $\hat{Y}$ has a column for each output variable.

$$\boxed{\hat{Y} \;=\; \mathbf{f}(XB)} \tag{10.62}$$

The m-by-n_y negative of the error matrix E is the difference between the predicted and actual/target output/response.

$$\boxed{E \;=\; \hat{Y} \,-\, Y} \tag{10.63}$$

The m-by-n_y delta matrix Δ adjusts the error according to the slopes within $\mathbf{f}'(XB)$ and is the element-wise matrix (Hadamard) product of $\mathbf{f}'(XB)$ and E.

$$\boxed{\Delta \;=\; \mathbf{f}'(XB) \odot E} \tag{10.64}$$

In math, the Hadamard product may be denoted by the $\odot$ symbol, while in ScalaTion it is denoted by $\odot$ or $*\sim$. Finally, the n-by-n_y parameter matrix B is updated by $-X^\intercal \Delta\eta$.

$$\boxed{B \;=\; B - X^\intercal \Delta\eta} \tag{10.65}$$

The `train` Method

The `train` method calls the appropriate optimizer and records statistics about the number of epochs. The corresponding code for the `train` method is shown below:

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output matrix

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
        val epochs = opti.optimize2 (x_, y_, bb, eta, Array (f))  // optimize parameters bb
        println (s"ending epoch = $epochs")
        estat.tally (epochs._2)
    end train
```

For `NeuralNet_2L`, the bulk of the work is done by the `optimize2` method, in for example, the `Optimizer_SGD` class. The initializattion part of this method is defined as follows:

```
    @param x    the m-by-nx input matrix (training data consisting of m input vectors)
    @param y    the m-by-ny output matrix (training data consisting of m output vectors)
    @param bb   the array of parameters (weights & biases) between every two adjacent layers
    @param eta  the initial learning/convergence rate
    @param ff   the array of activation function family for every two adjacent layers

    def optimize2 (x: MatrixD, y: MatrixD,
                   bb: NetParams, eta: Double, ff: Array [AFF]): (Double, Int) =
        val permGen   = permGenerator (x.dim)                     // permutation vector gen
        val b         = bb(0)                                     // net-param: weight matrix
                                                                  // and bias vector
        val f         = ff(0)                                     // activation function
        val bSize     = min (hp("bSize").toInt, x.dim)            // batch size
        val maxEpochs = hp("maxEpochs").toInt                     // maximum number of epochs
        val upLimit   = hp("upLimit").toInt                       // limit on increasing lose
        var η         = eta                                       // set initial learning rate
        val nB        = x.dim / bSize                             // the number of batches
```

The main loop iterates up to `maxEpochs`, but may exist early depending on what `stopWhen` returns. The learning rate η is periodically adjusted.

```
        var sse_best_   = -0.0
        var (go, epoch) = (true, 1)
        cfor (go && epoch <= maxEpochs, epoch += 1) {             // iterate over each epoch
            val batches = permGen.igen.chop (nB)                  // permute indices & chop

            for ib <- batches do b -= updateWeight (x(ib), y(ib))   // update parameter b

            val sse = (y - f.fM (b * x)).normFSq                  // recompute sse
            collectLoss (sse)                                     // collect loss per epoch
            val (b_best, sse_best) = stopWhen (Array (b), sse)
            if b_best != null then
                b.set (b_best (0))
                sse_best_ = sse_best                            // save best in sse_best_
                go = false
            else
                if epoch % ADJUST_PERIOD == 0 then η *= ADJUST_FACTOR
            end if
        } // cfor
```

The parameters are updated in the `updateWeight` method.

```
        inline def updateWeight (x: MatrixD, y: MatrixD): MatrixD =
            val α  = η / x.dim                                  // eta over the batch size
            val yp = f.fM (b * x)                               // prediction: Yp = f(XB)
            val ϵ  = yp - y                                     // negative of error matrix
            val δ  = f.dM (yp) ⊙ ϵ                              // delta matrix for y

            x.𝒯 * δ * α                                         // return change in params
        end updateWeight
```

The end of the `optimize2` method returns the best value for the loss function and the number of epochs.

```
        if go then ((y - f.fM (b * x)).normFSq, maxEpochs)      // return sse and # epochs
        else       (sse_best_, epoch - upLimit)
    end optimize2
```

Note: `f.fM` is the matrix version of the activation function and it is created using the `matrixize` high-order function that takes a vector function as input.

```
    def matrixize (f: FunctionV2V): FunctionM2M = (x: MatrixD) => x.map (f(_))
    val fM = matrixize (f_)
```

Similary, `f.dM` is the matrix version of the derivative of the activation function. The `NeuralNet_2L` class also provides `train2` (with built in η search) methods.

Also note: ScalaTion provides the following alternatives for (a) Hadamard product: $\odot$ or $*\sim$, and (b) Transpose: T-like Unicode symbol or `transpose`.

10.6.5 `NeuralNet_2L` Class

Class Methods:

```
    @param x        the m-by-n input/data matrix (full/training data having m input vectors)
    @param y        the m-by-ny output/response matrix (full/training data having m vectors)
    @param fname_   the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters for the model/network (defaults to Optimizer.hp)
    @param f        the activation function family for layers 1->2 (input to output)
    @param itran    the inverse transformation function returns response matrix to original
                    scale

    class NeuralNet_2L (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                        hparam: HyperParameter = Optimizer.hp,
                        f: AFF = f_sigmoid, val itran: FunctionM2M = null)
          extends PredictorMV (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    override def train2 (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD) =
    override def makePlots (yy_ : MatrixD, yp: MatrixD): Unit =
    def predict (v: VectorD): VectorD = f.f_ (bb(0) dot v)
    override def predict (v: MatrixD = x): MatrixD = f.fM (bb(0) * v)
    def buildModel (x_cols: MatrixD): NeuralNet_2L =
    def summary2 (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                  b_ : MatrixD = parameter): String =
```

10.6.6 NeuralNet_2L Object

The NeuralNet_2L companion object provides factory methods for buidling two-layer neural nets: The apply method creates a NeuralNet_2L with automatic resclaing from a combined data matrix. The rescale method creates a NeuralNet_2L with automatic rescaling from a data matrix and response matrix. The perceptron method creates a NeuralNet_2L with automatic rescaling from a data matrix and response vector. As the number of output nodes is one in this case, it is effectively a perceptron that is typically better than the Perceptron class that is intentionally keep simple.

Although all modeling techniques provide factory methods for convenience, due to the increased need for **rescaling of data**, these methods are more important for neural networks.

Object Methods:

```
object NeuralNet_2L extends Scaling:

    @param xy       the combined input and output matrix
    @param fname    the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to Optimizer.hp)
    @param f        the activation function family for layers 1->2 (input to output)
    @param col      the first designated response column (defaults to the last column)

    def apply (xy: MatrixD, fname: Array [String] = null,
               hparam: HyperParameter = Optimizer.hp, f: AFF = f_sigmoid)
               (col: Int = xy.dim2 - 1): NeuralNet_2L =

    @param x        the input/data m-by-n matrix
    @param y        the output/response m-by-ny matrix
    @param fname    the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to Optimizer.hp)
    @param f        the activation function family for layers 1->2 (input to output)

    def rescale (x: MatrixD, y: MatrixD, fname: Array [String] = null,
                 hparam: HyperParameter = Optimizer.hp, f: AFF = f_sigmoid): NeuralNet_2L =

    @param x        the input/data m-by-n matrix
    @param y_       the output/response m-vector
    @param fname    the feature/variable names (defaults to null)
    @param hparam   the hyper-parameters (defaults to Optimizer.hp)
    @param f        the activation function family for layers 1->2 (input to output)

    def perceptron (x: MatrixD, y_ : VectorD, fname: Array [String] = null,
                    hparam: HyperParameter = Optimizer.hp, f: AFF = f_sigmoid):
                    NeuralNet_2L =
```

10.6.7 Exercises

1. The dataset in `Example_Concrete` consists of 7 input variables and 3 output variables.

```
    // Input Variables (7) (component kg in one M^3 concrete):
    // 1. Cement
    // 2. Blast Furnace Slag
    // 3. Fly Ash
    // 4. Water
    // 5. Super Plasticizer (SP)
    // 6. Coarse Aggregate
    // 7. Fine Aggregate
    // Output Variables (3):
    // 1. SLUMP (cm)
    // 2. FLOW (cm)
    // 3. 28-day Compressive STRENGTH (Mpa)
```

   Create a `NeuralNet_2L` model to predict values for the three outputs y_0, y_1 and y_2. Compare with the results of using three `Perceptron`s.

2. Create a `NeuralNet_2L` model to predict values for the one output for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`.

3. Were the results in for the `AutoMPG` dataset the same for `Perceptron` and `NeuralNet_2L`? Please explain. In general, is a `NeuralNet_2L` equivalent to n_y `Perceptron`s?

4. Compare the convergence for the `AutoMPG` dataset of the following three optimization algorithm by plotting the drop in the loss function versus the number of epochs.

   (a) Gradient Descent (GD) from the `train` method in `Perceptron`.

   (b) Stochastic Gradient Descent (SGD) from the `train` method in `NeuralNet_2L` using `Optimizer_SGD`. This will require recompilation as `Optimizer_SGDM` is the default optimizer for `NeuralNet_2L`.

```
//  val opti = new Optimizer_SGD ()                  // SGD parameter optimizer
    val opti = new Optimizer_SGDM ()                 // SGDM parameter optimizer
```

   (c) Stochastic Gradient Descent with Momentum (SGDM) from the `train` method in `NeuralNet_2L`. using `Optimizer_SGDM`.

5. Explain how the ADAM optimizer works and redo the above exercise using Keras comparing GD, SGD, SGDM and Adam.

6. Draw a `NeuralNet_2L` with $n = 4$ input nodes and $n_y = 2$ output nodes. Label the eight edges with weights from the 4-by-2 weight matrix $B = [b_{jk}]$. Write the two model equations, one for y_0 and one for y_1. Combine these two equations into one vector equation for $\mathbf{y} = [y_0, y_1]$. Given column vector $\mathbf{x} = [1, x_1, x_2, x_3]$, express $\hat{\mathbf{y}} = \mathbf{f}(B^\intercal \mathbf{x})$ at the scalar level.

10.7 Three-Layer Neural Networks

The `NeuralNet_3L` class supports 3-layer (input, hidden and output) Neural Networks. The inputs into a Neural Net are given by the input vector $\mathbf{x}$, while the outputs are given by the output vector $\mathbf{y}$. Between these two layers is a single hidden layer, whose intermediate values will be denoted by the vector $\mathbf{z}$. Each input x_j is associated with an input node in the network, while each output y_k is associated with an output node in the network, as shown in Figure 10.9. The input layer consists of n input nodes, the hidden layer consists of n_z hidden nodes, and the output layer consists of n_y output nodes.

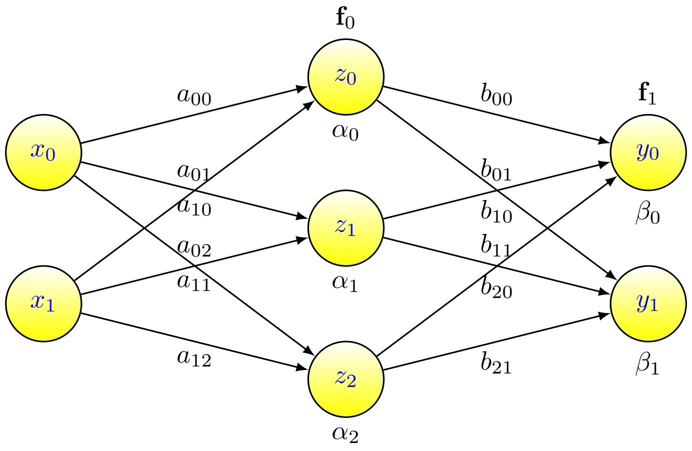


Figure 10.9: Three-Layer (input, hidden, output) Neural Network

There are two sets of edges. Edges in the first set connect each input node with each hidden node, i.e., there are nn_z such edges in the network. The parameters (or edge weights) for the first set of edges are maintained in matrix $A = [a_{jh}]_{n \times n_z}$. Edges in the second set connect each hidden node with each output node, i.e., there are $n_z n_y$ such edges in the network. The parameters (or edge weights) for the second set of edges are maintained in matrix $B = [b_{hk}]_{n_z \times n_y}$.

There are now two activation functions, f_0 and f_1. f_0 is applied at each node in the hidden layer, while f_1 plays this role for the output layer. Having two activation functions allows greater capability to approximate a variety of functional forms (for more information see [36, 81] on Universal Approximation Theorems).

10.7.1 Model Equation

The model equation for `NeuralNet_3L` can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{f}_0(A \cdot \mathbf{x})) + \boldsymbol{\epsilon} \;=\; \mathbf{f}_1(B^{\intercal} \mathbf{f}_0(A^{\intercal} \mathbf{x})) + \boldsymbol{\epsilon}} \tag{10.66}$$

The innermost matrix-vector product multiplies the transpose of the n-by-n_z matrix A by the n-by-1 vector $\mathbf{x}$, producing an n_z-by-1 vector, which is passed into the $\mathbf{f}_0$ vectorized activation function. The outermost matrix-vector product multiplies the transpose of the n_z-by-n_y matrix B by the n_z-by-1 vector results, producing an n_y-by-1 vector, which is passed into the $\mathbf{f}_1$ vectorized activation function.

Intercept/Bias

As before, one may include an intercept in the model (also referred to as bias) by having a special input node (say x_0) that always provides the value 1. A column of all one in an input matrix (see below) can achieve this. This approach could be carried forward to the hidden layer by including a special node (say z_0) that always produces the value 1 (referred to as the bias trick). In such case, the computation performed at node z_0 would be thrown away and replaced with 1, although a clever implementation could avoid the extra calculation. The alternative is to replace the uniform notion of parameters with two types of parameters, weights and biases. SCALATION supports this with the `NetParam` case class.

```scala
    @param w  the weight matrix
    @param b  the bias/intercept vector

    case class NetParam (var w: MatrixD, var b: VectorD = null):

        ...
        def dot (x: VectorD): VectorD = (w dot x) + b
```

Following this approach, there is no need for the special nodes and the `dot` product is re-defined to add the bias `b` to the regular matrix-vector `dot` product (`w dot x`). Note, the `NetParam` class defines several other methods as well. The vector version of the predict method in `NeuralNet_3L` uses this `dot` product to make predictions.

```scala
    @param v  the new input vector

    def predict (v: VectorD): VectorD =
        val yp = f1.f_ (bb(1) dot f.f_ (bb(0) dot v))   // scaled? prediction
        if itran == null then yp
        else itran (MatrixD (yp))(0)                    // back to original scale
    end predict
```

Note: $\mathbf{f}_0$ corresponds to `f.f_` in the code, while $\mathbf{f}_1$ corresponds to `f1.f_` in the code.

Including the bias vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ and splitting the computation of the predicted value $\hat{\mathbf{y}}$ into two steps yields,

$$\mathbf{z} = \mathbf{f}_0(A^\intercal \mathbf{x} + \boldsymbol{\alpha}) \tag{10.67}$$

$$\hat{\mathbf{y}} = \mathbf{f}_1(B^\intercal \mathbf{z} + \boldsymbol{\beta}) \tag{10.68}$$

10.7.2 Ridge Functions

While perceptrons allow the hyperplane to be bent, the fact that they are restricted to a single ridge function, limits their capabilities. Furthermore, as two layer neural networks are equivalent to having a perceptron for each output node, their capabilities are limited as well. The fundamental improvement begins with three layer (one hidden layer) neural networks, as they allow the superposition of ridge functions, as depicted in Figure 10.11. Somewhat like waves on an ocean, the ridge functions may collide to form flexible response surfaces.

One may see the superposition of ridge functions when the input is two dimensional, e.g., x_1 = `weight` and x_2 = `modelyear` for the AutoMPG dataset. Letting $\mathbf{f}_1$ be the identity function `id`, the model equation (or rather the corresponding prediction equation) reduces to

$$\hat{y} \; = \; B \cdot \mathbf{f}_0(A \cdot \mathbf{x}) \tag{10.69}$$

where matrix $A \in \mathbb{R}^{2\times 2}$ and matrix $B \in \mathbb{R}^{2\times 1}$. Expanding the outer dot product results in the following:

$$\hat{y} \; = \; b_{00} f_0(\mathbf{a}_{:0} \cdot \mathbf{x}) \, + \, b_{10} f_0(\mathbf{a}_{:1} \cdot \mathbf{x}) \tag{10.70}$$

Including the biases produces

$$\hat{y} \; = \; b_{00} f_0(\mathbf{a}_{:0} \cdot \mathbf{x} + \alpha_0) \, + \, b_{10} f_0(\mathbf{a}_{:1} \cdot \mathbf{x} + \alpha_1) + \beta_0 \tag{10.71}$$

The corresponding network diagram is shown in Figure 10.10.

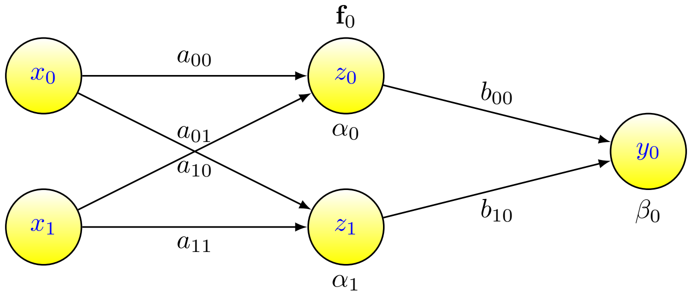


Figure 10.10: A Simple Three-Layer (input, hidden, output) Neural Network

Each hidden node brings in a ridge function and since the second activation is the identity function, the output $\hat{y}$ is a linear combination of the two (e.g., sigmoid) ridge functions.

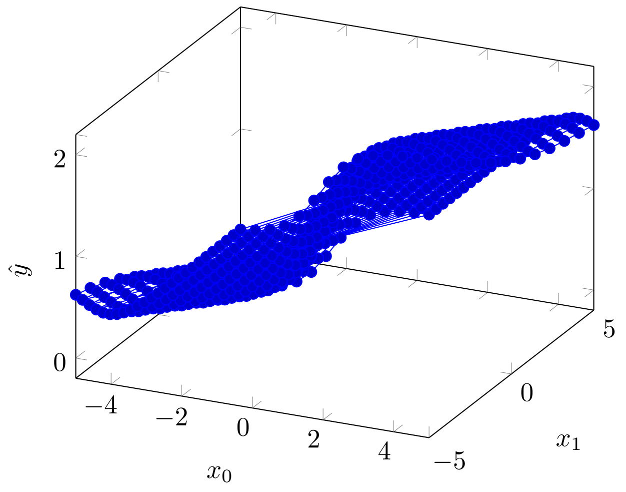


Figure 10.11: Ridge Functions: $\hat{y} =$ Superposition of Two Ridge Functions

The equation (not optimized) used in Figure 10.11 is

$$\hat{y} \;=\; \frac{1}{1+e^{-(2x_0+x_1+.5)}} \;+\; \frac{1}{1+e^{-(x_0-x_1+.5)}}$$

See the subsection on Response Surface and the exercises for an optimized equation.

Note, the use of identity `id` activation function for the last layer means that the final layer preforms a linear transformation, so that rescaling of the outputs $\mathbf{y}$ may be avoided.

10.7.3 Training

Given a training dataset made up of an m-by-n input matrix X and an m-by-n_y output matrix Y, training consists of making a prediction $\hat{Y}$,

$$\boxed{\hat{Y} \;=\; \mathbf{f}_1(\mathbf{f}_0(XA)B)} \tag{10.72}$$

and determining the error in prediction $E = Y - \hat{Y}$ with the goal of minimizing the error.

$$\min_{A,B} \|Y - \mathbf{f}_1(\mathbf{f}_0(XA)B)\|_F \tag{10.73}$$

Training involves an iterative procedure (e.g., stochastic gradient descent) that adjusts parameter values (for weights and biases) to minimize a loss function such as *sse* or rather half *sse* (or $\mathcal{L}$). Before the main loop, random parameter values (for weights and biases) need to be assigned to `NetParam` A and `NetParam` B. Roughly as outlined in section 3 of [157], the training can be broken into four steps:

1. Compute predicted values for output $\hat{\mathbf{y}}$ and compare with actual values $\mathbf{y}$ to determine the error $\mathbf{y} - \hat{\mathbf{y}}$.

2. Back propagate the adjusted error to determine the amount of correction needed at the output layer. Record this as vector $\boldsymbol{\delta}^1$.

3. Back propagate the correction to the hidden layer and determine the amount of correction needed at the hidden layer. Record this as vector $\boldsymbol{\delta}^0$.

4. Use the delta vectors, $\boldsymbol{\delta}^1$ and $\boldsymbol{\delta}^0$, to makes updates to `NetParam` A and `NetParam` B, i.e., the weights and biases.

10.7.4 Optimization

In this subsection, the basic elements of the back-propagation algorithm are presented. In particular, we now go over the four steps outlined above in more detail. Biases are ignored for simplicity, so the A and B `NetParams` are treated as weight matrices. In the code, the same logic includes the biases (so nothing is lost, see exercises). Note that $\mathcal{L}$ denotes a loss function, while h is an index into the hidden layer.

1. Compute predicted values: Based on the randomly assigned weights to the A and B matrices, predicted outputs $\hat{\mathbf{y}}$ are calculated. First values for the hidden layer $\mathbf{z}$ are calculated, where the value for hidden node h, z_h, is given by

$$z_h \;=\; f_0(\mathbf{a}_{:h} \cdot \mathbf{x}) \qquad \text{for } h = 0, \ldots, n_z - 1$$

where f_0 is the first activation function (e.g., sigmoid), $\mathbf{a}_{:h}$ is column-h of the A weight matrix, and $\mathbf{x}$ is an input vector for a training sample/instance (row in the data matrix). Typically, several samples (referred to as a *mini-batch*) are used in each step. Next, the values computed at the hidden layer are used to produce predicted outputs $\hat{\mathbf{y}}$, where the value for output node k, $\hat{y}_k$, is given by

$$\hat{y}_k \;=\; f_1(\mathbf{b}_{:k} \cdot \mathbf{z}) \qquad \text{for } k = 0, \ldots, n_y - 1$$

where the second activation function f_1 may be the same as (or different from) the one used in the hidden layer and $\mathbf{b}_{:k}$ is column-k of the B weight matrix. Now the difference between the actual and predicted output can be calculated by simply subtracting the two vectors, or element-wise, the error for the k^{th} output, ϵ_k, is given by

$$\epsilon_k \;=\; y_k - \hat{y}_k \qquad \text{for } k = 0, \ldots, n_y - 1$$

Obviously, for subsequent iterations, the updated/corrected weights rather than the initial random weights are used.

2. Back propagate from output layer: Given the computed error vector $\boldsymbol{\epsilon}$, the delta/correction vector $\boldsymbol{\delta}^1$ for the output layer may be calculated, where for output node k, δ^1_k is given by

$$\boxed{\delta^1_k \;=\; [-\epsilon_k]\, f_1'(\mathbf{b}_{:k} \cdot \mathbf{z})} \qquad \text{for } k = 0, \ldots, n_y - 1 \tag{10.74}$$

where f_1' is the derivative of the activation function (e.g., for sigmoid, $f'(u) = f(u)[1 - f(u)]$). The partial derivative of the loss function $\mathcal{L}$ with respect to the weight connecting hidden node h with output node k, b_{hk}, is given by

$$\frac{\partial \mathcal{L}}{\partial b_{hk}} \;=\; z_h \delta^1_k \tag{10.75}$$

3. Back propagate from hidden layer: Given the delta/correction vector $\boldsymbol{\delta}^1$ from the output layer, the delta vector for the hidden layer $\boldsymbol{\delta}^0$ may be calculated, where for hidden node h, δ^0_h is given by

$$\boxed{\delta^0_h \;=\; [\mathbf{b}_h \cdot \boldsymbol{\delta}^1]\, f_0'(\mathbf{a}_{:h} \cdot \mathbf{x})} \qquad \text{for } h = 0, \ldots, n_z - 1 \tag{10.76}$$

This equation is parallel to the one given for δ^1_k in that an error-like factor multiplies the derivative of the activation function. In this case, the error-like factor is the weighted combination of the δ^1_k for output nodes connected to hidden node h times row-h of weight matrix B. The weighted combination is computed using the dot product.

$$\mathbf{b}_h \cdot \boldsymbol{\delta}^1 \;=\; \sum_{k=0}^{n_y-1} b_{hk}\, \delta^1_k$$

The partial derivative of $\mathcal{L}$ with respect to the weight connecting input node j with hidden node h, a_{jh}, is given by

$$\frac{\partial \mathcal{L}}{\partial a_{jh}} \;=\; x_j \delta^0_h \tag{10.77}$$

4. Update weights: The weight matrices A and B, connecting input to hidden and hidden to output layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate η which moderates the adjustments to the weights.

$$b_{hk} \; -= \; z_h \, \delta^1_k \, \eta \tag{10.78}$$

$$a_{jh} \; -= \; x_j \, \delta^0_h \, \eta \tag{10.79}$$

Figure 10.12 shows the forward ($\rightarrow$) and backward ($\leftarrow$) propagation through the Neural Network and depicts the role of δ in updating the parameters from the perspective of hidden node z_h.

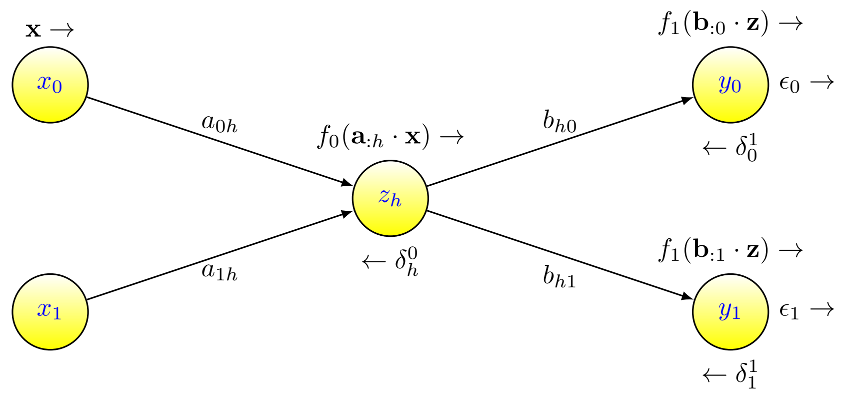


Figure 10.12: View for Hidden Node z_h

To improve the stability of the algorithm, weights are adjusted based on accumulated corrections over a mini-batch of instances, where a mini-batch is a sub-sample of the training dataset and may be up to the size the of the entire training dataset (for $i = 0, \ldots, m-1$). Once training has occurred over the current mini-batch including, at the end, updates to the A and B estimates, the current epoch is said to be complete. Correspondingly, the above equations may be vectorized/matrixized so that calculations are performed over many instances in a mini-batch using matrix operations. Each outer iteration (epoch) typically should improve the A and B estimates. Simple stopping rules include specifying a fixed number of iterations or breaking out of the outer loop when the loss function $\mathcal{L}$ fails to decrease for q iterations.

10.7.5 Matrix Version

Given a training dataset consisting of an input/data matrix $X \in \mathbb{R}^{m \times n}$ and an output/response matrix $Y \in \mathbb{R}^{m \times n_y}$, the optimization equations may be re-written in matrix form as shown below.

The gradient descent optimizer used by the `train` method has one main loop, while the stochastic gradient descent optimizer used by the `train` in `Optimizer_SGD` has two main loops. The outer loop iterates over *epochs* which serve to improve the parameters/weights with each iteration. If the fit does not improve in several epochs, the algorithm likely should break out of this loop.

The four boxed equations from the previous section become seven due to the extra layer. The optimizers compute predicted outputs and take differences between the actual/target values and these predicted values to compute an error matrix. These results are then used to compute delta matrices that form the basis for updating the parameter/weight matrices $A \in \mathbb{R}^{n \times n_z}$ and $B \in \mathbb{R}^{n_z \times n_y}$.

1. The hidden (latent) values for all m instances and all n_z hidden nodes are computed by applying the first matrixized activation function $\mathbf{f}_0$ to the matrix product XA to produce the **latent feature matrix** $Z \in \mathbb{R}^{m \times n_z}$.

$$\boxed{Z \;=\; \mathbf{f}_0(XA)} \tag{10.80}$$

   The **predicted output matrix** $\hat{Y} \in \mathbb{R}^{m \times n_y}$ is similarly computed by applying the second matrixized activation function $\mathbf{f}_1$ to the matrix product ZB.

$$\boxed{\hat{Y} \;=\; \mathbf{f}_1(ZB)} \tag{10.81}$$

2. The **negative of the error matrix** $E \in \mathbb{R}^{m \times n_y}$ is just the difference between the predicted and actual/target values.

$$\boxed{E \;=\; \hat{Y} \;-\; Y} \tag{10.82}$$

3. This information is sufficient to calculate delta matrices: Δ^1 for adjusting B and Δ^0 for adjusting A. The **output-layer delta matrix** $\Delta^1 \in \mathbb{R}^{m \times n_y}$ is the element-wise matrix (Hadamard) product of E and $\mathbf{f}_1'(ZB)$.

$$\boxed{\Delta^1 \;=\; E \odot \mathbf{f}_1'(ZB)} \tag{10.83}$$

   The **hidden-layer delta matrix** $\Delta^0 \in \mathbb{R}^{m \times n_z}$ is the element-wise matrix (Hadamard) product of $\Delta^1 B^\intercal$ and $\mathbf{f}_0'(XA)$.

$$\boxed{\Delta^0 \;=\; [\Delta^1 B^\intercal] \odot \mathbf{f}_0'(XA)} \tag{10.84}$$

4. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate η) for updating the parameter/weight matrices, A and B.

$$\boxed{B \;-\!=\; Z^\intercal \Delta^1 \eta} \tag{10.85}$$

$$\boxed{A \;-\!=\; X^\intercal \Delta^0 \eta} \tag{10.86}$$

10.7.6 `train` Method

The corresponding ScalaTion code for the `train` method is show below.

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output matrix

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
        val epochs = opti.optimize3 (x_, y_, bb, eta, Array (f, f1))  // optimize param bb
        println (s"ending epoch = $epochs")
        estat.tally (epochs._2)
    end train
```

10.7.7 Stochastic Gradient Descent Algorithm

The stochastic gradient descent algorithm for `NeuralNet_3L` places the above matrix equations into a loop to iteratively update the weight matrices and bias vectors (unified via the `NetParam` class). The work is done by the `optimize3` method, e.g., in the `Optimizer_SGD` class. The code below initializes values for the `optimize3` method. These include a permutation generator that is used for producing randomized mini-batches, convenient aliases for the two sets of net-parameters as well as for the two activation functions, three values from hyper-parameters (`bSize`, `maxEpochs`, and `upLimit`), the initial learning rate, and the number of batches.

```
    @param x    the m-by-n input matrix (training data consisting of m input vectors)
    @param y    the m-by-ny output matrix (training data consisting of m output vectors)
    @param bb   the array of parameters (weights & biases) between two adjacent layers
    @param eta  the initial learning/convergence rate
    @param ff   the array of activation function family for every two adjacent layers

    def optimize3 (x: MatrixD, y: MatrixD,
                   bb: NetParams, eta: Double, ff: Array [AFF]): (Double, Int) =
        val permGen   = permGenerator (x.dim)                   // permutation vector gen
        val (a, b)    = (bb(0), bb(1))                          // two sets of net-parameters
        val (f, f1)   = (ff(0), ff(1))                          // two activation functions
        val bSize     = min (hp("bSize").toInt, x.dim)          // batch size
        val maxEpochs = hp("maxEpochs").toInt                   // maximum number of epochs
        val upLimit   = hp("upLimit").toInt                     // limit on increasing lose
        var η         = eta                                     // set initial learning rate
        val nB        = x.dim / bSize                           // the number of batches
```

The main part of the `optimize3` method is a nested loop: The outer loop iterates up to `maxEpochs`, but may exist early depending on what `stopWhen` returns. The inner loop iterates through the mini-batches (`ib` is the i^{th} mini-batch) The learning rate η is periodically adjusted.

```
        var sse_best_   = -0.0
        var (go, epoch) = (true, 1)
        cfor (go && epoch <= maxEpochs, epoch += 1) {           // iterate over each epoch
            val batches = permGen.igen.chop (nB)                // permute indices & chop

            for ib <- batches do
                val ab = updateWeight (x(ib), y(ib))            // update parameters a & b
                a -= ab._1; b -= ab._2
            end for

            val sse = (y - b * f1.fM (f.fM (a * x))).normFSq
            collectLoss (sse)                                  // collect the loss per epoch
            val (b_best, sse_best) = stopWhen (Array (a, b), sse)
            if b_best != null then
                a.set (b_best(0))
                b.set (b_best(1))
                sse_best_ = sse_best                           // save best in sse_best_
                go = false
            else
                if epoch % ADJUST_PERIOD == 0 then η *= ADJUST_FACTOR
            end if
        } // cfor
```

The updateWeight method is used to update the parameters (weights and biases). It performs both forward and back propagation on the mini-batch passed in.

```
        inline def updateWeight (x: MatrixD, y: MatrixD): (NetParam, NetParam) =
            val α  = η / x.dim                                 // eta over batch size
            var z  = f.fM (a * x)                              // Z  = f(XA)
            var yp = f1.fM (b * z)                             // Yp = f(ZB)
            var ϵ  = yp - y                                    // negative of error matrix
            val δ1 = f1.dM (yp) ⊙ ϵ                            // delta matrix for y
            val δ0 = f.dM (z) ⊙ (δ1 * b.w.𝒯)                   // delta matrix for z

            (NetParam (x.𝒯 * δ0 * α, δ0.mean * η),             // change a params
             NetParam (z.𝒯 * δ1 * α, δ1.mean * η))             // change b params
        end updateWeight
```

The end of the optimize3 method returns the best value for the loss function and the number of epochs.

```
        if go then ((y - b * f1.fM (f.fM (a * x))).normFSq, maxEpochs)   // ret. sse, epochs
        else       (sse_best_, epoch - upLimit)
    end optimize3
```

For stochastic gradient descent in the Optimizer_SGD class, the inner loop divides the training dataset into nB *mini-batches*. A batch is a randomly selected group/batch of rows. Each batch (ib) is passed to the updateWeight (x(ib), y(ib)) method that updates the A and B parameter/weight matrices.

Neural networks may be used for prediction/regression as well as classification problems. For prediction/regression, the number of output nodes would corresponding to the number of responses. For example, in the ExampleConrete example there are three response columns, requiring three instances of Regression or one instance of NeuralNet_3L. Three separate NeuralNet_3L instances each with one output node could be used as well. Since some activation functions have limited ranges, it is common practice for these types of problems to let the activation function in the last layer be identity id. If this is not done, response columns need to be re-scaled based on the training dataset. Since the testing dataset may have values outside this range, this approach may not be ideal.

Softmax Activation Function

For classification problems, it is common to have an output node for each response value for the categorical variable, e.g., "no", "yes" would have y_0 and y_1, while "red", "green", "blue" would have y_0, y_1 and y_2. The softmax activation function is a common choice for the last layer for classification problems.

$$f_i(\mathbf{u}) = \frac{e^{u_i}}{\mathbf{1} \cdot e^{\mathbf{u}}} \qquad \text{for } i = 0, \dots, n-1$$

The values produced by the softmax activation function can be thought of as giving a probability score to the particular class label, e.g., the image shows a “cat” vs. “dog”.

An alternative for binary classification ($k = 2$) is to have one output node and use sigmoid activation for the last layer.

10.7.8 Example Error Calculation Problem

Draw a 3-layer (input, hidden and output) Neural Network (with sigmoid activation), where the number of nodes per layer are $n = 2, n_z = 2$ and $n_y = 1$.

Input to Hidden Layer Parameters

Initialize bias vector $\boldsymbol{\alpha}$ to $[.1, .1]$ and weight matrix A (n-by-n_z) to

$$\begin{bmatrix} .1 & .2 \\ .3 & .4 \end{bmatrix}$$

Hidden to Output Layer Parameters

Initialize bias vector $\boldsymbol{\beta}$ to $[.1]$ and weight matrix B (n_z-by-n_y) to

$$\begin{bmatrix} .5 \\ .6 \end{bmatrix}$$

Compute the Error for the First Iteration

Let $\mathbf{x} = [x_0, x_1] = [2, 1]$ and $y_0 = .8$, and then compute the error $\epsilon_0 = y_0 - \hat{y_0}$, by feeding the values from vector $\mathbf{x}$ forward. First compute values at the hidden layer for $\mathbf{z}$.

$$\begin{aligned}
z_h &= f_0(\mathbf{a}_{:h} \cdot \mathbf{x} + \alpha_h) \\
z_0 &= f_0(\mathbf{a}_{:0} \cdot \mathbf{x} + \alpha_0) \\
z_0 &= f_0([0.1, 0.3] \cdot [2.0, 1.0] + 0.1) \\
z_0 &= f_0(0.6) \;=\; 0.645656 \\
z_1 &= f_0(\mathbf{a}_{:1} \cdot \mathbf{x} + \alpha_1) \\
z_1 &= f_0([0.2, 0.4] \cdot [2.0, 1.0] + 0.1) \\
z_1 &= f_0(0.9) \;=\; 0.710950
\end{aligned}$$

One may compute the values for sigmoid activation function as follows:

```
    println (ActivationFun.sigmoid_ (VectorD (0.6, 0.9)))
```

Then compute predicted values at the output layer for $\hat{\mathbf{y}}$.

$$\begin{aligned}
\hat{y_k} &= f_1(\mathbf{b}_{:k} \cdot \mathbf{z} + \beta_k) \\
\hat{y_0} &= f_1(\mathbf{b}_{:0} \cdot \mathbf{z} + \beta_0) \\
\hat{y_0} &= f_1([0.5, 0.6] \cdot [0.645656, 0.71095] + 0.1) \\
\hat{y_0} &= f_1(0.849398) \;=\; 0.7004408
\end{aligned}$$

Therefore, the error is

$$\begin{aligned}
&\epsilon_0 = y_0 - \hat{y_0} \\
&\epsilon_0 = 0.8 - 0.7004408 \;=\; 0.0995592
\end{aligned}$$

10.7.9 Response Surface

The response surface for a 3-Layer Neural Network on AutoMPG based on the best combination of two features, `weight` and `modelyear`, can be shown in a 3D plot. The surface can be calculated from how input matrix `x` is rescaled and from the model parameter values `bb` found by the optimizer. Recall that some activation functions have an active range outside of which the neural network may get stuck. In such cases, the each column of the input/data matrix needs to be rescaled into the active range of the activation function. This is done by the `rescaleX` method.

```
    rescaleX: from (VectorD(1.61300, 70.0000), VectorD(5.14000, 82.0000)) to (-2.0, 2.0)

    parameter  bb  = Array (b.w = MatrixD(-2.12262, -0.743867,
                                          -0.200314, 1.62988)
                            b.b = VectorD(-2.15785, -1.65227)
                            b.w = MatrixD(15.7250,
                                          13.1971)
                            b.b = VectorD(13.4702))
```

From these a prediction formula may be created that closely approximates the `predict` method.

$$\begin{aligned}
\mathbf{x} &= [(4/3.537)(x_1 - 1.61300) - 2, ((4/12)(x_2 - 70) - 2] \\
\hat{y} &= 15.7250\,\text{sigmoid}\,([-2.12262, -0.200314] \cdot \mathbf{x} - 2.15785) + \\
&\quad 13.1971\,\text{sigmoid}\,([-0.743867, 1.62988] \cdot \mathbf{x} - 1.65227) + \\
&\quad 13.4702
\end{aligned}$$

See the exercises.

10.7.10 `NeuralNet_3L` Class

Class Methods:

```
    @param x        the m-by-n input/data matrix (full/training data having m vectors)
    @param y        the m-by-ny output/response matrix (full/training data having m vectors)
    @param fname_   the feature/variable names (defaults to null)
```

```scala
    @param nz       the number of nodes in hidden layer (-1 => use default formula)
    @param hparam   the hyper-parameters for the model/network (defaults to Optimizer.hp)
    @param f        the activation function family for layers 1->2 (input to output)
    @param f1       the activation function family for layers 2->3 (hidden to output)
    @param itran    the inverse transformation returns response matrix to original scale

    class NeuralNet_3L (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                        private var nz: Int = -1, hparam: HyperParameter = Optimizer.hp,
                        f: AFF = f_sigmoid, f1: AFF = f_id,
                        val itran: FunctionM2M = null)
          extends PredictorMV (x, y, fname_, hparam)
             with Fit (dfm = x.dim2, df = x.dim - x.dim2):         // under-estimates df

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    override def train2 (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD) =
    override def makePlots (yy_ : MatrixD, yp: MatrixD): Unit =
    def predict (v: VectorD): VectorD =
    override def predict (v: MatrixD = x): MatrixD =
    def buildModel (x_cols: MatrixD): NeuralNet_3L =
    def summary2 (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                  b_ : MatrixD = parameter): String =
```

10.7.11 Exercises

1. **Delta Vectors**: For the example error calculation problem given in this section, calculate the $\boldsymbol{\delta}^1 = [\delta^1_0]$ vector using the following formula.

$$\delta^1_k \;=\; -\,\epsilon_k\, f_1'(\mathbf{b}_{:k} \cdot \mathbf{z})$$

Rework the problem using separate weights ($\mathbf{b}_{:k}$) and biases (β_k) (not lumped together as parameters). to compute δ^1_k.

$$\delta^1_k \;=\; -\,\epsilon_k\, f_1'(\mathbf{b}_{:k} \cdot \mathbf{z} + \beta_k)$$

Calculate the $\boldsymbol{\delta}^0 = [\delta^0_0, \delta^0_1]$ vector using the analogous reworked formula having separate weights ($\mathbf{a}_{:h}$) and biases (α_h) to compute δ^0_h.

$$\delta^0_h \;=\; [\mathbf{b}_h \cdot \boldsymbol{\delta}^1]\, f_0'(\mathbf{a}_{:h} \cdot \mathbf{x} + \alpha_h)$$

2. **Parameter Update Equations**: Use the $\boldsymbol{\delta}_1$ vector to update weight matrix B, i.e., for each row h,

$$\mathbf{b}_h \;-\!=\; z_h \boldsymbol{\delta}^1 \eta$$

and update the bias vector $\boldsymbol{\beta}$ as follows:

$$\boldsymbol{\beta} \ -= \ \boldsymbol{\delta}^1\eta$$

Use the $\boldsymbol{\delta}_0$ vector to update weight matrix A, i.e., for each row j,

$$\mathbf{a}_j \ -= \ x_j\boldsymbol{\delta}^0\eta$$

and update the bias vector $\boldsymbol{\alpha}$ as follows:

$$\boldsymbol{\alpha} \ -= \ \boldsymbol{\delta}^0\eta$$

3. Derive the equation for the partial derivative of the loss function $\mathcal{L}$ w.r.t. b_{hk},

$$\frac{\partial\mathcal{L}}{\partial b_{hk}} \ = \ z_h\delta_k^1$$

by defining pre-activation value $v_k = \mathbf{b}_{:k} \cdot \mathbf{z}$ and applying the following chain rule:

$$\frac{\partial\mathcal{L}}{\partial b_{hk}} \ = \ \frac{\partial\mathcal{L}}{\partial v_k}\frac{\partial v_k}{\partial b_{hk}}$$

4. Explain the formulations for the two delta matrices.

$$\begin{aligned}\Delta^1 &= E \odot \mathbf{f}_1'(ZB) \\ \Delta^0 &= [\Delta^1 B^{\intercal}] \odot \mathbf{f}_0'(XA)\end{aligned}$$

5. The dataset in `Example_Concrete` consists of 7 input variables and 3 output variables. See the `NeuralNet_2L` section for details. Create a `NeuralNet_3L` model to predict values for the three outputs y_0, y_1 and y_2. Compare with the results of using a `NeuralNet_2L` model.

6. Create a `NeuralNet_3L` model to predict values for the one output for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`, (c) `NeuralNet_2L`.

7. For the `AutoMPG` dataset let the **activation function** for the last layer be `id`, so that rescaling of the output/response `y` is not needed. Then try all of ScalaTion's activation functions and compare the QoF for (a) in-sample testing, (b) validation, i.e., train-n-test split (TnT), and (c) cross-validation.

8. Use the formula for $\hat{y}$ given in the Response Surface subsection to plot $\hat{y}$ vs. x_1 and x_2.

9. Conduct a literature study on the effectiveness and efficiency of various training algorithms for neural networks, for example see [31].

10.8 Multi-Hidden Layer Neural Networks

The `NeuralNet_XL` class supports basic x-layer (input, {hidden} and output) Neural Networks. For example a four layer neural network (see Figure 10.13) with have four layers of nodes with (one input layer numbered 0, two hidden layers numbered 1 and 2, and one output layer numbered 3). Note, since the input layer's purpose is just to funnel the input into the model, it is also common to refer to such a neural network as a three layer network. This has the advantage that the number of layers now corresponds to the number parameter/weight matrices.

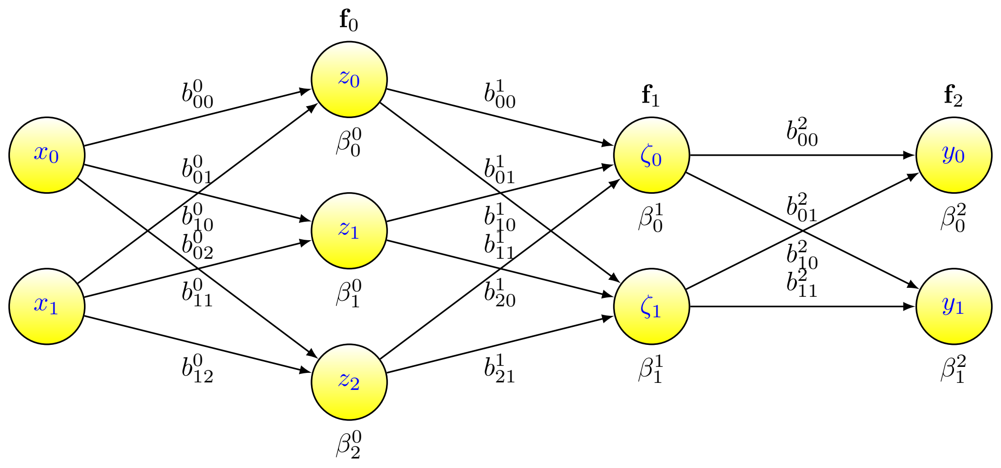


Figure 10.13: Four-Layer (input, hidden ($l = 0$), hidden ($l = 1$), output ($l = 2$)) Neural Network

In ScalaTion, the number of active layers is denoted by `nl` (which in this case equals 3 with $l = 0, 1, 2$). Since arrays of matrices are used in the ScalaTion code, multiple layers of hidden nodes are supported. The matrix $B_l = [b^l_{jk}]$ maintains the weights for layer l, while $\boldsymbol{\beta}_l = [\beta^l_k]$ maintains the biases for layer l.

In particular, parameter `b` which holds the weights and biases for all layers is of type `NetParams` where

```
    type NetParams = Array [NetParam]
```

For simplicity, in the model equation below rolls the biases into the weights using `NetParam`.

10.8.1 Model Equation

The equations for `NeuralNet_XL` are the same as those used for `NeuralNet_3L`, except that the calculations are repeated layer by layer in a forward direction for prediction. The model equation for a four layer `NeuralNet_XL` can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f}_2(B_2 \cdot \mathbf{f}_1(B_1 \cdot \mathbf{f}_0(B_0 \cdot \mathbf{x}))) + \boldsymbol{\epsilon}} \tag{10.87}$$

where B_l is the `NetParam` (weight matrix and bias vector) connecting layer l to layer $l + 1$ and $\mathbf{f}_l$ is the vectorized activation function at layer $l + 1$.

10.8.2 Training

As before, the training dataset consists of an m-by-n input matrix X and an m-by-n_y output matrix Y. During training, the predicted values $\hat{Y}$ are compared to actual/target values Y,

$$\boxed{\hat{Y} \;=\; \mathbf{f}_2(\mathbf{f}_1(\mathbf{f}_0(XB_0)B_1)B_2)} \tag{10.88}$$

to compute an error matrix $E = Y - \hat{Y}$, to be minimized.

$$\min_B \|Y - \mathbf{f}_2(\mathbf{f}_1(\mathbf{f}_0(XB_0)B_1)B_2)\|_F \tag{10.89}$$

Corrections based on these errors are propagated backward through the network to improve the parameter estimates (weights and biases) layer by layer.

10.8.3 Optimization

The seven boxed equations from the previous section become six due to unification of the last two. As before, the optimizers compute a predicted output matrix and then take differences between the actual/target values and these predicted values to compute an error matrix. These computed matrices are then used to compute delta matrices that form the basis for updating the weight matrices. Again for simplicity, biases are ignore in the equations below, but are taken care of in the code through the `NetParam` abstraction. See the exercises for details.

1. The values are feed forward through the network, layer by layer. For layer l, these values are stored in matrix Z_l. The first layer is the input, so $Z_0 \;=\; X$. For the rest of the layers, Z_{l+1} equals the result of activation function $\mathbf{f}_l$ being applied to the product of the previous layer's Z_l matrix times its parameter matrix B_l.

$$\boxed{Z_0 \;=\; X} \tag{10.90}$$

   For each layer l in the forward direction:

$$\boxed{Z_{l+1} \;=\; \mathbf{f}_l(Z_l B_l)} \tag{10.91}$$

2. The negative of the error matrix E is just the difference between the predicted and actual/target values, where $\hat{Y} = Z_{nl}$.

$$\boxed{E \;=\; \hat{Y} - Y} \tag{10.92}$$

3. This information is sufficient to calculate delta matrices Δ^l. For the last layer:

$$\boxed{\Delta^{nl-1} \;=\; \mathbf{f}'_{nl-1}(Z_{nl-1}B_{nl-1}) \odot E} \tag{10.93}$$

   For the rest of layers in the backward direction with l being decremented:

$$\boxed{\Delta^l \;=\; \mathbf{f}'_l(Z_l B_l) \odot (\Delta^{l+1} B^{\intercal}_{l+1})} \tag{10.94}$$

4. As mentioned, the delta matrices form the basis (a matrix transpose × delta × the learning rate η) for updating the parameter/weight matrices, B_l for each layer l.

$$\boxed{B_l \;\; -= \;\; Z_l^{\intercal} \Delta^l \eta} \tag{10.95}$$

The implementation of the `train` encodes these equation and uses gradient descent to improve the parameters B_l over several `epochs`, terminating early when the objective/cost function fails to improve.

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output matrix

    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
        if flayer >= 0 then opti.freeze (flayer)              // optimizer to freeze flayer
        val epochs = opti.optimize (x_, y_, bb, eta, f)       // optimize parameters bb
        println (s"ending epoch = $epochs")
        estat.tally (epochs._2)
    end train
```

The code for the `optimize` method may be found in the `Optiimzer_SGD` and `Optimizer_SGDM` classes. Again, the `train2` method includes limited auto-tuning of hyper-parameters.

10.8.4 Number of Nodes in Hidden Layers

The array `nz` gives the number of nodes for each of the hidden layers. If the user does not specify the number of nodes for each hidden layer, then based on the number of input nodes `n` and number of output nodes `ny`, defaults are utilized according to one of two rules.

```
    if nz == null then nz = compute_nz        // Rule default # nodes for hidden layers
```

If the array is `null`, then default numbers for the hidden layers are utilized. The default rule sets the first hidden layer to `2 * n + 1` and divides this by the layer number for subsequent layers.

The number of nodes in each layer is currently used as a very rough estimate of the Degrees of Freedom for the neural network. Also, the Degrees of Freedom is only considered for the first output variable (y_0). There is ongoing research to characterize Generalized Degrees of Freedom for neural networks [61]. As this work matures, plans are for ScalaTion to include them.

10.8.5 Avoidance of Overfitting

If efforts are not made to avoid over-fitting, NeuralNet_XL models are likely to suffer from this problem. When $R^2, \bar{R}^2$ are much higher than R^2_{cv} there may be two causes. One is that the tuning of hyper-parameters on the full dataset, is different from the tuning on training set slices. Two is that the optimization algorithm finished with the signal and continued on to fit the noise. The simplest way to reduce over-fitting is to make the optimizer quit before focusing its efforts on the noise. If only we knew. A crude way to this is to reduce the maximum number of epochs. A better way to do this is to split a training set into two parts, one for training (iteratively adjusting the parameters) and other for the stopping rule. The stopping rule would compute the objective/loss function only using the validation data. Drop-out may be used in which a node temporarily taken out of the model. Regularization of the parameters, as was done for Ridge and Lasso Regression, may help as well.

10.8.6 Deep Learning

When the number of hidden layers are increased beyond the base levels of one or two, the learning may be described as deep. In addition, special types of layers or units may be included such as convolutional or recurrent.

There are Universal Approximation Theorems that indicate that Neural Networks with one hidden layer (with arbitrary width) can approximate a board class of functions mapping inputs to outputs. However, other theorems show the using more hidden layers allows the width to be reduced. This is the realm of deep learning that has showed success in many applications areas, including automatic speech recognition, image classification, computer vision and natural language processing.

10.8.7 `NeuralNet_XL` Class

Class Methods:

```
    @param x        the m-by-n input/data matrix (full/training data having m input vectors)
    @param y        the m-by-ny output/response matrix (full/training data having m vectors)
    @param fname_   the feature/variable names (defaults to null)
    @param nz       the number of nodes in each hidden layer, e.g.,
                    Array (9, 8) => 2 hidden of sizes 9 and 8 (null => use default formula)
    @param hparam   the hyper-parameters for the model/network (defaults to Optimizer.hp)
    @param f        the array of activation function families between every pair of layers
    @param itran    the inverse transformation function returns response to original scale

    class NeuralNet_XL (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                        private var nz: Array [Int] = null,
                        hparam: HyperParameter = Optimizer.hp,
                        f: Array [AFF] = Array (f_sigmoid, f_sigmoid, f_id),
                        val itran: FunctionM2M = null)
          extends PredictorMV (x, y, fname_, hparam)
             with Fit (dfm = x.dim2, df = x.dim - x.dim2):      // under-estimate of df

    def getNetParam (layer: Int = 1): NetParam = bb(layer)
    def freeze (layer: Int): Unit = flayer = layer
    def compute_nz: Array [Int] =
    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    override def train2 (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD) =
    override def makePlots (yy_ : MatrixD, yp: MatrixD): Unit =
    def predict (v: VectorD): VectorD =
    override def predict (v: MatrixD = x): MatrixD =
    def buildModel (x_cols: MatrixD): NeuralNet_XL =
    def summary2 (x_ : MatrixD = getX, fname_ : Array [String] = fname,
                  b_ : MatrixD = null): String =
```

10.8.8 Exercises

1. Examine the implementation of the `train` method and the `NetParam` case class, where the net parameter `b` has two parts: the weight matrix `b.w` and the bias vector `b.b`. Show how the biases affect the calculation of prediction matrix $\hat{Y} = Z_{nl}$ in the feed forward process.

2. Examine the implementation of the `train` method and the `NetParam` case class and show how the biases affect the update of the weights `b.w` in the back-propagation process.

3. Examine the implementation of the `train` method and the `NetParam` case class and show how the biases `b.b` are updated in the back-propagation process.

4. The dataset in `Example_Concrete` consists of 7 input variables and 3 output variables. See the `NeuralNet_2L` section for details. Create a `NeuralNet_XL` model with four layers to predict values for the three outputs y_0, y_1 and y_2. Compare with the results of using a `NeuralNet_3L` model.

5. Create a `NeuralNet_XL` model with four layers to predict values for the one output for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`, (c) `NeuralNet_2L`, (d) `NeuralNet_3L`.

6. **Tuning the Hyper-Parameters**: The learning rate η (`eta` in the code) needs frequent tuning. ScalaTion as with most packages has limited auto-tuning of the learning rate. Tune the other hyper-parameters for the `AutoMPG` dataset.

```scala
object Optimizer:

    /** hyper-parameters for tuning the optimization algorithms - user tuning
     */
    val hp = new HyperParameter
    hp += ("eta", 0.1, 0.1)                     // learning/convergence rate
    hp += ("bSize", 20, 20)                     // mini-batch size, common range 10 to 40
    hp += ("maxEpochs", 400, 400)               // maximum number of epochs/iterations
    hp += ("lambda", 0.01, 0.01)                // regularization/shrinkage hyper-parameter
    hp += ("upLimit", 4, 4)                     // up-limit hyper-parameter for stopping rule
    hp += ("beta", 0.9, 0.9)                    // momentum decay hyper-parameter
    hp += ("nu", 0.9, 0.9)                      // interpolates between SGD (nu = 0) and
                                                // (normalized) SHB (nu = 1)

    // example adjustments - to be done before creating the neural network model
    hp("eta")       = 0.05
    hp("bSize")     = 25
    hp("maxEpochs") = 200
    hp("lambda")    = 0.02
    hp("upLimit")   = 5
    hp("beta")      = 0.8
    hp("nu")        = 0.8
```

Note, in other packages `patience` is number of upward steps, not the number of subsequent upward steps. Below are additional neural network hyper-parameters for a future release of ScalaTion.

```scala
    hp += ("dropout", 0.05, 0.05)               // probability of neuron dropping out
    hp += ("valSplit", 0.1, 0.1)                // training-validation set split fraction
```

7. **Tuning the Network Architecture**: The architecture of the neural network can be tuned by

   (a) changing the number of layers,

   (b) changing the number of nodes in each hidden layer, and

   (c) changing the activation functions.

   Tune the architecture for the `AutoMPG` dataset. The number of layers and number of nodes in each layer should only be increased when there is non-trivial improvement.

8. **Feature Selection for Neural Networks**. Although Neural Networks may be used without Feature Selection, it can still be useful to consider. An improvement over Forward Selection and Backward Elimination is Stepwise Regression. Start with no variables in the model and add one variable that improves the selection criterion the most. Add the second best variable for step two. After the second step determine whether it is better to add or remove a variable. Continue in this fashion until no improvement in the selection criterion is found. For Forward Selection and Backward Elimination it may instructive to continue all the way to the end (all variables for forward/no variables for backward).

   Stepwise regression may lead to coincidental relationships being included in the model, particularly if a penalty-free QoF measure such as R^2 is used. Typically, this approach is used when there a penalty for having extra variables/parameters, e.g., R^2 adjusted $\bar{R}^2$, R^2 cross-validation R^2_{cv} or Akaike Information Criterion (AIC). See the section on Maximum Likelihood Estimation for a definition of AIC. Alternatives to Stepwise Regression include Lasso Regression (ℓ^1 regularization) and to a lesser extent Ridge Regression (ℓ^2 regularization).

   Perform Forward Selection, Backward Elimination, and Stepwise Regression with all four criteria: R^2, $\bar{R}^2$, R^2_{cv}, and AIC. Plot the curve for each criterion, determine the best number of variables and what these variables are. Compare the four criteria.

   As part of a larger project compare this form of feature selection with that provided by Ridge Regression and Lasso Regression.

   Use the following types of models: `TranRegression`, `Perceptron`, `NeuralNet_3L`, and `NeuralNet_XL` (4 layers).

   In addition to the `AutoMPG` dataset, use the `Concrete` dataset and three more datasets from UCI Machine Learning Repository. The UCI datasets should have more instances (m) and variables (n) than the first two datasets. The testing is to be done in ScalaTion and Keras.

9. **Double Descent in Deep Learning**. Conventional wisdom indicates that as more features, nodes, or parameters are added the network/model, the training loss (e.g., based on mse or sse) will continue to decrease, but the testing loss (out-of-sample) will start to increase. Recent research suggests that as even more parameters are added to the network/model, the testing loss may begin to decrease for a second time. Try this with some of the datasets from the last question. Note, that a decrease in the loss function corresponds to an increase in R^2 for training or R^2_{cv} for testing.

10.9 Convolutional Neural Networks

One way to think about a Convolution Neural Network (CNN) is to take an ordinary Neural Network such as a 2-layer Neural Network and add a special non-neural layer in front of it, as depicted in Figure 10.14. This special layer is called the convolutional layer (there may be more than one). Its purpose is to reduce the volume of input in such a way that the salient features are brought forward. When the input is huge, such as image data, it is important to reduce the amount of data sent to the last layers, now referred to as the Fully-Connected (FC) layers.

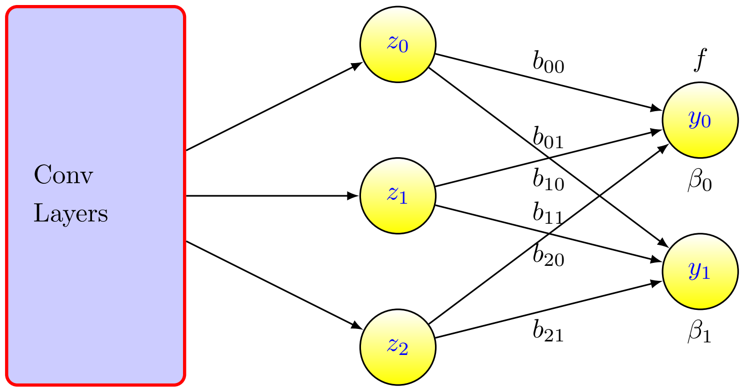


Figure 10.14: High-Level View of a Convolutional Neural Network

The convolution layers will take the input and apply *convolutional filtering* along with *pooling* operations. Inputs to the convolutional layers may be 1D (vector), 2D (matrix) or 3D or higher (tensor). After the convolutional layers, a flattening layer is used to turn matrices/tensors into vectors. Whatever the dimensionality of the input, the output of the flattening layer will be one dimensional. The flattening layer will fully connect to the first FC layer using a parameter matrix for edge weights. In general, the box labeled Conv Layers may consist of multiple convolutional layers with multiple pooling layer interspersed, followed by a flattening layer. Some of the advantages of convolutional networks are listed below:

- Input Data Reduced for Subsequent Layers.
- Learning Involves Fewer Parameters.
- Faster Network Training Time.
- Bringing Salient Features to the Forefront.

For additional background, there are multiple basic introductions to convolutional networks including [97, 108]. In the next two sections, 1D and 2D convolutional networks will be presented.

10.10 1D CNN

A 1D CNN takes the simplest form of input, that of a one-dimensional vector. Such input is found in sequence data such a time series, sound/audio and text. More well known are 2D CNNs where the input is two-dimensional as in grayscale images. Color images would be 3D as there are typically three color channels.

Convolutional Neural Networks can be deep and complex, but in their simplest form they may be viewed as a modified Fully-Connected Neural Network. The example shown in Figure 10.15 of a one-dimensional convolutional network replaces the first dense layer with a convolutional layer. There are two differences: First the connections are *sparse* (at least not full) and the weights/parameters are *shared* by all second layer nodes. Notice that each node z_h is only connected to three input nodes forming a *window* of width $n_c = 3$ starting at index h,

$$\mathbf{x}_{[h,n_c]} \;=\; \mathbf{x}_{h:h+n_c-1} \;=\; [x_h, x_{h+1}, x_{h+2}] \qquad \text{for } h = 0, 1, 2 \tag{10.96}$$

and that the parameters are always $\mathbf{c} = [c_0, c_1, c_2]$. The second set of edges fully connect the $\mathbf{z}$ nodes with the $\mathbf{y}$ nodes.

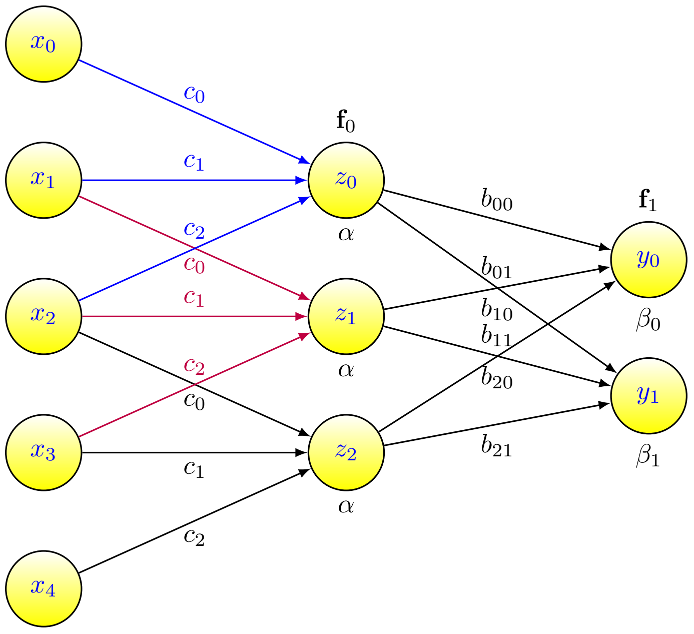


Figure 10.15: Simple 1D CNN

Convolutional Filter

As the edge weights are shared, rather than duplicating them for multiple edges, we may think of the vector $\mathbf{c} = [c_0, c_1, c_2]$ as a convolutional filter (or cofilter) that is used to calculate second layer node values,

$$z_h \;=\; \mathbf{x}_{[h,3]} \cdot \mathbf{c} \;=\; \mathbf{x}_{h:h+2} \cdot \mathbf{c} \tag{10.97}$$

for each index value h by shifting the window/slice $\mathbf{x}_{h:h+2}$ and taking the dot product with the cofilter $\mathbf{c}$. For example, if the input $\mathbf{x} = [1, 2, 3, 4, 5]$ and the cofilter $\mathbf{c} = [0.5, 1.0, 0.5]$, then

$$\mathbf{z} \;=\; [4, 6, 8]$$

To keep things simple, we assume the bias α is zero and that the convolutional layer's activation function f_0 is the identity function. The $\mathbf{z}$ vector is then propagated through the fully connected layer to obtain predicted output $\hat{\mathbf{y}}$.

$$\hat{y}_k \;=\; f_1(\mathbf{b}_{:k} \cdot \mathbf{z})$$

10.10.1 Model Equation

The calculation of second layer node values can be vectorized by introducing a convolutional operator $*_c$ that shifts the cofilter $\mathbf{c}$ over the vector $\mathbf{x}$, slicing and computing dot products, which are collected into vector $\mathbf{z}$.

$$\mathbf{z} \;=\; \mathbf{x} *_c \mathbf{c} \tag{10.98}$$

In cases where activation function f_0 is not the identity function, the equation becomes

$$\mathbf{z} \;=\; \mathbf{f}_0(\mathbf{x} *_c \mathbf{c}) \tag{10.99}$$

ScalaTion's `conv` function in object `CoFilter_1D` implements this convolution operator $*_c$. Note, the subscript c is used to avoid confusion with element-wise vector multiplication.

```
    @param c  the cofilter vector of coefficient
    @param x  the input/data vector

    def conv (c: VectorD, x: VectorD): VectorD =
        val y = new VectorD (x.dim - c.dim + 1)
        for k <- y.indices do y(k) = x(k until k + c.dim) dot c
        y
    end conv
```

The second equation takes the $\mathbf{z}$ vector and propagates it through the Fully-Connected (FC) component of the network

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{z}) + \boldsymbol{\epsilon} \tag{10.100}$$

where the B matrix is a 3-by-2 matrix in the example.

Recall the model equation for `NeuralNet_3L` written in vector form.

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{f}_0(A \cdot \mathbf{x})) + \boldsymbol{\epsilon}$$

For 1D CNNs, the model equation is very similar.

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{f}_0(\mathbf{x} *_c \mathbf{c})) + \boldsymbol{\epsilon} \tag{10.101}$$

Rather than taking the dot product with a matrix of unshared parameters/weights A, the input vector $\mathbf{x}$ undergoes convolution with cofilter vector $\mathbf{c}$. Adding in the shared scalar bias α for the convolutional layer and the unshared bias vector $\boldsymbol{\beta}$ for the fully connected layer gives

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{f}_0(\mathbf{x} *_c \mathbf{c} + \alpha) + \boldsymbol{\beta}) + \boldsymbol{\epsilon} \tag{10.102}$$

When input vector $\mathbf{x}$ is large it may be useful to further reduce the vector sizes by pooling, e.g., max-pooling with stride $s = 2$, would take the max of two adjacent elements and then move on to the next two. ScalaTion's `pool` function in object `CoFilter_1D` implements the pooling operator.

```
    @param x  the input/data vector
    @param s  the size of the pooling window

    def pool (x: VectorD, s: Int = 2): VectorD =
        val p = new VectorD (x.dim / s)
        for j <- p.indices do
            val jj = s * j
            p(j)   = x(jj until jj+s).max ()
        end for
        p
    end pool
```

10.10.2 Training

Training 1D Convolutional Neural Networks is more complicated, although more efficient, than training Fully-Connected Neural Networks [97]. Training involves finding values for the parameters $B \in \mathbb{R}^{n_z \times n_y}$, $\boldsymbol{\beta} \in \mathbb{R}^{n_y}$, $\mathbf{c} \in \mathbb{R}^{n_c}$ and $\alpha \in \mathbb{R}$ that minimize a given loss function $\mathcal{L}$. To simplify the development, the biases will be ignored.

For a single input vector $\mathbf{x}$ yielding

$$\hat{\mathbf{y}} = \mathbf{f}_1(B \cdot \mathbf{f}_0(\mathbf{x} *_c \mathbf{c})) \tag{10.103}$$

the ℓ^2 loss function will be

$$\mathcal{L}(B, \mathbf{c}) = \frac{1}{2}[(\mathbf{y} - \hat{\mathbf{y}}) \cdot (\mathbf{y} - \hat{\mathbf{y}})] \tag{10.104}$$

10.10.3 Optimization

As with `NeuralNet_3L`, there are basically four steps in optimization.

1. **Compute predicted values**: Based on the randomly assigned weights to vector $\mathbf{c}$ and matrix B, predicted outputs $\hat{\mathbf{y}}$ are calculated. First values for the hidden layer $\mathbf{z}$ are calculated, where the value for hidden node h, z_h, is given by

$$z_h \;=\; f_0(\mathbf{x}_{[h,n_c]} \cdot \mathbf{c}) \qquad \text{for } h = 0, \ldots, n_z - 1$$

where f_0 is the first activation function (e.g., reLU), $\mathbf{c}$ is the convolutional filter which contains the shared weights/parameters, and $\mathbf{x}$ is an input vector for a training sample/instance (row in the data matrix). Next, the values computed at the hidden layer are used to produce predicted outputs $\hat{\mathbf{y}}$, where the value for output node k, $\hat{y}_k$, is given by

$$\hat{y}_k \;=\; f_1(\mathbf{b}_{:k} \cdot \mathbf{z}) \qquad \text{for } k = 0, \dots, n_y - 1$$

where the second activation function f_1 may be the same as (or different from) the one used in the hidden layer and $\mathbf{b}_{:k}$ is column-k of the B weight matrix. Now the difference between the actual and predicted output, the error, for the k^{th} output, ϵ_k, is given by

$$\epsilon_k \;=\; y_k - \hat{y}_k \qquad \text{for } k = 0, \dots, n_y - 1$$

2. **Back propagate from output layer**: Given the computed error vector $\boldsymbol{\epsilon}$, the delta/correction vector $\boldsymbol{\delta}^1$ for the output layer may be calculated, where for output node k, δ^1_k is given by

$$\boxed{\delta^1_k \;=\; [-\epsilon_k]\, f_1'(\mathbf{b}_{:k} \cdot \mathbf{z})} \qquad \text{for } k = 0, \dots, n_y - 1 \tag{10.105}$$

where f_1' is the derivative of the activation function. The partial derivative of the loss function $\mathcal{L}$ with respect to the weight connecting hidden node h with output node k, b_{hk}, is given by

$$\frac{\partial \mathcal{L}}{\partial b_{hk}} \;=\; z_h \delta^1_k \tag{10.106}$$

3. **Back propagate from hidden layer**: Given the delta/correction vector $\boldsymbol{\delta}^1$ from the output layer, the delta vector for the hidden layer $\boldsymbol{\delta}^0$ may be calculated, where for hidden node h, δ^0_h is given by

$$\boxed{\delta^0_h \;=\; [\mathbf{b}_h \cdot \boldsymbol{\delta}^1]\, f_0'(\mathbf{x}_{[h,n_c]} \cdot \mathbf{c})} \qquad \text{for } h = 0, \dots, n_z - 1 \tag{10.107}$$

The partial derivative of $\mathcal{L}$ with respect to the j^{th} weight in the convolutional filter, c_j, is given by

$$\frac{\partial \mathcal{L}}{\partial c_j} \;=\; \sum_{h=0}^{n_z-1} x_{j+h} \delta^0_h \;=\; \mathbf{x}_{[j,n_z]} \cdot \boldsymbol{\delta}^0 \tag{10.108}$$

4. **Update weights**: The weights $\mathbf{c}$ and B, in the convolutional and full-connected layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate η which moderates the adjustments to the weights.

$$b_{hk} \;-\!=\; z_h\, \delta^1_k\, \eta \tag{10.109}$$

$$c_j \;-\!=\; \mathbf{x}_{[j,n_z]} \cdot \boldsymbol{\delta}^0\, \eta \tag{10.110}$$

10.10.4 Matrix Version

1. The hidden (latent) values for all m instances and all n_z hidden nodes are computed by applying the first matrixized activation function $\mathbf{f}_0$ to the convolution $X *_c c$ to produce the **latent feature matrix** $Z \in \mathbb{R}^{m \times n_z}$.

$$\boxed{Z \;=\; \mathbf{f}_0(X *_c c)} \tag{10.111}$$

The **predicted output matrix** $\hat{Y} \in \mathbb{R}^{m \times n_y}$ is similarly computed by applying the second matrixized activation function $\mathbf{f}_1$ to the matrix product ZB.

$$\boxed{\hat{Y} \;=\; \mathbf{f}_1(ZB)} \tag{10.112}$$

2. The **negative of the error matrix** $E \in \mathbb{R}^{m \times n_y}$ is just the difference between the predicted and actual/target values.

$$\boxed{E \;=\; \hat{Y} - Y} \tag{10.113}$$

3. This information is sufficient to calculate delta matrices: Δ^1 for adjusting B and Δ^0 for adjusting A. The **output-layer delta matrix** $\Delta^1 \in \mathbb{R}^{m \times n_y}$ is the element-wise matrix (Hadamard) product of E and $\mathbf{f}_1'(ZB)$.

$$\boxed{\Delta^1 \;=\; E \odot \mathbf{f}_1'(ZB)} \tag{10.114}$$

The **hidden-layer delta matrix** $\Delta^0 \in \mathbb{R}^{m \times n_z}$ is the element-wise matrix (Hadamard) product of $\Delta^1 B^\intercal$ and $\mathbf{f}_0'(XA)$.

$$\boxed{\Delta^0 \;=\; [\Delta^1 B^\intercal] \odot \mathbf{f}_0'(X *_c c)} \tag{10.115}$$

4. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate η) for updating the parameter/weight matrix B and cofilter vector $\mathbf{c}$.

$$\boxed{B \;-\!\!=\; Z^\intercal \Delta^1 \eta} \tag{10.116}$$

$$\boxed{c_j \;-\!\!=\; [X^\intercal_{:,[j,n_z]} \Delta^0].\text{mean}\, \eta} \tag{10.117}$$

10.10.5 Gradient Descent Algorithm

The basic gradient descent algorithm adapts the above equations into a loop to iteratively update the weight matrices and bias vectors (unified via the `NetParam` class). The corresponding ScalaTion code for the `train` method is shown below. Note: `*:` defined in `NetParam` computes value-matrix * weigth-matrix + bias-vector.

```
    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
        var sse0 = Double.MaxValue                              // hold prior value of sse

        var (go, epoch) = (true, 1)
        cfor (go && epoch <= maxEpochs, epoch += 1) {
            var z  = f.fM (conv (x_ , c))                       // Z  = f(conv (X, c))
            var yp = f1.fM (b * z)                              // Yp = f(ZB)
            val ϵ  = yp - y_                                    // negative of error matrix
            val δ1 = f1.dM (yp) ⊙ ϵ                                 // delta matrix for y
            val δ0 = f.dM (z) ⊙ (δ1, b.w.𝒯)                          // delta matrix for z
            CNN_1D.updateParam (x_, z, δ0, δ1, eta, c, b)

            val yp_ = f1.fM (f0.fM (conv (x, c)) *: b)           // updated predictions
            val sse = sseF (y_, yp_)
            if sse > sse0 then return                            // return early if moving up
            sse0 = sse                                           // save prior sse
        end for
    end train
```

Again note: ScalaTion provides the following alternatives for (a) Hadamard product: ⊙ or *~, and (b) Transpose: T-like Unicode symbol or `transpose`.

The `updateParam` method produces new values for `c` and `b`.

```
    def updateParam (x_ : MatrixD, z: MatrixD, δ0: MatrixD, δ1: MatrixD, eta: Double,
                     c: VectorD, b: NetParam) =
        for j <- c.indices do
            var sum = 0.0
            for i <- x_.indices; h <- z.indices2 do sum += x_(i, h+j) * δ0(i, h)
            c(j) -= (sum / x_.dim) * eta                           // update c in conv filter
        end for
        b -= (z.𝒯 * δ1 * eta, δ1.mean * eta)                       // update b weights/biases
    end updateParam
```

10.10.6 Example Error Calculation Problem

Given the following problem, compute the negative error matrix.

```
    val x = MatrixD ((2, 5), 1, 2, 3, 4, 5,
                             6, 7, 8, 9, 10)
    val y = MatrixD ((2, 2),  6,  9,
                             16, 24)
    val c = VectorD (0.5, 1, 0.5)
    val b = NetParam (MatrixD ((3, 2), 0.1, 0.2,
                                       0.3, 0.4,
                                       0.5, 0.6))
```

Use the following code to compute the negative error matrix E.

```
    val z  = f0.fM (conv (x, c))                                // Z  = f(conv (X, c))
    val yp = f1.fM (z *: b)                                     // Yp = f(ZB)
    val ϵ  = yp - y                                             // negative error E  = Yp - Y
    println (s"ϵ = $ϵ")
```

10.10.7 Two Convolutional Filters

Typically, a convolutional layer may have multiple cofilters. Figure 10.16 has two cofilters: The first cofilter incident upon the yellow nodes in the middle has weights $\mathbf{c^0} = [c^0_0, c^0_1, c^0_2]$, while the second cofilter incident upon the orange nodes in the middle has weights $\mathbf{c^1} = [c^1_0, c^1_1, c^1_2]$. The superscripts are left off in the figure due to crowding of edge labels.

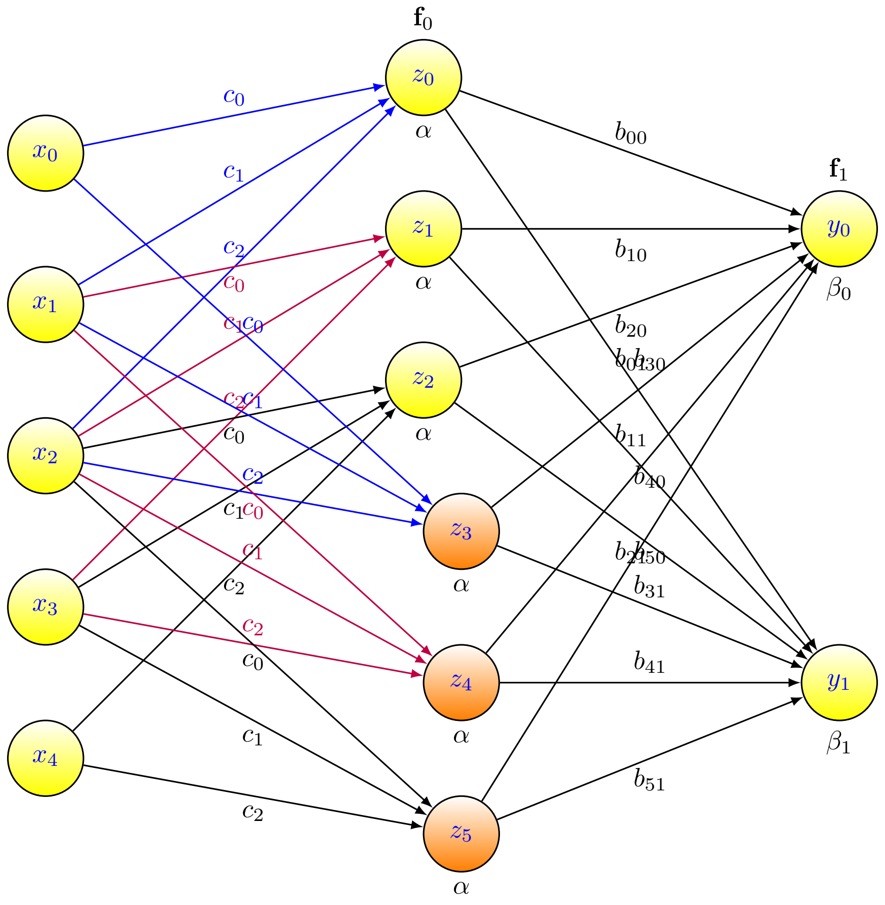


Figure 10.16: A 1D CNN with Two Convolution Filters $\mathbf{c^0}$ and $\mathbf{c^1}$

The cofilter $\mathbf{c^0}$ will move over the input nodes producing feature map $\boldsymbol{\phi}^0$ (the yellow nodes in the middle), while cofilter $\mathbf{c^1}$ will also move over the input nodes producing feature map $\boldsymbol{\phi}^1$ (the orange nodes in the middle). As there are no other layers before the fully-connected layer, these two feature maps are combined (or flattened) and activated to form a hidden ($\mathbf{z} = [z_0, z_1, z_2, z_3, z_4, z_5]$) layer. The stride indicates how far the cofilter moves (e.g., down one node, down two nodes) on each step. Here the stride is equal to one.

Figure 10.17 shows the same Convolutional Neural Network from the point of view of tensors flowing through the network (in this case the tensors are just vectors). Cofilter $\mathbf{c^0}$ moves through the input vector $\mathbf{x}$ taking the dot product of itself with the blue, purple/crimson and black windows/slices, respectively. These three dot products produce the values in feature map $\boldsymbol{\phi}^0$. (Note that blue line from cofilter $\mathbf{c^0}$ to feature map $\boldsymbol{\phi}^0$ has utilized all the values in the cofilter and all values in the blue window and is shown as a single line to reduce clutter.) The same logic is used for the second cofilter $\mathbf{c^1}$ to create feature map $\boldsymbol{\phi}^1$. The difference lies only in the values in each of these two convolution vectors.

Now the two feature maps $\boldsymbol{\phi}^0$ and $\boldsymbol{\phi}^1$ need to be combined. In general this is done by a flattening operation. For the 1D case, the feature maps are already vectors so flattening reduces to vector concatenation. This flattened vector is activated using $\mathbf{f}_0$ to produce $\mathbf{z}$ that serves as an entry point into the fully-connected part of the network. Further computations produce the output vector $\hat{\mathbf{y}}$ based on vector $\mathbf{z}$ and parameter matrix B, with their product activated using $\mathbf{f}_1$.

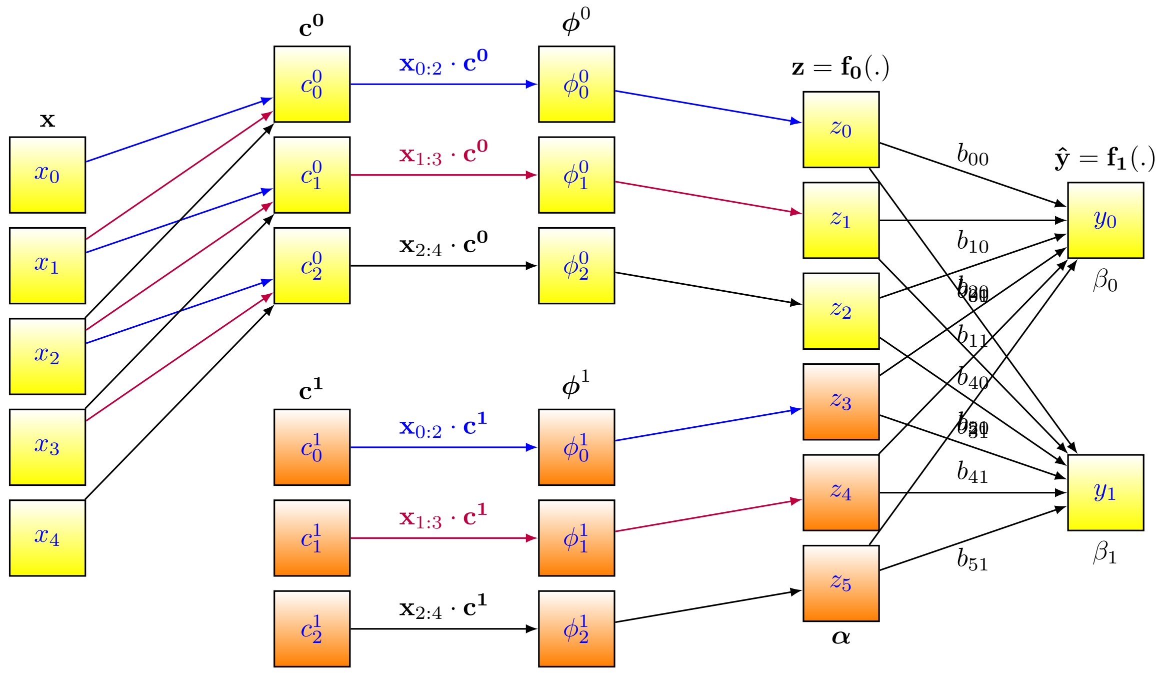


Figure 10.17: Tensor Flow Diagram Showing Two Convolution Filters $\mathbf{c^0}$, $\mathbf{c^0}$ and Two Feature Maps $\boldsymbol{\phi}^0$, $\boldsymbol{\phi}^0$ followed by a Fully-Connected Layer

A Convolutional Neural Network may have multiple convolutional layers as well as pooling layers (discussed in more detail in the next section). Activation (e.g., using `reLU`) occurs at the end of paired convolutional-pooling layers. In addition, the fully-connected part may consist of multiple layers.

10.10.8 `CNN_1D` Class

Class Methods:

```
    @param x        the input/data matrix with instances stored in rows
    @param y        the output/response matrix, where y_i = response for row i of matrix x
    @param fname_   the feature/variable names (defaults to null)
    @param nf       the number of filters for this convolutional layer
    @param nc       the width of the filters (size of cofilters)
    @param hparam   the hyper-parameters for the model/network
    @param f        the activation function family for layers 1->2 (input to hidden)
    @param f1       the activation function family for layers 2->3 (hidden to output)
    @param itran    the inverse transformation function returns responses to original scale

    class CNN_1D (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                  nf: Int = 1, nc: Int = 3,
                  hparam: HyperParameter = Optimizer.hp,
                  f: AFF = f_reLU, f1: AFF = f_reLU,
                  val itran: FunctionM2M = null)
          extends PredictorMV (x, y, fname_, hparam)
```

```scala
            with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def filter (i: Int, f: Int): VectorD =
    def updateFilterParams (f: Int, vec2: VectorD): Unit = filt(f).update (vec2)
    override def parameters: NetParams = Array (NetParam (MatrixD.fromVector (c)), b)
    def train (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    override def train2 (x_ : MatrixD = x, y_ : MatrixD = y): Unit =
    def test (x_ : MatrixD = x, y_ : MatrixD = y): (MatrixD, MatrixD) =
    def predict (z: VectorD): VectorD = f1.f_ (b dot f.f_ (conv (z, c)))
    override def predict (z: MatrixD = x): MatrixD = f1.fM (b * f.fM (conv (z, c)))
    def buildModel (x_cols: MatrixD): CNN_1D =
```

10.10.9 Exercises

1. **Delta Vectors**: For the example error calculation problem given in this section, calculate the $\boldsymbol{\delta}^1$ and $\boldsymbol{\delta}^0$ vectors.

2. **Parameter Update Equations**: For the same example, use the $\boldsymbol{\delta}^1$ vector to update weight matrix B for the fully-connected layer. Use the $\boldsymbol{\delta}^0$ vector to update weight vector $\mathbf{c}$ for the convolutional layer.

3. For the same example, make four training steps and see what what happens to *sse*.

4. Create a `CNN_1D` model for the `AutoMPG` dataset.

```scala
    val cn = new CNN_1D (x, MatrixD (y))
```

5. Create and demo a 1D CNN for the MIT-BIH Arrhythmia Database using ScalaTion's `CNN_1D` class and Keras. See `https://physionet.org/content/mitdb/1.0.0`.

10.11 2D CNN

Two-dimensional Convolution Networks (2D CNN2) are needed when the data instances are two-dimensional and based on a regular grid pattern where the values in the grid can be mapped to a matrix. For 2D CNNs, flattening operators are added to the convolutional and pooling operators. The following references focus on 2D Convolutional Networks [88, 214]. To clarify what can happen in a 2D CNN convolutional layer, consider the well-known MNIST problem to classify hand written digits 0 to 9. Given 10,000 grayscale images, each having 28-by-28 pixels, the goal is to train a convolutional network to classify an image as one of 10 digits. The following simple network architecture may be used as a starting point for the MNIST problem:

- Convolutional Layer. A convolutional layer consists of one or more convolutional filters, for which a convolutional operation is applied to the input for each filter, conceptually in parallel. The result of the convolutions may then be passed into an activation function. For this problem, assume there is just one 5-by-5 convolutional filter.

- Pooling Layer. A pooling layer takes the output of a convolutional layer and reduces/aggregates it. A common pooling operation is generally applied to the results of the convolution. For this problem, assume there is a 2-by-2 max-pooling operation applied to the one filter.

- Flattening Layer. Between the convolutional layers and fully-connected layers a flattening layer turns the matrices (or tensors for 3D CNN) flowing through the convolutional layers and pooling layers into vectors as required by the fully-connected layers.

- Fully Connected Output Layer. A 10 node layer where the i^{th} node represents the i^{th} digit. Whichever output node gets the highest score is the classification result. In general, there may be multiple fully-connected layers.

Advanced network architectures may have many convolutional layers and pooling layers. Since one layer comes after the other, the signals are processed in series (not parallel). For this problem, assume there is just one convolutional layer.

10.11.1 Filtering Operation

A filter is a small matrix, e.g., a 5-by-5 matrix that scans across the image taking the dot product of its values with the values in a 5-by-5 window of the image, as depicted in Figure 10.18. Commonly a (convolutional) stride of 1 is used that moves the window one position/pixel at a time. This process produces a 24-by-24 result matrix referred to as a feature map. Padding can be used to maintain the size of the image. The commonly used zero-padding would put zeros around the image. Each of the filters will produce its own feature map.

The MNIST dataset consisting of many images can be collected into a 3-dimensional tensor $\mathbf{X}$. The i^{th} row (or renamed as a sheet) of the tensor $\mathbf{X}$, X_i, will be a 28-by-28 input matrix.

Instead of looking at a large input matrix X, consider one of the matrices given on the following Webpage `https://cs231n.github.io/convolutional-networks`. It is a 5-by-5 matrix, extracted below.

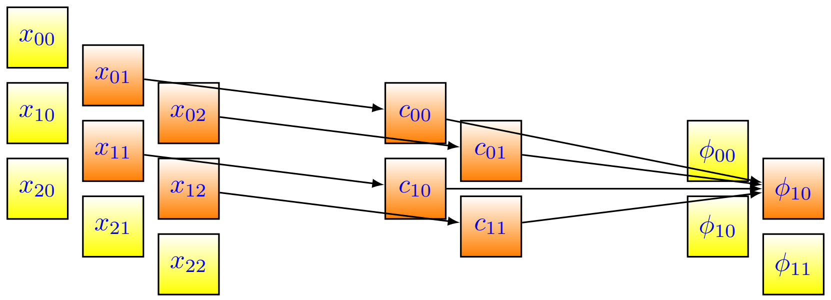


Figure 10.18: Portion of an Input Image/Matrix X - Convolutional Filter C - Feature Map Φ

$$X = \begin{bmatrix} 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 \\ 1 & 2 & 2 & 0 & 2 \\ 2 & 0 & 0 & 0 & 1 \\ 2 & 2 & 2 & 0 & 1 \end{bmatrix} \tag{10.118}$$

The effect of applying the following 2-by-2 convolutional filter matrix C to X,

$$C = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{10.119}$$

is produced by using the convolution operator ($*_c$), which will give the following feature map matrix Φ,

$$\Phi = X *_c C \tag{10.120}$$

where each element in the resulting 4-by-4 feature map matrix Φ is the dot product of a subimage and the convolutional filter. The subimage is a shifting 2-by-2 window/slice of the original image with top-left position (j, k).

$$\phi_{jk} = X_{j:j+1,k:k+1} \cdot C \tag{10.121}$$

The resulting feature map matrix Φ is shown below.

$$\Phi = \begin{bmatrix} 0 & 2 & 4 & 3 \\ 2 & 2 & 1 & 5 \\ 3 & 4 & 2 & 3 \\ 4 & 2 & 0 & 2 \end{bmatrix} \tag{10.122}$$

Notice for this example, if a reLU activation function is applied, the final feature map is unaltered. ScalaTion's `conv` function in the `CoFilter_2D` object implements the convolution operator.

```
def conv (x: MatrixD, c: MatrixD): MatrixD =
    val (m, n) = (c.dim, c.dim2)
    val phi = new MatrixD (x.dim + 1 - m, x.dim2 + 1 - n)
```

```
        for j <- phi.indices; k <- phi.indices2 do
            phi(j, k) = (x(j until j+m, k until k+n) *~ c).sum
        end for
        phi
    end conv
```

10.11.2 Pooling Operation

Each feature map can be further reduced to a pooled feature map using a pooling operation. Max pooling takes the maximum value in each non-overlapping 2-by-2 region as the value for the pooled feature map. Each of the pooled feature maps will be 12-by-12 matrices. All these values will be placed in the flattening layer consisting of $12 \times 12 = 144$ nodes.

Using 2-by-2 max pooling, the maximum of each non-overlapping 2-by-2 submatrix is taken.

$$p_{jk} = \max \Phi_{2j:2j+1,2k:2k+1} \tag{10.123}$$

to produce the 2-by-2 matrix P. Note, having a pooling stride of 2 will make the regions non-overlapping.

$$P = \begin{bmatrix} 2 & 5 \\ 4 & 3 \end{bmatrix} \tag{10.124}$$

ScalaTion's `pool` function in the `CoFilter_2D` object implements the pooling operator.

```
    def pool (x: MatrixD, s: Int = 2): MatrixD =
        val p = new MatrixD (x.dim / s, x.dim2 / s)
        for j <- p.indices; k <- p.indices2 do
            val (jj, kk) = (s*j, s*k)
            p(j, k) = x.max (jj until jj+s, kk until kk+s)
        end for
        p
    end pool
```

10.11.3 Flattening Operation

Flattening takes either a matrix or tensor and forms a single vector that includes all the elements. A matrix may be flattened in either row-major or column-major style. The ScalaTion method in `MatrixD` shown below flattens a matrix in row-major fashion.

```
    def flatten: VectorD =
        val a = Array.ofDim [Double] (dim * dim2)
        var k = 0
        for i <- indices do
            val v_i = v(i)
            var j = 0
            cfor (j < dim2, j += 1) { a(k) = v_i(j); k += 1 }
        end for
        new VectorD (a.length, a)
    end flatten
```

Therefore, the flattened pooled matrix P is

```
P.flatten = [ 2, 5, 4, 3 ]
```

10.11.4 Model Equation

The first equation for a 2D CNN computes the intermediate $\mathbf{z}$ vector for a particular image, X, by taking the following steps: (i) convolve X with convolutional filter C, (ii) apply the first activation function $\mathbf{f}_0$ (e.g., reLU), (iii) pool to reduce size, and (iv) flatten matrix into a vector.

$$\mathbf{z} \;=\; [\text{pool}(\mathbf{f}_0(X *_c C))].\text{flatten} \tag{10.125}$$

The second equation takes the $\mathbf{z}$ vector and propagates it through the Fully-Connected (FC) layer of the network. The model equation includes these two steps with an error term added.

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot \mathbf{z}) + \boldsymbol{\epsilon} \tag{10.126}$$

where the B matrix connects each of the 144 nodes in the flattened layer to the 10 output nodes of the FC layer. Combining these two equations yields

$$\mathbf{y} \;=\; \mathbf{f}_1(B \cdot [\text{pool}(\mathbf{f}_0(X *_c C))].\text{flatten}) + \boldsymbol{\epsilon} \tag{10.127}$$

More generally, multiple convolutional filters (along with their pooling) can be used simultaneously. See the exercises for an example with multiple convolutional filters. In addition, multiple combination convolution-pooling layers are often used, as in Deep Learning. The fully-connected part of the network may also consist of multiple layers as well.

10.11.5 Training

Training involves finding values for the parameters $B \in \mathbb{R}^{n_z \times n_y}$, $\boldsymbol{\beta} \in \mathbb{R}^{n_y}$, $C \in \mathbb{R}^{n_c \times n_c}$ and $\alpha \in \mathbb{R}$ that minimize a given loss function $\mathcal{L}$. To simplify the development, the biases will be ignored.

For a single input matrix X yielding

$$\hat{\mathbf{y}} = \mathbf{f}_1(B \cdot [\text{pool}(\mathbf{f}_0(X *_c C))].\text{flatten}) \tag{10.128}$$

the ℓ^2 loss function will be

$$\mathcal{L}(B, C) = \frac{1}{2}[(\mathbf{y} - \hat{\mathbf{y}}) \cdot (\mathbf{y} - \hat{\mathbf{y}})] \tag{10.129}$$

10.11.6 Optimization

10.11.7 Exercises

1. Develop the modeling equations for a 2D CNN that consists of the following layers:

   (a) Convolutional Layer with two feature maps. 5×5 convolutional filters, no padding, stride 1 and reLU activation.

   (b) Pooling Layer using max-pooling. 2×2 pooling with a stride of 2.

   (c) Flattening Layer to convert matrices into a vector.

(d) Fully-Connected Layer with 10 output nodes.

2. If the above network is used for a 28×28 MNIST image/matrix, give the dimensions for all components in the network. How many parameters will need to be optimized?

3. Create and demo a 2D CNN for the MNIST dataset using Keras. See `http://yann.lecun.com/exdb/mnist`.

4. It is common in data science to *normalize* input data. This is especially true when there are nonlinear transformations. For CNNs and deep learning, normalizing the initial input may be insufficient, so other types of normalization have been introduced, such as batch normalization, and layer normalization. Discuss the advantages and disadvantages of the various normalization techniques.

5. The convolution operator slides a window the size of the convolution filter over the image. For a dilated CNN, when the *dilation rate* is 2, the window size is enlarged in each dimension since every other pixel is skipped. Given a 3-by-3 convolution filter, the window size (size of the receptive field in the image) is 3-by-3. In this case the dilation rate is 1. In general, a dilation rate of n, will mean $n - 1$ pixels will skipped for every pixel used. What would the window size if the dilation rate is 2?

6. Create a 3D CNN that processes color images using Keras. It will take a four-dimensional input tensor $\mathbf{X} = [x_{ijkl}]$ where i indicates which image, j indicates which (horizontal) row, k indicates which (vertical) column, and l indicates which color channel. Use the CIFAR-10 dataset `https://www.cs.toronto.edu/~kriz/cifar.html`.

10.12 Transfer Learning

Transfer learning is applicable to many types of machine learning problems [140]. In this section, we focus on transfer learning for neural networks [190].

The first sets of transformations (from input to the first hidden layer and between the first and second hidden layers) in a Neural Network allow nonlinear effects to be created to better capture characteristics of the system under study. These are taken in combination in later latter layers of the Neural Network. The thought is that related problems share similar nonlinear effects. In other words, two datasets on related problems used to train two Neural Networks are likely to develop similar nonlinear effects at certain layers. If this is the case, the training of the first Neural Network could expedite the training of the second Neural Network. Some research has shown the Quality of Fit (QoF) may also be enhanced as well.

The easiest way to imagine this is to have two four-layer Neural Networks, say with 30 inputs for the first and 25 for the second. Let the first hidden layer have 50 nodes and second have 20 nodes, with an output layer for the single output/response value. The only difference in node count is in the input layer. For the first Neural Network, the sizes of the parameter/weight matrices are 30-by-50, 50-by-20 and 20-by-1. The only difference in the second Neural Network is that the first matrix is 25-by-50. After the first Neural Network is trained, its second matrix (50-by-20) along with its associated bias vector could be transferred to the second Neural Network. When training starts for the second Neural Network, random initialization is skipped for this matrix (and its associated bias vector). A training choice is whether to freeze this layer or allow its values to be adjusted during back-propagation.

10.12.1 Definition of Transfer Learning

In its simplest form transfer learning involves two related datasets and two related tasks.

- Dataset. A dataset D may be defined as an ordered pair of tensors $D = (\mathbf{X}, \mathbf{Y})$ where $\mathbf{X}$ is the input tensor and $\mathbf{Y}$ is the output tensor. It is assumed that there is an unknown function f and a noise generator $\boldsymbol{\epsilon}$ that characterizes the relationship between input tensor $\mathbf{X}$ and output tensor $\mathbf{Y}$.

$$\mathbf{Y} \;=\; f(\mathbf{X}) + \boldsymbol{\epsilon} \tag{10.130}$$

- Task. In a limited sense, a task τ may be viewed a procedure to produce a model function f_m that approximates f. Secondary goals include interpretability and generalizability. The closeness of the approximation can be measured by some norm of the differences between the functions or more generally by a task inspired loss function (e.g., MSE for regression taks or cross entropy for classifications tasks),

$$f_m \;=\; \min\{f_k : \|f - f_k\| \text{ for } f_k \in \mathcal{F}\} \tag{10.131}$$

  where $\mathcal{F}$ is a function space. For example, if the input tensor is two-dimensional (a matrix) and the output tensor is one-dimensional (a vector), $\mathcal{F}$ could be the set of all linear transformations of the column space of the input matrix to $\mathbb{R}$. The goal of multiple linear regression is to find a point in the column space of matrix X closest to the vector $\mathbf{y}$.

- Transfer Learning. Now given two datasets D_1 and D_2, and two tasks τ_1 and τ_2, the question is whether the efforts in task τ_1 can reduce the effort and/or improve the quality of outcome of task τ_2.

Transfer learning is ideally applied when both the datasets and the tasks are similar (or related in some sense) and when the first dataset is large enough to support accurate training. Applying transfer learning to the second task may be motivated by limited time/computational resources or lack of data. The lack of data often happens for classification tasks with missing labels (y values).

10.12.2 Type of Transfer Learning

Transfer Learning can involve domain adaptation and instance selection. For neural networks, components of one network developed for task τ_1 may be transferred to another network under development for task τ_2.

Transfer Learning in Neural Networks

An area in which transfer learning has demonstrated considerable success is in Convolutional Networks. Therefore, consider a dataset where the input is a three-dimensional tensor $\mathbf{X} \in \mathbb{R}^{m \times w \times h}$, where m is the number of grayscale images, w is their pixel width and h is their pixel height. For image classification, the output is a one-dimensional vector $\mathbf{y} \in \{0, \ldots K-1\}^m$, where K is the number of classes in classification C.

For simplicity, assume that $m_2 < m_1$, $w_2 = w_1$, $h_2 = h_1$, and $K_2 \leq K_1$. We are left with two issues: how similar/related are the two sets of images and what is the logical/semantic connection between classification C_2 and classification C_1. The similarity of classification schemes could be based on the fraction of classes in C_2 that are also in C_1.

$$r \;=\; \frac{|C_2 \cap C_1|}{|C_2|} \tag{10.132}$$

This can be refined when there is an underlying ontology to provide meaning to the classifications and their class labels, as well as, a metric for semantic distance. Note, some pre-trained models may have been trained to discriminate between hundreds or thousands of image types, e.g., ImageNet [98].

Selection of a source dataset may now be decomposed into the following steps:

1. Initial Screening. Initial screening of potential candidate source datasets can be done by comparing their similarity to that of the target dataset D_2. Information divergence (e.g., KL divergence) or distances based on optimal transport may be used to explore the similarity of datasets (or their underlying probability measures). Source datasets sufficiently similar and with pre-trained models should be tested in step 2.

2. Choosing Candidates. Datasets identified from step 1 may be selected as candidates by applying them to the target problem in the following way. Replace the last fully-connected layer of the source convolutional network with a new one, where the output layer is reduced from K_1 to K_2 nodes. Call this the adapted model. Freeze all the previous layers and train the last layer from scratch. A layer being frozen means that its parameters (weights and biases) will not be changed during back-propagation. Choose the models with better Quality of Fit (QoF) measures or lower loss functions.

3. Fine-Tuning. Let D_1 be one of the candidate datasets. One option would be to train the adapted model/network from scratch. Unfortunately, this would be time consuming at best and likely to fail at worst due to lack of class labels. Retraining involves a choice for each layer, whether to freeze, fine-time, or train from scratch. Making the right choice for each layer can lead to a more accurate model. Unfortunately, a convolution network with 16 trainable layers, such as VGG-16 [189], would

require $3^{16} = 43,047,721$ models to train. Therefore, establishing criteria for deciding how to handle each layer is important. Although it tends to be generally the case, that the later layers are specific to the given task and that earlier layers are more generic and hence transferable as is, recent research has shown benefits in fine-tuning some early layers [66].

Domain Adaptation

To summarize, given two datasets $D_1 = (\mathbf{X_1}, \mathbf{Y_1})$ and $D_2 = (\mathbf{X_1}, \mathbf{Y_2})$, and two tasks τ_1 and τ_2, utilze trained model/network f_{m_1} to produce a new model f_{m_2} by reusing much of f_{m_1} to make training have similar or better outcomes in less time.

Typically in domain adaptation, we may assume that $\tau_1 = \tau_2$ and the task is classification (thus, the outputs are vectors). It may be further assumed that a label mapping function can be defined so that class labels in classification 1 can be mapped without loss of information to labels in classification 2. Thus, we may stipulate that the output vectors are points in the following label spaces.

$$\mathbf{y}_1 \in [0, \dots, K-1]^{m_1} \tag{10.133}$$

$$\mathbf{y}_2 \in [0, \dots, K-1]^{m_2} \tag{10.134}$$

The focus in domain adaptation is on applying transformations on the input datasets, $\mathbf{X_1}$ and $\mathbf{X_2}$. Let the form of inputs now be reduced to matrices $X_1 \in \mathbb{R}^{m_1 \times n_1}$ and $X_2 \in \mathbb{R}^{m_2 \times n_2}$. To facilitate visualization, let the number of columns in both matrices be 2 ($n_1 = n_2 = 2$). Now, two-dimensional histograms may be produced for each input data matrix.

Instance Selection

10.12.3 `NeuralNet_XLT` Class

Class Methods:

```
    @param x          the m-by-n input matrix (training data consisting of m input vectors)
    @param y          the m-by-ny output matrix (training data consisting of m output vectors
    )
    @param fname_     the feature/variable names (defaults to null)
    @param nz         the number of nodes in each hidden layer, e.g., Array (9, 8) => 2
    hidden of sizes 9 and 8
                      (null => use default formula)
    @param hparam     the hyper-parameters for the model/network
    @param f          the array of activation function families between every pair of layers
    @param l_tran     the layer to be transferred in (defaults to first hidden layer)
    @param transfer   the saved network parameters from a layer of a related neural network
                      trim before passing in if the size does not match
    @param itran      the inverse transformation function returns responses to original scale

    class NeuralNet_XLT (x: MatrixD, y: MatrixD, fname_ : Array [String] = null,
                          nz: Array [Int] = null, hparam: HyperParameter = hp,
                          f: Array [AFF] = Array (f_sigmoid, f_sigmoid, f_id),
                          l_tran: Int = 1, transfer: NetParam = null,
                          itran: FunctionM2M = null)
```

```
          extends NeuralNet_XL (x, y, fname_, nz, hparam, f, itran):

    def trim (tl: NetParam, lt: Int = l_tran): NetParam = tl.trim (sizes(lt), sizes(lt+1))
```

10.12.4 Exercises

1. Find two related datasets at `https://huggingface.co/datasets/inria-soda/tabular-benchmark` and use the larger dataset to transfer in a layer and see if this improves the QoF. Compare the following three scenarios:
   (a) QoF of small dataset trained with its own data
   (b) QoF of small dataset after a layer is transferred in and not frozen
   (c) QoF of small dataset after a layer is transferred in and frozen (see `freeze` method in `NeuralNet_XL`)

10.13 Extreme Learning Machines

An Extreme Learning Machine (ELM) may be viewed as three-layer Neural Network with the first set of fixed-parameters (weights and biases) frozen. The values for these parameters may be randomly generated or transferred in from a Neural Network. With the first set of fixed-parameters frozen, only the second set of parameters needs to be optimized. When the second activation function is the identity function (`id`), the optimization problem is the same as for the Regression problem, so that matrix factorization may be used to train the model. This greatly reduces the training time over Neural Networks. Although the first set of fixed-parameters is not optimized, nonlinear effects are still created at the hidden layer and there may be enough flexibility left in the second set of parameters to retain some of the advantages of Neural Networks.

10.13.1 Model Equation

A relatively simple form of Extreme Learning Machine in ScalaTion is `ELM_3L1`. It allows for a single output/response variable y and multiple input/predictors variables $\mathbf{x} = [x_0, x_1, \dots x_{n-1}]$. The number of nodes per layer are n for input, n_z for hidden and $n_y = 1$ for output. The second activation function $\mathbf{f}_1$ is implicitly `id` and be left out.

The model equation for `ELM_3L1` can written in vector form as follows,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{f}_0(A^{\intercal}\mathbf{x})) + \epsilon} \tag{10.135}$$

where A is the first layer `NetParam` consisting of an n-by-n_z weight matrix and an n_z bias vector. In ScalaTion, these parameters are initialized as follows (see exercises for details):

```
    private var a = new NetParam (weightMat3 (n, nz, s),
                                  weightVec3 (nz, s))
```

The second layer is simply an n_z weight vector $\mathbf{b}$. The first layer's weights and biases A are frozen, while the second layer parameters $\mathbf{b}$ are optimized.

10.13.2 Training

Given a dataset $(X, \mathbf{y})$, training will be used to adjust values of the parameter vector $\mathbf{b}$. The objective is to minimize the distance between the actual and predicted response vectors.

$$\boxed{f_{obj} \;=\; \|\mathbf{y} - \mathbf{b} \cdot \mathbf{f}_0(XA)\|} \tag{10.136}$$

10.13.3 Optimization

An optimal value for parameter vector $\mathbf{b}$ may be found using `Regression`.

```
    @param x_  the training/full data/input matrix
    @param y_  the training/full response/output vector

    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
        val z = f.fM (a *  x_)                          // layer 1->2: Z   = f(XA)
        val reg = new Regression (z, y_)                // layer 2-3: delegate to ‘Regression
    ‘
        reg.train ()
        b = reg.parameter
    end train
```

Extreme Learning Machines typically are competitive with lower order polynomial regression and may have fewer parameters than `CubicXRegression`.

10.13.4 ELM_3L1 Class

Class Methods:

```
    @param x        the m-by-n input matrix (training data consisting of m input vectors)
    @param y        the m output vector (training data consisting of m output scalars)
    @param fname_   the feature/variable names (if null, use x_j s)
    @param nz       the number of nodes in hidden layer (-1 => use default formula)
    @param hparam   the hyper-parameters for the model/network
    @param f        the activation function family for layers 1->2 (input to hidden)
    @param itran    the inverse transformation function returns responses to original scale

    class ELM_3L1 (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                   private var nz: Int = -1, hparam: HyperParameter = null,
                   f: AFF = f_tanh, val itran: FunctionV2V = null)
          extends Predictor (x, y, fname_, hparam)
             with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

    def compute_df_m (nz_ : Int): Int = nz_
    def parameters: VectorD = b
    def train (x_ : MatrixD = x, y_ : VectorD = y): Unit =
    def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
    override def predict (z: VectorD): Double = b dot f.f_ (a dot z)
    override def predict (z: MatrixD = x): VectorD = f.fM (a * z) * b
    def buildModel (x_cols: MatrixD): ELM_3L1 =
```

When the second activation function is not `id`, then the optimization of the second set of parameters works like it does for Transformed Regression. Also, when multiple outputs are needed, the `ELM_3L` may be used.

10.13.5 Exercises

1. Create an `ELM_3L1` model to predict values for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`, (c) `NeuralNet_2L`, (d) `NeuralNet_3L`, (e) `NeuralNet_XL`

2. Time each of the six model given above using ScalaTion's `time` method (in the `scalation` package).

   ```
       def time [R] (block: => R): R =
   ```

   This method can time the execution of any block of code (`time { block }`).

3. Compare the following strategies for initializing `NetParam` A (weights and biases for the first layer).

   (a) randomly generated weights

(b) binary sparse weights

(c) ternary sparse weights

For an explanation of these three weight initialization schemes, see `https://core.ac.uk/download/pdf/301130498.pdf`.

Chapter 11

Time Series/Temporal Models

For *time series forecasting*, time series/temporal models are used in order to make forecasts into the future. Applications are numerous and include weather, sales projections, energy consumption, economic indicators, financial instruments and traffic conditions. For *time series classification*, related models are applied to *sequential data* that includes Speech Recognition and Natural Language Processing (NLP).

Until this chapter, predictive models have been of the form

$$y = f(\mathbf{x}; \mathbf{b}) + \epsilon$$

where $\mathbf{x}$ is the vector of predictive variables/features, y is the response variable and ϵ is the residual/error.

In order to fit the parameters $\mathbf{b}$, m samples are collected into a data/input matrix X and a response/output vector $\mathbf{y}$. The samples are treated as though they are independent. In many case, such as data collected over time, they are often not independent. For example, the current Gross Domestic Product (GDP) will likely show high dependency upon the previous quarter's GDP. If the model is to forecast the next quarter's GDP, surely it should take into account the current (or recent) GDP values.

General Time Series Model

One may think about forecasting as follows: Given that you know today's high temperature y_t, predict (or forecast) tomorrow's high temperature y_{t+1}. Of course, using historical data (past temperatures) would be likely to improve the quality of the forecasts.

To begin with, one could focus on forecasting the value of response y at time $t+1$ as a function of current and past (or lagged) values of y, e.g.,

$$\boxed{y_{t+1} \;=\; f(y_t, y_{t-1}, \dots, y_{t-p'}; \,\boldsymbol{\phi}) \;+\; \epsilon_{t+1}} \tag{11.1}$$

This indicates the forecasted value is a function the most recent p (back $p' = p - 1$ to 0 time units) values. In vector form, the equation becomes,

$$\boxed{y_{t+1} \;=\; f(\mathbf{y}_{t-p':t}; \,\boldsymbol{\phi}) \;+\; \epsilon_{t+1}} \tag{11.2}$$

where f is a function of the slice of the past p values $\mathbf{y}_{t-p':t}$ and the parameter vector $\boldsymbol{\phi}$ is analogous to the $\mathbf{b}$ vector used in previous chapters.

The lags may also be dilated, e.g., with a *dilation rate* of 2,

$$[y_{t-2p+1}, \dots, y_{t-3}, y_{t-1}]$$

In addition to past values of the response vector, due to the time-oriented nature of the data, one can literally use past errors to improve forecasts. This is easier to rationalize if they are termed *shocks* or *innovations*, e.g., sometime unexpected happened that caused a forecast to be off. Obviously, other variables besides the response variable itself may be used. In some models, these are referred to as *exogenous* variables.

Some forecasting techniques such as Time Series Regression with Lagged Variables can select the features using for example forward selection, backward elimination or stepwise refinement.

Initially in this chapter, time will be treated as discrete time.

11.1 Forecaster

The `Forecaster` trait within the `scalation.modeling.forecasting` package provides a common framework for several forecasters. The key methods are the following:

- The `train` method must be called first as it matches the parameter values to patterns in the data. Training can occur on a training set or the full dataset/time series.
- The `test` method is used to assess the Quality of Fit of the trained model in making *predictions* (one time-unit ahead forecasts). It can be applied to the full dataset for in-sample testing, or to the testing set for out-of-sample testing.
- The `trainNtest` calls `train`, `test` and `report`.
- The `testF` method takes testing to the next level. Once predictions are judged to be satisfactory, the model can be assessed in terms of its ability to make longer term *forecasts*. Obviously, the farther into the future the forecasts are made, the greater the challenge. The quality of the forecasts are likely degrade as the *forecasting horizon* h increases.
- The `predict` method gives a one-step ahead forecast. Given the current time t (e.g., think of it as today) based on for example the current value y_t, past values, etc. predict the next (e.g., tomorrow's) value y_{t+1}.
- The `predictAll` method extends this over the times-series (or a sub-range of it)
- The `forecast` method is the analog of `predict`, but for multi-horizon forecasting, returning a vector of forecasts for 1 to h time units (e.g., days) ahead.
- The `forecastAt` method will provide all the forecasts for a particular forecasting horizon h.
- The `forecastAll` method extends this over all forecasting horizon from 1 up to and including h. The commonly used *recursive method* for forecasting into the future, requires forecasts to be generated and stored in a matrix (called `yf`) one horizon at a time. The first forecast is made entirely from actual data, while the next one dependent on the prior forecast. Depending on how many past values are used in the model, eventually the recursive methods will produce forecasts solely based on forecasted values. The alternative, *direct method*, will be discussed later.

`Forecaster` **Trait**

Trait Methods:

```
@param y        the response vector (time series data)
@param tt       the time vector, if relevant (index as time may suffice)
@param hparam   the hyper-parameters for models extending this trait

trait Forecaster (y: VectorD, tt: VectorD = null, hparam: HyperParameter = null)
      extends Model:

def cap: Int = 1                                            // how far into the past
def getY: VectorD = y
```

```scala
    def getFname: Array [String] = Array ("no-x features")
    def train (x_null: MatrixD, y_ : VectorD): Unit
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD)
    protected def testSetup (y_ : VectorD, doPlot: Boolean = true): (VectorD, VectorD) =
    def trainNtest (y_ : VectorD = y)(yy: VectorD = y): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD)
    protected def testSetupF (y_ : VectorD, h: Int, doPlot: Boolean = true):
                              (VectorD, VectorD) =
    def hparameter: HyperParameter = hparam
    def parameter: VectorD = new VectorD (0)                    // vector with no elements
    def nparams: Int       = parameter.dim                      // number of parameters
    def residual: VectorD = { if e == null then flaw ("residual",
                               "must call test method first"); e }
    def predict (z: VectorD): Double =
    def predict (t: Int, y_ : VectorD): Double
    def predictAll (y_ : VectorD): VectorD =
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD
    def forecastAll (y_ : VectorD, h: Int): MatrixD =
    def forwardSel (cols: Set [Int], idx_q: Int = QoF.rSqBar.ordinal):
                  (Int, Forecaster) = ???
    def forwardSelAll (idx_q: Int = QoF.rSq.ordinal, cross: Boolean = false):
                     (Set [Int], MatrixD) =
```

11.1.1 `Stats4TS` Case Class

The following class is used to produce basic statistics about a time series, passed in as a vector. Even if the model will not end up using a large number of lags, it is a good idea to collect statistics for several lags to assess which ones may be important. The `Stats4TS` case class in the `scalation.mathstat` package defines the Auto-Correlation Function (ACF). It holds the mean, variance, the auto-covariance vector and the auto-correlation vector.

Class Methods:

```scala
    @param y       the response vector (time series data) for the training/full dataset
    @param lags_   the maximum number of lags

    case class Stats4TS (y: VectorD, lags_ : Int):

        val lags = min (y.dim-1, lags_)                    // lags can't exceed dataset size
        val mu   = y.mean                                  // sample mean
        val sig2 = y.variance                              // sample variance
        val acv  = new VectorD (lags + 1)                  // auto-covariance vector
        for k <- acv.indices do acv(k) = y acov k          // k-th lag auto-covariance
        val acr  = acv / acv(0)                            // auto-correlation function
        override def toString: String =

    end Stats4TS
```

11.1.2 Auto-Correlation Function

To better understand the dependencies in the data, it is useful to look at the auto-correlation. Consider the following time series data used in forecasting lake levels recorded in the Lake Level Times-series Dataset. (see `cran.r-project.org/web/packages/fpp/fpp.pdf`):

```
val m = 98
val t = VectorD.range (0, m)
val y = VectorD (580.38, 581.86, 580.97, 580.80, 579.79, 580.39, 580.42, 580.82, 581.40, 581.32,
                 581.44, 581.68, 581.17, 580.53, 580.01, 579.91, 579.14, 579.16, 579.55, 579.67,
                 578.44, 578.24, 579.10, 579.09, 579.35, 578.82, 579.32, 579.01, 579.00, 579.80,
                 579.83, 579.72, 579.89, 580.01, 579.37, 578.69, 578.19, 578.67, 579.55, 578.92,
                 578.09, 579.37, 580.13, 580.14, 579.51, 579.24, 578.66, 578.86, 578.05, 577.79,
                 576.75, 576.75, 577.82, 578.64, 580.58, 579.48, 577.38, 576.90, 576.94, 576.24,
                 576.84, 576.85, 576.90, 577.79, 578.18, 577.51, 577.23, 578.42, 579.61, 579.05,
                 579.26, 579.22, 579.38, 579.10, 577.95, 578.12, 579.75, 580.85, 580.41, 579.96,
                 579.61, 578.76, 578.18, 577.21, 577.13, 579.10, 578.25, 577.91, 576.89, 575.96,
                 576.80, 577.68, 578.38, 578.52, 579.74, 579.31, 579.89, 579.96)
```

The data also available in the `scalation.modeling.forecasting` package.

```
    import Example_LakeLevels.{t, y}
```

First plot this dataset and then look at its Auto-Correlation Function (ACF).

```
    new Plot (t, y, null, "Plot of y vs. t", true)
```

The Auto-Correlation Function (ACF) measures how much the past can influence a forecast. If the forecast is for time $t+1$, then the current/past time points are $t, t-1, \dots, t-p'$. The k-lag auto-covariance (auto-correlation), γ_k (ρ_k) is the covariance (correlation) of y_t and y_{t-k}.

$$\gamma_k = \mathbb{C}\,[y_t, y_{t-k}] \tag{11.3}$$

$$\rho_k = \text{corr}(y_t, y_{t-k}) \tag{11.4}$$

Note that $\gamma_0 = \mathbb{V}\,[y_t]$ and $\rho_k = \gamma_k/\gamma_0$. These equations assume the stochastic process $\{y_t | t \in [0, m-1]\}$ is *covariance stationary* (see exercises).

Although vectors need not be created, to compute $\text{corr}(y_t, y_{t-2})$ one could imagine computing the correlation between `y(2 until m)` and `y(0 until m-2)`. In ScalaTion, the ACF is provided by the `acF` function from the `Correlogram` trait in the `scalation.mathstat` package.

```
@main def correlogramTest (): Unit =

    val y = VectorD (1, 2, 5, 8, 3, 6, 9, 4, 5, 11,
                     12, 16, 7, 6, 13, 15, 10, 8, 14, 17)

    object CT extends Correlogram (y)

    banner ("Plot Data")
    new Plot (null, y, null, "y vs. t", lines = true)

    banner ("Test Correlogram")
    CT.makeCorrelogram ()
    val acf  = CT.acF
    val pacf = CT.pacF
    println (s"acF = $acf")
    println (s"pacF = $pacf")
    CT.plotFunc (acf, "ACF")
    CT.plotFunc (pacf, "PACF")

end correlogramTest
```

The first point in plot is the auto-correlation of y_t with itself, while the rest of the points are ACF(k), the k-lag auto-correlation.

11.1.3 Correlogram

A correlogram shows how a time series correlates with lagged versions of itself and allows one to visualize the important dependencies in the time series.

`Correlogram` Trait

Trait Methods:

```
    @param y  the time series data (response vector)

    trait Correlogram (y: VectorD):

    def makeCorrelogram (y_ : VectorD = y): Unit =
    def acF: VectorD = stats.acr
    def pacF: VectorD = pacf
    def psiM: MatrixD = psi
    def statsF: Stats4TS = stats
    def durbinLevinson (g: VectorD, ml: Int): MatrixD =
    def plotFunc (fVec: VectorD, name: String, show: Boolean = true): Unit =
```

11.1.4 Quality of Fit (QoF) for Time Series Data

In addition to the `Forecaster` trait, models must extend the `Fit` trait and so several QoF measures are available, including R^2, $\bar{R}^2$, MAE, MSE, RMSE, and AIC.

Five QoF measures that are widely used in time series analysis are the following (assuming y_t and $\hat{y}_t$ are properly aligned): The first two are in the units of the response variable (e.g., meters), while the last three are relative measures (e.g., MAPE ranges from 0 to 100% and sMAPE ranges from 0 to 200%). In all cases, the smaller the value the better.

1. Mean Absolute Error (MAE)

$$\text{MAE} = \frac{1}{m}\sum_{t=0}^{m-1} |y_t - \hat{y}_t| \tag{11.5}$$

Problem: If y_t is large, a difference of 10 may be negligible.

2. Root Mean Squared Error (RMSE)

$$\text{RMSE} = \sqrt{\left[\frac{1}{m}\sum_{t=0}^{m-1} (y_t - \hat{y}_t)^2\right]} \tag{11.6}$$

Problem: Same as for MAE plus can be overly influenced by outliers (or near outliers).

3. Mean Absolute Percentage Error (MAPE)

$$\text{MAPE} = \frac{100}{m}\sum_{t=0}^{m-1} \frac{|y_t - \hat{y}_t|}{|y_t|} \tag{11.7}$$

Problem: If some values for y_t are 0, it will return infinity. Also, an upward shift of the values will reduce/improve the score (e.g, changing from Celsius to Kelvins)

4. symmetric Mean Absolute Percentage Error (sMAPE)

$$\text{sMAPE} = \frac{200}{m}\sum_{t=0}^{m-1} \frac{|y_t - \hat{y}_t|}{|y_t| + |\hat{y}_t|} \tag{11.8}$$

Problem: Same problem, but to a lesser degree, as both y_t and $\hat{y}_t$ need to be zero. Also, an upward shift of the values will reduce the score.

5. Mean Absolute Scaled Error (MASE)

$$\text{MASE} = \frac{MAE}{MAE\_n} \tag{11.9}$$

where the denominator is the MAE for the Naïve Model (Simple Random Walk) that is used as the standard baseline model to compare other models with.

$$\text{MAE\_n} = \frac{1}{m-1}\sum_{t=1}^{m-1} |y_t - y_{t-1}| \tag{11.10}$$

While the standard baseline model for prediction is the Null Model (guess the mean), it tends not to perform well for time series forecasting, so the Naïve Model (guess the previous value) is used in its place. Therefore, MASE measures how well the forecasting model works relative to the Naïve Model. When MASE is 1, the model performance is on par with the Naïve Model, values below 1 are better and values above 1 are worse. Note, for long horizon forecasts MASE values above 1 are likely to happen.

11.2 Baseline Models: Random Walk, Null and Trend Models

There are three simple baseline models for time series data: the Random Walk, the Null Model and the Trend Model. These models are very simple and serve as baselines for other forecasting models to compete against.

11.2.1 Random Walk Model

The physical property of *inertia* would suggest that many processes may be accurately modeled as a Random Walk, at least in the short run. The Random Walk Model (or Naïve Model) states the future value y_{t+1} may be modeled as the current value y_t disturbed by white noise ϵ_{t+1}.

$$y_{t+1} \;=\; y_t \,+\, \epsilon_{t+1} \tag{11.11}$$

Notice that this model has no parameters to estimate and as such is considered to be a baseline model for more sophisticated models to improve upon. If they cannot, their value is questionable.

11.2.2 White Noise

A stochastic process $\{\epsilon_t | t \in \{0, \dots, m-1\}\}$ is said to be white noise if

$$\begin{aligned}
\mathbb{E}\left[\epsilon_t\right] &= 0 && \text{zero mean} && (11.12)\\
\mathbb{E}\left[{\epsilon_t}^2\right] &= \sigma^2 && \text{constant variance} && (11.13)\\
\mathbb{E}\left[\epsilon_t \epsilon_{t-k}\right] &= 0 \qquad \text{for} \quad k \geq 1 && \text{uncorrelated} && (11.14)
\end{aligned}$$

Several distributions may be used to generate white noise with the most common being the Gaussian (Normal) distribution, $\mathbf{N}(\mathbf{0}, \sigma^2 I)$.

11.2.3 Detecting Random Walks

To determine whether a stochastic process is a random walk, take its first difference.

$$y'_t \;=\; y_t - y_{t-1} \;=\; \epsilon_t \tag{11.15}$$

Then apply the Ljung-Box Test to see if it qualifies as white noise.

$$q \;=\; m(m+2) \sum_{k=1}^{h} \frac{\rho_k^2}{m-k} \;\sim\; \chi_h^2 \tag{11.16}$$

where ρ_k is the k-lag autocorrelation and m is the length of the time series. Large values for q indicate significant autocorrelation. Hyndman [84] recommends that the number of lags h be 10 for non-seasonal processes. See the section on SARIMA for a discussion of seasonal models.

11.2.4 `RandomWalk` Class

Class Methods:

```
    @param y        the response vector (time series data)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters (none => use null)

    class RandomWalk (y: VectorD, tt: VectorD = null, hparam: HyperParameter = null)
          extends Forecaster (y, tt, hparam)
             with Correlogram (y)
             with Fit (dfm = 1, df = y.dim - 1):

    def train (x_null: MatrixD, y_ : VectorD): Unit =
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
    def predict (t: Int, y_ : VectorD): Double = y_(t)
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
```

11.2.5 Null Model

A simpler baseline modeling technique that is typically not as accurate as `RandomWalk` is the `NullModel`. The Null Model (or Mean Model) simply always forecasts future values to be equal to the training mean. When the process mean is stable over time, this a reasonable baseline model to consider.

$$y_{t+1} \;=\; \mu_y \,+\, \epsilon_{t+1} \tag{11.17}$$

where μ_y is the mean of the response variable estimated from the training set. For in-sample assessment it is computed over the full dataset. For out-of-sample assessment using rolling validation, each retraining may produce a slightly different value for the mean. See the section on Rolling Validation.

11.2.6 `NullModel` Class

Class Methods:

```
    @param y        the response vector (time series data)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters (none => use null)

    class NullModel (y: VectorD, tt: VectorD = null, hparam: HyperParameter = null)
          extends Forecaster (y, tt, hparam)
             with Correlogram (y)
             with Fit (dfm = 0, df = y.dim - 1):

    def train (x_null: MatrixD, y_ : VectorD): Unit =
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
    override def parameter: VectorD = VectorD (mu)
    def predict (t: Int, y_ : VectorD): Double = mu
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
```

11.2.7 Trend Model

When the process mean changes over time, a (Linear) Trend Model will usually work better than the Null Model. While the Null Model produces a forecasting line with slope 0, as its generalization, the Trend Model allows positive (trending up) or negative slopes (trending down). The predictions/forecasts are made based on the following linear model of time t.

$$y_{t+1} \;=\; b_0 \,+\, b_1(t+1) \,+\, \epsilon_{t+1} \tag{11.18}$$

where b_0 is the intercept and b_1 is the slope. The `SimpleRegression` class can be used to estimate the parameter/coefficient vector $\mathbf{b} = [b_0, b_1]$.

Of course, the more sophisticated trend models (e.g., quadratic) may be applied, but these are usually not considered baseline. Also, care must be taken for these complex models when applied over long forecasting horizons as higher order terms may become unrealistically large.

11.2.8 `TrendModel` Class

Class Methods:

```
    @param y       the response vector (time series data)
    @param tt      the time vector (required for the trend model)
    @param hparam  the hyper-parameters (none => use null)

    class TrendModel (y: VectorD, tt: VectorD, hparam: HyperParameter = null)
          extends Forecaster (y, tt, hparam)
             with Correlogram (y)
             with Fit (dfm = 1, df = y.dim - 1):

    def train (x_null: MatrixD, y_ : VectorD): Unit =
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
    override def parameter: VectorD = b
    def predict (t: Int, y_ : VectorD): Double = reg.predict (VectorD (1, t))
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
```

11.2.9 Forecasting Lake Levels - Battle of the Baselines

As a simple example, run and compare the three baseline modeling techniques, `RandomWalk`, `NullModel` and `TrendModel` on the Lake Level Time Series Dataset. The three baseline models are compared based on their in-sample Quality of Fit (Qof) measures. See the exercises for out-of-sample quality assessment.

```scala
@main def randomWalkTest2 (): Unit =

    import Example_LakeLevels.{t, y}

    banner (s"Tests: RandomWalk on LakeLevels Dataset")
    val mod = new RandomWalk (y)                              // create a Random Walk Model
    mod.trainNtest ()()                                       // train-test model on full data

    banner (s"Tests: NullModel on LakeLevels Dataset")
    val nm = new NullModel (y)                                // create a Null Model
    nm.trainNtest ()()                                        // train-test model on full data

    banner (s"Tests: TrendModel on LakeLevels Dataset")
    val tm = new TrendModel (y)                               // create a Trend Model
    tm.trainNtest ()()                                        // train-test model on full data

    banner ("Select model based on ACF and PACF")
    mod.plotFunc (mod.acF, "ACF")                             // Auto-Correlation Function (ACF)
    mod.plotFunc (mod.pacF, "PACF")                           // Partial Auto-Correlation Func.

end randomWalkTest2
```

Note, computation of `mase` requires the first value of **y** (`y(0)`) which is eliminated by the `test` method as it aligns the time series `y` and `yp`. Hence, `mase` is computed separately from the other QoF measures.

Results from Random Walk

In the plot, the forecasts (in red) simply follow the actual values (in black) with a one-step delay. The report for `RandomWalk` shows the quality of fit.

```
REPORT
    ----------------------------------------------------------------------------
    modelName  mn  = RandomWalk
    ----------------------------------------------------------------------------
    hparameter hp  = null
    ----------------------------------------------------------------------------
    features   fn  = Array(no-x features)
    ----------------------------------------------------------------------------
    parameter  b   = VectorD()
    ----------------------------------------------------------------------------
    fitMap     qof = LinkedHashMap(rSq -> 0.676806, rSqBar -> 0.673440, sst -> 166.664699,
    sse -> 53.865000, mse0 -> 0.555309, rmse -> 0.745191, mae -> 0.585567, dfm -> 1.000000,
    df -> 96.000000, fStat -> 201.035387, aic -> -105.107880, bic -> -99.958458,
    mape -> 0.101146, smape -> 0.101154)
    ----------------------------------------------------------------------------

    mase = 1.0
```

In particular, the R^2 is 0.676806, the Mean Absolute Error (MAE) is 0.585567, the Mean Absolute Percentage Error (MAPE) is 0.101146, the symmetric MAPE is 0.101154, and the Mean Absolute Scaled Error (MASE) is 1.0 (as expected).

Results from Null Model

In the plot, the forecasts (in red) are a flat line that cuts the actual values (in black) in the middle (mean). The report for `NullModel` shows a reduced quality of fit.

```
REPORT
    ----------------------------------------------------------------------------
    modelName  mn  = NullModel
    ----------------------------------------------------------------------------
    hparameter hp  = null
    ----------------------------------------------------------------------------
    features   fn  = Array(no-x features)
    ----------------------------------------------------------------------------
    parameter  b   = VectorD(578.990)
    ----------------------------------------------------------------------------
    fitMap     qof = LinkedHashMap(rSq -> 0.000000, rSqBar -> 0.000000, sst -> 166.664699,
    sse -> 166.664699, mse0 -> 1.718193, rmse -> 1.310799, mae -> 1.057230, dfm -> 1.000000,
    df -> 96.000000, fStat -> 0.000000, aic -> -159.888779, bic -> -154.739357,
    mape -> 0.182630, smape -> 0.182614)
    ----------------------------------------------------------------------------

    mase = 1.805481
```

In this case, the R^2 is 0.0 (as expected), the Mean Absolute Error (MAE) is 1.057230, the Mean Absolute Percentage Error (MAPE) is 0.182630, the symmetric MAPE is 0.182614, and the Mean Absolute Scaled Error (MASE) is 1.805481.

Results from Trend Model

In the plot, the forecasts (in red) are a flat line that cuts the actual values (in black) in the middle (mean). The report for `TtendModel` shows a reduced quality of fit.

```
REPORT
    ----------------------------------------------------------------------------
    modelName  mn  = TrendModel
    ----------------------------------------------------------------------------
    hparameter hp  = null
    ----------------------------------------------------------------------------
    features   fn  = Array(no-x features)
    ----------------------------------------------------------------------------
    parameter  b   = VectorD(580.169,-0.0240708)
    ----------------------------------------------------------------------------
    fitMap     qof = LinkedHashMap(rSq -> 0.264379, rSqBar -> 0.264379, sst -> 166.664699,
    sse -> 122.602045, mse0 -> 1.263939, rmse -> 1.124250, mae -> 0.920808, dfm -> 1.000000,
    df -> 96.000000, fStat -> 34.501992, aic -> -144.997325, bic -> -139.847903,
    mape -> 0.159076, smape -> 0.159073)
```

```
----------------------------------------------------------------------------

mase = 1.572507
```

For the final baseline model, the R^2 is 0.264379, the Mean Absolute Error (MAE) is 0.920808, the Mean Absolute Percentage Error (MAPE) is 0.159076, the symmetric MAPE is 0.159073, and the Mean Absolute Scaled Error (MASE) is 1.572507

Figure 11.1 shows in-sample, one-step ahead forecasting plots, where the actual values `y` are shown in red, Random Walk forecasts `y-RW` are shown in green, Null Model forecasts `y-NM` are shown in blue, and Trend Model forecasts `y-TM` are shown in black.

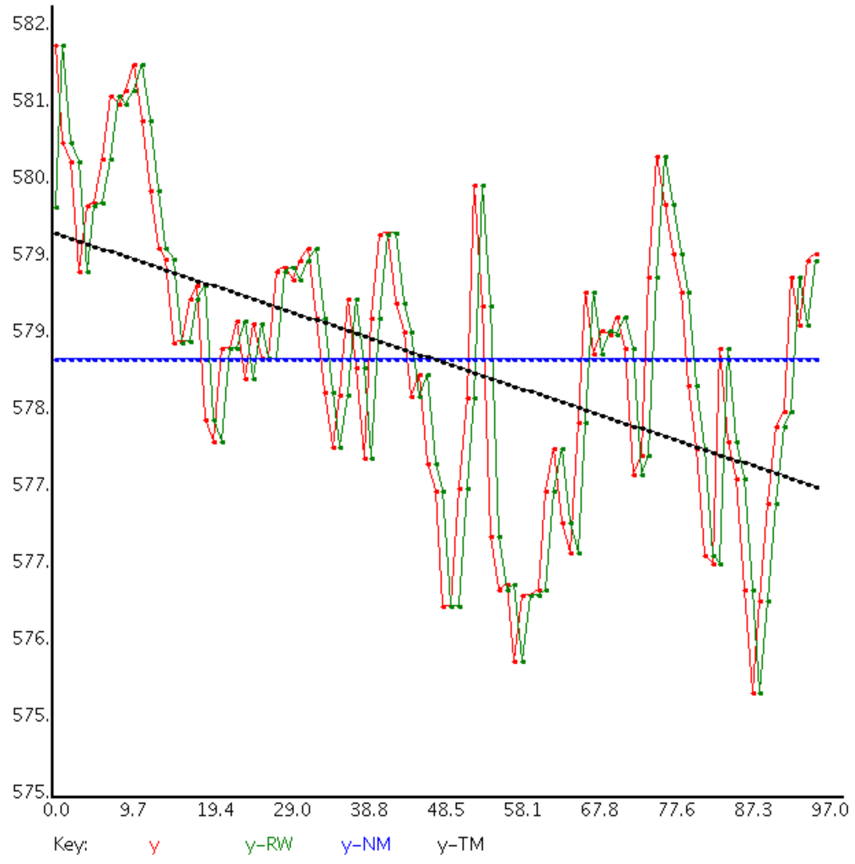


Figure 11.1: Comparison of Baseline Time Series Models: Lake Level vs. Year

Notice that `y-RW` follows the actual time series value with a lag of one time unit, while `y-NM` and `y-TM` are straight lines, with the former being a horizontal line at the mean value for the time series and the latter showing negative slope indicating a downward trend.

A useful time series model should have a MASE smaller than minimum of the values from the three baselines (`RandomWalk`, `NullModel` and `TrendModel`).

11.2.10 Exercises

1. Generate Gaussian white noise with a variance of one and plot it.

```scala
    val noise = Normal (0, 1)
    val y = VectorD (for i <- 0 until 100 yield noise.gen)
    new Plot (null, y, null, "white noise")
```

2. Create a Random Walk model for the above white noise process, plot the process value vs. the forecasted value and assess the Quality of Fit (QoF). Also, examine the Auto-Correlation Function (ACF) plot. What would it mean if a peak in the ACF plot was outside the error bands. Ignore the first value corresponding to the zeroth lag where the correlation ρ_0 should one and the partial correlation ψ_{00} should be zero.

```scala
    val mod = new RandomWalk (y)                    // time series model
    mod.trainNtest ()()                             // train-test model on full dataset
    mod.plotFunc (mod.acF, "ACF")
```

   Note: `plotFunc` includes error bands/confidence intervals.

3. For the Lake-Level Dataset, apply the Ljung-Box Test to see if y_t or ϵ_t are white noise. Determine the values for q for each case and the 95% critical value of χ^2_{10}.

```scala
    import scalation.variate.Quantile.chiSquareInv
    println (chiSquareInv (0.95, 10))
```

4. Give the model equations for a Random Walk with Drift. How is the drift parameter estimated? What are its advantages?

5. Explain the results from the three baseline models, Random Walk, Null and Trend Models, for out-of sample horizon $h = 1$ assessment show below. See the section on Rolling Validation for how out-of-sample QoF measures are produced.

   Random Walk: rollValidate: for horizon h = 1:

```
LinkedHashMap(rSq -> 0.511033, rSqBar -> 0.511033, sst -> 74.797020, sse -> 36.573300,
mse0 -> 0.746394, rmse -> 0.863941, mae -> 0.690000, dfm -> 1.000000, df -> 49.000000,
fStat -> 51.211192, aic -> -59.547204, bic -> -55.763564, mape -> 0.119283, smape -> 0.119301)
```

   Null Model: rollValidate: for horizon h = 1:

```
LinkedHashMap(rSq -> -0.668405, rSqBar -> -0.668405, sst -> 74.797020, sse -> 124.791686,
mse0 -> 2.546769, rmse -> 1.595860, mae -> 1.313594, dfm -> 1.000000, df -> 49.000000,
fStat -> -19.630623, aic -> -90.603986, bic -> -86.820346, mape -> 0.227405, smape -> 0.227085)
```

   Trend Model: rollValidate: for horizon h = 1:

```
LinkedHashMap(rSq -> -0.433676, rSqBar -> -0.463545, sst -> 74.797020, sse -> 107.234717,
mse0 -> 2.188464, rmse -> 1.479346, mae -> 1.255724, dfm -> 1.000000, df -> 48.000000,
fStat -> -14.519640, aic -> -89.187547, bic -> -85.403906, mape -> 0.216952, smape -> 0.217159)
```

6. Compare the three baseline models, Random Walk, Null and Trend Models, using out-of sample assessment for $h > 1$ via the `RollValidate` object. See the section on Rolling Validation. What can be said about the three baseline model as a group? What about the relative strengths of each baseline modeling technique as the forecasting horizon increases?

7. Two of the three baseline models can be unified using the Simple Moving Average Model. While `RandomWalk` takes the last value as the forecast, `SimpleMovingAverage` takes the average the last q-values as the forecast.

```scala
    @param y        the response vector (time series data)
    @param tt       the time points, if needed
    @param hparam   the hyper-parameters

   class SimpleMovingAverage (y: VectorD, tt: VectorD = null,
                              hparam: HyperParameter = SimpleMovingAverage.hp)
         extends Forecaster (y, tt, hparam)
            with Correlogram (y)
            with Fit (dfm = 1, df = y.dim - 1):
```

Show that for $q = 1$, `SimpleMovingAverage` gives the same results as `RandomValk`, while as q becomes large, it approximates the results of the Mean Model, i.e., `NullModel`.

8. Consider a stochastic process generated from a Random Walk with $y_0 = 0$ and $\epsilon_{t+1} \sim N(0, \sigma^2)$

$$y_{t+1} \;=\; y_{t-1} + \epsilon_{t+1} \tag{11.19}$$

Show that $\mathbb{E}[y_t] = 0$ and $\mathbb{V}[y_t] = t\,\sigma^2$. The fact that the variance grows linearly in time shows that an Random Walk is not a covariance stationary process (see the section on ARIMA).

11.3 Simple Exponential Smoothing

The basic idea of Simple Exponential (SES) is to start with Random Walk,

$$\hat{y}_{t+1} \;=\; y_t \tag{11.20}$$

but stabilize/smooth out the forecast by maintaining a state variable the summarizes the past.

The state variable s_{t+1} is a weighted combination of the most recent value y_t and the most recent value of the state variable s_t. The forecasted value $\hat{y}_{t+1}$ is taken as the value of the state variable s_{t+1}.

$$s_{t+1} \;=\; \alpha\, y_t + (1-\alpha)s_t \tag{11.21}$$

$$\hat{y}_{t+1} \;=\; s_{t+1} \tag{11.22}$$

The state variable may be initialized as the initial value of the time series (other options include taking an average or letting it be a second parameter passed to an optimizer).

$$s_0 \;=\; y_0 \tag{11.23}$$

The parameter $\alpha \in [0,1]$ is referred to as the smoothness parameter. When $\alpha = 1$, SES becomes Random Walk as older values in the time series are ignored. When $\alpha = 0$, SES the state remains locked at its initial value. Thus, $\alpha \in (0,1]$ is a more practical range.

The reason it is called exponential smoothing is that the impact of older values in the time series decreases exponentially.

11.3.1 Model Equation

The model equation indicates that next forecasted value (one step ahead) is the value of the state variable plus a random element/shock.

$$y_{t+1} \;=\; s_{t+1} \,+\, \epsilon_{t+1} \tag{11.24}$$

11.3.2 Training

For SES, one may minimize the sum of squared errors (sse). The error at time t is given by

$$\epsilon_t \;=\; y_t \,-\, \hat{y}_t \tag{11.25}$$

In vector form $\boldsymbol{\epsilon} = [\epsilon_0 \ldots, \epsilon_{m-1}]$, the equation becomes,

$$\boldsymbol{\epsilon} \;=\; \mathbf{y} \,-\, \hat{\mathbf{y}} \tag{11.26}$$

A loss function $\mathcal{L}(\alpha) = sse = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}$ may be minimized to determine the parameter α (alternatively, both s_0 and α).

$$\mathcal{L}(\alpha) \;=\; (\mathbf{y} \,-\, \hat{\mathbf{y}}) \cdot (\mathbf{y} \,-\, \hat{\mathbf{y}}) \tag{11.27}$$

In ScalaTion, the `train` method is used to find an optimal value for α (`a`).

```
    override def train (x_null: MatrixD, y_ : VectorD): Unit =

       def f_obj (x: VectorD): Double = (y_ - smooth (x(0), y_)).normSq

       if opt then
           val optimizer = new L_BFGS_B (f_obj, l = lo, u = up)   // Quasi-Newton optimizer
           val opt = optimizer.solve (VectorD (a), toler = TOL)   // optimize value for a
           a = (opt._2)(0)                                        // pull from result
       end if
       s = smooth (a)                                             // predicted values
   end train
```

y_ holds the vector of y_t values, while `smooth` calculates $\hat{y}_t$

11.3.3 Effect of the Smoothing Parameter

As the smoothing parameter α get smaller, the forecast curve becomes more smooth, as shown in the plot produced by the following code applied to the Lake Level database.

```
@main def simpleExpSmoothingTest5 (): Unit =

    import Example_LakeLevels.{t, y}

    val mod = new SimpleExpSmoothing (y)                  // time series model
    mod.toggleOpt ()                                      // switch auto optimization off

    for i <- 0 to 5 do
        val a = i.toDouble / 5.0
        banner (s"Build SimpleExpSmoothing model with a = $a")
        mod.reset (a)
        mod.train (null, y)                               // train the model on full dataset
        val (yp, qof) = mod.test (null, y)                // test the model on full dataset
        println (mod.report (qof))                        // report on Quality of Fit (QoF)
    end for

end simpleExpSmoothingTest5
```

11.3.4 `SimpleExpSmoothing` Class

Class Methods:

```
    @param y       the response vector (original time series data)
    @param tt      the time vector, if relevant (time index may suffice)
    @param hparam  the hyper-parameters

    class SimpleExpSmoothing (y: VectorD, tt: VectorD = null,
                              hparam: HyperParameter = SimpleExpSmoothing.hp)
          extends Forecaster (y, tt, hparam)
             with Correlogram (y)
             with Fit (dfm = 1, df = y.dim - 1):

    def reset (a: Double): Unit = α = a
    def toggleOpt (): Unit = opt = ! opt
```

```scala
    def smooth (a: Double = α, y_ : VectorD = y): VectorD =
    def train (x_null: MatrixD, y_ : VectorD): Unit =
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
    override def parameter: VectorD = VectorD (α)
    def predict (t: Int, y_ : VectorD): Double = s(t+1)
    override def predictAll (y_ : VectorD): VectorD = s
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAll (h: Int, y_ : VectorD): MatrixD =
```

11.3.5 Exercises

1. Test customized ($\alpha = 0.5$) vs. optimized (α optimizer determined) smoothing on the following synthetic data.

```scala
@main def simpleExpSmoothingTest3 (): Unit =

    val m = 50
    val r = Random ()
    val y = VectorD (for i <- 0 until m yield i + 10.0 * r.gen)

    val mod = new SimpleExpSmoothing (y)              // smooth time series data: y vs.
     t

    banner ("Customized Simple Exponential Smoothing")
    mod.smooth (0.5)                                  // use customized parameters, don
    't train
    val (yp, qof) = mod.test (null, y)                // test the model on full dataset
    println (mod.report (qof))                        // report on Quality of Fit (QoF)
    println (s"mase = ${Fit.mase (y, yp)}")

    banner ("Optimized Simple Exponential Smoothing")
    mod.train (null, y)                               // train to use optimal α
    val (yp2, qof2) = mod.test (null, y)              // test the model on full dataset
    println (mod.report (qof2))                       // report on Quality of Fit (QoF)
    println (s"mase = ${Fit.mase (y, yp2)}")

end simpleExpSmoothingTest3
```

2. For some optimization software it is necessary to pass in an objective function and derivative function(s). For this type of optimization, $\hat{\mathbf{y}}$ needs to be replaced.

$$\hat{y}_t \;=\; \alpha y_{t-1} + \alpha(1-\alpha)y_{t-2} + \alpha(1-\alpha)^2 y_{t-3} + \dots \tag{11.28}$$

As a sum this becomes,

$$\hat{y}_t \;=\; \alpha \sum_{k=0}^{t-1} (1-\alpha)^k y_{t-1-k} \tag{11.29}$$

Therefore, the loss function is expressed as a double sum.

$$\mathcal{L}(\alpha) \;=\; \sum_{t=0}^{m-1} \left[y_t - \alpha \sum_{k=0}^{t-1} (1-\alpha)^k y_{t-1-k} \right]^2 \tag{11.30}$$

Create a formula for the derivative of $\mathcal{L}(\alpha)$ and pass the function and its derivative into the `NewtonRaphson` class found in the `scalation.optimization` package to find an optimal value for α.

11.4 Auto-Regressive (AR) Models

In order to predict future values for a response variable y, the obvious thing to do is to find variables that may influence the value of y. The most obvious is prior (or lagged) values of itself. Often, other variables may be helpful as well, some of which may be time series themselves. Given the variety of information potentially available, it should not be surprising that there are numerous modeling techniques used for time series forecasting [20, 84].

One of the simplest types of forecasting models of the form given in the first section of this chapter is to make the future value y_{t+1} be linearly dependent on the last p values of y. In particular, a p^{th}-order Auto-Regressive AR(p) model predicts the next value y_{t+1} from the sum of the last p values each weighted by its own coefficient/parameter ϕ_j,

$$\boxed{y_{t+1} \;=\; \delta \,+\, \boldsymbol{\phi} \cdot^{\leftarrow} \mathbf{y}_{t-p':t} \,+\, \epsilon_{t+1}} \tag{11.31}$$

where

- $p' \;=\; p-1$
- $\cdot^{\leftarrow}$ is the dot product with the second argument reversed,
- the parameter vector $\boldsymbol{\phi} \;=\; [\phi_0, \dots, \phi_{p'}]$,
- the slice of the past p values $\mathbf{y}_{t-p':t-1} \;=\; [y_{t-p'}, \dots, y_t]$,
- the constant/intercept/drift $\delta \;=\; \mu(1 - \mathbf{1}\cdot\boldsymbol{\phi})$, and
- the error/noise represented by ϵ_{t+1}.

This equation may be expanded out into

$$\boxed{y_{t+1} \;=\; \delta \,+\, \phi_0 y_t \,+\, \phi_1 y_{t-1} \,+\, \dots \,+\, \phi_{p'} y_{t-p'} \,+\, \epsilon_{t+1}} \tag{11.32}$$

ϵ_t represents noise that shocks the system at each time step. We require that $\mathbb{E}[\epsilon_t] = 0$, $\mathbb{V}[\epsilon_t] = \sigma^2_\epsilon$, and that all the noise shocks be independent.

Zero-Centered

To better capture the dependency, the data can be zero-centered, which can be accomplished by subtracting the mean μ, $z_t = y_t - \mu$.

$$z_{t+1} \;=\; \phi_0 z_t \,+\, \phi_1 z_{t-1} \,+\, \dots \,+\, \phi_{p'} z_{t-p'} \,+\, \epsilon_{t+1} \tag{11.33}$$

Notice that since z_t is zero-centered, δ drops out (see exercises) and the formulas for the mean, variance and covariance are simplified.

$$\mathbb{E}[z_t] \;=\; 0 \tag{11.34}$$
$$\mathbb{V}[z_t] \;=\; \mathbb{E}\left[{z_t}^2\right] \;=\; \gamma_0 \tag{11.35}$$
$$\mathbb{C}[z_t, z_{t-k}] \;=\; \mathbb{E}[z_t z_{t-k}] \;=\; \gamma_k \tag{11.36}$$

Recall that covariance $\mathbb{C}[x, y] = \mathbb{E}[(x - \mu_x)(y - \mu_y)]$.

11.4.1 AR(1) Model

When the future is mainly dependent only on the most recent value, e.g., ρ_1 is high and rest ρ_2, ρ_3, etc. are decaying, then an AR(1) model may be sufficient.

$$\boxed{z_{t+1} \;=\; \phi_0 z_t \,+\, \epsilon_{t+1}} \tag{11.37}$$

An estimate for the parameter ϕ_0 may be determined from the Auto-Correlation Function (ACF).

Solving for the ϕ_0 Parameter

Take this equation from z_t and multiply it by z_{t-k}.

$$z_t z_{t-k} \;=\; \phi_0 z_{t-1} z_{t-k} \,+\, \epsilon_t z_{t-k}$$

Taking the expected value of the above equation gives,

$$\mathbb{E}\left[z_t z_{t-k}\right] \;=\; \phi_0 \mathbb{E}\left[z_{t-1} z_{t-k}\right] + \mathbb{E}\left[\epsilon_t z_{t-k}\right] \tag{11.38}$$

Using the definition for γ_k given above for $k \geq 1$, this can be rewritten as

$$\gamma_k \;=\; \phi_0 \gamma_{k-1} \,+\, 0$$

The zero is due to the fact that $\mathbb{E}\left[\epsilon_t z_{t-k}\right] = \mathbb{C}\left[\epsilon_t, z_{t-k}\right]$ and past value z_{t-k} is independent of future noise shock ϵ_t. Now dividing by the variance γ_0 yields

$$\rho_k \;=\; \phi_0 \rho_{k-1} \tag{11.39}$$

An estimate for parameter ϕ_0 may be easily determined by simply setting $k = 1$ in the above equation.

$$\phi_0 \;=\; \rho_1 \tag{11.40}$$

Furthermore, ρ_0 is 1 and

$$\rho_k \;=\; \phi_0^k \qquad \text{for } \; |\phi_0| < 1 \tag{11.41}$$

Solving for the Noise Variance

When $k = 0$, multiplying by z_t gives,

$$\mathbb{E}\left[z_t z_t\right] \;=\; \phi_0 \mathbb{E}\left[z_{t-1} z_t\right] \,+\, \mathbb{E}\left[\epsilon_t z_t\right] \tag{11.42}$$

Using gamma notation and replacing z_t with $\hat{z}_t + \epsilon_t$ yields

$$\gamma_0 \;=\; \phi_0 \gamma_1 \,+\, \mathbb{E}\left[\epsilon_t(\hat{z}_t + \epsilon_t)\right]$$

Since $\hat{z}_t$ is independent of ϵ_t, the last term becomes $\mathbb{E}\left[\epsilon_t \epsilon_t\right]$ and as $\gamma_1 = \gamma_0 \rho_1$, the variance of the noise may be written as follows:

$$\sigma_\epsilon^2 \;=\; \mathbb{V}\left[\epsilon_t\right] \;=\; \gamma_0(1 - \phi_0 \rho_1) \;=\; \gamma_0(1 - \phi_0^2) \tag{11.43}$$

11.4.2 AR(p) Model

The zero-centered model equation for an AR(p) model includes the past p values.

$$\boxed{z_{t+1} \;=\; \phi_0 z_t \;+\; \phi_1 z_{t-1} \;+\; \ldots \;+\; \phi_{p'} z_{t-p'} \;+\; \epsilon_{t+1}}$$

The forecasted value for one step ahead ($h = 1$) is calculated as follows:

$$\hat{z}_{t+1} \;=\; \phi_0 z_t \;+\; \phi_1 z_{t-1} \;+\; \ldots \;+\; \phi_{p'} z_{t-p'}$$

Using the reverse dot product the equation becomes,

$$\hat{z}_{t+1} \;=\; \boldsymbol{\phi} \cdot^{\leftarrow} \mathbf{z}_{t-p':t} \tag{11.44}$$

This is depicted in Figure 11.2 that shows each subsequent parameter is paired with a value from the time series which is farther back in time. The paired values are multiplied and their products are summed to provide the next forecasted value.

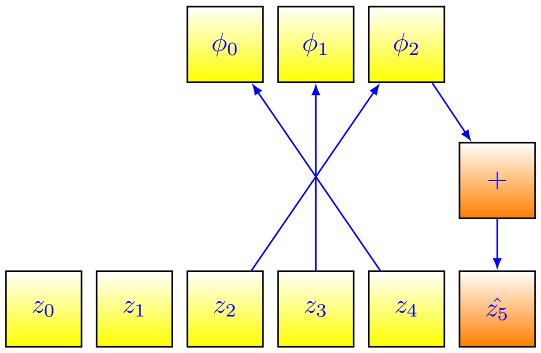


Figure 11.2: AR(p) Model with $p = 3$

As with convolutional filters, the parameter vector $\boldsymbol{\phi}$, can slide along the zero-centered time series z_t to compute forecasted values $\hat{z}_t$. To obtain forecasted values for the original time series, simply add back the mean,

$$y_{t+1} \;=\; z_{t+1} \;+\; \mu \tag{11.45}$$

Determining Parameter Values from the ACF

Working with the zero-centered equations, one can derive a system of linear equations to solve for the parameters. First, multiplying z_t by z_{t-k} gives

$$z_t z_{t-k} \;=\; \phi_0 z_{t-1} z_{t-k} \;+\; \phi_1 z_{t-2} z_{t-k} \;+\; \ldots \;+\; \phi_{p-1} z_{t-p} z_{t-k} \;+\; \epsilon_t z_{t-2}$$

and then taking the expectation produces

$$\gamma_k \;=\; \phi_0 \gamma_{k-1} \;+\; \phi_1 \gamma_{k-2} \;+\; \ldots \;+\; \phi_{p-1} \gamma_{k-p}$$

Dividing by γ_0 yields equations relating the parameters and correlations,

$$\rho_k \;=\; \phi_0\rho_{k-1} + \phi_1\rho_{k-2} + \ldots + \phi_{p-1}\rho_{k-p} \tag{11.46}$$

These equations contains p unknowns and by letting $k = 1, 2, \ldots, p$, it can be used to generate p equations, or one matrix equation.

Yule-Walker Equations

The equations below are known as the Yule-Walker equations. Note that $\rho_{-j} = \rho_j$, ρ_0 equals 1, and k advances row by row.

$$\begin{aligned}
\rho_1 &= \phi_0\rho_0 &&+ \phi_1\rho_1 &&+ \ldots + \phi_{p-1}\rho_{p-1} \\
\rho_2 &= \phi_0\rho_1 &&+ \phi_1\rho_0 &&+ \ldots + \phi_{p-1}\rho_{p-2} \\
\ldots & && && \\
\rho_p &= \phi_0\rho_{p-1} &&+ \phi_1\rho_{p-2} &&+ \ldots + \phi_{p-1}\rho_0
\end{aligned}$$

Letting $\boldsymbol{\rho}$ be the p-dimensional vector of lag auto-correlations and $\boldsymbol{\phi}$ be the p-dimensional vector of parameters/coefficients, we may concisely write

$$\boxed{\boldsymbol{\rho} \;=\; R\,\boldsymbol{\phi}} \tag{11.47}$$

where R is a p-by-p symmetric Toeplitz (one value per diagonal) matrix of correlations with ones on the main diagonal.

$$R \;=\; \begin{bmatrix} 1 & \rho_1 & \ldots & \rho_{p-2} & \rho_{p-1} \\ \rho_1 & 1 & \ldots & \rho_{p-3} & \rho_{p-2} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \rho_{p-2} & \rho_{p-3} & \ldots & 1 & \rho_1 \\ \rho_{p-1} & \rho_{p-2} & \ldots & \rho_1 & 1 \end{bmatrix} \tag{11.48}$$

One way to solve for the parameter vector $\boldsymbol{\phi}$ is to take the inverse (or use related matrix factorization techniques).

$$\boldsymbol{\phi} \;=\; R^{-1}\,\boldsymbol{\rho} \tag{11.49}$$

The noise variance σ_ϵ^2 is given by (see exercises)

$$\sigma_\epsilon^2 \;=\; \gamma_0(1 - \boldsymbol{\rho}\cdot\boldsymbol{\phi}) \;=\; \gamma_0(1 - \boldsymbol{\rho}^\intercal R^{-1}\boldsymbol{\rho}) \tag{11.50}$$

Due to the special structure of the R matrix, more efficient techniques may be used, see the next subsection.

Equations for AR(2)

The $\boldsymbol{\rho} = R\boldsymbol{\phi}$ equation for an AR(2) model becomes the following:

$$\begin{bmatrix} \rho_1 \\ \rho_2 \end{bmatrix} = \begin{bmatrix} 1 & \rho_1 \\ \rho_1 & 1 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix}$$

An easy way to solve for $\boldsymbol{\phi}$ is to apply LU Factorization using the augmented matrix below.

$$\left[\begin{array}{cc|c} 1 & \rho_1 & \rho_1 \\ \rho_1 & 1 & \rho_2 \end{array}\right]$$

11.4.3 Training

The steps in training include computing the Auto-Correlation Function, executing the Durbin-Levinson algorithm, and zero-centering the response. The `train` method of the `AR` class is implemented as follows:

```
    def train (x_null: MatrixD, y_ : VectorD): Unit =
        m = y_.dim                                        // length of relevant time series
        resetDF (pnq, m - pnq)                            // reset the degrees of freedom
        makeCorrelogram (y_)                              // correlogram computes psi matrix
        φ = psiM(p)(1 until p+1)                          // coefficients = p-th row, columns 1..p
        δ = statsF.mu * (1 - φ.sum)                       // compute drift/intercept
    end train
```

The drift δ is computed from the mean and the values of the $\boldsymbol{\phi}$. The parameter/coefficient vector $\boldsymbol{\phi}$ can be estimated using multiple training algorithms/estimation procedures [194]:

1. **Method of Moments (MoM)**. The first two moments, mean and covariance, are used in the Yule-Walker Method. In particular the auto-correlations in ACF are used.

2. **Maximum Likelihood Estimation (MLE)**. Minimize the negative log-likelihood. See the ARMA section.

3. **Least Squares Estimation (LSE)**. Minimize the conditional sum of squared errors. See the ARMA section.

In ScalaTion, the coefficients $\boldsymbol{\phi}$ are estimated using the Durbin-Levinson algorithm and extracted from the p^{th} row of the ψ (`psi`) matrix. Define ψ_{kj} to be ϕ_j for an AR(k) model. Letting k range up to p allows the ϕ_j parameters to be calculated. Letting k range up to the maximum number of lags (`ml`) allows the Partial Auto-Correlation Function (PACF) to be computed.

Invoke the `durbinLevinson` method [151] passing in the auto-covariance vector $\boldsymbol{\gamma}$ (`g`) and the maximum number of lags (`ml`). From 1 up to the maximum number of lags, iteratively compute the following:

$$\begin{aligned} \psi_{kk} &= \frac{\gamma_k - \Sigma_{j=1}^{k-1} \psi_{k-1,j}\, \gamma_{k-j}}{r_{k-1}} \\ \psi_{kj} &= \psi_{k-1,j} - \psi_{kk}\, \psi_{k-1,k-j} \\ r_k &= r_{k-1}(1 - \psi_{kk}^2) \end{aligned}$$

```
def durbinLevinson (g: VectorD, ml: Int): MatrixD =
    val ψ = new MatrixD (ml+1, ml+1)                    // psi matrix (ml = max lags)
    val r = new VectorD (ml+1); r(0) = g(0)

    for k <- 1 to ml do                                 // range up to max lags
        var sum = 0.0
        for j <- 1 until k do sum += ψ(k-1, j) * g(k-j)
        val a = (g(k) - sum) / r(k-1)
        ψ(k, k) = a
        for j <- 1 until k do ψ(k, j) = ψ(k-1, j) - a * ψ(k-1, k-j)
        r(k) = r(k-1) * (1 - a * a)
    end for
    ψ
end durbinLevinson
```

In particular, ψ_{kk} is the k-lag partial auto-correlation and the parameter vector is

$$\boldsymbol{\phi} \ = \ \boldsymbol{\psi}_p \qquad\qquad \text{the } p^{th} \text{ row of the } \Psi \text{ matrix} \tag{11.51}$$

Partial Auto-Correlation Function

The Partial Auto-Correlation Function (PACF) that is extracted from the main diagonal of the Ψ matrix and can be used along with ACF to select the appropriate type of model. As k increases, the k-lag auto-correlation ρ_k will decrease and eventually adding more parameters/coefficients will become of little help.

Deciding where to cut off the model based on ρ_k from the ACF is somewhat arbitrary, but the k-lag partial auto-correlation ψ_{kk} drops toward zero more abruptly giving a stronger signal as to what model to select. The partial auto-correlation differs from auto-correlation in that it removes indirect correlation. For example, if $\rho_1 = .7$, one would expect ρ_2 to equal $\rho_1^2 = .49$ as y_{t-2} is correlated with y_{t-1} and y_{t-1} is correlated with y_t. The 2-lag partial auto-correlation ψ_{22} measures whether there is any direct correlation between y_{t-2} and y_t. For this example, when $p_2 = .49$, $\psi_{22} = 0$ indicating no direct correlation and implying that the $\phi_1 z_{t-1}$ term need not be included in the model.

$$z_{t+1} \ = \ \phi_0 z_t + \phi_1 z_{t-1} + \epsilon_{t+1}$$

See the exercises and the ARMA section for more details.

11.4.4 Forecasting

After the parameters/coefficients have been estimated as part of the `train` method, the AR(p) model can be used for forecasting.

```
@main def aRTest4 (): Unit =

    val m  = y.dim                              // number of data points
    val hh = 2                                  // maximum forecasting horizon

    banner (s"Test Forecasts: AR(1) on LakeLevels Dataset")
    val mod = new AR (y)                        // create time series AR(1) model
    mod.trainNtest ()()                         // train-test model on full dataset

    val yf = mod.forecastAll (y, hh)                     // forecast h-steps ahead (1..hh)
    println (s"y.dim = ${y.dim}, yp.dim = ${yp.dim}, yf.dims = ${yf.dims}")
    assert (yf(?, 0)(0 until m) == y)                    // col 0 must agree with actual values
    differ (yf(?, 1)(1 until m), yp)
    assert (yf(?, 1)(1 until m) == yp)                   // col 1 must agree with 1 step pred.

    for h <- 1 to hh do
        val (yfh, qof) = mod.testF (h, y)
        val yy = y(h until m)
        println (s"Evaluate QoF for horizon $h:")
        println (FitM.fitMap (qof, QoF.values.map (_.toString)))
        println (s"Fit.mae (y, yfh, h)  = ${Fit.mae (y, yfh, h)}")
        println (s"Fit.mae_n (y, 1)     = ${Fit.mae_n (y, 1)}")
        println (s"Fit.mase (y, yfh, h) = ${Fit.mase (y, yfh, h)}")
    end for

end aRTest4
```

Prediction may be thought of as a one-step ahead forecast.

$$\hat{y}_{t+1} \;=\; \phi_0 y_t + \ldots + \phi_{p'} y_{t-p'} \tag{11.52}$$

ScalaTion will not predict a value corresponding to y_0 (although some packages offer the option of back-casting). When p is greater than 1, it repeats the y_0 value into negative times, hence the `y_(max (0, t-j))` factor.

```
    @param t   the time point from which to make prediction
    @param y_  the actual values to use in making predictions

    def predict (t: Int, y_ : VectorD): Double =
        var sum = δ                                            // intercept
        for j <- 0 until p do sum += ϕ(j) * y_(max (0, t-j))      // add $\phi_j y_{t-j}$
        sum
    end predict
```

Multi-horizon forecasting is more challenging than one-step ahead predictions. When the forecasting horizon $h > 1$,

$$\hat{y}_{t+h} \;=\; \phi_0 \hat{y}_{t+h-1} + \ldots + \phi_{h-1} y_t + \ldots + \phi_{p'} y_{t+h-p} \tag{11.53}$$

As h increases, fewer actual values are used to make the forecast. The `forecastAt` method makes forecasts for horizon h. It uses the last p actual values to make the first forecast, and then uses the last $p-1$ actual values and the first forecast to make the next forecast. As expected, the quality of the forecast will degrade as `h` gets larger.

```
    @param yf  the forecasting matrix (time x horizons)
    @param y_  the actual values to use in making forecasts
    @param h   the forecasting horizon, number of steps ahead to produce forecasts

    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
        if h < 1 then flaw ("forecastAt", s"horizon h = $h must be at least 1")
        for t <- y_.indices do                           // make forecasts over time horizon h
            val t1  = t + h - 1                          // time point prior to horizon
            var sum = δ
            for j <- 0 until p do sum += ϕ(j) * yf(max (0, t1-j), max (0, h-1-j))
            yf(t+h, h) = sum                            // forecast down the diagonal
        end for
        yf(?, h)                                        // return h-step ahead forecast vector
    end forecastAt
```

The forecasting matrix `yf` is required since the next horizon's forecasts depends on those from the previous horizon (the `forecastAll` method in the `Forecaster` traits sorts all this out).

11.4.5 `AR` Class

Class Methods:

```
    @param y        the response vector (time series data)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters

    class AR (y: VectorD, tt: VectorD = null, hparam: HyperParameter = SARIMAX.hp)
          extends Forecaster (y, tt, hparam)
             with Correlogram (y)
             with Fit (dfm = hparam("p").toInt, df = y.dim - hparam("p").toInt):

    def train (x_null: MatrixD, y_ : VectorD): Unit =
    def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
    override def parameter: VectorD = ϕ :+ δ
    def predict (t: Int, y_ : VectorD): Double =
    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
    def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
```

11.4.6 Exercises

1. Compute ACF for the Lake Level time series dataset. For AR(1) set parameter ϕ_0 to ρ_1, the first lag auto-correlation. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$ and for `k <- 1 until y.dim`

$$\begin{aligned} z_t &= y_t - \mu \\ \hat{y}_t &= \rho_1 z_{t-1} + \mu \end{aligned}$$

   Plot $\hat{y}_t$ and y_t versus t.

2. Consider the following AR(2) Model.

$$z_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \epsilon_t$$

   Derive the following equation:

$$\rho_k \;=\; \phi_0 \rho_{k-1} \;+\; \phi_1 \rho_{k-2}$$

Setting $k = 1$ and then $k = 2$ produces two equations which have two unknowns ϕ_0 and ϕ_1. Solve for ϕ_0 and ϕ_1 in terms of the first and second lag auto-correlations, ρ_1 and ρ_2, by completing the LU Factorization of the Augmented Yule-Walker Matrix. Given these two formulas, plug in values for (a) ρ_1 and ρ_2 for the Lake Level Dataset, see Table 11.1, (b) $\rho_1 = .7$ and $\rho_2 = .6$.

For both (a) and (b), compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$, $\hat{y}_1 = y_1$ and for `k <- 2 until y.dim`

$$z_t \;=\; y_t \;-\; \mu$$
$$\hat{y}_t \;=\; \phi_0 z_{t-1} \;+\; \phi_1 z_{t-2} \;+\; \mu$$

Plot $\hat{y}_t$ and y_t versus t.

3. Use the ScalaTion class `AR` to develop Auto-Regressive Models for $p = 1, 2, 3$, for the Lake Level time series dataset. Plot $\hat{y}_t$ and y_t versus t for each model. Also, compare the first two models with those developed in the previous exercises.

4. Generate a dataset for an AR(p) model as follows:

```
    val y = makeTSeries ()
```

Now create an AR(1) model, train it and show its report.

```
    val mod = new AR (y)
    mod.trainNtest ()()
```

Also, plot $\hat{y}_t$ and y_t versus t. Look at the ACF and PACF to see if some other AR(p) might be better.

```
    mod.plotFunc (ar.acF, "ACF")
    mod.plotFunc (ar.pacF, "PACF")
```

Choose a value for p and create an AR(p) model. Explain your choice of p. Also, plot $\hat{y}_t$ and y_t versus t.

5. The k-lag auto-covariance is useful when the stochastic process $\{y_t \,|\, t \in \{0, m-1\}\}$ is covariance stationary.

$$\gamma_k \;=\; \mathbb{C}\left[y_t, y_{t-k}\right]$$

When y_t is covariance stationary, the covariance is only determined by the lag between the variables and not where they occur in the time series, e.g., $\mathbb{C}[y_4, y_3] = \mathbb{C}[y_2, y_1] = \gamma_1$. Thus, the covariance matrix Γ is Toeplitz. Covariance stationarity also requires that $\mathbb{E}[y_t] = \mu$, i.e., the mean is time invariant. Repeat the previous exercise, but generate a time series from a process that is not covariance stationary. What can be done to transform such a process into a covariance stationary process? Hint: read ahead to the section on ARIMA models or additional reading such as [168], Chapter 23.

6. For a covariance stationary process y_t, derive the relationship between the drift parameter δ and the mean μ. Further, given an AR(p) model,

$$y_t \;=\; \delta + \phi_0 y_{t-1} + \phi_1 y_{t-2} + \ldots + \phi_{p-1} y_{t-p} + \epsilon_t$$

Show that δ drops out of the equation when $z_t + \mu$ is substituted for y_t in the above equation.

$$z_t \;=\; \phi_0 z_{t-1} + \phi_1 z_{t-2} + \ldots + \phi_{p-1} z_{t-p} + \epsilon_t$$

7. Assume the time indexed, m dimensional vector $\mathbf{y}$ is a zero-centered, covariance stationary Gaussian process $\mathbf{N}(0, \Gamma)$ having the following probability density function (pdf),

$$f(\mathbf{y}; \boldsymbol{\phi}, \sigma^2) \;=\; [(2\pi)^m \det\Gamma]^{-\frac{1}{2}} \, e^{-\frac{1}{2}\mathbf{y}^t \Gamma^{-1} \mathbf{y}}$$

Recall, that the covariance matrix for a covariance stationary process is Toeplitz, meaning

$$\Gamma \;=\; [\gamma_{i-j}]_{i,j=0,m-1}$$

Write the likelihood and log-likelihood functions. Discuss how to create an algorithm for Maximum Likelihood Estimation. Be sure to exploit the special structure of the covariance matrix.

8. Derive the formula for the noise variance $\sigma_\epsilon^2 = \mathbb{V}[z_t] = \mathbb{E}[z_t z_t]$,

$$\sigma_\epsilon^2 \;=\; \gamma_0 (1 - \boldsymbol{\rho} \cdot \boldsymbol{\phi})$$

from the following equation:

$$\gamma_0 \;=\; \phi_0 \gamma_1 + \phi_1 \gamma_2 + \ldots + \phi_{p-1} \gamma_p + \mathbb{E}[\epsilon_t \epsilon_t]$$

9. Create a process with ϕ_0 and ϕ_1 nonzero, such that ρ_1 is approximately zero, while ρ_2 is large. Compare the Quality of Fit (QoF) of an AR(2) model versus an AR(1) model for this process. Must complete the code before running.

```
    val m = 100
    val y = new VectorD (m)
    val noise = Normal (0, 1)
    val φ = VectorD (?, ?)
    y(0) = noise.gen; y(1) = noise.gen
    for t <- 2 until m do y(t) = φ(0) * y(t-1) + φ(1) * y(t-2) + noise.gen
```

Plot the ACF to view the auto-correlations.

10. An AR(1) process

$$y_{t+1} \;=\; \delta + \phi_0 y_t + \epsilon_{t+1}$$

is stationary, unit-root, or explosive, if $|\phi_0|$ is less than, equal to, or greater than 1, respectively. Letting $y_0 = 10$, $\delta = 0$, and `noise = Normal (0, 1)`, examine the time evolution of the process for $\phi_0 = .7$, 1, and 2. Explain what is happening. Which ones are covariance stationary?

11. For the Lake Level Dataset, the first ten values for the ACF ρ_k and PACF ψ_{kk} are given in Table 11.1.

Table 11.1: Correlogram (ACF ρ_k, PACF ψ_{kk}) for the Lake Level Dataset

k	ρ_k	ψ_{kk}
0	1.000000	0.000000
1	0.840488	0.840488
2	0.622644	-0.285357
3	0.472722	0.147962
4	0.386269	0.032072
5	0.343057	0.069543
6	0.303435	-0.026385
7	0.285146	0.108148
8	0.287510	0.047882
9	0.283758	0.003677
10	0.203506	-0.237926

As `AR` extends `Correlogram` the ACF and PACF functions may be plotted. Any points inside the two lines (upper and lower bounds) are of lesser significance.

```scala
@main def aRTest3 (): Unit =

    banner (s"Test Predictions: AR(1) on LakeLevels Dataset")
    val mod = new AR (y)                       // create model for time series data AR(1)
    mod.trainNtest ()()                        // train and test on full dataset

    banner ("Select model based on ACF and PACF")
    mod.plotFunc (mod.acF, "ACF")              // Auto-Correlation Function (ACF)
    mod.plotFunc (mod.pacF, "PACF")            // Partial Auto-Correlation Function (PACF)

end aRTest3
```

Based on the values in Table 11.1 and the ACF and PACF plots, explain your choices of p for candidate AR(p) models. Make a loop `for p <- 1 to 10 do` and check the QoF (R^2, MAE, and sMAPE) for your choices versus the other models. Justify your choices.

11.5 Moving-Average (MA) Models

The Auto-Regressive (AR) models predict future values based on past values. Let us suppose the daily values for a stock are 100, 110, 120, 110, 90. To forecast the next value, one could look at these values or focus on each days errors or shocks. Whether previous values of the time series (AR) or unexpected changes (MA) (e.g., while predictions indicated that the Dow-Jones Industrial Average would continue its slow rise, news of the its large drop has investor nervous) are more important depends on the situation. In the simplest case an MA model is of the following form,

$$y_{t+1} = \mu + \theta_0 \epsilon_t + \epsilon_{t+1}$$

To make the data zero-centered, again let $z_{t+1} = y_{t+1} - \mu$

$$z_{t+1} = \theta_0 \epsilon_t + \epsilon_{t+1} \tag{11.54}$$

Errors are computed in the usual way.

$$\epsilon_{t+1} = y_{t+1} - \hat{y}_{t+1}$$

The forecast for the next time point t+1 is

$$\hat{y}_{t+1} = \mu + \theta_0 \epsilon_t \tag{11.55}$$

$$\hat{z}_{t+1} = \theta_0 \epsilon_t \tag{11.56}$$

that is, the mean of the process plus some fraction of yesterday's shock. Let us assume that $\mu = 100$ and the parameter/coefficient θ_0 had been estimated to be 0.8. Now, if the mean is 100 dollars and yesterday's shock was that stock went up an extra 10 dollars, then the new forecast would be $100 + 8 = 108$ (day 2). This can be seen more clearly in the Table 11.2.

Table 11.2: MA(1) Forecast Chart (to one decimal place)

t	y_t	z_t	ϵ_{t-1}	$\hat{z}_t = \theta_0 \epsilon_{t-1}$	$\hat{y}_t$	ϵ_t	ϵ_t^2
0	100.0	0.0	-	-	-	0.0	0.0
1	110.0	10.0	0.0	0.0	100.0	10.0	100.0
2	120.0	20.0	10.0	8.0	108.0	12.0	144.0
3	110.0	10.0	12.0	9.6	109.6	0.4	0.2
4	90.0	-10.0	0.4	0.3	100.3	-10.3	106.1

The question of AR(1) versus MA(1) may be viewed as which column z_{t-1} or ϵ_{t-1} leads to better forecasts. It depends on the data, but considering the GDP example again, z_{t-1} indicates how far above or below the mean the current GDP is. One could imagine shocks ϵ_{t-1} to GDP, such as new tariffs, tax cuts or bad weather being influencial shocks to the economy. In such cases, an MA model may provide better forecasts than an AR model.

11.5.1 MA(q) Model

The model equation for an MA(q) model includes the past q values of ϵ_t.

$$\boxed{y_{t+1} \;=\; \delta \,+\, \boldsymbol{\theta} \cdot \overleftarrow{\boldsymbol{\epsilon}_{\mathbf{t-q':t}}} \,+\, \epsilon_{t+1}} \tag{11.57}$$

where $q' = q - 1$, the parameter vector $\boldsymbol{\theta} = [\theta_0, \ldots, \theta_{q-1}]$, the drift $\delta = \mu$, and the error/noise is represented by ϵ_{t+1}. This may be expanded out into

$$\boxed{y_{t+1} \;=\; \mu \,+\, \theta_0 \epsilon_t \,+\, \ldots \,+\, \theta_{q'} \epsilon_{t-q'} \,+\, \epsilon_{t+1}}$$

Zero-Centered

Zero-centering the data $z_t = y_t - \mu$ produces

$$z_t \;=\; \theta_0 \epsilon_{t-1} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-q} \,+\, \epsilon_t \tag{11.58}$$

In order to estimate the parameter vector $\boldsymbol{\theta} = [\theta_0, \ldots, \theta_{q-1}]$, we would like to develop a system of equations like the Yule-Walker equation. Proceeding likewise, the auto-covariance function for MA(q) is

$$\gamma_k \;=\; \mathbb{C}\left[z_t, z_{t-k}\right] \;=\; \mathbb{E}\left[z_t, z_{t-k}\right]$$

$$\gamma_k \;=\; \mathbb{E}\left[(\theta_0 \epsilon_{t-1} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-q} \,+\, \epsilon_t)(\theta_0 \epsilon_{t-k-1} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-k-q} \,+\, \epsilon_{t-k})\right] \tag{11.59}$$

When $k = 0$

Letting $k = 0$ will give the variance $\gamma_0 = \mathbb{V}\left[z_t\right]$

$$\gamma_0 \;=\; \mathbb{E}\left[(\theta_0 \epsilon_{t-1} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-q} \,+\, \epsilon_t)(\theta_0 \epsilon_{t-1} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-q} \,+\, \epsilon_{t-k})\right]$$

Since the noise shocks are independent, i.e., $\mathbb{C}\left[\epsilon_t, \epsilon_u\right] = 0$ unless $t = u$, many of the terms in the product drop out.

$$\gamma_0 \;=\; \theta_0^2 \mathbb{E}\left[{\epsilon_{t-1}}^2\right] \,+\, \ldots \,+\, \theta_{q-1}^2 \mathbb{E}\left[{\epsilon_{t-q}}^2\right] \,+\, \mathbb{E}\left[{\epsilon_t}^2\right]$$

The variance of the noise shocks is defined, for any t, to be $\mathbb{V}\left[\epsilon_t\right] = \mathbb{E}\left[{\epsilon_t}^2\right] = \sigma_\epsilon^2$, so

$$\gamma_0 \;=\; (\theta_0^2 \,+\, \theta_1^2 \,+\, \ldots \,+\, \theta_{q-1}^2 \,+\, 1)\, \sigma_\epsilon^2 \tag{11.60}$$

When $k \geq 1$

For $k \in [1, \ldots, q]$, similar equations can be created, only with parameter index shifted so the noise shocks match up.

$$\gamma_1 \;=\; \mathbb{E}\left[(\theta_0 \epsilon_{t-1} \,+\, \theta_1 \epsilon_{t-2} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-q} \,+\, \epsilon_t)(\theta_0 \epsilon_{t-1-1} \,+\, \theta_1 \epsilon_{t-1-2} \,+\, \ldots \,+\, \theta_{q-1} \epsilon_{t-1-q} \,+\, \epsilon_{t-1})\right]$$

$$\gamma_1 \;=\; (\theta_0 \,+\, \theta_1 \theta_0 \,+\, \theta_2 \theta_1 \,+\, \ldots \,+\, \theta_{q-1} \theta_{q-2})\, \sigma_\epsilon^2$$

$$\gamma_k \;=\; \mathbb{E}\left[(\theta_0\epsilon_{t-1} + \theta_1\epsilon_{t-2} + \ldots + \theta_{q-1}\epsilon_{t-q} + \epsilon_t)(\theta_0\epsilon_{t-k-1} + \theta_1\epsilon_{t-k-2} + \ldots + \theta_{q-1}\epsilon_{t-k-q} + \epsilon_{t-k})\right]$$

$$\gamma_k \;=\; (\theta_{k-1} + \theta_k\theta_0 + \theta_{k+1}\theta_1 + \ldots + \theta_{q-1}\theta_{q-k-1})\,\sigma_\epsilon^2 \tag{11.61}$$

As before the equation for the k-lag auto-correlation (part of the Auto-Correlation Function (ACF)) is simply γ_k/γ_0,

$$\rho_k \;=\; (\theta_{k-1} + \theta_k\theta_0 + \theta_{k+1}\theta_1 + \ldots + \theta_{q-1}\theta_{q-k-1})\,\frac{\sigma_\epsilon^2}{\gamma_0} \tag{11.62}$$

Notice that when $k > q$, the ACF will be zero. This is because only the last q noise shocks are included in the model, so any earlier noise shocks before that are forgotten. MA processes will tend to exhibit a more rapid drop off of the ACF compared to the slow decay for AR processes.

Unfortunately, the system of equations that can be generated from these equations are nonlinear. Consequently, training is more difficult and less efficient.

11.5.2 Training

Training for MA(1)

Training for Moving-Average models is easiest to understand for the case of a single parameter θ_0. In general, training is about errors and so rearranging the MA equation gives the following:

$$e_t \;=\; z_t - \theta_0 e_{t-1} \tag{11.63}$$

Given a value for parameter θ_0, this is a recursive equation that can be used to compute subsequent errors from previous ones. Unfortunately, the equation cannot be used to compute the first error e_0. One approach to deal with indeterminacy of e_0 is to assume (or condition on) it being 0.

$$e_0 \;=\; 0 \tag{11.64}$$

Note, this affects more than the first error, since the first affects the second and so on. Next, we may compute a sum of squared errors, in this case called Conditional Sum of Squared Errors (denoted in ScalaTion as `csse`.

$$csse \;=\; \sum_{t=0}^{m-1} e_t^2 \;=\; \sum_{t=1}^{m-1} (z_t - \theta_0 e_{t-1})^2 \tag{11.65}$$

One way to find a near optimal value for θ_0 is to minimize the Conditional Sum of Squared Error.

$$\text{argmin}_{\theta_0} \sum_{t=1}^{m-1} (z_t - \theta_0 e_{t-1})^2 \tag{11.66}$$

As the parameter $\theta_0 \in (-1, 1)$, an optimal value minimizing $csse$ may be found using Grid Search. A more efficient approach is to use the Newton method for optimization.

$$\theta_0^i \;=\; \theta_0^{i-1} - \frac{d\,csse}{d\theta_0} / \frac{d^2 csse}{d\theta_0^2} \tag{11.67}$$

where

$$\begin{aligned}\frac{d\,csse}{d\theta_0} &= -\,2\sum_{t=1}^{m-1} e_{t-1}(z_t - \theta_0 e_{t-1}) \\ \frac{d^2\,csse}{d\theta_0^2} &= 2\sum_{t=1}^{m-1} e_{t-1}^2\end{aligned}$$

Substituting gives

$$\theta_0^i \;=\; \theta_0^{i-1} + \sum_{t=1}^{m-1} e_{t-1}(z_t - \theta_0 e_{t-1}) / \sum_{t=1}^{m-1} e_{t-1}^2 \tag{11.68}$$

Maximum Likelihood Estimation (MLE) is also used for MA(q) parameter estimation.

11.5.3 Exercises

1. For an MA(1) model, solve for θ_0 using the equation for the noise variance and the equation for the correlation.

   $$\begin{aligned}\gamma_0 &= (1\,+\,\theta_0^2)\,\sigma_\epsilon^2 \\ \rho_1 &= (\theta_0)\,\frac{\sigma_\epsilon^2}{\gamma_0} \;=\; \frac{\theta_0}{1+\theta_0^2}\end{aligned}$$

   Solve for θ_0 in terms of ρ_1.

2. Develop an MA(1) model for the Lake Level time series dataset using the solution you derived in the previous question. Plot y_t and $\hat{y_t}$ vs. t. Estimate the mean Lake level and then plot z_t and ϵ_t.

3. Use the MA(1) solution you derived above for θ_0 and the fact that $\phi_0 = \rho_1$ for AR(1) to produce two Forecast Charts for the first 10 values in the Lake-Level Dataset, one for MA(1) and one for AR(1).

   $$\begin{aligned}\hat{z}_t &= \theta_0\epsilon_{t-1} \,+\, \epsilon_t && \text{MA(1)} \\ \hat{z}_t &= \phi_0 y_{t-1} \,+\, \epsilon_t && \text{AR(1)}\end{aligned}$$

   Compute R^2 and sMAPE from information in the charts. Which model has a better Quality of Fit (QoF)?

4. Use ScalaTion to assess the quality of an MA(1) model versus an MA(2) model for the Lake Level time series dataset.

5. The position in the ACF plot where ρ_k drops off can be used for q, the order of the model. What does this plot suggest for the Lake Level time series dataset?

6. Deduce the fact the drift $\delta = \mu$ for all MA(q) models.

11.6 Auto-Regressive, Moving-Average (ARMA) Models

The `ARMA` class provides basic time series analysis capabilities for Auto-Regressive (AR) and Moving-Average (MA) models. In an ARMA(p, q) model, p and q refer to the order of the Auto-Regressive and Moving-Average components of the model. ARMA models are often used for forecasting.

Recall that a p^{th}-order Auto-Regressive AR(p) model predicts the next value y_{t+1} from a weighted combination of prior values and that a q^{th}-order Moving-Average MA(q) model predicts the next value y_{t+1} from the combined effects of prior noise/disturbances.

A combined p^{th}-order Auto-Regressive, q^{th}-order Moving-Average ARMA(p, q) model predicts the next value y_{t+1} from both a weighted combination of prior values and the combined effects of prior noise/disturbances.

$$\boxed{y_{t+1} \;=\; \delta + \boldsymbol{\phi} \cdot^{\leftarrow} \mathbf{y}_{t-p':t} \;+\; \boldsymbol{\theta} \cdot^{\leftarrow} \boldsymbol{\epsilon}_{\mathbf{t-q':t}} + \epsilon_{t+1}} \tag{11.69}$$

This can be expanded into

$$\boxed{y_{t+1} \;=\; \delta + \phi_0 y_t + \ldots + \phi_{p'} y_{t-p'} \;+\; \theta_0 \epsilon_t + \ldots + \theta_{q'} \epsilon_{t-q'} + \epsilon_{t+1}} \tag{11.70}$$

where $p' = p - 1$, $q' = q - 1$, and δ is the drift.

11.6.1 Selection Based on ACF and PACF

The Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF) may be used for choosing values for the hyper-parameters p and q, for AR(p), MA(q) and ARMA(p, q) models. The ACF is computed to a maximum number of lags and when plotted shows the auto-correlation ρ_k versus the number of lags. When the ACF as a function of k drops in value toward zero, one may select a value for q around this k.

$$\rho_k \;=\; \frac{\mathbb{C}\left[z_t, z_{t-k}\right]}{\mathbb{V}\left[z_t\right]} \;=\; \frac{\mathbb{E}\left[z_t z_{t-k}\right]}{\mathbb{E}\left[z_t z_t\right]} \;=\; \frac{\gamma_k}{\gamma_0} \tag{11.71}$$

Similarly, the PACF is computed to a maximum number of lags and when plotted shows the partial auto-correlation ψ_{kk} versus the number of lags. When the PACF as a function of k drops in value toward zero, one may select a value for p around this k.

$$\psi_{kk} \;=\; \left[R^{-1} \boldsymbol{\rho}_{\mathbf{1:k}}\right]_k \tag{11.72}$$

Figure 11.3 shows the ACF ρ_k and PACF ψ_{kk} values versus the number of lags k. The PACF indicates the correlation between z_t and z_{t-k} beyond the chaining effect. For example, z_t and z_{t-2} would naturally be correlated at the level ρ_1^2, since the correlation between z_t and z_{t-1} is ρ_1, and z_{t-1} and z_{t-2} is ρ_1. Thus, partial correlation is a better indication of whether a term $\phi_k z_{t-k}$ contributes to the model.

From Figure 11.3 three partial correlations are worthy of consideration: $\rho_1 = 0.831911$, $\rho_2 = -0.285357$, and maybe $\rho_{10} = -0.237926$. From this perspective, candidate models are AR(1), AR(2) and AR(10). Of course, AR(10) will include several terms that contribute very little, so if a software package supports feature selection, they can be removed so that only three terms will be included in the model. Other options include maximizing $\bar{R}^2$ or minimizing AIC. See the exercise.

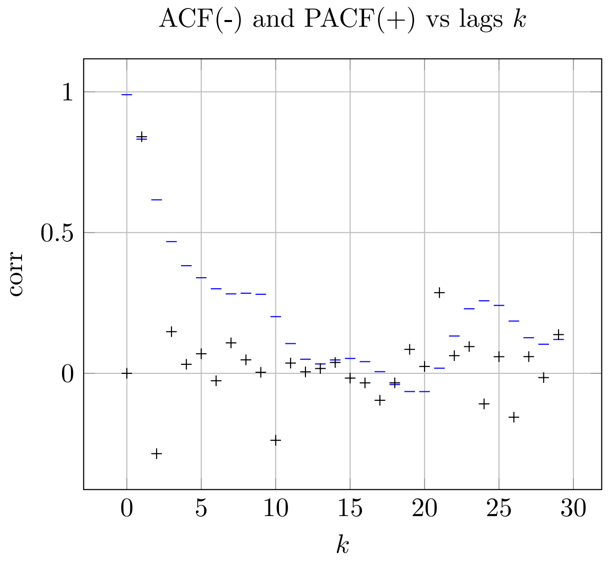


Figure 11.3: Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF)

11.6.2 Training

Training for ARMA models involves simultaneously finding p values for the AR part and q values for the MA part. We have seen that are multiple optimization algorithms for finding the $\boldsymbol{\phi}$ parameters for AR models.

Table 11.3: AR Optimization Algorithms

Technique	**Description**
MoM	Durbin-Levinson Solution to Yule-Walker Equations
OLS	Ordinary Least Squares
WLS	Weighted Least Squares
GLS	Generalized Least Squares
MLE	Maximum Likelihood Estimation

Similarly, there are also multiple, more complex, optimization algorithms for finding the $\boldsymbol{\theta}$ parameters for MA models.

The `train` method in the `ARMA` class shown below uses conditional sum of squared errors (csse) as the loss function. The parameter vector **b** collects all the parameters, $\boldsymbol{\phi}, \boldsymbol{\theta}$, and δ and passes then into a BFGS optimizer. The optimal values are determined and unpacked back into $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, and δ.

Table 11.4: MA Optimization Algorithms

Technique	Description
MoI	Method of Innovations
cSSE	Conditional Sum of Squares
MLE	Maximum Likelihood Estimation

```
override def train (x_null: MatrixD, y_ : VectorD): Unit =
    m = y_.dim                                       // length of relevant time series
    resetDF (pnq, m - pnq)                           // reset the degrees of freedom
    makeCorrelogram (y_)                             // correlogram computes ψ matrix

    val mu = y_.mean                                 // sample mean of y_
    val z  = y_ - mu                                 // opt. is better using zero-centered
    φ = new VectorD (p)                              // zeros for AR part
    θ = new VectorD (q)                              // zeros for MA part
    δ = 0.0                                          // drift for z (should be near zero)
    val b = φ ++ θ :+ δ                              // all parameters -> vector to optimize

    def csse (b: VectorD): Double =                  // conditional sum of squared errors
        φ = b(0 until p); θ = b(p until p+q); δ = b(b.dim-1)  // pull parameters from b
        ssef (z, predictAll (z))                     // compute loss function
    end csse

    val optimizer = new BFGS (csse)                  // apply Quasi-Newton BFGS optimizer
    val (fb, bb) = optimizer.solve (b)               // optimal solution and parameters

    φ = bb(0 until p); θ = bb(p until p+q); δ = bb(b.dim-1)  // recover parameters for z
    δ += mu * (1 - φ.sum)                            // uncenter
end train
```

The `predict` method gives a one-step ahead forecast $\hat{y}_{t+1}$ using the past p time series values and the most recent q errors/shocks. Note that errors before the start of the time series do not exist and cannot be predicted, hence the `if` in the second loop. Also, errors must be computed sequentially as predictions are made.

```
override def predict (t: Int, y_ : VectorD): Double =
    var sum = δ
    for j <- 0 until p do               sum += φ(j) * y_(max (0, t-j))
    for j <- 0 until q if t-j >= 0 do sum += θ(j) * e(t-j)
    if t < y_.dim-1 then e(t+1) = y_(t+1) - sum
    sum
end predict
```

11.6.3 `ARMA` Class

Class Methods:

```
    @param y        the response vector (time series data)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters
```

```

class ARMA (y: VectorD, tt: VectorD = null, hparam: HyperParameter = SARIMAX.hp)
      extends Forecaster (y, tt, hparam)
         with Correlogram (y)
         with Fit (dfm = pq (hparam), df = y.dim - pq (hparam)):

override def cap: Int = max (p, q)
def train (x_null: MatrixD, y_ : VectorD): Unit =
def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
def testF (h: Int, y_ : VectorD): (VectorD, VectorD) =
override def parameter: VectorD = φ ++ θ :+ δ
override def predict (t: Int, y_ : VectorD): Double =
override def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
override def forecastAt (yf: MatrixD, y_ : VectorD, h: Int): VectorD =
```

11.6.4 Exercises

1. Plot the PACF for the Lake Level time series dataset and use it to pick a value for p based on rule 1: "stop at the drop" and rule 2: "pull all peaks". Run AR for p ranging from 1 to 10. Assess the Quality of Fit (QoF) for each AR(p) model over this range. What would $\bar{R}^2$, AIC, and MAPE/sMAPE pick, respectively?

2. Rule 1 and especially rule 2 should be done within the context of significance, based on the following critical values,

$$\rho_k^* \;=\; \frac{z_{\alpha/2}^*}{\sqrt{m-k}} \tag{11.73}$$

   where m is the length of the time series, k is the lag, and α is the significance level (e.g., for $\alpha = .95$, $z_{\alpha/2}^* = 1.96$). Determine all peaks (both positive and negative) that are outside the interval $[-\rho_k^*, \rho_k^*]$.

3. Plot the ACF for the Lake Level time series dataset and use it to pick a value for q. Assess the quality of an MA(q) with this value for q. Try it for q being one lower and one higher.

4. Using the selected values for p and q from the two previous exercises, assess the quality of an ARMA(p, q). Try the four possibilities around the point (p, q).

5. For an ARMA (p, q) model, explain why the Partial Auto-correlation Function (PACF) is useful in choosing a value for the p AR hyper-parameter.

6. For an ARMA (p, q) model, explain why the Auto-correlation Function (ACF) is useful in in choosing a value for the q MA hyper-parameter.

11.7 Rolling-Validation

Although cross-validation is very useful for prediction and classification problems, there is no direct analog for time series data. Instances cannot be randomly selected for training and testing, because the instances are now time ordered and dependency is high between instances that are close in time (e.g., as measured by auto-correlation).

In time series, a rolling-validation can be used as a replacement for cross-validation. Rolling-validation works by defining two adjacent, non-overlapping time windows: the first containing training data and the second containing testing data. These two windows are rolled forward in time together.

11.7.1 1-Fold Rolling-Validation

The simplest form of rolling-validation is one-fold. The first time window is for training and has length l, while the second time window is for testing and has length $\lambda = m - l$.

$$\mathcal{W}^{(r)} = \{t_0, \dots, t_{l-1}\} \qquad \text{training} \tag{11.74}$$

$$\mathcal{W}^{(e)} = \{t_l, \dots, t_{m-1}\} \qquad \text{testing} \tag{11.75}$$

Based on these time windows, the response vector $\mathbf{y}$ is chopped into two sub-vectors:

$$\mathbf{y^{(r)}} = \mathbf{y}[\mathcal{W}^{(r)}] \qquad \text{training} \tag{11.76}$$

$$\mathbf{y^{(e)}} = \mathbf{y}[\mathcal{W}^{(e)}] \qquad \text{testing} \tag{11.77}$$

An algorithm for rolling validation may start by training the model on $\mathbf{y^{(r)}}$. Then iterate through $\mathbf{y^{(e)}}$ making a forecast for each element in this vector, producing a forecast vector $\hat{\mathbf{y}}$. The forecast error is then

$$\boldsymbol{\epsilon} = \mathbf{y^{(e)}} - \hat{\mathbf{y}} \tag{11.78}$$

This basic algorithm will only work for very short testing windows and thus the quality assessment may be unreliable.

Retraining

If the testing window is long, training may not be up to date, so one would expect the forecasts to degrade as the algorithm iterates through the testing data. This would argue for periodic retraining within the testing loop. Depending on the runtime efficiency of the training algorithm, the retaining may be more or less frequent. ScalaTion defines a retraining cycle `rc` that causes retraining to be performed after every `rc` forecasts. See the `RollingValidation` object in the `scalation.modeling.forecasting` package for details.

Multi-Horizon Forecasting

Although one step ahead forecasting is the most accurate, today's forecasting models are capable of producing useful forecasts for several steps ahead. As an example, Numerical Weather Prediction (NWP) produces 14-day forecasts. With a time-step on 6 hours, this would imply 56 step ahead forecasts. In this case, the forecasting horizon $h = 56$. Note, among other techniques, NWP uses Ensemble Kalman Filters, see the chapter on State-Space Models.

There are three main techniques available for multi-horizon forecasting [187]:

1. The **Direct Method** applies a trained forecasting function f to recent data to obtain an h-step ahead forecast.

$$\hat{y}_{t+h} = f(\mathbf{y}_{t-p':t}; \boldsymbol{\phi}) \tag{11.79}$$

$$\hat{y}_{t+h} = f([y_{t-p'}, \ldots, y_{t-2}, y_t]; \boldsymbol{\phi}) \tag{11.80}$$

   Clearly, the training must be based upon minimizing the h-step ahead forecast error (or maximizing the likelihood). For a single output model like ARMA, a separate model could be built for each forecasting horizon 1 through h. For a multiple output like a Neural Network, each output node would correspond to a particular horizon 1 through h.

2. The **Recursive Method** builds up forecasts one step at a time. For efficiency it is implemented in an iterative rather than recursive manner. For example, for $h = 3$ and $p \geq h$, forecasts would be produced as follows:

$$\hat{y}_{t+1} = f([y_{t-p'}, y_{t-p'+1}, \ldots, y_{t-1}, y_t]; \boldsymbol{\phi}) \tag{11.81}$$
$$\hat{y}_{t+2} = f([y_{t-p'+1}, y_{t-p'+2}, \ldots, y_t, \hat{y}_{t+1}]; \boldsymbol{\phi}) \tag{11.82}$$
$$\hat{y}_{t+3} = f([y_{t-p'+2}, y_{t-p'+3}, \ldots, \hat{y}_{t+1}, \hat{y}_{t+2}]; \boldsymbol{\phi}) \tag{11.83}$$

   Notice that the farther into the future the forecast is made, the more it depends on prior forecasts (not actual data). As errors can compound, longer term forecast may degrade.

   Warning Concerning Multi-Horizon Forecasting with MA(q) Models

   Predicting future shocks/errors is precarious, so order q needs to be at least the maximum forecasting horizon h for MA(q) models. When $h > q$, all the errors will need to be forecasted and the best estimate for future noise is zero, so MA(q) forecasts will quickly degenerate to the mean of the time series (like the NullModel).

3. **Hybrid Methods** try to combine the best the direct and recursive methods. One way to do this is to have a model for each forecasting horizon as with the direct method, but utilize forecasts from previous models (1 through $h - 1$) in the model for horizon h. See [187, 186] for more details.

11.7.2 Rolling Validation and the Forecasting Matrix

Consider the following time series y_t with $m = 24$ elements. The `forecastAll` method will produce forecasts for all time points and horizons, see `yf` in Table 11.5. The `rollValidate` method divides the training and testing set by splitting the time series in half (an adjustable value). At the start the first 12 values are used for training in order to predict the first value in the testing set. For the next value in testing, the actual first value in the testing set can be used for its prediction. This process continues until the end of the testing set. To keep the parameters from becoming stale, retraining occurs for every `rc` predictions (`rc = 4` in the example below). All of the previous values are used for retraining. In this way both (training and testing) roll forward.

```
    val y = VectorD.range (1, 25)
    val (rc, h) = (4, 2)                                  // horizon, retraining cycle
    val mod = new RandomWalk (y)                          // create an RW model
    mod.train (null, y)                                   // train the model on full dataset
    val (yp, qof) = mod.test (null, y)                    // test the model on full dataset
    println (mod.report (qof))                            // report on Quality of Fit (QoF)
    val yf = mod.forecastAll (y, h)                       // produce all forecasts up horizon h
    println (s"yf = $yf")                                 // print forecast matrix
    FitM.showQofStatTable (RollingValidation.rollValidate (mod, rc, h))
```

Table 11.5: Multi-Horizon ($h = 2$) Forecasting Matrix `yf` for RW

y_t	$\hat{y}_t@h = 1$	$\hat{y}_t@h = 2$	t
1.00000	0.00000	0.00000	0.00000
2.00000	1.00000	0.00000	1.00000
3.00000	2.00000	1.00000	2.00000
4.00000	3.00000	2.00000	3.00000
5.00000	4.00000	3.00000	4.00000
6.00000	5.00000	4.00000	5.00000
7.00000	6.00000	5.00000	6.00000
8.00000	7.00000	6.00000	7.00000
9.00000	8.00000	7.00000	8.00000
10.0000	9.00000	8.00000	9.00000
11.0000	10.0000	9.00000	10.00000
12.0000	**11.0000**	10.0000	11.00000
13.0000	12.0000	**11.0000**	12.00000
14.0000	13.0000	12.0000	13.00000
15.0000	14.0000	13.0000	14.00000
16.0000	15.0000	14.0000	15.00000
17.0000	16.0000	15.0000	16.00000
18.0000	17.0000	16.0000	17.00000
19.0000	18.0000	17.0000	18.00000
20.0000	19.0000	18.0000	19.00000
21.0000	20.0000	19.0000	20.00000
22.0000	21.0000	20.0000	21.00000
23.0000	22.0000	21.0000	22.00000
24.0000	23.0000	22.0000	23.00000
0.00000	24.0000	23.0000	24.00000
0.00000	0.00000	24.0000	25.00000

Lines in the Table 11.5 indicate the points at which training (or retraining) occurs:

- Training at time point times 12 (using 0-11) is used for forecasting values for 12 to 15.
- Training at time point times 16 (using 0-15) is used for forecasting values for 16 to 19.
- Training at time point times 20 (using 0-19) is used for forecasting values for 20 to 23.

Forecasts are produced for times 24 and 25, but as no actual values are available at these times, they cannot be used for quality assessment. In general, the last h rows of the `yf` matrix must be excluded from quality assessment. Do to the regular nature of the values in Table 11.5, calculation of QoF measures are easily carried out, as shown below.

- Sum of Squares Total:

$$sst \;=\; \sum_{t=12}^{23} (t+1-18.5)^2 \;=\; 83$$

- Sum of Squared Errors ($h = 1$):

$$sse \;=\; \sum_{t=12}^{23} (t+1-t)^2 \;=\; 12 \qquad R^2 \;=\; 1 - 12/83 \;=\; 0.855$$

- Sum of Squared Errors ($h = 2$):

$$sse \;=\; \sum_{t=12}^{23} (t+1-t-1)^2 \;=\; 48 \qquad R^2 \;=\; 1 - 48/83 \;=\; 0.422$$

Notice: (1) Random Walk has no training involved and was picked to make the patterns obvious. (2) The dataset is too small and regular to be anything but a toy example. (3) QoF measures get worse as the forecasting horizon becomes larger and for baseline models this may happen rapidly. (4) If the testing set is small, sst will be substantially reduced, which will make R^2 look very poor (this is one reason other QoF measures are preferred over R^2 for time series). For example, for this problem MAE = h (i.e., 1 or 2), independently of the size of testing set. See the exercises for application of other forecasting models to longer datasets.

Rolling Validation Algorithm

The `rollValidate` method in the `RollingValidation` object in the `scalation.modeling.forecasting` package implements the rolling validation algorithm.

```
    @param mod   the forecasting model being used (e.g., 'ARMA')
    @param rc    the retraining cycle (number of forecasts until retraining occurs)

    def rollValidate (mod: Forecaster & Fit, rc: Int): Unit =
        val y       = mod.getY                                     // get response vector
        val tr_size = trSize (y.dim)                               // size of training set
        val te_size = y.dim - tr_size                              // size of testing set
        debug ("rollValidate", s"train: tr_size = $tr_size; test: te_size = $te_size, rc = $
    rc")

        val yp = new VectorD (te_size)                                  // testing set y-predicted
        for i <- 0 until te_size do                                     // iterate over testing set
            val t = tr_size + i                                         // next time to forecast
            if i % rc == 0 then mod.train (null, y(0 until t))          // retrain on sliding set
            yp(i) = mod.predict (t-1, y)                                // predict the next value
        end for

        val (t, yy) = align (tr_size, y)                                // align vectors
        val df = max (0, mod.parameter.size - 1)                        // degrees of freedom model
        mod.resetDF (df, te_size - df)                                  // reset degrees of freedom
        new Plot (t, yy, yp, "Plot yy, yp vs. t", lines = true)
        println (FitM.fitMap (mod.diagnose (yy, yp), QoF.values.map (_.toString)))
    end rollValidate
```

An overloaded version handles the multi-horizon forecasting case.

```
@param mod  the forecasting model being used (e.g., `ARMA`)
@param rc   the retraining cycle (number of forecasts until retraining occurs)
@param h    the forecasting horizon (h-steps ahead)

def rollValidate (mod: Forecaster & Fit, rc: Int, h: Int): Unit =
    val y       = mod.getY                                          // get response vector
    val yf      = mod.forecastAll (y, h)                            // get forecasting matrix
    val tr_size = trSize (y.dim)                                    // size of training set
    val te_size = y.dim - tr_size                                   // size of testing set
    debug ("rollValidate", s"train: tr_size = $tr_size; test: te_size = $te_size, rc = $
rc")

    val yp = new VectorD (te_size)                                  // testing set y-predicted
    for i <- 0 until te_size do                                     // iterate over testing set
        val t = tr_size + i                                         // next time to forecast
        if i % rc == 0 then mod.train (null, y(0 until t))          // retrain on sliding set
        yp(i)  = mod.predict (t-1, y)                               // predict the next value
        val yd = mod.forecast (t-1, yf, y, h)                       // forecast next h-values
                                                                    // yf updated on diagonals
        assert (yp(i) =~ yd(0))                                     // forecasts =? predictions
    end for                                                         // yf updated on diagonals

    val (t, yy) = align (tr_size, y)                                // align vectors
    val df = max (0, mod.parameter.size - 1)                        // degrees of freedom model
    mod.resetDF (df, te_size - df)                                  // reset degrees of freedom
    new Plot (t, yy, yp, "Plot yy, yp vs. t", lines = true)

    for k <- 1 to h do
        val yfh = yf(tr_size until y.dim, k)
        new Plot (t, yy, yfh, s"Plot yy, yfh vs. t (h = $k)", lines = true)
        banner (s"rollValidate: for horizon h = $k:")
        println (FitM.fitMap (mod.diagnose (yy, yfh), QoF.values.map (_.toString)))
    end for
end rollValidate
```

The first algorithm is subsumed by the second. The main differences occur in the `for` loop that iterates through the testing set.

```
        for i <- 0 until te_size do                                 // iterate over testing set
            val t = tr_size + i                                     // next time to forecast
            if i % rc == 0 then mod.train (null, y(0 until t))      // retrain on sliding set
            yp(i)  = mod.predict (t-1, y)                           // predict the next value
            val yd = mod.forecast (t-1, yf, y, h)                   // forecast next h-values
                                                                    // yf updated on diagonals
            assert (yp(i) =~ yd(0))                                 // forecasts =? predictions
        end for                                                     // yf updated on diagonals
```

The time t is the training set size tr_size plus the index i into the testing set. If the index i modulus rc is zero retraining occurs. The predict method returns a one-step ahead prediction for time t using past known values. For an AR(p) model, here's the predict method.

```
    @param t   the time point from which to make prediction
    @param y_  the actual values to use in making predictions

    def predict (t: Int, y_ : VectorD): Double =
        var sum = δ                                             // intercept
        for j <- 0 until p do sum += φ(j) * y_(max (0, t-j))    // add φ_j y_{t-j}
        sum
    end predict
```

The forecast method returns a vector containing forecasts for horizons 1 to h. It uses the recursive method and requires the forecasting matrix yf that it updates as it goes making modifications down diagonals. The updated values are captured in vector called yd and returned. For an AR(p) model, here's the forecast method.

```
    @param t   the time point from which to make forecasts
    @param yf  the forecasting matrix (time x horizons)
    @param y_  the actual values to use in making predictions
    @param h   the forecasting horizon, number of steps ahead to produce forecasts

    def forecast (t: Int, yf: MatrixD, y_ : VectorD, h: Int): VectorD =
        if h < 1 then flaw ("forecast", s"horizon h = $h must be at least 1")
        val yd = new VectorD (h)                                // hold forecasts
        for k <- 1 to h do
            val t1  = t + k - 1                                 // time point prior to horizon
            var sum = δ
            for j <- 0 until p do sum += φ(j) * yf(max (0, t1-j), max (0, k-1-j))
            yf(t+k, k) = sum                                    // forecast down diagonal
            yd (k-1)   = sum                                    // record diagonal values
        end for
        yd                                                      // return forecast vector
    end forecast
```

Note the switch, predict uses y_, while forecast uses yf. The first column of the yf matrix contains actual values, while the rest contain forecasts. Obviously, actual values are preferred over forecasted values, but for multi-horizon forecasting, prior forecasts are used when actual values are not available (or would constitute cheating). Values are extracted from the yf matrix by tracking back up the diagonal until the first column is reached and then moving up that column. Until the first column is reached, only forecasted are available. Furthermore, for AR(p) if $h > p$, some forecasts will be based solely upon prior forecasts and not directly on any actual values. The assert statement makes sure the prediction agrees with the one-step forecast.

Example: Rolling Validation for an AR(3) Model

Consider how rolling validation would work for an AR(3) model for the Lake Levels dataset that consists of 98 years worth of time series data. Using the default rule that initially the first half (49) is used for training and second half (49) is used for testing, Table 11.6 shows the forecasting matrix around this split (made the same size as the previous table for easy of understanding).

Table 11.6: Multi-Horizon ($h = 2$) Forecasting Matrix `yf` for AR(3)

y_t	$\hat{y}_t@h = 1$	$\hat{y}_t@h = 2$	t
578.670	578.284	578.786	37.0000
579.550	578.945	578.512	38.0000
578.920	579.645	578.966	39.0000
578.090	578.617	579.431	40.0000
579.370	578.096	578.688	41.0000
580.130	579.809	578.379	42.0000
580.140	579.970	579.610	43.0000
579.510	579.832	579.641	44.0000
579.240	579.233	579.594	45.0000
578.660	579.212	579.204	46.0000
578.860	578.588	579.207	47.0000
578.050	**579.030**	578.725	48.0000
577.790	577.946	**579.047**	49.0000
576.750	578.045	578.220	50.0000
576.750	576.873	578.326	51.0000
577.820	577.298	577.436	52.0000
578.640	578.345	577.759	53.0000
580.580	578.789	578.458	54.0000
579.480	580.760	578.750	55.0000
577.380	578.783	580.220	56.0000
576.900	577.202	578.777	57.0000
576.940	577.436	577.775	58.0000
576.240	577.383	577.940	59.0000
576.840	576.509	577.793	60.0000
576.850	577.500	577.128	61.0000
576.900	577.140	577.870	62.0000

Again, rolling validation is used to roll forward through the testing set, with fresh data and periodic retraining, making out-of-sample multi-horizon forecasts. The first 2-steps ahead, out-of-sample forecast is for time-unit 49. It uses three previous values to make a 2-steps ahead forecast for 49 (shown in bold). The most reliable values are actual values (shown in the first column). Actual values are available for times 47 and 46. Using the actual value for 48 is not appropriate since this is tantamount to soothsaying (knowing the future). Although the data is yearly, suppose it is daily and today is Wednesday and you want for forecast two days ahead (Friday). It is fine to use actual values from Tuesday and Wednesday, but not Thursday (since it is tomorrow). When actual values are not appropriate, use forecasts from the least horizon that is appropriate (in this case the one-day ahead forecast for Thursday). Therefore, multi-horizon forecasting involves moving up the diagonal until column 0 is reached and then moving up column 0 (as indicated by the pattern of values in bold).

11.7.3 Exercises

1. Compute the following additional QoF measure for rolling-validation on the dataset given in Table 11.5: RMSE, sMAPE and MASE.

2. Rolling-Validation results were given for the Lake Levels Dataset for the three baseline models: Random Walk, Null and Trend models. See the Baseline Models section. Compare these to the modeling techniques of intermediate complexity covered so far in this text: Simple Exponential Smoothing (SES), Auto-Regressive (AR) and Auto-Regressive, Moving-Average (ARMA).

3. Find the optimal values for p and q in ARMA(p, q) models for the Lake Levels Dataset for each forecasting horizon $h = 1, 2, 3$. Compare in-sample and out-of-sample QoF measures. How do these results relate to suggestions implied by the Correlogram?

4. An alternative to the rolling validation algorithm would be to keep the size of training set the same size throughout, by discarding values at the beginning of the time series. Although in general this means less data for training, it could reduce staleness (where values from the distant pass have undue influence on parameters estimation). Try making this change to the code and assess it impact.

5. To further reduce staleness one could use a restricted training that is a user specified distance back from the testing window. Try making this change to the code and assess it impact.

6. Compare recursive and direct multi-horizon forecasting under rolling-validation for Auto-Regressive, Moving-Average (ARMA) models.

7. Compare two recursive-direct hybrid multi-horizon forecasting methods found in the literature. Find papers that compare the relative quality of recursive, direct and hybrid methods.

8. The complete forecasting matrix `yf` created, for example, by calling `forecastAll` is not needed for rolling-validation. Only `cap` values from training set combined with the testing set are needed. Make this change to the `forecast` method to improve its efficiency.

9. Design a k-fold rolling-validation algorithm by dividing the testing set into multiple folds and by using $j \in \{0, \dots, k-1\}$ to keep track of the current fold. Now each fold has two adjacent, non-overlapping time windows. The first time window is for training and has length l, while the second time window is for testing and has length $\lambda = \lfloor \frac{m-l}{k} \rfloor$.

$$\mathcal{W}_j^{(r)} = \{t_{\lambda j}, \dots, t_{l+\lambda j-1}\} \qquad \text{training} \qquad (11.84)$$

$$\mathcal{W}_j^{(e)} = \{t_{l+\lambda j}, \dots, t_{l+\lambda(j+1)-1}\} \qquad \text{testing} \qquad (11.85)$$

For example, let $m = 200, k = 10, l = 100$, then $\lambda = 10$ and the ten pairs of time windows are shown below.

$$\mathcal{W}_{:}^{(r)} = \{t_0, \dots, t_{99}\}, \{t_{10}, \dots, t_{109}\}, \dots, \{t_{90}, \dots, t_{189}\} \tag{11.86}$$

$$\mathcal{W}_{:}^{(e)} = \{t_{100}, \dots, t_{109}\}, \{t_{110}, \dots, t_{119}\}, \dots, \{t_{190}, \dots, t_{199}\} \tag{11.87}$$

(11.88)

11.8 ARIMA (Integrated) Models

In cases where a time series is not stationary, an ARIMA (p, d, q) may be used. The new hyper-parameter d is the number of times the time series is differenced. When there is a significant trend in the data (e.g., the values or levels are increasing/decreasing over time), an ARIMA (p, 1, q) may be effective. This model can take a **first difference** of the values in the time series, i.e.,

$$y'_t \;=\; \Delta y_t \;=\; y_t \;-\; y_{t-1} \tag{11.89}$$

This new 'differenced' time series can then be put into an ARMA(p, q) model to predict the next difference y'_{t+1} from both a weighted combination of prior values and the combined effects of prior noise/disturbances.

$$\boxed{y'_{t+1} \;=\; \delta + \boldsymbol{\phi} \cdot^{\leftarrow} \mathbf{y'_{t-p':t}} \;+\; \boldsymbol{\theta} \cdot^{\leftarrow} \boldsymbol{\epsilon}_{\mathbf{t-q':t}} \;+\; \epsilon_{\mathbf{t+1}}} \tag{11.90}$$

This can be expanded into

$$\boxed{y'_{t+1} \;=\; \delta + \phi_0 y'_t \;+\; \ldots \;+\; \phi_{p'} y'_{t-p'} \;=\; \theta_0 \epsilon_t \;+\; \ldots \;+\; \theta_{q'} \epsilon_{t-q'} \;+\; \epsilon_{t+1}} \tag{11.91}$$

Recall $p' = p - 1$ and q' $= q - 1$ where p is the AR order and q is the MA order.

A **second difference** ($d = 2$) is computed as the difference of the first difference.

$$y''_t \;=\; y'_t - y'_{t-1} \;=\; y_t - 2y_{t-1} + y_{t-2} \tag{11.92}$$

Higher differencing is possible, but is not commonly used. Think of the original/level time series as values, the first difference as rates of change, and the second difference as changes in rates (e.g., position, velocity, and acceleration).

Training for an ARIMA model can apply one of the optimization algorithms used for ARMA models. Note, the "I" in ARIMA could interpreted to mean, once you have taken a difference (like a derivative), it should be "Integrated" back to get the final forecast (see the subsection below).

11.8.1 Differencing

The `del` method (also Δ in the code) in `AR1MA.scala` in the `forecasting` package takes a time series `y` as a vector and returns the first difference. Note that the 'differenced' series has one less element than the original series.

```
    def del (y: VectorD): VectorD = VectorD (for t <- 0 until y.dim - 1 yield y(t+1) - y(t))
```

The first element in the original time series `y0` must be maintained to enable the original time series to be exactly restored from the 'differenced' series. Restoration of the original times-series is achieved using the `undel` (inverse difference) method.

```
    def undel (v: VectorD, y0: Double): VectorD =
        val y = new VectorD (v.dim + 1)
        y(0)  = y0
        for t <- 1 until y.dim do y(t) = v(t-1) + y(t-1)
        y
    end undel
```

11.8.2 Forecasting

Forecasts for the differenced time series may be produced using an ARMA model.

$$\hat{y}'_{t+1} \;=\; \delta \,+\, \boldsymbol{\phi} \cdot^{\leftarrow} \mathbf{y'_{t-p':t}} \,+\, \boldsymbol{\theta} \cdot^{\leftarrow} \boldsymbol{\epsilon}_{\mathbf{t-q':t}} \tag{11.93}$$

The level forecast then simply undoes the differencing.

$$\hat{y}_{t+1} \;=\; \hat{y}'_{t+1} + y_t \tag{11.94}$$

11.8.3 Backshift Operator

For more complex models, it become convenient to define the backshift (or lag) operator B^k,

$$B^k y_t \;=\; y_{t-k} \tag{11.95}$$

that moves back k positions in the time series. Consequently, an AR(p) model may be written as

$$y_t \;=\; \delta + [\phi_0 B^1 + \cdots + \phi_{p-1} B^p] y_t + \epsilon_t \tag{11.96}$$

Note the change from forecasting y_{t+1} to y_t as it is more convenient with the backshift operator. In vector notation $\boldsymbol{\phi} = [\phi_0, \dots \phi_{p-1}]$ and using a dot product, this becomes

$$y_t \;=\; \delta + \boldsymbol{\phi} \cdot [B^1, \dots, B^p] y_t + \epsilon_t \tag{11.97}$$

Similarly, an MA(q) model may be written as

$$y_t \;=\; \delta + \boldsymbol{\theta} \cdot [B^1, \dots, B^q] \epsilon_t + \epsilon_t \tag{11.98}$$

$$y_t \;=\; \delta + [1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q] \epsilon_t \tag{11.99}$$

Combining these two gives an ARMA(p, q) model.

$$y_t \;=\; \delta + \boldsymbol{\phi} \cdot [B^1, \dots, B^p] y_t + [1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q] \epsilon_t \tag{11.100}$$

Notice that $(1 - B^1) y_t = y_t - y_{t-1}$, i.e., the first difference and in general, $(1 - B^1)^d$ is the d^{th} difference.

The backshift operator can be used to reformulate an ARIMA(p, 0, q) model as follows:

$$[1, \boldsymbol{\phi}] \cdot [1, -B^1, \dots, -B^p] y_t \;=\; \delta + [1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q] \epsilon_t \tag{11.101}$$

This is just the ARMA model with the AR part collected on the right and the MA collected on the left. In particular, y_t is not repeated. So why the more obscure formulation. Well, it makes it easy to apply the differencing in an ARIMA(p, d, q) model.

$$\boxed{[1, \boldsymbol{\phi}] \cdot [1, -B^1, \dots, -B^p](1 - B)^d y_t \;=\; \delta + [1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q] \epsilon_t} \tag{11.102}$$

11.8.4 Stationarity Process

A general assumption for ARMA models is that time series y_t is covariance (or weakly) stationary, meaning that the first two moments are time invariant, i.e,

$$\mathbb{E}[y_t] = \mu(t) = \mu \tag{11.103}$$

$$\mathbb{V}[y_t] = \gamma_0(t) = \gamma_0 \tag{11.104}$$

$$\mathbb{C}[y_t, y_{t-k}] = \gamma_k(t) = \gamma_k \tag{11.105}$$

If the time series violates these conditions strongly enough, several approaches may be tried including transformations, detrending and differencing.

Unit-Root Process

It is not uncommon to have a stochastic process where the variance grows over time, The Random Walk process is an example. Interesting, for an AR(1) model, a slight change to the parameter/coefficient will change a process from a stationary process, to a unit-root process, and then to an explosive process (see the exercises). Although for an AR(1) process, this demarcation is obvious, a process is stationary, unit-root or explosive, if $|\phi_0|$ is less than, equal to, or greater than 1, respectively. As an analogy, think of exponential decay vs. growth with $e^{.9}$, e^1, and $e^{1.1}$. For AR(p) processes, the case is not so obvious, however, by finding the roots of the characteristic equation for a given model equation, the answer can be obtained.

Characteristic Equation

Starting with the boxed equation for ARIMA model equations with $\delta = 0$ formulated using the backshift operator,

$$[1, \boldsymbol{\phi}] \cdot [1, -B^1, \ldots, -B^p](1 - B)^d y_t = [1, \boldsymbol{\theta}] \cdot [1, B^1, \ldots, B^q]\epsilon_t$$

let $(p, d, q) = (1, 0, 0)$ to obtain an AR(1) model.

$$[1, \phi_0] \cdot [1, -B^1]y_t = [1] \cdot [1]\epsilon_t \tag{11.106}$$

Carrying out the dot products gives,

$$(1 - \phi_0 B^1)y_t = \epsilon_t \tag{11.107}$$

Similarly, for an AR(2) model,

$$(1 - \phi_0 B^1 - \phi_1 B^2)y_t = \epsilon_t \tag{11.108}$$

In general, these can be rewritten in terms of characteristic polynomials,

$$\phi(B)y_t = \epsilon_t \tag{11.109}$$

where for an AR(p) model the characteristic polynomial is

$$\phi(B) = 1 - \phi_0 B^1 - \phi_1 B^2 - \cdots - \phi_{p-1} B^p \tag{11.110}$$

Table 11.7: Characteristic Polynomials for ARIMA (p, d, q) Models

Characteristic Polynomial	**Notation**	**Formula**
AR(p)	$\boldsymbol{\phi}(B)$	$1 - \phi_0 B^1 - \phi_1 B^2 - \cdots - \phi_{p-1} B^p$
MA(q)	$\boldsymbol{\theta}(B)$	$1 + \theta_0 B^1 + \theta_1 B^2 + \cdots + \theta_{q-1} B^q$

An AR(p) process is covariance stationary, if all the root of the following characteristic equation,

$$\phi(B) = 0 \tag{11.111}$$

$$\phi(\zeta) = 0 \tag{11.112}$$

are outside the unit circle. As roots of a polynomial function may be complex numbers, we replace B with a complex variable $\zeta \in \mathbb{C}$.

For example, for an AR(1) process, the characteristic equation is

$$1 - \phi_0 \zeta = 0 \tag{11.113}$$

The root of this equation is clearly

$$\zeta = \frac{1}{\phi_0} \tag{11.114}$$

Based upon the magnitude of the complex number $|\zeta|$, three cases exist.

- When $|\zeta| > 1$, the coefficient $\phi_0 < 1$ and the process will be covariance stationary,
- When $|\zeta| = 1$, the coefficient $\phi_0 = 1$ and it will be a unit root process, and
- When $|\zeta| < 1$, the coefficient $\phi_0 > 1$ and the process will be explosive.

Note, unit circle denotes a circle with radius one in the complex plane.

For an AR(2) process, the characteristic equation is

$$1 - \phi_0 \zeta - \phi_1 \zeta^2 = 0 \tag{11.115}$$

The roots of this equation is given by the quadratic equation,

$$\zeta = \frac{\phi_0 \pm \sqrt{\phi_0^2 + 4\phi_1}}{-2\phi_1} \tag{11.116}$$

Trend Stationary Process

A (linear) trend stationary process can be constructed by combining a deterministic linear trend $\mu(t)$ with a covariance stationary process z_t.

$$y_t = \mu(t) + z_t \tag{11.117}$$

where the mean varies with time t, i.e., $\mu(t) = b_0 + b_1 t$. Simple Regression can be used to determine the coefficients b_0 and b_1.

Trend stationary processes have an advantage over unit-root processes in that a shock's influence will diminish over time and the process will revert to the mean [217]. Detrending can be accomplished by subtracting the deterministic trend $\mu(t)$ or by differencing. Note, for nonlinear trends (not considered here), differencing may not remove the trend.

Tests for Classifying Processes

In order to classify a process as covariance stationary, trend stationary or non-stationary, the following tests are available in ScalaTion.

The Augmented Dickey-Fuller (ADF) Test can be used to test whether a times-series has a unit root (alternatively, is covariance stationary)

$$\begin{aligned} H_0 : y_t & \quad \text{has a unit root} \\ H_1 : y_t & \quad \text{is covariance stationary} \end{aligned}$$

The Kwiatkowski–Phillips–Schmidt–Shin (KPSS) Test can be used to test whether a time series is trend stationary (alternatively, has a unit root).

$$\begin{aligned} H_0 : y_t & \quad \text{is trend stationary} \\ H_1 : y_t & \quad \text{has a unit root} \end{aligned}$$

These two popular tests can be applied in combination to classify a process as one of the following:

1. Covariance Stationary: ADF gives H_1 and KPSS gives H_0.
2. Trend Stationary: ADF gives H_0 and KPSS gives H_0 (may warrant further investigation).
3. Non-Stationary: ADF gives H_0 and KPSS gives H_1.

Unless the process is classified as Covariance Stationary, one may difference the time series and apply the tests again. For more information on on testing processes for stationarity, see [217].

As many real-world time series are not initially covariance stationary and differencing can lose some pattern information, there is ongoing research on analyzing non-stationary or locally stationary processes directly [38].

11.8.5 `ARIMA` Class

Class Methods:

```
    @param y        the original input vector (time series data)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters

    class ARIMA (y: VectorD, tt: VectorD = null, hparam: HyperParameter = SARIMAX.hp)
          extends ARMA (y, tt, hparam):
```

```
    protected def init (v: VectorD): Unit =
    def setPQ (pq: VectorI): Unit =
    def showParameterEstimates (): Unit =
    override def train (x_null: MatrixD, y_ : VectorD): Unit =
    protected def nll (b: VectorD): Double =
    protected def updateFittedValues (): Double =
    override def test (x_null: MatrixD, y_ : VectorD): (VectorD, VectorD) =
    override def predictAll (y_ : VectorD): VectorD =
    override def forecast (t: Int = y.dim, h: Int = 1): VectorD =
    override def forecastAll (y_ : VectorD, h: Int): MatrixD =
    def residuals: VectorD = if differenced then y - predictAll (y) else e
```

11.8.6 Exercises

1. Use the `del` method to create a version that takes the number of differences d as a second parameter. Do this (a) iteratively, (b) recursively.

2. Take a first difference of the Lake Level time series dataset and plot the values versus time.

3. Look at the ACF and PACF plots for the Lake Level 'differenced series' to determine values for p and q. Create an ARIMA (p, 1, q) model. Plot its in-sample forecast $\hat{y}$ versus the actual value y and determine its **in-sample** Quality of Fit (QoF) measures.

4. Use Grid Search on 0 to 19 for p and q to find the hyper-parameter values that (a) minimize AIC, (b) minimize sMAPE, (c) minimize MAE, (d) minimize RMSE, (e) maximize R^2, and (f) maximize $\bar{R}^2$. How do these compare?

5. Take the best model according to the consensus in the last question and determine its **out-of-sample** Quality of Fit (QoF) measures for forecasting horizon $h = 1$. Plot its out-of-sample forecast $\hat{y}$ versus the actual value y. Try this for a training to testing ratio of (a) 60-40 and (b) 70-30.

6. Determine the **out-of-sample** Quality of Fit (QoF) measures for forecasting horizons $h = 2$ to 14 (two weeks ahead). How rapidly do the QoF measures decline.

7. Repeat the questions on the Lake-Level dataset for the COVID-19 dataset.

8. Compare an ARIMA (0, 1, 0) model and a Random Walk with Drift model. Remove the constant/drift from this ARIMA model and determine what other model type it matches.

9. Compare an ARIMA (0, 1, 1) model and Simple Exponential Smoothing with Growth model. Remove the constant/drift from this ARIMA model and determine what other model type it matches.

10. Generate and plot the following four types of stochastic processes: covariance stationary process, unit root process, explosive process, and (linear) trend stationary process.

$$\begin{aligned}
y_{t+1} &= 0.99\, y_t + \epsilon_t \\
y_{t+1} &= 1.00\, y_t + \epsilon_t \\
y_{t+1} &= 1.01\, y_t + \epsilon_t \\
y_{t+1} &= 1 + 0.1(t+1) + \epsilon_t
\end{aligned}$$

11.9 SARIMA (Seasonal) Models

In this context, the word "seasonal" is used to mean any time period longer than one. For example, vehicle traffic may exhibit periodic behavior every 24 hours. For Caltrans PeMS data, the time resolution is 5 minutes (smoothed to 15), so the period or seasonal length $s = 288$ time units (96 for the smoothed data). In COVID-19 pandemic forecasting there is a strong weekly period in daily reported deaths, so the seasonal period $s = 7$.

11.9.1 Determination of the Seasonal Period

The two cases mentioned above have obvious periods that are apparent due to knowledge of the process, plotting the time series, or examination of the correlelogram. For example, the Auto-Correlation Function (ACF) may have peaks at lags of 7, 14 and 21 days (diminishing with increasing lag length).

11.9.2 Seasonal Differencing

Seasonal ARIMA allows for seasonal differences, so a SARIMA$(p, d, q)_\times(0, D, 0)_s$ is

$$[1, \boldsymbol{\phi}] \cdot [1, -B^1, \dots, -B^p](1 - B)^d(1 - B^s)^D y_t \;=\; \delta + [1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q]\epsilon_t \tag{11.118}$$

The seasonal differencing $(1-B^s)^D$ happens first, followed by regular differencing $(1-B)^d$ and then essentially an ARMA model.

Considering the COVID-19 daily data, when $s = 7$, (seven days), $D = 1$ (one seasonal difference), and $d = 0$ (no regular difference), the differenced series would be

$$z_t \;=\; (1 - B^s)^D y_t \;=\; y_t \;-\; y_{t-7} \tag{11.119}$$

This differenced time series would indicate how $(y_t$ changes each day from what it was last week on the same date (e.g., Monday to Monday). Although one might think it would be more useful to know the change from the day before, the strong seasonal pattern may override the basic principle that the most recent data is the most useful. A case study of COVID-19 is given later in the section to illustrate this.

11.9.3 Seasonal AR and MA Terms

In addition to the regular Auto-Regressive (AR) and Moving-Average (MA) terms, Seasonal Auto-Regressive parameter vector $\boldsymbol{\Phi} \in \mathbb{R}^P$ and Seasonal Moving-Average parameter vector $\boldsymbol{\Theta} \in \mathbb{R}^Q$ may be included to form a SARIMA$(p, d, q)_\times(P, D, Q)_s$ model.

$$\big[[1, \boldsymbol{\phi}] \cdot [1, -B^1, \dots, -B^p]\big] \big[[1, \boldsymbol{\Phi}] \cdot [1, -B^s, \dots, -B^{Ps}]\big] (1 - B)^d(1 - B^s)^D y_t \;= \tag{11.120}$$

$$\delta + \big[[1, \boldsymbol{\theta}] \cdot [1, B^1, \dots, B^q]\big] \big[[\boldsymbol{\Theta}, 1] \cdot [1, B^s, \dots, B^{Qs}]\big] \epsilon_t \tag{11.121}$$

The expression $\big[[1, \boldsymbol{\phi}] \cdot [1, -B^1, \dots, -B^p]\big]$ may be viewed as a polynomial and the polynomials are multiplied. Because of this, it is referred to as a multiplicative SARIMA model. Other types of SARIMA models include additive and subset [183].

Breaking this into steps gives z_t after seasonal and regular differencing.

$$z_t \;=\; (1-B)^d (1-B^s)^D y_t \tag{11.122}$$

$$\boldsymbol{\phi}(B)\,\boldsymbol{\Phi}(B^s) z_t \;=\; \delta \,+\, \boldsymbol{\theta}(B)\,\boldsymbol{\Theta}(B^s)\,\epsilon_t \tag{11.123}$$

Table 11.8: Characteristic Polynomials for SARIMA Models

Characteristic Polynomial	Notation	Formula
Regular AR(p)	$\boldsymbol{\phi}(B)$	$1-\phi_0 B^1 - \phi_1 B^2 - \cdots - \phi_{p-1} B^p$
Seasonal AR(P)	$\boldsymbol{\Phi}(B^s)$	$1-\phi_0 B^s - \phi_1 B^{2s} - \cdots - \phi_{P-1} B^{Ps}$
Regular MA(q)	$\boldsymbol{\theta}(B)$	$1+\theta_0 B^1 + \theta_1 B^2 + \cdots + \theta_{q-1} B^q$
Seasonal MA(Q)	$\boldsymbol{\Theta}(B^s)$	$1+\Theta_0 B^s + \Theta_1 B^{2s} + \cdots + \Theta_{Q-1} B^{Qs}$

11.9.4 Case Study: COVID-19

The following case study will focus on forecasting COVID-19 Daily Deaths y_t. In this chapter, the forecasts will be based solely in previous values of y_t and shocks ϵ_t. The next chapter on multivariate time series will consider several related variables such as positive-test-rate, hospitalizations, patients-in-ICU, etc.

First, models built using the original times-series y_t will be considered, followed first regular differences, and finally first seasonal differences. Combining both differences is left as an exercise.

Modeling y_t

Modeling $(1-B)y_t$

Modeling $(1-B^2)y_t$

11.9.5 `SARIMA` Class

Class Methods:

```
@param y        the original input vector (time series data)
@param dd       the order of seasonal differencing
@param period   the seasonal period (at least 2)
@param tt       the time vector, if relevant (time index may suffice)
@param hparam   the hyper-parameters

class SARIMA (y: VectorD, dd: Int = 0, period: Int = 2,
              tt: VectorD = null, hparam: HyperParameter = SARIMAX.hp)
      extends ARIMA (y, tt, hparam):

override def modelName: String =
protected override def init (v: VectorD): Unit =
override def setPQ (pq: VectorI): Unit =
override def train (x_null: MatrixD, y_ : VectorD): Unit =
protected override def nll (b: VectorD): Double =
protected override def updateFittedValues (): Double =
override def parameter: VectorD = ϕ ++ θ ++ FF ++ TH
```

```
    override def predictAll (y_ : VectorD): VectorD =
    override def forecast (t: Int = y.dim, h: Int = 1): VectorD =
```

11.9.6 Exercises

1. Write out a SARIMA$(5, 1, 3)_\times(4, 1, 2)_7$ model, with and without using the backshift operator.
2. In predicting a value for y_{t+1} how far back in time would the above SARIMA model need to go?

11.10 Further Reading

1. Forecasting: Principles and Practice [84]
   `https://otexts.com/fpp3`
2. A Course in Time Series Analysis [151]
   `https://web.stat.tamu.edu/~suhasini/teaching673/time_series.pdf`
3. Time Series Analysis: Forecasting and Control [20]

Chapter 12

Multivariate and Nonlinear Time Series

Univariate Time Series (UTS) models only consider how a time series is influenced by past values of the time series itself. Some such models also allow for a trend as a simple function (e.g., linear, quadratic) of time t to be included. Still, UTS modeling and forecasting might be considered myopic, ignoring other potentially important variables or factors.

Multivariate Time Series (MTS) models consider the interaction of multiple related time series. MTS models consider endogenous and exogenous variables. A variable is considered to be exogenous when it is not correlated with the error term. This ideal implies that the values of predictor exogenous variables will not influence the errors. See the exercises in the Auto-Regressive with eXogenous Variables (ARX) section for further discussion of this issue and the notion weak exogeneity. Values from exogenous variables are recorded in the datasets and may be used by models that utilize exogenous variables, such as ARX, SARIMAX, VARX and VARMAX models. However, generally exogenous values are not predicted nor forecasted by the models. Consider the following simple model equation,

$$y_t \;=\; \delta + \phi_0 y_{t-1} + \beta_0 x_t + \epsilon_t \tag{12.1}$$

While x_t being exogenous is uncorrelated with ϵ_t, y_t is influenced by the value of the error term and is hence, endogenous. In time series, it is the endogenous variables that are predicted or forecasted. The discussion in this chapter will start with models that allow only one endogenous variable, the ARX and SARIMAX models. Next up are Vector Auto-Regressive (VAR) models that only have endogenous variables and face the challenge in multi-horizon forecasting of compounding errors across time and series, e.g., errors in forecasts of `hospitalizations` feed into future forecasts of `new_deaths` and their errors. A recent survey of Multivariate Time Series [122] discusses the evolution of such modeling techniques.

Nonlinear Time Series models allow additional functional forms to be fit and therefore, forecast the time series. Linear models are defined, as they were for regression, to be linear in the parameters. Hence the inclusion of quadratic terms does not make the model nonlinear. Note that while linear regression allows for efficient optimization using matrix factorization, having an MA components in the time series model (e.g., $\theta_0\epsilon_{t-1}$) will necessitate the use of nonlinear optimization. The discussion will start with Nonlinear Auto-Regressive (NAR) models and extend into various types of Neural Network architectures.

12.1 Auto-Regressive with eXogenous variables (ARX) Models

Suppose there is another time series x_t that may be a leading indicator for y_t. An example in pandemic modeling would when x_t is hospitalizations and y_t is deaths. Shifting the x_t curve to the right, will tend to make it match up (somewhat) with the y_t. This time shift indicates how much the hospitalizations lead deaths. Note, this tendency may be diffuse, e.g., 8 to 10 days. In this case three lags from x_t could help in predicting y_t.

An Auto-Regressive with eXogenous variables (ARX) model allows for this type of connection, where actual values from x_t can be used to help make y_t forecasts. The X indicates that one or more eXogenous variables are included in the model. Other variables may include the number of patients in ICU, number of positive cases, etc. Related modeling techniques include SARIMAX, VAR, VARX, VARMA and VARMAX models; these are discussed in the next two sections.

12.1.1 The ARX(p) Model

The simplest such model ARX(p) has a model equation that extends the AR(p) model by adding the most recently available value for one exogenous variable.

$$y_t \;=\; \delta \,+\, \phi_0 y_{t-1} \,+\, \phi_1 y_{t-2} \,+\, \ldots \,+\, \phi_{p'} y_{t-p} \,+\, \beta_0 x_{t-1} \,+\, \epsilon_t \tag{12.2}$$

This is the same as the AR(p) model equation from the last chapter with the $\beta_0 x_{t-1}$ term added in. Recall that $p' = p - 1$.

12.1.2 The ARX(p, $[a, b]$) Model

A more useful model is to pick an interval of lags $[a, b]$ over which the exogenous variable has the greatest influence or affect on y_t.

$$y_t \;=\; \delta \,+\, \phi_0 y_{t-1} \,+\, \phi_1 y_{t-2} \,+\, \ldots \,+\, \phi_{p'} y_{t-p} \,+\, \beta_0 x_{t-a} \,+\, \ldots \,+\, \beta_{b-a+1} x_{t-b} \,+\, \epsilon_t \tag{12.3}$$

As x_t may be a leading indicator, starting at x_{t-a} may be more effective than starting at $x_{t=1}$. Note, if $a = 0$, $x_{t-a} = x_t$ is called a *contemporaneous variable*, e.g., for two daily time series, one may be made public before the other and therefore, be put to use in forecasting.

12.1.3 The ARX(p, n, $[a, b]$) Model

Limiting the number of exogenous variable to one, excludes potentially useful information that may improve the forecasting accuracy of the model. Therefore, the next generalization is to allow n exogenous variables that follow the same lag pattern. (Of course, this simplification could be relaxed as well.) The model equation for an ARX(p, n, $[a, b]$) may be written by vectorizing the exogenous part $\mathbf{x}_t = [x_{t0}, x_{t1}, \ldots x_{t,n-1}]$,

$$y_t \;=\; \delta \,+\, \phi_0 y_{t-1} \,+\, \phi_1 y_{t-2} \,+\, \ldots \,+\, \phi_{p'} y_{t-p} \,+\, \boldsymbol{\beta}_0 \cdot \mathbf{x}_{t-a} \,+\, \ldots \,+\, \boldsymbol{\beta}_{b-a+1} \cdot \mathbf{x}_{t-b} \,+\, \epsilon_t \tag{12.4}$$

where parameter/coefficient $\boldsymbol{\beta}_j \in \mathbb{R}^n$.

12.1.4 Determining the Exogenous Lag Interval $[a, b]$

While the Partial AutoCorrelation Function (PACF) can be used to establish a value for the number of endogenous lags p (although hyper-parameter search may be more effective), cross-correlation (or measures derived derived from it) may be used to select the interval $[a, b]$.

12.1.5 Time Series Regression

From another perspective, rather than starting with an AR model and extending it, one may start with a Regression model and extend it. Given a predictor time series x_t and a response time series y_t, a simple times-series regression model may use the current value of x_t to predict y_t.

$$y_t = \beta_0 + \beta_1 x_t + \epsilon_t \tag{12.5}$$

Knowing previous values of x_t often helps with the predictions, so it is common to introduce a lagged predictor variable x_{t-1}

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 x_{t-1} + \epsilon_t \tag{12.6}$$

One may also introduce a lagged response variable y_{t-1}.

$$y_t = \beta_0 + \phi_1 y_{t-1} + \beta_1 x_t + \beta_2 x_{t-1} + \epsilon_t \tag{12.7}$$

Note, when contemporaneous values are not known, x_t should be dropped from the model equation. The parameters can estimated using the efficient Ordinary Least Squares (OLS) algorithm. Doing so may introduce estimation problems for OLS due to temporal dependencies (we are not in the IID world anymore). It may be beneficial to use regularized regression or Generalized Least Square (GLS) [92].

12.1.6 ARX$^A(p, n, k)$ Model

For multi-horizon forecasting, the previous versions of ARX suffer from not being able to use recent values for exogenous variables. The solution of using models from the VAR family introduces more ways in which errors can compound on each other. One intermediate solution is to utilize simple forecasting for the exogenous variables. An ARX Averaged (ARX^A) model allows exogenous variables to be forecasted using Simple Moving Averages.

$$\bar{\mathbf{x}}_t = \frac{1}{k}\sum_{i=0}^{k-1} \mathbf{x}_{t-i} \tag{12.8}$$

Such forecasts tend to be conservative and do not require any parameters to be estimated for the exogenous variables. Then the modeling equation becomes,

$$y_t = \delta + \phi_0 y_{t-1} + \phi_1 y_{t-2} + \ldots + \phi_{p'} y_{t-p} + \boldsymbol{\beta}_0 \cdot \bar{\mathbf{x}}_{t-1} + \ldots + \boldsymbol{\beta}_{p'} \cdot \bar{\mathbf{x}}_{t-p} + \epsilon_t \tag{12.9}$$

Suppose that t represents today, and one wants a two-day ahead forecast (i.e., the forecasting horizon $h = 2$). First a forecast for tomorrow is made $\hat{y}_{t+1}$ to be used with older actual values as follows:

$$y_{t+2} = \delta + \phi_0 y_{t+1} + \phi_1 y_t + \ldots + \phi_{p'} y_{t+2-p} + \boldsymbol{\beta}_0 \cdot \bar{\mathbf{x}}_{t+1} + \ldots + \boldsymbol{\beta}_{p'} \cdot \bar{\mathbf{x}}_{t+2-p} + \epsilon_{t+2} \tag{12.10}$$

When $k = 1$, the averaging will correspond to using a Random Walk Model for each exogenous variable, while when k is large, it will approximate the Null (or Mean) Model.

12.1.7 ARX^A_MV Model

The ARX^A_MV Model uses `RegressionMV` to directly forecast multiple future values, e.g., y_{t+2}, y_{t+1}. Unlike the recursive method that first forecasts $\hat{y}_{t+1}$ and uses it to forecast $\hat{y}_{t+2}$, the direct method fits separate parameters for each future value.

$$y_{t+1} = \delta^{(1)} + \phi_0^{(1)} y_t + \phi_1^{(1)} y_{t-1} + \ldots + \phi_{p'}^{(1)} y_{t-p'} + \boldsymbol{\beta}_0^{(2)} \cdot \bar{\mathbf{x}}_t + \ldots + \boldsymbol{\beta}_{p'}^{(2)} \cdot \bar{\mathbf{x}}_{t-p'} + \epsilon_{t+1} \quad (12.11)$$

$$y_{t+2} = \delta^{(2)} + \phi_0^{(2)} y_t + \phi_1^{(2)} y_{t-1} + \ldots + \phi_{p'}^{(2)} y_{t-p'} + \boldsymbol{\beta}_0^{(2)} \cdot \bar{\mathbf{x}}_t + \ldots + \boldsymbol{\beta}_{p'}^{(2)} \cdot \bar{\mathbf{x}}_{t-p'} + \epsilon_{t+2} \quad (12.12)$$

Note, when $k = 1$, the ARX^A_MV Model reduces to a ARX_MV Model where each $\bar{\mathbf{x}}_t$ is replaced with $\mathbf{x}_t$.

12.1.8 `ARX` Class

Class Methods:

```
@param x        the input/predictor matrix built out of lags of y
                    (and optionally from exogenous variables ex)
@param yy       the output/response vector trimmed to match x.dim
@param lags     the maximum lag included (inclusive)
@param fname    the feature/variable names
@param hparam   the hyper-parameters (use Regression.hp for default)

class ARX (x: MatrixD, yy: VectorD, lags: Int, fname: Array [String] = null,
           hparam: HyperParameter = Regression.hp)
      extends Regression (x, yy, fname, hparam)
         with ForecasterX (lags):

def forecast (t: Int, yf: MatrixD, h: Int): VectorD =
def forecastAt (yf: MatrixD, yx: MatrixD, h: Int): VectorD =
def testF (h: Int, y_ : VectorD, yx: MatrixD): (VectorD, VectorD) =
```

As stated, `yy` is a trimmed version of the given endogenous time series `y`. This is because the first response cannot be forecasted due to missing past values, as there is no value at time $t = -1$ or earlier. Therefore, the size of the `yy` vector is reduced to `y.dim - 1`. Note, some forecasting packages perform back-casting to make a forecast $\hat{y}_0$, while others will require all p past values to be available and therefore do not start making forecasts until $\hat{y}_p$. Thus, care needs to be taken when comparing results from different forecasting packages.

Rather than passing in a user supplied predictor matrix built out of lags of the endogenous variable y_t, the `apply` method in the companion object may be called.

```
@param y        the original un-expanded output/response vector
@param lags     the maximum lag included (inclusive)
@param hparam   the hyper-parameters (use Regression.hp for default)

def apply (y: VectorD, lags: Int, hparam: HyperParameter = Regression.hp): ARX =
```

When there are exogenous variables, $\mathbf{x}_t = [x_{t0}, x_{t1}, \dots x_{t,n-1}]$, the following method should be called.

```
    @param y        the original un-expanded output/response vector
    @param lags     the maximum lag included (inclusive)
    @parax ex       the input matrix for exogenous variables (one per column)
    @param hparam   the hyper-parameters (use Regression.hp for default)
    @param elag1    the minimum exo lag included (inclusive)
    @param elag2    the maximum exo lag included (inclusive)

    def exo (y: VectorD, lags: Int, ex: MatrixD, hparam: HyperParameter = Regression.hp)
            (elag1: Int = max (1, lags / 5), elag2: Int = max (1, lags)): ARX =
```

12.1.9 ARX_MV Object

Object Methods:

```
object ARX_MV:

    @param y           the original un-expanded output/response vector
    @param lags        the maximum lag included (inclusive)
    @param h           the forecasting horizon (1, 2, ... h)
    @param intercept   whether to add a column of all ones to the matrix (intercept)
    @param hparam      the hyper-parameters (use Regression.hp for default)

    def apply (y: VectorD, lags: Int, h: Int, intercept: Boolean = true,
               hparam: HyperParameter = Regression.hp): RegressionMV =
        val (x_, yy) = buildMatrix4TS (y, lags, h)                      // column per lag
        val x = if intercept then VectorD.one (yy.dim) +^: x_ else x_  // add column of ones

        val mod = new RegressionMV (x, yy, null, hparam)
        mod.modelName = s"ARX_MV$lags"
        mod
    end apply
```

When averaging is to used, the corresponding class and object are ARXA and ARXA_MV.

12.1.10 Exercises

1. Compare ARX(p, 0, []) to AR(p) models for the Lake Level dataset. This is the case when there are no exogenous variables. The two models will likely differ slightly, since AR uses the Method of Moments for estimation, while ARX uses either OLS via the `Regression` class or regularized OLS via the `RidgeRegression` class.

2. Compare ARX(p, 0, []) to AR(p) models for the COVID-19 dataset.

3. When errors are correlated, e.g., $\mathbb{E}[\epsilon_{t-1}\epsilon_t] \neq 0$, the significance of relationships may be over-estimated. This is referred to *spurious regression* [60]. Discuss the ramifications of this problem.

4. Weak Exogeneity. OLS results rely on the error at time t being uncorrelated with the current value of the exogenous variable. See `https://www.reed.edu/economics/parker/s12/312/notes/Notes9.pdf`, `https://www.reed.edu/economics/parker/312/tschapters/S13_Ch_2.pdf`.

$$\mathbb{E}\left[\epsilon_t | x_t\right] \;=\; 0 \tag{12.13}$$

Estimate the conditional expectation of errors given `hospitalizations` for COVID-19 dataset. Recall that `new_deaths` is the endogenous variables.

5. For the previous problem, use GLS for parameter estimation, standard errors and forecasting. Compare with the results given by OLS.

12.2 SARIMAX Models

The natural scale up from an ARX model is a Seasonal Auto-Regressive, Integrated, Moving-Average, with eXogenous variables (SARIMAX) model. It adds the ability to consider the effects of shocks and some long-term periodic effects. In addition, it allows differencing of the time series.

The specification of a SARIMAX model subsumes the ARX specification,

$$\text{SARIMAX}(p, d, q) \times (P, D, P)_s[a, b] \tag{12.14}$$

where

- p is the number of Auto-Regressive (AR) terms/lagged endogenous values, e.g., temperature for five ($p = 5$) previous days used to estimate tomorrow's temperature;
- d is the number of stride-1 Differences (Integrations (I)) to take, e.g., focus on the daily change in temperature rather than the temperature itself;
- q is the number of Moving-Average (MA) terms/lagged shocks, e.g., a shock (unexpected change that induces forecast errors) due to the emergence of a new more virulent strain;
- P is the number of Seasonal (stride-s) Auto-Regressive (AR) terms/lagged endogenous values, e.g., for traffic forecasts, values from the previous Mondays will likely work better that values from the previous days;
- D is the number of Seasonal (stride-s) Differences to take; e.g., for daily time series data, taking a difference based on one week may work better that one day;
- Q is the number of Seasonal (stride-s) Moving-Average (MA) terms/lagged shocks; e.g., the start of college football season is a dramatic shock for Saturday traffic forecasts;
- s is the Seasonal period (e.g., week, month, or whatever time period best captures the pattern), e.g., COVID-19 daily data exhibit a strong weekly pattern, so setting $s = 7$ tends to improve the accuracy of the model; and
- $[a, b]$ is the range of eXogenous (X) lags to include (where as a shorthand $[b] = [1, b]$), e.g., as hospitalization data tends to lead new deaths by several days, finding appropriate values for a and b can improve model accuracy.

12.2.1 Model Equations

The SARIMAX model equations extend those given for SARIMA models.

$$z_t = (1 - B)^d (1 - B^s)^D y_t \tag{12.15}$$

$$\boldsymbol{\phi}(B)\, \boldsymbol{\Phi}(B^s) z_t = \delta + \boldsymbol{\theta}(B)\, \boldsymbol{\Theta}(B^s)\, \epsilon_t + \boldsymbol{\beta}(B) x_t \tag{12.16}$$

Again, there may be multiple exogenous variables, i.e., replace x_t with a vector $\mathbf{x}_t$. For COVID-19 forecasting of `new_deaths` as the endogenous variable, potentially useful exogenous variables include `icu_patients, hosp_patients, new_tests, people_vaccinated`, see the exercises.

12.2.2 SARIMAX Object

Class Methods:

```
    @param y        the original endogenous input vector (time series data)
    @param x        the exogenous time series data as an input matrix
    @param dd       the order of seasonal differencing
    @param period   the seasonal period (at least 2)
    @param tt       the time vector, if relevant (time index may suffice)
    @param hparam   the hyper-parameters

    class SARIMAX (y: VectorD, x: MatrixD, dd: Int = 0, period: Int = 2,
                   tt: VectorD = null, hparam: HyperParameter = SARIMAX.hp)
          extends SARIMA (y, dd, period, tt, hparam):
```

12.2.3 Exercises

1. Start with a SARIMA model for forecasting weekly `new_death` and find the best exogenous variable to add to improve the accuracy of the forecasts. Pick the best variable from `icu_patients`, `hosp_patients`, `new_tests`, `people_vaccinated`,

12.3 Vector Auto-Regressive (VAR) Models

Multivariate Time Series Analysis extends univariate time series by analyzing multiple variables. It works on multiple interrelated time series,

$$\mathbf{y}_t = [y_{t0}, y_{t1}, \dots y_{t,n-1}] \tag{12.17}$$

with one time series for each of the n variables [201]. By convention in this text, the time series are oriented in a matrix so that the j^{th} variable is held in the j^{th} column and the t^{th} time point is held in the t^{th} row. This way the matrix holding the data will correspond to how data is typically organized in a csv data file.

The forecasted value for the j^{th} variable at time t, y_{tj}, can depend on the previous (or lagged) values of all the variables. This notion is captured in Vector Auto-Regressive Models [176, 218]. A Vector Auto-Regressive Model of order p with n variables, VAR(p, n), will utilize the most recent p values for each variable to produce a forecast.

12.3.1 VAR(p, 2)

When $n = 1$, VAR(p, n) becomes AR(p), so the simplest actual vector case is VAR(p, 2) also known as a bivariate VAR(p) model. Such models can be useful in traffic forecasting as flow and speed are related variables for which time series data is maintained for each, i.e., y_{t0} is the traffic flow at a particular sensor at time t and y_{t1} is the traffic speed at that sensor at time t.

For each lag, each of the lag variables is used to predict each response variable, so the complete set of parameters forms a tensor consisting of p matrices.

$$\mathbf{\Phi} \;=\; [\Phi^{(0)}, \Phi^{(1)}, \dots, \Phi^{(p-1)}] \quad \text{where} \;\; \Phi^{(l)} \in \mathbb{R}^{n \times n} \tag{12.18}$$

Written in matrix-vector form [80], a bivariate ($n = 2$) order ($p = 3$), VAR(3, 2) model is

$$\mathbf{y}_t \;=\; \boldsymbol{\delta} + \Phi^{(0)}\mathbf{y}_{t-3} + \Phi^{(1)}\mathbf{y}_{t-2} + \Phi^{(2)}\mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t \tag{12.19}$$

and the forecast $\hat{\mathbf{y}}_\mathbf{t}$ is

$$\hat{\mathbf{y}}_\mathbf{t} \;=\; \boldsymbol{\delta} + \Phi^{(0)}\mathbf{y}_{t-3} + \Phi^{(1)}\mathbf{y}_{t-2} + \Phi^{(2)}\mathbf{y}_{t-1} \tag{12.20}$$

where $\boldsymbol{\delta} \in \mathbb{R}^n$ is a constant, $\Phi^{(0)} \in \mathbb{R}^{n\times n}$ is the parameter matrix (first row of tensor) for first lags, $\Phi^{(1)} \in \mathbb{R}^{n\times n}$ is the parameter matrix (second row of tensor) for second lags, $\Phi^{(2)} \in \mathbb{R}^{n\times n}$ is the parameter matrix (third row of tensor) for third lags, and $\boldsymbol{\epsilon}_t \in \mathbb{R}^n$ is the residual/error/shock vector. Notice that $\Phi^{(0)}\mathbf{y}_{t-3}$ is a matrix-vector product yielding an n-dimensional vector. Also see the note about index (superscript/subscript) ordering in the section on AR models.

The equations for each component of the vector $\mathbf{y}_t = [y_{t0}, y_{t1}]$ are

$$\begin{aligned} y_{t0} &= \delta_0 + \boldsymbol{\phi}_0^{(0)} \cdot \mathbf{y}_{t-3} + \boldsymbol{\phi}_0^{(1)} \cdot \mathbf{y}_{t-2} + \boldsymbol{\phi}_0^{(2)} \cdot \mathbf{y}_{t-1} + \epsilon_{t1} && \text{flow} \\ y_{t1} &= \delta_1 + \boldsymbol{\phi}_1^{(0)} \cdot \mathbf{y}_{t-3} + \boldsymbol{\phi}_1^{(1)} \cdot \mathbf{y}_{t-2} + \boldsymbol{\phi}_1^{(2)} \cdot \mathbf{y}_{t-1} + \epsilon_{t1} && \text{speed} \end{aligned}$$

where for example, the vector $\boldsymbol{\phi}_0^{(1)} = [\phi_{00}^{(1)}, \phi_{01}^{(1)}]$ is the first row of matrix $\Phi^{(1)}$. The two equations may be combined and rewritten in expanded matrix-vector notation.

$$\mathbf{y}_t \;=\; \begin{bmatrix} y_{t0} \\ y_{t1} \end{bmatrix} \;=\; \begin{bmatrix} \delta_0 \\ \delta_1 \end{bmatrix} \;+\; \begin{bmatrix} \phi^{(0)}_{00} & \phi^{(0)}_{01} \\ \phi^{(0)}_{10} & \phi^{(0)}_{11} \end{bmatrix} \mathbf{y}_{t-3} \;+\; \begin{bmatrix} \phi^{(1)}_{00} & \phi^{(1)}_{01} \\ \phi^{(1)}_{10} & \phi^{(1)}_{11} \end{bmatrix} \mathbf{y}_{t-2} \;+\; \begin{bmatrix} \phi^{(2)}_{00} & \phi^{(2)}_{01} \\ \phi^{(2)}_{10} & \phi^{(2)}_{11} \end{bmatrix} \mathbf{y}_{t-1} \;+\; \boldsymbol{\epsilon}_t$$

Convention: known values are shown in black and unknown values are shown in blue. For example, given values from the past (e.g., $\mathbf{y}_{t-1}, \mathbf{y}_{t-2}$), before the beginning of day t, make a forecast for the value of $\mathbf{y_t}$. Note, the forecast is known $\hat{\mathbf{y}}_\mathbf{t}$, but the actual value $\mathbf{y_t}$ is not yet known.

12.3.2 VAR(p, n)

A general Vector Auto-Regressive VAR(p, n) model with n variables each with p lags will have p parameter matrices. The equation for the response vector $\mathbf{y}_t \in \mathbb{R}^n$ may be written in matrix-vector form as follows:

$$\mathbf{y}_t \;=\; \boldsymbol{\delta} \,+\, \Phi^{(0)}\mathbf{y}_{t-p} \,+\, \Phi^{(1)}\mathbf{y}_{t-p+1} \,+\, \dots \,+\, \Phi^{(p-1)}\mathbf{y}_{t-1} \,+\, \boldsymbol{\epsilon}_t \qquad (12.21)$$

where constant vector $\boldsymbol{\delta} \in \mathbb{R}^n$, parameter matrices $\Phi^{(l)} \in \mathbb{R}^{n\times n}$ $(l = 0, \dots p-1)$, and error vector $\boldsymbol{\epsilon}_t \in \mathbb{R}^n$.

12.3.3 Training

One way to train a VAR model is to treat each variable as a separate regression problem and use least squares to estimate the parameters, e.g., the parameters for y_{tj} are matrix $\Phi_{:j}$. A more efficient approach taken by ScalaTion is to use the `RegressionMV` class that fits multiple responses to multiple predictor variables.

12.3.4 `VAR` Object

The `apply` method in the `VAR` object takes a multivariate time series as a `y` matrix and created predictor variables by taking lagged values for each of the variables to form the input matrix `x`. For example, in the bivariate case where `y` has 2 columns, `lags = 3`, and `intercept = true` the `x` matrix will consist of 7 columns.

```scala
object VAR:

    @param y           the original un-expanded output/response matrix
    @param lags        the maximum lag included (inclusive)
    @param h           the forecasting horizon (1, 2, ... h)
    @param intercept   whether to add a column of all ones to the matrix (intercept)
    @param hparam      the hyper-parameters (use Regression.hp for default)

    def apply (y: MatrixD, lags: Int, h: Int, intercept: Boolean = true,
               hparam: HyperParameter = Regression.hp): RegressionMV =

        var x  = ARX.makeExoCols (lags, y, 1, lags+1)       // add cols for each lagged vars
        val yy = y(1 until y.dim)                           // trim y to match x
        if intercept then x = VectorD.one (yy.dim) +^: x    // add first column of all ones

        val mod = new RegressionMV (x, yy, null, hparam)
        mod.modelName = s"VAR_$lags"
        mod
    end apply
```

For example, when run on the `GasFurnace` dataset that consists of 296 rows (time steps) and 2 columns (gas flow rate and Carbon Dioxide concentration variables), the `VAR (3, 2)` produces the following results for In-Sample testing.

```
MatrixD (210.004,      1467.40,                  // intercept
         -0.170147,    0.243170,                 // lag 3 for y_0
         -0.0755910,   -0.454592,                // lag 2 for y_0
         1.14269,      0.0743863,                // lag 1 for y_0
         0.0152980,    0.416977,                 // lag 3 for y_1
         -0.0660577,   -1.60076,                 // lag 2 for y_1
         0.0589660,    2.17496)                  // lag 1 for y_1
```

The above matrix captures all the parameters as follows:

$$\boldsymbol{\delta} \;=\; [\delta_0, \delta_1] \;=\; [210.004, 1467.40]$$

$$\Phi^{(0)} \;=\; \begin{bmatrix} \phi^{(0)}_{00} & \phi^{(0)}_{01} \\ \phi^{(0)}_{10} & \phi^{(0)}_{11} \end{bmatrix} \;=\; \begin{bmatrix} -0.170147 & 0.243170 \\ 0.0152980 & 0.416977 \end{bmatrix}$$

$$\Phi^{(1)} \;=\; \begin{bmatrix} \phi^{(1)}_{00} & \phi^{(1)}_{01} \\ \phi^{(1)}_{10} & \phi^{(1)}_{11} \end{bmatrix} \;=\; \begin{bmatrix} -0.0755910 & -0.454592 \\ -0.0660577 & -1.60076 \end{bmatrix}$$

$$\Phi^{(2)} \;=\; \begin{bmatrix} \phi^{(2)}_{00} & \phi^{(2)}_{01} \\ \phi^{(2)}_{10} & \phi^{(2)}_{11} \end{bmatrix} \;=\; \begin{bmatrix} 1.14269 & 0.0743863 \\ 0.0589660 & 2.17496 \end{bmatrix}$$

12.3.5 AR$^*(p, n)$

An AR$^*(p, n)$ model is a VAR(p, n) model where all of the parameter matrices are diagonal. Thus, each variable/time series y_{tj} is modeled independently, so for an AR$^*(3, n)$ model, the j^{th} equation is

$$y_{tj} \;=\; \delta_j \,+\, \phi^{(0)}_{jj} y_{t-3,j} \,+\, \phi^{(1)}_{jj} y_{t-2,j} \,+\, \phi^{(2)}_{jj} y_{t-1,j} \,+\, \epsilon_{tj}$$

12.3.6 Exercises

1. Plot the flow y_{t0} and speed y_{t1} given in the Traffic Sensor Dataset (`traffic.csv`). Estimate values for the four parameters contained in the one parameter matrix $\Phi^{(0)}$ of a VAR (1, 2) model ($p = 1, n = 2$). Compute the Quality of Fit (QoF).

2. Use the Traffic Sensor Dataset (`traffic.csv`) to estimate values for the three parameter matrices $\Phi^{(0)}$, $\Phi^{(1)}$ and $\Phi^{(2)}$ of a VAR (3, 2) model ($p = 3, n = 2$). Compute the Quality of Fit (QoF) and compare with the VAR(1, 2) model.

3. Consider what happens if two times series $\{y_{t0}\}$ and $\{y_{t1}\}$ are following similar trends. What difficulties could this cause in a VAR model.

4. How could a Vector Error Correction Model (VECM) handle the above problem?
5. Compare an ARX model with a VAR model for the COVID-19 weekly dataset.

12.4 Nonlinear Time Series Models

12.4.1 Nonlinear Auto-Regressive (NAR)

As was the case for Nonlinear Regression, Nonlinear Time Series models have the potential for better fitting models.

A p^{th} order Nonlinear Auto-Regressive NAR(p) model is a generalization of an AR(p) model. The forecasted value at time t, y, is a function of previous values of y, e.g.,

$$\mathbf{x}_t = [y_{t-p}, \ldots, y_{t-1}] \tag{12.22}$$

$$y_t = f(\mathbf{x}_t; \boldsymbol{\phi}) + \epsilon_t \tag{12.23}$$

where $\boldsymbol{\phi}$ is a vector of p parameters and p is also taken the number of lagged values to use. In general, for nonlinear models the number of lags may or may not equal the number of parameters p.

12.4.2 Auto-Regressive Neural Network (ARNN)

A special case of a NAR(p) is an Auto-Regressive Neural Network ARNN(p) model. It is a three-layer (one hidden) neural network, where the input layer has a node for each of the p lags, the hidden layer also has p nodes, and the output layer has 1 node.

$$\mathbf{x}_t = [y_{t-p}, \ldots, y_{t-1}] \tag{12.24}$$

$$y_t = \mathbf{w} \cdot \mathbf{f}(\Phi\, \mathbf{x}_t) + \epsilon_t \tag{12.25}$$

The $p \times p$ matrix Φ holds the parameters/weights connecting the input and hidden layers, while the p dimensional vector $\mathbf{w}$ holds the parameters/weights connecting the hidden and output layers. There is only an activation function (vectorized $\mathbf{f}$) for the hidden layer. Note that $\Phi\, \mathbf{x}_t$ is matrix-vector multiplication. Of course, the basic ARNN model can be embellished.

12.4.3 Nonlinear Auto-Regressive, Moving-Average (NARMA)

A (p, q) order Nonlinear Auto-Regressive, Moving-Average NARMA(p, q) model, is a generalization of an ARMA(p, q) model.

$$\mathbf{x}_t = [y_{t-p}, \ldots, y_{t-1}] \qquad \text{input : past values} \tag{12.26}$$

$$\mathbf{e}_t = [\epsilon_{t-p}, \ldots, \epsilon_{t-1}] \qquad \text{input : past shocks} \tag{12.27}$$

$$y_t = f(\mathbf{x}_t, \mathbf{e}_t; \boldsymbol{\phi}, \boldsymbol{\theta}) + \epsilon_t \qquad \text{output} \tag{12.28}$$

12.5 Recurrent Neural Networks (RNN)

Regular neural networks are often referred to as feed-forward neural networks, as there is no feedback in the calculations. Recurrent Neural Networks (RNN) provide a straightforward mechanism that uses feedback to allow the past to be captured (metaphorically remembered).

As exponential smoothing maintains a state variable that summarizes the past, with the importance of past values decaying with age, recurrent neural networks maintain a *hidden state vector* $\mathbf{h}_t$. The new state vector is computed as a combination of the prior state vector and the current input vector. The forecasted value (output) is determined from the state vector. For simplicity, here it is assumed the dimensionality of the output vector ($k = 1$).

A single hidden layer RNN (p, n_h) has one input layer, one hidden layer, and one output layer, much like a 3-layer Fully-Connected Neural Network. The difference is that the hidden state vector is fed back into the calculation for the next time step. The hyper-parameter p can denote the dimensionality of the input vector, while n_h can denote the number of hidden units (or the dimensionality of the hidden state vector).

The size (and shape) of the input depends on the modeling approach. The size p (as in AR(p) models) could be used to indicate the number of lags. One may say that the past is captured by the hidden state, but it may be helpful to explicitly include lags. Now in multi-variate time-series, there will be multiple variables, so p could denote the number of variables (e.g., `new_deaths`, `hosp_patients` for Covid-19 forecasting) Often, it will be useful to include both, so here we use p for lags and n_v for variables, and then the input at time t becomes a matrix X_t Thus, the input over time becomes a 3D tensor $\mathbf{X}$ (time $\times$ lags $\times$ variables).

12.5.1 RNN$(1, 1)$

The simplest Recurrent Neural Network has input dimension one and hidden dimension one, i.e., it has one hidden unit/node and makes forecasts based on the most recent information. For such models, the hidden state vector is one dimensional (i.e., scalar). In order to forecast a value for y_t, denoted $\hat{y}_t$, a weighted combination of the previous value $x_t = y_{t-1}$ and the previous state h_{t-1} are passed to an activation function f.

$$\begin{aligned}
x_t &= y_{t-1} && \text{input scalar} \\
h_t &= f(\phi\, x_t + w\, h_{t-1} + \beta^{(h)}) && \text{hidden state scalar} \\
\hat{y}_t &= v h_t + \beta^{(y)} && \text{output scalar}
\end{aligned}$$

where ϕ and w are scalar parameters/weights and $\beta^{(h)}$ and $\beta^{(y)}$ are the scalar biases. The above RNN model is a NARMA$(1, 1)$ model [33].

The hidden state variable h_t is defined recursively and as such provides *memory* to the model, since its value results from all previous values [124].

Notice that if the activation function f is the identity function, $v = 1$, and $\beta^{(y)} = 0$, then the middle equation becomes,

$$h_t \;=\; \beta^{(h)} \,+\, \phi\, x_t \,+\, w\, h_{t-1} \tag{12.29}$$

so in this case it is an ARMA$(1, 1)$ model, since $h_{t-1} = y_{t-1} - \epsilon_{t-1}$.

12.5.2 RNN(p, n_h)

A single hidden layer Recurrent Neural Network of order (p, n_h) makes forecasts based on information going back p lags, with a n_h-dimensional hidden state vector. The scalar parameters ϕ and w now become parameter matrices Φ (renamed U) and W. They are given in pre-transposed form to facilitate the direct application of matrix multiplication with the need to take a transpose.

$$\begin{aligned}
\mathbf{x}_t &= [y_{t-p}, \dots, y_{t-1}] && \text{input vector} \\
\mathbf{h}_t &= \mathbf{f}(U\mathbf{x}_t + W\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(h)}) && \text{hidden state vector} \\
\hat{y}_t &= g(V\mathbf{h}_t + \beta^{(y)}) && \text{output scalar}
\end{aligned}$$

where the variables are

- $\mathbf{x}_t \in \mathbb{R}^p$ holds the collected inputs (time series values from the recent past)
- $\mathbf{h}_t \in \mathbb{R}^{n_h}$ holds the current hidden state
- $\hat{y}_t \in \mathbb{R}^k$ holds the 1-step to k-steps ahead forecasts (assumed $k = 1$)

and the parameters (weights and biases) are

- $U \in \mathbb{R}^{n_h \times p}$ is the input to hidden layer (auto-regressive) weight matrix
- $W \in \mathbb{R}^{n_h \times n_h}$ is the hidden unit to hidden unit weight matrix
- $V \in \mathbb{R}^{k \times n_h}$ is the hidden to output layer weight matrix
- $\boldsymbol{\beta}^{(h)} \in \mathbb{R}^{n_h}$ is the n_h-dimensional hidden layer bias vector
- $\beta^{(y)} \in \mathbb{R}^k$ is the k-dimensional (assumed $k = 1$) output layer bias vector

and the activation functions are

- $\mathbf{f} : \mathbb{R}^{n_h} \to \mathbb{R}^{n_h}$ hidden layer activation function defaults to the tanh function (vectorized)
- $g : \mathbb{R}^{n_h} \to \mathbb{R}^k$ output layer activation function defaults to the identity function (scalar when $k = 1$)

Note, the hidden layer is recurrent, while the output layer may be dense. This RNN model is a NARMA model [33].

The computation of the forecasted value $\hat{y}_t$ is depicted in Figure 12.1 where the recurrent unit is within the purple box.

The hidden layer is typically implemented by looping through all of the time steps, taking the next input $\mathbf{x}_t$ and the prior hidden state vector $\mathbf{h}_{t-1}$ into the unit for calculation.

Figure 12.2 show a small, yet complete Recurrent Neural Network (RNN) with $p = 2$ (two lags), $n_h = 2$ (two hidden nodes/units), and $k = 1$ (size of output). As the RNN iterates over time t the calculations are repeated with new inputs $\mathbf{x_t}$ and the hidden state $\mathbf{h}_{t-1}$ saved from the last iteration. This saved vector is shown in the orange box.

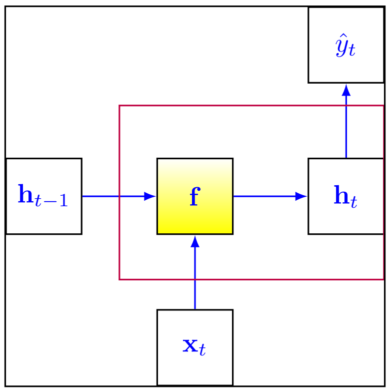


Figure 12.1: Recurrent Neural Network (RNN)

As vector equations, the hidden layer calculations are the following:

$$h_{t0} = f(\mathbf{u}_0 \cdot \mathbf{x_t} + \mathbf{w}_0 \cdot \mathbf{h}_{t-1} + \beta_0^{(h)}) \tag{12.30}$$
$$h_{t1} = f(\mathbf{u}_1 \cdot \mathbf{x_t} + \mathbf{w}_1 \cdot \mathbf{h}_{t-1} + \beta_1^{(h)}) \tag{12.31}$$

Similarly, the output layer calculation is the following:

$$\hat{y}_t = g(\mathbf{v}_0 \cdot \mathbf{h_t} + \beta^{(y)}) \tag{12.32}$$

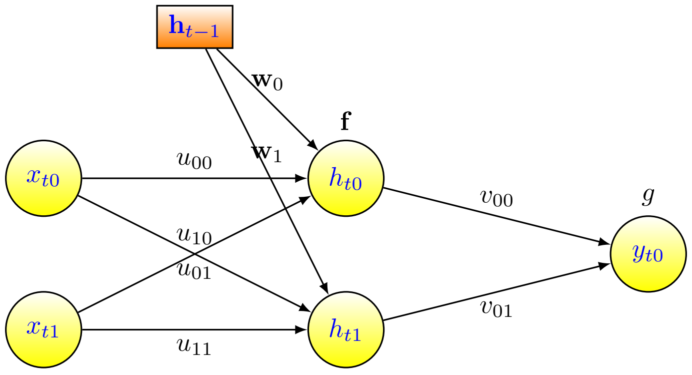


Figure 12.2: Three-Layer (input, hidden, output) Recursive Neural Network (RNN) with Biases Removed

Unfortunately, such neural networks may have stability problems, so work moved unto units with gates that add stability (e.g., to avoid vanishing or exploding gradients). Gates are used to control how much of the previous state is preserved as signals propagate through the units. This allows gated units to have longer memories that simple RNNs.

12.5.3 RNN(p, n_h, n_v)

When there are multiple variables (e.g., $n_v = 2$ with one for COVID new hospitalizations and one for new deaths) the situation becomes more complex. In this case, the input input is a matrix X_t and the output/response is a vector, $\mathbf{y_t} = [y_{t0}, y_{t1}, \dots, y_{t,n_v-1}]$.

$$
\begin{aligned}
X_t &= [\mathbf{y}_{t-p}, \dots, \mathbf{y}_{t-1}] && \text{input matrix} \\
\mathbf{h}_t &= \mathbf{f}(U\,\text{flatten}(X_t) + W\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(h)}) && \text{hidden state} \\
\hat{\mathbf{y}}_\mathbf{t} &= \mathbf{g}(V\mathbf{h}_t + \boldsymbol{\beta}^{(y)}) && \text{output vector}
\end{aligned}
$$

where the variables are

- $X_t \in \mathbb{R}^{n_v \times p}$ holds the collected inputs (time series values from the recent past)
- $\mathbf{h}_t \in \mathbb{R}^{n_h}$ holds the current hidden state
- $\hat{\boldsymbol{y}}_t \in \mathbb{R}^k$ holds the 1-step to k-steps ahead forecasts

and the parameters (weights and biases) are

- $U \in \mathbb{R}^{n_h \times n_v p}$ is the input to hidden layer (auto-regressive) weight matrix
- $W \in \mathbb{R}^{n_h \times n_h}$ is the hidden unit to hidden unit weight matrix
- $V \in \mathbb{R}^{k \times n_h}$ is the hidden to output layer weight matrix
- $\boldsymbol{\beta}^{(h)} \in \mathbb{R}^{n_h}$ is the n_h-dimensional hidden layer bias vector
- $\boldsymbol{\beta}^{(y)} \in \mathbb{R}^k$ is the k-dimensional output layer bias vector

and the activation functions are

- $\mathbf{f} : \mathbb{R}^{n_h} \to \mathbb{R}^{n_h}$ hidden layer activation function defaults to the tanh function (vectorized)
- $\mathbf{g} : \mathbb{R}^{n_h} \to \mathbb{R}^k$ output layer activation function defaults to the identity function (vectorized)

12.5.4 Training

Below is a simple gradient descent implementation for train the RNN on a dataset (full or training).

```
def train (): Unit =
    for it <- 1 to max_epochs do
        forward ()             // forward propagate: get intermediate and output results

        println (s"train: for epoch $it: loss function L = $L")
        banner (s"train: for epoch $it: total loss function L.sum = ${L.sum}")

        backward ()            // back propagate: calculate gradients (partial derivatives)

        update_params ()       // update parameters (weights and biases)
    end for
end train
```

12.5.5 Optimization

Forward Pass

The forward method performs forward propagatiion to calculate yp, loss and intermediate variables for each step.

```
    def forward (): Unit =
        for t <- 0 until n_seq do
            val h_pre = if t == 0 then h_m1 else h(t-1)        // get previous hidden state
            h(t) = tanh_ (U * x(t) + W * h_pre + b_h)          // compute new hidden state
            if CLASSIF then
                yp(t) = softmax_ (V * h(t) + b_y)              // act: softmax for classif
                L(t)  = (-y(t) * log_ (yp(t))).sum             // cross-entropy loss funct
            else
                yp(t) = V * h(t) + b_y                         // act: id for forecasting
                L(t)  = (y(t) - yp(t)).normSq                  // sse loss function
            end if
        end for
    end forward
```

Backward Pass

The backward method performs back-propagation to calculate gradients using chain rules.

```
    def backward (): Unit =

        import ActivationFun.tanhD

        // start back-propagation with the final/feed-forward (ff) layer (uses id)

        val e = yp - y                                         // negative error matrix
        db_y  = e.sumVr                                        // vector of row sums
        for t <- 0 until n_seq do dV += outer (e(t), h(t))     // outer vector product
        val dh_ff = e * V                                      // partial w.r.t. h
        var dh = new VectorD (dh_ff.dim2)                      // hold partial dh @ time t
        var dIn: VectorD = null

        // calculate the derivative contribution of each step and add them up

        for t <- n_seq-1 to 1 by -1 do                         // move back in time to t=1
            dh += dh_ff(t)                                     // update partial for dh @ t
            dIn = dh * tanhD (hg(t))                           // input to tanh for hidden
            hg += (dIn, x(t), h(t-1))                          // update partials for hidden
            dh = W.𝒯 * dIn                                     // 𝒯 => matrix transpose
        end for

        // end case @ time t = 0 -> use h_m1 for hidden state

        dh += dh_ff(0)                                         // update partial for dh @ t=0
        dIn = dh * tanhD (hg(0))
        hg += (dIn, x(0), h_m1)                                // update hidden gate @ t=0
        dh_m1 = W.𝒯 * dIn
    end backward
```

Parameter Update

Based on the calculated partial derivatives, update the parameters (weights and biases).

```
    def update_params (): Unit =
        // hidden state (h)
        U   -= hg.dU * eta
        W   -= hg.dW * eta
        b_h -= hg.db * eta

        // output layer
        V   -= dV * eta
        b_y -= db_y * eta
    end update_params
```

12.5.6 Exercises

1. Use ScalaTion's `RNN` class to make forecasts based on the Lake Level dataset given in `Example_LakeLevels`.

2. Create a `SimpleRNN` model using Keras, Guide: `https://keras.io/guides/working_with_rnns/`, `https://faroit.com/keras-docs/2.0.5/layers/recurrent`, API: `https://keras.io/api/layers/recurrent_layers/simple_rnn/` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

3. Create a `RNN` model using PyTorch, `https://pytorch.org/docs/stable/generated/torch.nn.RNN.html` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

12.6 Gated Recurrent Unit (GRU) Networks

Gated units have alleviated some of the problems with traditional RNNs. As the simplest gated unit, a Minimal Gated Unit (MGU) has a single gate and a minimal number of parameters compared to other gated units.

A Gated Recurrent Unit (GRU) [30, 29] is slightly more complex, but is more commonly use than an MGU. It adds a second gate, a third pair of parameter matrices and a third bias vector. The two gates in a GRU are the *reset gate* and the *update gate*. These are used to control the degree to which the previous state $\mathbf{h}_{t-1}$ factors, along with the current input $\mathbf{x}_t$, into the calculation of the new state $\mathbf{h}_t$, which will be a mixture of the previous state $\mathbf{h}_{t-1}$ and new candidate state $\tilde{\mathbf{h}}_t$ (or `c(t)` in the code) coming out of the activation function $\mathbf{f}$. Figure 12.3 shows how signals propagate through one unit. This unit would be connected to one on the left taking input $\mathbf{x}_{t-1}$ and one on the right taking input $\mathbf{x}_{t+1}$. The state provides memory for the next unit.

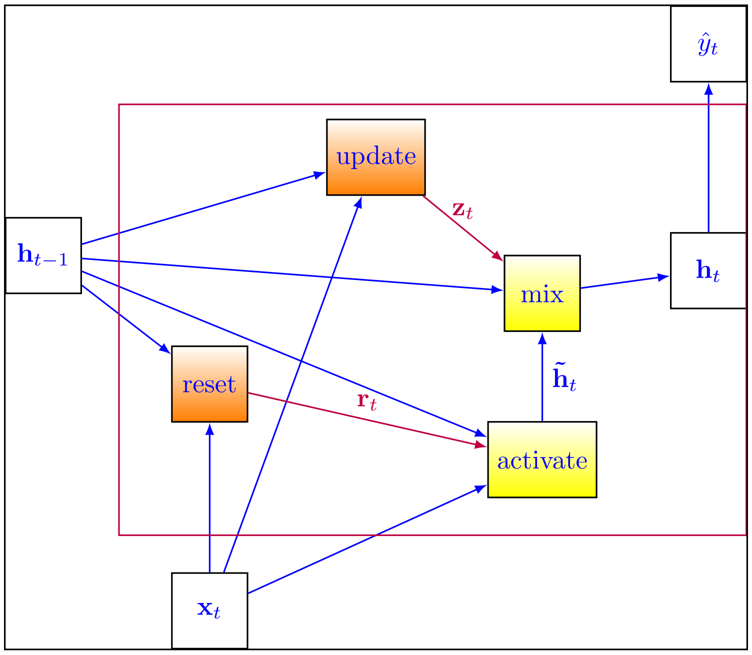


Figure 12.3: Gated Recurrent Unit (GRU)

The elements in the control vectors, reset $\mathbf{r}_t$ and update $\mathbf{z}_t$, come through sigmoid activation so they are always between 0 (open circuit) and 1 (closed circuit) and are shown in purple.

The following two equations indicate one way that a GRU could be set up to handle univariate time series data, e.g., a NAR(p) model.

$$\begin{aligned} \mathbf{x}_t &= [y_{t-1}, \dots, y_{t-p}] && \text{input} \\ \hat{y}_t &= g(\mathbf{h}_t) && \text{forecast} \end{aligned}$$

Other forms of feature selection or engineering could be done as well; one could include time t, shocks, etc. or extend to multivariate time series analysis.

GRU Equations

The equations below show how information flows through a gated recurrent unit [30, 215, 211].

Reset Gate. The degree to which the previous state $\mathbf{h}_{t-1}$ influences the new candidate state $\tilde{\mathbf{h}}_t$ is controlled by the reset gate. When the reset gate is open ($\mathbf{r}_t$ is close to zero) the previous state is all but ignored, whereas, when the gate is closed, its influence will be strong.

$$\mathbf{r}_t \ = \ \mathbf{f}_\sigma(U_r \mathbf{x}_t + W_r \mathbf{h}_{t-1} + \boldsymbol{\beta}^{(r)}) \tag{12.33}$$

Update Gate. The update gate controls the relative mixture of the previous state $\mathbf{h}_{t-1}$ and the new candidate state $\tilde{\mathbf{h}}_t$ used to form the new state $\mathbf{h}_t$. When the update gate is open ($\mathbf{z}_t$ is close to zero) the previous state is passed through almost intact, whereas, when the gate is closed, the new state is essentially the candidate state.

$$\mathbf{z}_t \ = \ \mathbf{f}_\sigma(U_z \mathbf{x}_t + W_z \mathbf{h}_{t-1} + \boldsymbol{\beta}^{(z)}) \tag{12.34}$$

Activate. A candidate state $\tilde{\mathbf{h}}_t$ can be created by applying the activation function f (e.g., tanh or reLU) to the weighted combination of the previous state $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$. However, the reset control $\mathbf{r}_t$ can be used to cut off the influence of the previous state.

$$\tilde{\mathbf{h}}_t \ = \ \mathbf{f}(U_c \mathbf{x}_t + W_c[\mathbf{r}_t * \mathbf{h}_{t-1}] + \boldsymbol{\beta}^{(c)}) \tag{12.35}$$

Mix. The new state $\mathbf{h}_t$ is created as a mixture of the previous state $\mathbf{h}_{t-1}$ and the newly created candidate state $\tilde{\mathbf{h}}_t$. The update control $\mathbf{z}_t$ determines how much of each are put into the new state $\mathbf{h}_t$.

$$\mathbf{h}_t \ = \ (\mathbf{1} - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \tilde{\mathbf{h}}_t \tag{12.36}$$

When the update gate is nearly open ($\mathbf{z}_t$ near $\mathbf{0}$), the new state preserves much of previous state, while when it is nearly closed ($\mathbf{z}_t$ near $\mathbf{1}$), the new state is mainly given by the candidate state. Notice that when both $\mathbf{r}_t$ and $\mathbf{z}_t = \mathbf{1}$, the GRU equations reduce to those of the simple RNN given in the last section (see exercises). Recall that $*$ (or $\odot$) is the element-wise vector product.

GRU Variables and Parameters

A Gated Recurrent Unit introduces a greater number of variables and parameters than a dense Feed Forward Neural Network. The variables for a GRU unit consist of one variable that serves as input $\mathbf{x}_t$, one that represents state and is modified by the unit having before $\mathbf{h}_{t-1}$ and after $\mathbf{h}_t$ values, a candidate state $\tilde{\mathbf{h}}_t$, and two control variables, $\mathbf{r}_t$ and $\mathbf{z}_t$. These five vector-valued variables are listed in Table 12.1. All but one the variables have dimension n_h (or `n_mem` in the code), the dimensionality of state variables (memory size). That one, $\mathbf{x}_t$ has dimension p corresponding to the number of lags in univariate time series (or course feature engineering can be used to add more). For multi-variate time series it corresponds to the number of variables n_v (`n_var` in the code). Note, using both multiple lags and multi-variate time series will require $\mathbf{x}_t$ to be a 2-dimensional matrix.

$$X_t \in \mathbb{R}^{n_v \times p} \qquad \text{input matrix at time } t \tag{12.37}$$

Table 12.1: GRU Variables

variable	**dimensions**	**name**
$\mathbf{x}_t$	$\mathbb{R}^p$	input vector
$\mathbf{r}_t$	$\mathbb{R}^{n_h}$	reset control vector
$\mathbf{z}_t$	$\mathbb{R}^{n_h}$	update control vector
$\tilde{\mathbf{h}}_t$	$\mathbb{R}^{n_h}$	candidate state vector
$\mathbf{h}_t$	$\mathbb{R}^{n_h}$	hidden state vector

The parameters for a GRU unit consist of six weight/parameter matrices (in three pairs) and three bias vectors as listed in Table 12.2.

Table 12.2: GRU Parameters (Weight Matrices and Bias Vectors)

parameter	**dimensions**	**name**
U_r	$\mathbb{R}^{n_h \times p}$	input-to-reset weight matrix
W_r	$\mathbb{R}^{n_h \times n_h}$	state-to-reset weight matrix
$\boldsymbol{\beta}^{(r)}$	$\mathbb{R}^{n_h}$	reset bias vector
U_z	$\mathbb{R}^{n_h \times p}$	input-to-update weight matrix
W_z	$\mathbb{R}^{n_h \times n_h}$	state-to-update weight matrix
$\boldsymbol{\beta}^{(z)}$	$\mathbb{R}^{n_h}$	update bias vector
U_c	$\mathbb{R}^{n_h \times p}$	input-to-activate matrix
W_c	$\mathbb{R}^{n_h \times n_h}$	state-to-activate weight matrix
$\boldsymbol{\beta}^{(c)}$	$\mathbb{R}^{n_h}$	activate bias vector

The matrices may be thought of a pre-transposed to facilitate application of matrix multiplication without the need to transpose.

12.6.1 A GRU Layer

So far, a GRU has been discussed in isolation. To better visualize the role of GRUs in a neural network, first consider the three-layer Dense Feed-Forward Neural Network consisting of an input, hidden and output layer as shown in 12.4.

Now, swap a GRU layer in for the dense hidden layer. Let the number of lags $p = 2$. Then at time t, the network maps input $\mathbf{x}_t = [y_{t-2}, y_{t-1}]$ to output y_t.

The U weights shown in Figure 12.5 are meant to represent the weights in the three weight matrices, U_r, U_z and U_c, while the W weights are meant to represent the weights in the three weight matrices, W_r, W_z and W_c. Each of the two gates has its own weight matrices and the candidate state has it own as well.

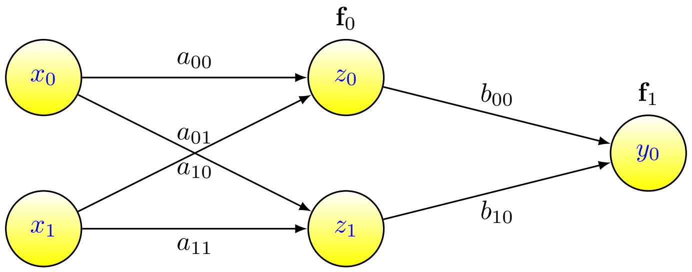


Figure 12.4: Three-Layer (input, hidden, output) Neural Network with Biases Removed

Each of the two units takes the input vector $\mathbf{x}_t$ and the previous state $\mathbf{h}_{t-1}$ and computes the next state $\mathbf{h}_t$. Computations, are performed over all n_h units in the form of matrix-vector multiplications, e.g., $W_r\mathbf{h}_{t-1}$, as depicted in Figure 12.3. The states coming out of the GRU layer are feed into a final dense layer to make one step-head forecasts $\hat{y}_t$. (This treatment can be extended for multi-horizon forecasts.)

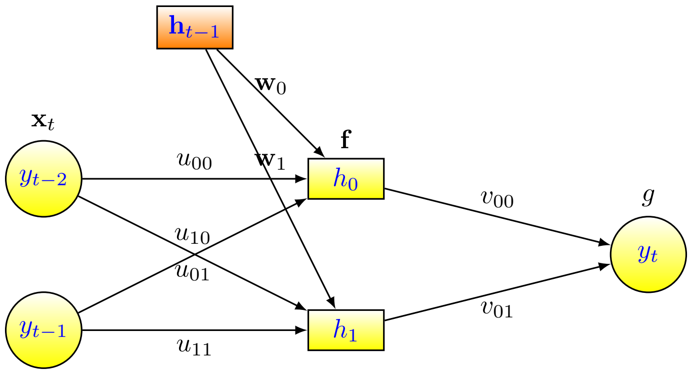


Figure 12.5: Three-Layer (input, hidden, output) Neural Network with a GRU Layer

The GRU iterates through time (using a loop in implementation). A useful way to visualize the execution is to duplicate the unit for each of the m timestamps in the time series. Figure 12.6 illustrates this, imagining h_0 and h_1 executing at time 0, $t-1$, t, $t+1$, and $m-1$. Note, unless imputation or back-casting is used, there is no input for time 0. Consequently, for this example, one would set $\mathbf{x}_0 = \mathbf{0}$ and $\mathbf{h}_0 = \mathbf{0}$.

See [44, 211] for additional details.

12.6.2 Training

Below is a simple gradient descent implementation for train the GRU on a dataset (full or training).

```
def train (): Unit =
    for it <- 1 to max_epochs do
        forward ()            // forward propagate: get intermediate and output results

```

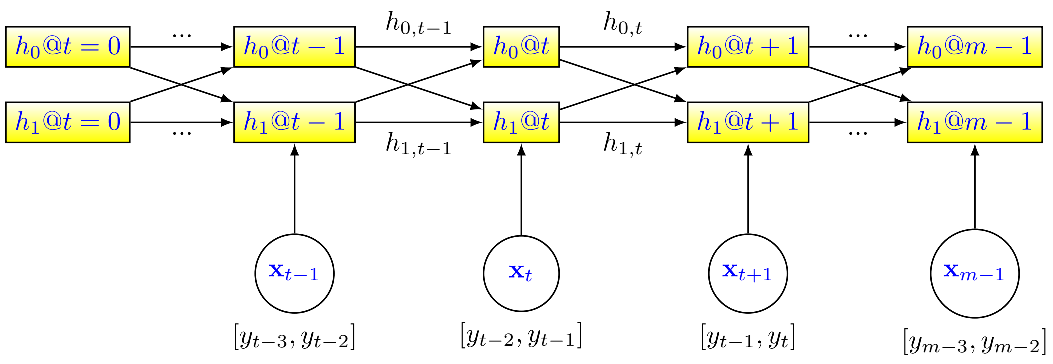


Figure 12.6: GRU in Execution over Time

```
        println (s"train: for epoch $it: loss function L = $L")
        banner (s"train: for epoch $it: total loss function L.sum = ${L.sum}")

        backward ()           // back propagate: calculate gradients (partial derivatives)

        update_params ()      // update parameters (weights and biases)
    end for
end train
```

See the exercises for how to replace gradient descent with stochastic gradient descent using mini-batches.

12.6.3 Optimization

As shown in the code above, there are three parts: a forward pass, a backward pass and a parameter update.

Forward Pass

Using the identity activation function for the hidden to output layer yields the following forward propagation equations.

$$\begin{aligned}
\mathbf{r}_t &= \mathbf{f}_\sigma(U_r\mathbf{x}_t + W_r\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(r)}) \\
\mathbf{z}_t &= \mathbf{f}_\sigma(U_z\mathbf{x}_t + W_z\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(z)}) \\
\tilde{\mathbf{h}}_t &= \mathbf{f}(U_c\mathbf{x}_t + W_c[\mathbf{r}_t * \mathbf{h}_{t-1}] + \boldsymbol{\beta}^{(c)}) \\
\mathbf{h}_t &= (\mathbf{1} - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \tilde{\mathbf{h}}_t \\
\hat{y}_t &= V\mathbf{h}_t \; + \; \beta^{(y)}
\end{aligned}$$

This case is for univariate times series, i.e., $\mathbf{y}_t = [y_0, y_1, \dots y_{m-1}]$, in which case $V \in \mathbb{R}^{1 \times n_h}$.

```
def forward (): Unit =
    for t <- 0 until n_seq do
        val h_pre = if t == 0 then h_m1 else h(t-1)            // get previous hidden state
        r(t) = sigmoid_ (Ur * x(t) + Wr * h_pre + b_r)         // reset gate
        z(t) = sigmoid_ (Uz * x(t) + Wz * h_pre + b_z)         // update gate
        c(t) = tanh_ (Uc * x(t) + Wc * (r(t) * h_pre) + b_c)  // candidate state
        h(t) = (_1 - z(t)) * h_pre + z(t) * c(t)               // hidden state
        if CLASSIF then
            yp(t) = softmax_ (V * h(t) + b_y)                 // act: softmax for classif
            L(t)  = (-y(t) * log_ (yp(t))).sum                // cross-entropy loss function
        else
            yp(t) = V * h(t) + b_y                            // act: id for forecasting
            L(t)  = (y(t) - yp(t)).normSq                     // sse loss function
        end if
    end for
end forward
```

Consequently, the sse (or divide by $m-1$ for mse) loss function on the training (or full) dataset of size m is

$$\mathcal{L} \;=\; \sum_{t=1}^{m-1} (y_t - \hat{y}_t)^2 \tag{12.38}$$

As there is no data for predicting $\hat{y}_0$, it is not considered. The following are the trainable parameters: weight matrices: $U_r, W_r, U_z, W_z, U_c, W_c, V$, and bias vectors: $\beta^{(r)}, \beta^{(z)}, \beta^{(c)}, \beta^{(y)}$.

Backward Pass

The `Gate` case class holds information on the gate's value and its partial derivatives.

```
@param n_seq  the length of the time series
@param n_mem  the size for hidden state (h) (dimensionality of memory)
@param n_var  the number of variables

case class Gate (n_seq: Int, n_mem: Int, n_var: Int):

    val v  = new MatrixD (n_seq, n_mem)                    // gate value: time x state
    var dU = new MatrixD (n_mem, n_var)                    // partial w.r.t. weight matrix U
    var dW = new MatrixD (n_mem, n_mem)                    // partial w.r.t. weight matrix W
    var db = new VectorD (n_mem)                           // partial w.r.t. bias vector b

    def apply (t: Int): VectorD = v(t)

    def update (t: Int, vv: VectorD): Unit = v(t) = vv
    def += (dIn: VectorD, x_t: VectorD, h_tm1: VectorD): Unit =
            { dU += outer (dIn, x_t); dW += outer (dIn, h_tm1); db += dIn }

end Gate
```

$$\begin{aligned}
\partial_U \mathcal{L} \;&{+}{=}\; \mathbf{d}_{in} \otimes \mathbf{x}_t \\
\partial_W \mathcal{L} \;&{+}{=}\; \mathbf{d}_{in} \otimes \mathbf{h}_{t-1} \\
\partial_b \mathcal{L} \;&{+}{=}\; \mathbf{d}_{in}
\end{aligned}$$

Candidate c Mixin

$$\begin{aligned}
\mathbf{d}_{hbk} &= \partial_{\mathbf{h}_t}\mathcal{L} \\
\mathbf{d}_{in} &= \partial_{\mathbf{h}_t}\mathcal{L} * (\mathbf{1} - \mathbf{z}_t) * \tanh'(\tilde{\mathbf{h}}_t) \\
\tilde{\mathbf{h}}_t &\mathrel{+}= (\mathbf{d}_{in}, \mathbf{x}_t, \mathbf{h}_{t-1} * \mathbf{r}_t) \\
\mathbf{d}_{hr} &= W_c^{\intercal} * \mathbf{d}_{in} \\
\partial_{\mathbf{h}_t}\mathcal{L} &= \mathbf{d}_{hr} * \mathbf{r}_t
\end{aligned}$$

Reset r Gate

$$\begin{aligned}
\mathbf{d}_{in} &= \mathbf{d}_{hr} * \mathbf{h}_{t-1} * \text{sigmoid}'(\mathbf{r}_t) \\
\mathbf{r}_t &\mathrel{+}= (\mathbf{d}_{in}, \mathbf{x}_t, \mathbf{h}_{t-1}) \\
\partial_{\mathbf{h}_t}\mathcal{L} &\mathrel{+}= W_r^{\intercal} * \mathbf{d}_{in} + \mathbf{d}_{hbk} * \mathbf{z}_t
\end{aligned}$$

Update z Gate

$$\begin{aligned}
\mathbf{d}_{in} &= \mathbf{d}_{hbk} * (\tilde{\mathbf{h}}_t - \mathbf{h}_{t-1}) * \text{sigmoid}'(\mathbf{z}_t) \\
\mathbf{z}_t &\mathrel{+}= (\mathbf{d}_{in}, \mathbf{x}_t, \mathbf{h}_{t-1}) \\
\partial_{\mathbf{h}_t}\mathcal{L} &\mathrel{+}= W_z^{\intercal} * \mathbf{d}_{in} + \mathbf{d}_{hbk} * \mathbf{z}_t
\end{aligned}$$

The `backward` method has three parts: (1) start back-propagation with the final/feed-forward (ff) layer (uses id for activation); (2) loop back in time adding to the partials for U, W and $\mathbf{b}$, as well as for the state at time t, $\mathbf{h_t}$; (3) handle the end case for $t = 0$ where $\mathbf{h_{-1}}$ becomes `h_m1`.

```
def backward (): Unit =
    val e = yp - y                                           // negative error matrix
    db_y  = e.sumVr                                          // vector of row sums
    for t <- 0 until n_seq do dV += outer (e(t), h(t))    // outer vector product
    val dh_ff = e * V                                        // partial w.r.t. h: n_seq by n_mem
    var dh = new VectorD (dh_ff.dim2)                        // hold partial for hidden state
    var dIn, dhr: VectorD = null

    for t <- n_seq-1 to 1 by -1 do                           // move back in time to t = 1
        dh += dh_ff(t)                                       // update partial: hidden state @ t
        val dh_bk = dh                                       // save dh

        dIn = dh * (_1 - z(t)) * tanhD (c(t))                // input to tanh: candidate mixin c
        c  += (dIn, x(t), h(t-1) * r(t))                     // update partials: c mixin
        dhr = Wc.𝒯 * dIn                                     // matrix transpose
        dh  = dhr * r(t)

        dIn = dhr * h(t-1) * sigmoidD (r(t))                 // input to sigmoid: reset gate r
        r  += (dIn, x(t), h(t-1))                            // update partials: r gate
        dh += Wr.𝒯 * dIn + dh_bk * z(t)

        dIn = dh_bk * (c(t) - h(t-1)) * sigmoidD (z(t))      // input to sigmoid: update gate z
        z  += (dIn, x(t), h(t-1))                            // update partials: z gate
        dh += Wz.𝒯 * dIn
    end for
    ...
end backward
```

For completeness the code (corresponding to the ... above) for the end case ($t = 0$) is show below.

```
    dh += dh_ff(0)                                            // update partial: h hidden @ t = 0

    dIn = dh * (_1 - z(0)) * tanhD (c(0))
    c  += (dIn, x(0), h_m1 * r(0))                            // update partials: c mixin @ t = 0
    dhr = Wc.𝒯 * dIn
    dh_m1 += dhr * r(0)

    dIn = dhr * h_m1 * sigmoidD (r(0))
    r  += (dIn, x(0), h_m1)                                   // update partials: r gate @ t = 0
    dh_m1 += Wr.𝒯 * dIn + dh * z(0)

    dIn = dh * (h_m1 - c(0)) * sigmoidD (z(0))
    z  += (dIn, x(0), h_m1)                                   // update partials: z gate @ t = 0
    dh_m1 += Wz.𝒯 * dIn
```

Parameter Update

After computing values for the variables in the forward pass and computing partial derivatives in the backward pass, the partials moderated by the learning rate η (or `eta`) can be subtracted from the current values for the parameters (weight matrices and bias vectors).

```
def update_params (): Unit =
    // update gate (z)
    Uz  -= z.dU * eta
    Wz  -= z.dW * eta
    b_z -= z.db * eta

    // reset gate (r)
    Ur  -= r.dU * eta
    Wr  -= r.dW * eta
    b_r -= r.db * eta

    // candidate state (c)
    Uc  -= c.dU * eta
    Wc  -= c.dW * eta
    b_c -= c.db * eta

    // output layer
    V   -= dV * eta
    b_y -= db_y * eta
end update_params
```

12.6.4 Exercises

1. Use ScalaTion's `GRU` class to make forecasts based on the Lake Level dataset given in `Example_LakeLevels`.

2. Create a `GRU` model using Keras, Guide: `https://keras.io/guides/working_with_rnns/`, `https://faroit.com/keras-docs/2.0.5/layers/recurrent`, API: `https://keras.io/api/layers/recurrent_layers/gru/` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

3. Create a `GRU` model using PyTorch, `https://pytorch.org/docs/stable/generated/torch.nn.GRU.html` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

4. When both the reset gate $\mathbf{r}_t$ and the update gate $\mathbf{z}_t$ are fixed at $\mathbf{1}$, show that the GRU equations reduce to those of the simple RNN given in the last section.

5. Show that the weight matrices U_r and W_r can be combined into one to make a slightly more concise formula for the reset gate $\mathbf{r}_t$ (same for the update gate).

$$\mathbf{r}_t \;=\; \mathbf{f}_\sigma(U_r\mathbf{x}_t + W_r\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(r)})$$

   Given $U_r \in \mathbb{R}^{n_h \times p}$, $W_r \in \mathbb{R}^{n_h \times n_h}$, $\mathbf{h}_{t-1} \in \mathbb{R}^{n_h}$, and $\mathbf{x}_t \in \mathbb{R}^p$, show that if $W = [U_r, W_r] \in \mathbb{R}^{n_h \times (p+n_h)}$, then

$$W \begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} \;=\; U_r\mathbf{x}_t \;+\; W_r\mathbf{h}_{t-1} \tag{12.39}$$

6. Use the above identity to rewrite the GRU equations using three parameter matrices and three bias vectors.

7. Write a formula for the total number of trainable parameters in a GRU where the state vector is n_h-dimensional and the input vector is p-dimensional.

8. Replace gradient descent with stochastic gradient descent using mini-batches.

9. Explain the difference between stateless and stateful training of a GRU (or MGU, LSTM) [44].

12.7 Minimal Gated Unit (MGU) Networks

A Minimal Gated Unit (MGU) [215] has a single gate and a minimal number of parameters compared to other gated units. Note, the GRU reset and update gates as well as their corresponding vectors are unified into the forget gate in an MGU. An MGU has a *forget gate* and its purpose is to weigh accumulated past information versus new information. The greater the forgetting the greater the reliance on recent data.

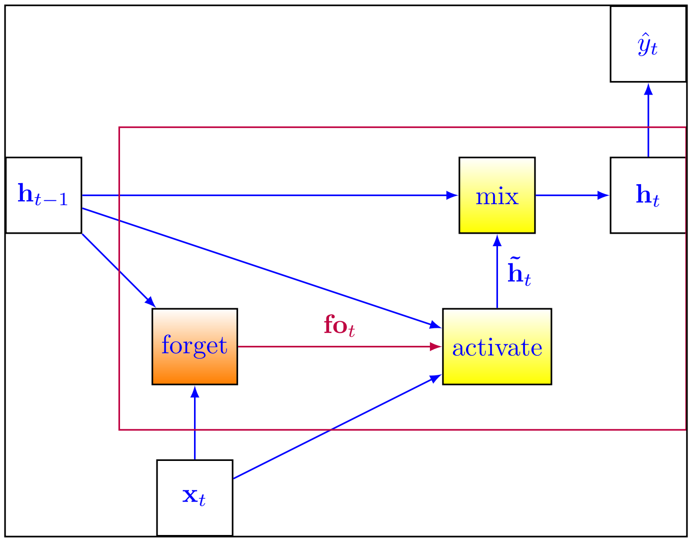


Figure 12.7: Minimal Gated Unit (MGU)

The input can be engineered like it was for GRUs, e.g.,

$$\mathbf{x}_t = [y_{t-1}, \dots, y_{t-p}]$$

MGU Equations

The equations below show how information flows through a minimal gated unit [215, 211].

$$\begin{aligned}
\mathbf{fo}_t &= \mathbf{f}_\sigma(U_{fo}\mathbf{x}_t + W_{fo}\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(fo)}) && \text{forget gate} \\
\tilde{\mathbf{h}}_t &= \mathbf{f}(U_c\mathbf{x}_t + W_c[\mathbf{fo}_t * \mathbf{h}_{t-1}] + \boldsymbol{\beta}^{(c)}) && \text{candidate state} \\
\mathbf{h}_t &= (\mathbf{1} - \mathbf{fo}_t) * \mathbf{h}_{t-1} + \mathbf{fo}_t * \tilde{\mathbf{h}}_t && \text{state}
\end{aligned}$$

Notice that when $\mathbf{fo}_t$ is essentially $\mathbf{1}$, the equations reduce to those of a simple RNN.

MGU Variables and Parameters

The variables for an MGU unit consist of one variable that serves as input $\mathbf{x}_t$, one that represents state and is modified by the unit having before $\mathbf{h}_{t-1}$ and after $\mathbf{h}_t$ values, a candidate state $\tilde{\mathbf{h}}_t$, and one control variable, $\mathbf{fo}_t$. These four vector-valued variables are listed in Table 12.3.

Table 12.3: MGU Variables

variable	dimensions	name
$\mathbf{x}_t$	$\mathbb{R}^p$	input vector
$\mathbf{fo}_t$	$\mathbb{R}^{n_h}$	forget control vector
$\tilde{\mathbf{h}}_t$	$\mathbb{R}^{n_h}$	candidate state vector
$\mathbf{h}_t$	$\mathbb{R}^{n_h}$	hidden state vector

The parameters consist of two pairs of weight/parameter matrices along with the biases, see Table 12.4.

Table 12.4: MGU Parameters (Weight Matrices and Bias Vectors)

parameter	dimensions	name
U_{fo}	$\mathbb{R}^{n_h \times p}$	input-to-forget weight matrix
W_{fo}	$\mathbb{R}^{n_h \times n_h}$	state-to-forget weight matrix
$\boldsymbol{\beta}^{(fo)}$	$\mathbb{R}^{n_h}$	forget bias vector
U_c	$\mathbb{R}^{n_h \times p}$	input-to-activate matrix
W_c	$\mathbb{R}^{n_h \times n_h}$	state-to-activate weight matrix
$\boldsymbol{\beta}^{(c)}$	$\mathbb{R}^{n_h}$	activate bias vector

The first activation function serves as a switch (typically sigmoid) and the second activation function defaults to tanh, but other activation functions (e.g., reLU) may be used.

In the above formulation, the dimensions of the vectors are follows: $\mathbf{x}_t \in \mathbb{R}^p$, $\mathbf{fo}_t, \tilde{\mathbf{h}}_t$ and $\mathbf{h}_t \in \mathbb{R}^{n_h}$, where p is the number features engineered into the input and n_h is the number of units.

Forget Gate. The purpose of the forget gate is to determine how much of the previous state to forget. When the forget gate is open ($\mathbf{fo}_t$ is close to zero) the previous state $\mathbf{h}_{t-1}$ is passed through almost intact, whereas, when the gate is closed, the new state is essentially the candidate state. The value of $\mathbf{fo}_t$ also determines the influence of the previous state $\mathbf{h}_{t-1}$ has over the new candidate state $\tilde{\mathbf{h}}_t$.

12.8 Long Short Term Memory (LSTM) Networks

Long Short Term Memory (LSTM) [75] Networks provide increased memory by introducing another state variable called the cell state $\mathbf{c}_t$ that works in parallel with the hidden state $\mathbf{h}_t$. For additional control of how past information is propagated, three gates are used: a forget gate, an input gate, and an output gate. The corresponding vectors, $\mathbf{fo}_t$, $\mathbf{in}_t$, and $\mathbf{ou}_t$, hold values in $(0, 1)$ and thus act as switches to control the flow.

An LSTM has advantages over a GRU when its longer memory is beneficial and the time series is long enough for effective training. An LSTM has more parameters to train than a GRU, so it takes longer to train.

LSTM Equations

The equations below show how information flows through a Long Short-Term Memory network [75, 211].

Forget Gate. The new cell state takes a fraction of the previous cell state and a fraction of the new candidate cell state (see below). The forget gate determines how much of the previous cell state is kept; when it is open (near zero) this information is forgotten, while when it is closed (near one) it is strongly remembered. (It may be more intuitive to think of this as a remember gate.)

$$\mathbf{fo}_t \;=\; \mathbf{f}_\sigma(U_{fo}\mathbf{x}_t + W_{fo}\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(fo)}) \tag{12.40}$$

Input Gate. The new candidate cell state (see below) as a weighted combination of the input $\mathbf{x_t}$ and the previous hidden state $\mathbf{h}_{t-1}$ is a key calculation in an LSTM. The input gate controls how much of the new candidate cell state goes into the actual new cell state. When the input gate is open (near zero) the new candidate cell state has little influence, while when it is closed, the new candidate cell state enters the calculation at full strength.

$$\mathbf{in}_t \;=\; \mathbf{f}_\sigma(U_{in}\mathbf{x}_t + W_{in}\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(in)}) \tag{12.41}$$

Output Gate. The output gate comes into play at the end of the unit calculations and is used to moderate the activated cell state before it assigned to the hidden state. Open dampens it, while closed does not. This provides additional stability in the hidden state $\mathbf{h_t}$, which is important as it widely fed back into most of the LSTM equations.

$$\mathbf{ou}_t \;=\; \mathbf{f}_\sigma(U_{ou}\mathbf{x}_t + W_{ou}\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(ou)}) \tag{12.42}$$

Candidate Cell State. As with a GRU, an LSTM first calculates a candidate state, a candidate for the cell state (the computed values in an LSTM are on the cell state, and at the end a final calculation is performed to determine the hidden state). The candidate cell state $\mathbf{cc}_t$ is computed by applying the activation function f (e.g., tanh or reLU) to the weighted combination of the current input $\mathbf{x}_t$ and previous state $\mathbf{h}_{t-1}$.

$$\mathbf{cc}_t \;=\; \mathbf{f}(U_{cc}\mathbf{x}_t + W_{cc}\mathbf{h}_{t-1} + \boldsymbol{\beta}^{(cc)}) \tag{12.43}$$

Cell State. The actual cell state $\mathbf{c}_t$ is a combination of the previous cell state $\mathbf{c}_{t-1}$ and the candidate cell state $\mathbf{cc}_t$, where the fraction of $\mathbf{c}_{t-1}$ included is determined by the forget gate, while the fraction of $\mathbf{cc}_t$ included is determined by the input gate.

$$\mathbf{c}_t = \mathbf{fo}_t * \mathbf{c}_{t-1} + \mathbf{in}_t * \mathbf{cc}_t \tag{12.44}$$

Hidden State. As the principal output of LSTM units, the hidden state $\mathbf{h}_t$ may be thought of the cell state $\mathbf{c}_t$ with an activation function applied to it. For added stability (remember an RNN has problems with calculations blowing up), this value is moderated since the value of the output gate is in $(0, 1)$.

$$\mathbf{h}_t = \mathbf{ou}_t * \mathbf{f}(\mathbf{c}_t) \tag{12.45}$$

For more details on how information flows through an LSTM, see [75, 52, 44].

LSTM Variables and Parameters

The variable for a LSTM unit consist of three control variables, two state variables and one internal candiate state variable described in Table 12.5.

Table 12.5: LSTM Variables

variable	**dimensions**	**name**
$\mathbf{x}_t$	$\mathbb{R}^p$	input vector
$\mathbf{fo}_t$	$\mathbb{R}^{n_h}$	forget control vector
$\mathbf{in}_t$	$\mathbb{R}^{n_h}$	input control vector
$\mathbf{ou}_t$	$\mathbb{R}^{n_h}$	output control vector
$\mathbf{cc}_t$	$\mathbb{R}^{n_h}$	candidate cell state vector
$\mathbf{c}_t$	$\mathbb{R}^{n_h}$	cell state vector
$\mathbf{h}_t$	$\mathbb{R}^{n_h}$	hidden state vector

The parameters for a LSTM unit consist of six weight/parameter matrices (in three pairs) and three bias vectors as listed in Table 12.6.

Table 12.6: GRU Parameters (Weight Matrices and Bias Vectors)

parameter	**dimensions**	**name**
U_{fo}	$\mathbb{R}^{n_h \times p}$	input-to-forget gate weight matrix
W_{fo}	$\mathbb{R}^{n_h \times n_h}$	state-to-forget gate weight matrix
$\boldsymbol{\beta}^{(fo)}$	$\mathbb{R}^{n_h}$	forget gate bias vector
U_{in}	$\mathbb{R}^{n_h \times p}$	input-to-input gate weight matrix
W_{in}	$\mathbb{R}^{n_h \times n_h}$	state-to-input gate weight matrix
$\boldsymbol{\beta}^{(in)}$	$\mathbb{R}^{n_h}$	input gate bias vector
U_{ou}	$\mathbb{R}^{n_h \times p}$	input-to-output gate weight matrix
W_{ou}	$\mathbb{R}^{n_h \times n_h}$	state-to-output gate weight matrix
$\boldsymbol{\beta}^{(ou)}$	$\mathbb{R}^{n_h}$	output gate bias vector
U_{cc}	$\mathbb{R}^{n_h \times p}$	input-to-candiate cell matrix
W_{cc}	$\mathbb{R}^{n_h \times n_h}$	state-to-candiate cell weight matrix
$\boldsymbol{\beta}^{(c)}$	$\mathbb{R}^{n_h}$	candiate cell bias vector

12.8.1 Exercises

1. Use ScalaTion's `LSTM` class to make forecasts based on the Lake Level dataset given in `Example_LakeLevels`.

2. Create a `LSTM` model using Keras, Guide: `https://keras.io/guides/working_with_rnns/`, `https://faroit.com/keras-docs/2.0.5/layers/recurrent`, API: `https://keras.io/api/layers/recurrent_layers/lstm/` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

3. Create a `LSTM` model using PyTorch, `https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html` for Lake Level dataset. Compare the results to those obtained with ScalaTion.

12.9 Encoder-Decoder Architectures

One may consider ways to improve LSTM (or GRU) networks and incremental improvement may be obtained by adding LSTM and/or Feed-Forward layers. It may be beneficial to replace or augment the Feed-Forward layer with a decoder, for example one containing LSTM units.

The purpose of the *encoder* it to create a context vector that captures the input in a summarized/encoded form. This may simply be the hidden state at time t or $\mathbf{h}_t$ produced by LSTM units. It may be more complex and include multiple hidden state vectors.

The *context vector* serves as the initial state vector for the decoder/second portion of the network architecture. In order to adjust the dimensionality of the data (inputs vs. outputs), a single layer Feed-Forward layer may be used (in line with the V matrix discussed in prior sections).

While the *encoder* can be thought of as encoding the past, the decoder uses this encoding to forecast the future. The hope is that the two parts can specialize their skill, with the encoder striving to better capture the patterns in data and the decoder at using patterns to make accurate forecasts.

12.9.1 Simple Encoder-Decoder Consisting of Two GRU Cells

Let us define cell to mean all the units used in the computation of $\mathbf{h}_t$ over time. For example, the figure in the GRU section had two units (in practice it could 64 or more). Figure 12.8 compresses the figure in the GRU section to the cell level. Although the two figures share much in common, there are two cells and each has its own set of trainable parameters. (Remember the figure shows the cells unrolled over time, there are only two with each executed in a loop.)

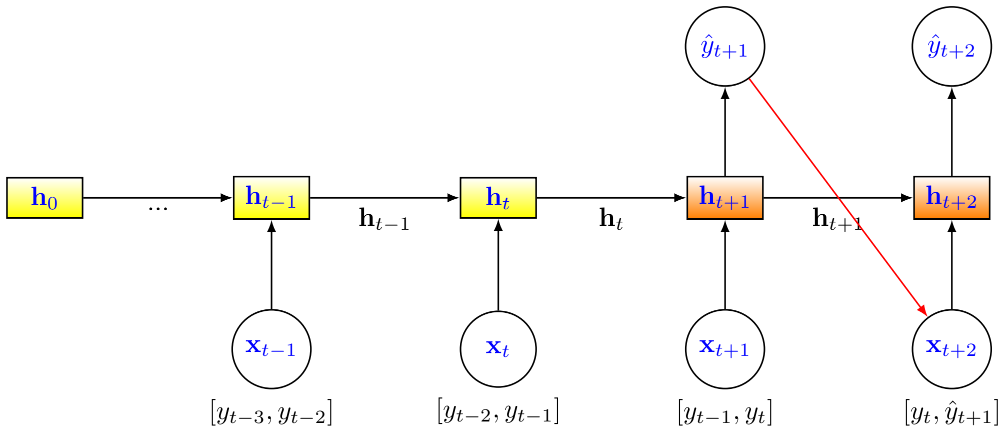


Figure 12.8: Encoder(yellow)-Decoder(orange) Inference over Time

The intuition is that training the parameters in the first GRU (the encoder) will be optimized to capture patterns in the data, while parameters in the second GRU will optimized on making accurate forecasts given the encoding saved in the context vector. Depending on the number of lags making up the input $\mathbf{x}_t$, forecasts at later horizons will need prior forecasts to be feed into them as shown by the red line in the figure.

12.9.2 Teacher Forcing

The figure shows what happens during inferencing (actual forecasting). Training is a bit different. First there is Back-Propagation Through Time (BPTT), ... Second, actual values, such as y_{t+1} may be used rather than forecasted values $\hat{y}_{t+1}$. This is not possible (or allowed) during actual forecasting, as it would constitute knowing the future. However, some degree of teacher forcing, using actual for forecasted values, has been shown potential [204].

12.9.3 Attention Mechanisms

Notice that the decoder, as discussed so far, only uses the last hidden state vector $\mathbf{h}_t$ from the encoder. Furthermore, the longer the time series (or sequence), the more challenging it is for $\mathbf{h}_t$ to stand in for what the encoder "knows". Of course, with an LSTM, the cell state $\mathbf{c}_t$ could be used as well. However, the vast majority of all the calculated state vectors are simply ignored. Taking all this information may overwhelm the decoder with marginally relevant information. If there was only a way to choose the most relevant information. This can be cast as another learning problem that involves computing *alignment scores* and normalizing them as *attention weights*.

Alignment Scores

An alignment score between encoder state $\mathbf{h}_\tau$ and decoder state $\mathbf{s}_t$ (renamed to distinguish it from an encoder states) is given by $a_{\tau t}$

$$a_{\tau t} \;=\; \mathbf{v}_a \cdot \tanh(U_a \mathbf{s}_{t-1} + W_a \mathbf{h}_\tau) \tag{12.46}$$

The vector $\mathbf{v}_a$ and matrices U_a, and W_a are trainable parameters.

Attention Weights

The attention weights are found by normalizing the alignments scores to the range $(0, 1)$ using the softmax function.

$$\alpha_{\tau t} = \frac{e^{a_{\tau t}}}{\sum_\tau e^{a_{\tau t}}} \tag{12.47}$$

Context Vector

The context vector used by the decoder at time t is then

$$\mathbf{c}_t \;=\; \sum_{\tau=1}^{m} \alpha_{\tau t} \mathbf{h}_\tau \tag{12.48}$$

i.e., it uses state vectors from the encoder $\mathbf{h}_\tau$ weighted by their importance in calculating $\mathbf{s}_t$ as specified by their attention weights. Define the function context to consist of the above three equations,

$$\mathbf{c}_t \;=\; \text{context}(\mathbf{s}_{t-1}, H) \tag{12.49}$$

where the matrix H maintains the hidden state vectors from the encoder GRU, such that the τ^{th} row of H stores $\mathbf{h}_\tau$.

Modifications to GRU Equations

A slight modification to the GRU equations for the decoder is needed [213]. There is useful state information from the previous time as well as the from context vector. These need to be combined in some fashion, for example, by vector concatenation.

$$\mathbf{cs}_t = [\mathbf{c}_t \,|\, \mathbf{s}_t] \tag{12.50}$$

The simple change is just to replace the state vector from the GRU section $\mathbf{h}$ with the concatenated state vector $\mathbf{cs}$ in the GRU equations.

$$\begin{aligned}
\mathbf{r}_t &= \mathbf{f}_\sigma(U_r\mathbf{x}_t + W_r\mathbf{cs}_{t-1} + \boldsymbol{\beta}^{(r)}) \\
\mathbf{z}_t &= \mathbf{f}_\sigma(U_z\mathbf{x}_t + W_z\mathbf{cs}_{t-1} + \boldsymbol{\beta}^{(z)}) \\
\tilde{\mathbf{h}}_t &= \mathbf{f}(U_{\tilde{h}}\mathbf{x}_t + W_{\tilde{h}}[\mathbf{r}_t * \mathbf{cs}_{t-1}] + \boldsymbol{\beta}^{(\tilde{h})}) \\
\mathbf{c}_t &= \text{context}(\mathbf{s}_{t-1}, H) \\
\mathbf{s}_t &= (\mathbf{1} - \mathbf{z}_t) * \mathbf{s}_{t-1} + \mathbf{z}_t * \tilde{\mathbf{h}}_t \\
\mathbf{cs}_t &= [\mathbf{c}_t \,|\, \mathbf{s}_t] \\
\hat{y}_t &= V\mathbf{cs}_t \; + \; \beta^{(y)}
\end{aligned}$$

12.9.4 Exercises

1. Notice in the equations above, the context vector and state vector are treated the same way throughout, e.g., both the context vector and state vector are regulated by the reset control vector $\mathbf{r}_t$ when computing $\tilde{\mathbf{h}}_t$. It may be beneficial to only do this to the state vector and not the control vector. Rewrite the above equations to achieve this.

   Hint: keep the context and state vectors separate and do not use $\mathbf{cs}_t$; also introduce new parameter matrices to include $\mathbf{c}_t$ into the equations.

2. Consider other ways of combining the context and state vectors, besides concatenation.

12.10 Transformer Models

Transformers [197] are well-suited to finding patterns in sequential data, including natural language and time series. While Recurrent Neural Networks utilize a hidden state vector that summarizes what happened in the past, a transformer can utilize temporal relationships/dependencies between any two elements in the time series. This enriched view of dependencies is referred to as the self-attention mechanism.

12.10.1 Self-Attention

Given a multi-variate time series Y consisting of m time steps and n_v variables,

$$Y = [y_{tj}] \tag{12.51}$$

the inputs $\mathbf{x}_t \in \mathbb{R}^{n_v}$ into the transformer can be defined as follows:

$$\mathbf{x}_t = [y_{t-1,0}, y_{t-1,1}, \dots y_{t-1,n_v-1}] \tag{12.52}$$

The goal now is to create a sequence of context vectors $\mathbf{c_t}$. Recall for encoder-decoder architectures, these were formed from state vectors $\mathbf{h_t}$ weighted by attentions scores/weights. The state vectors were based on the input sequence $\mathbf{x}_t$. As there is no recurrence to compute state vectors in Transformers, the following equation/linear transformation may be used instead,

$$\mathbf{v}_t \;=\; W_v \mathbf{x}_t \tag{12.53}$$

where $W_v \in \mathbb{R}^{n_u \times n_v}$ is the value matrix (its dimensions are number of units $\times$ size of input vector). Note that the transformation can have two effects: the new values are likely to be in an higher dimensional space (controlled by the user) and the values are rescaled (more suitable for neural computation).

Now attention weights $[\alpha_{t\tau}]$ are applied to produce a context vector at time t, $\mathbf{c}_t$,

$$\mathbf{c}_t \;=\; \sum_{\tau=0}^{m-1} \alpha_{t\tau} \mathbf{v}_\tau \;=\; \boldsymbol{\alpha}_t V \tag{12.54}$$

where $V = [\mathbf{v}_0, \mathbf{v}_1, \dots \mathbf{v}_{m-1}]^\intercal$ is the value vectors captured in a matrix. The question remains of how determine the attention scores/weights. We start by defining the following two learned views of the input sequence,

$$\begin{aligned} \mathbf{q}_t &= W_q \mathbf{x}_t \\ \mathbf{k}_t &= W_k \mathbf{x}_t \end{aligned}$$

where W_q and $W_k \in \mathbb{R}^{n_k \times n_v}$ are learned weight matrices. As with n_u, n_k is chosen by the user. (Note, in Natural Language Processing (NLP) the dimension of word vector $\mathbf{x}_t$ is typically very high, so the matrices are referred to projection matrices (project to a lower dimensional space), while for time series they would typically do the opposite.)

Let us consider how input $\mathbf{x}_t$ is related to the other inputs $\mathbf{x}_\tau$. In the transformed space $\mathbf{q}_t$ is its representative in making such an inquiry (query). One simple way to measure relatedness of two vectors is to take their dot product (proportional to the cosine of the angle between them).

$$\omega_{t\tau} \;=\; \mathbf{q}_t \cdot \mathbf{k}_\tau \tag{12.55}$$

These are referred to as the unnormalized attention scores. Notice that the other vector is represented by transformed vector $\mathbf{k}_t$. Using separate vectors $\mathbf{q}_t$ and $\mathbf{k}_t$ makes it possible for the attention scores to be asymmetric, meaning the influence (and therefore need for attention) between input $\mathbf{x}_t$ and $\mathbf{x}_\tau$ need not be the same in both direction. (Recall as measures of dependence correlation is symmetric, while conditional entropy is not).

The scores may be efficiently computed for all τ using matrix-vector multiplication,

$$\boldsymbol{\omega}_t \;=\; K\mathbf{q}_t \tag{12.56}$$

where $K = [\mathbf{k}_0, \mathbf{k}_1, \ldots \mathbf{k}_{m-1}]^\intercal$ is the key vectors captured in a matrix.

The following equation is used for computing (normalized) attention scores.

$$\boldsymbol{\alpha}_t \;=\; \text{softmax}\left[\frac{\boldsymbol{\omega}_t}{\sqrt{n_k}}\right] \tag{12.57}$$

The unnormalized attention scores are first divided by $\sqrt{n_k}$ and then passed through the softmax function to put the elements in the interval $(0, 1)$. Dividing by $\sqrt{n_k}$ (or $\sqrt{d_k}$ in other papers) may improve the stability of calculations. According to [197], "We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\sqrt{d_k}$."

Single-Head Self-Attention

Putting the formulas together yields a means for computing a context vector $\mathbf{c}_t$.

$$\mathbf{c}_t \;=\; V^\intercal \text{softmax}\left[\frac{K\mathbf{q}_t}{\sqrt{n_k}}\right] \tag{12.58}$$

This is implemented in ScalaTion as follows:

```
@param q_t  the query vector at time t (based on input vector x_t)
@param k    the key matrix K
@param v    the value matrix V

def context (q_t: VectorD, k: MatrixD, v: MatrixD): VectorD =
    val root_n = sqrt (q_t.dim)
    v.transpose * f_softmax.f_ (k * (q_t / root_n))
end context
```

Actually, all the context vectors (also referred to as an attention matrix) can be computed together using matrix multiplication.

$$C \;=\; \text{attention}(Q, K, V) \;=\; \text{softmax}\left[\frac{QK^\intercal}{\sqrt{n_k}}\right] V \tag{12.59}$$

The `attention` method computes attention weights.

```
@param q  the query matrix Q (q_t over all time)
@param k  the key matrix K
@param v  the value matrix V
```

```

    def attention (q: MatrixD, k: MatrixD, v: MatrixD): MatrixD =
        val root_n = sqrt (q.dim2)
        f_softmax.fM (q * (k.transpose / root_n)) * v
    end attention
```

The `context` and `attention` methods are provided by the `Attention` trait.

```
    @param n_var  the size of the input vector x_t (number of variables)
    @param n_mod  the size of the output (dimensionality of the model, d_model)
    @param heads  the number of attention heads
    @param n_v    the size of the value vectors

    trait Attention (n_var: Int, n_mod: Int = 512, heads: Int = 8, n_v: Int = -1):

    def queryKeyValue (x: MatrixD, w_q: MatrixD, w_k: MatrixD, w_v: MatrixD):
        (MatrixD, MatrixD, MatrixD) =
    def context (q_t: VectorD, k: MatrixD, v: MatrixD): VectorD =
    def attention (q: MatrixD, k: MatrixD, v: MatrixD): MatrixD =
    def attentionMH (q: MatrixD, k: MatrixD, v: MatrixD,
                     w_q: TensorD, w_k: TensorD, w_v: TensorD,
                     w_o: MatrixD): MatrixD =
```

Multi-Head Self-Attention

This self-attention mechanism is said to be bundled into an *attention head*. The transformer architecture allows for the use of multiple attention heads. For example, [197] suggests having 8 heads.

$$\text{attentionMH}(Q, K, V; \mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v, W^o) \;=\; \text{concat}_{i=0}^{\text{heads}-1} \left[\text{attention}(Q\mathbf{W}_i^q, K\mathbf{W}_i^k, V\mathbf{W}_i^v)\right] W^o \qquad (12.60)$$

The `attentionMH` method in the `Attention` trait computes multi-head attention weights.

```
    @param q    the query matrix Q (q_t over all time)
    @param k    the key matrix K
    @param v    the value matrix V
    @param w_q  the weight tensor for query Q (w_q(i) matrix for i-th head)
    @param w_v  the weight tensor for key K (w_k(i) matrix for i-th head)
    @param w_v  the weight tensor for value V (w_v(i) matrix for i-th head)
    @param w_o  the overall weight matrix to be applied to concatenated attention

    def attentionMH (q: MatrixD, k: MatrixD, v: MatrixD,
                     w_q: TensorD, w_k: TensorD, w_v: TensorD,
                     w_o: MatrixD): MatrixD =
        var aw = attention (q * w_q(0), k * w_k(0), v * w_v(0))
        for i <- 1 until heads do aw = aw ++^ attention (q * w_q(i), k * w_k(i), v * w_v(i))
        att * w_o
    end attentionMH
```

12.10.2 Positional Encoding

Rather than having a Recurrent Neural Network where information flows over time and the input is processed in time order, an alternative would be to turn each value y_t in the time series into a pair (t, y_t). A natural way to do this in a neural environment is through positional encoding.

Sinusoidal positional encoding is often used ... Given a time value t, a positional vector of length d is created by alternately calling sin and cos functions,

$$\mathbf{p}_t = [\sin(\omega_1 \mathrm{t}), \cos(\omega_1 \mathrm{t}), \sin(\omega_2 \mathrm{t}), \cos(\omega_2 \mathrm{t}), \ldots, \sin(\omega_{\mathrm{d}/2} \mathrm{t}), \cos(\omega_{\mathrm{d}/2} \mathrm{t})] \tag{12.61}$$

where the wavelengths $2\pi\omega_k$ decrease exponentially.

Absolute Positional Encoding

Relative Positional Encoding

Although, the commonly used sinusoidal positional encoding could be used, some studies have shown that other approaches may be better for time series forecasting [125].

12.10.3 Encoder-Decoder Architecture for Transformers

A common architecture for a Transformer consists of an encoder side and an decoder side with each side typically having multiple (around 6 [197]) encoder and decoder layers, allowing the input to be refined in phases.

Encoder

An encoder layer consists of (1) a multi-head self-attention module (shown in orange), followed by (2) a feed-forward neural network that leads this layer's output (shown in lime). This output is feed into the next encoder layer as input. The input to the first encoder layer is $[\mathbf{x}_t]$ (typically adjusted based upon positional encoding). In addition, after completion of each module, upstream information is added in (skip connection) followed by layer normalization (see exercises). Addition and normalization are shown in yellow as depicted in Figure 12.9.

Each of these steps is described below:

1. Attention: For a single head, attention(Q, K, V) computes ... For multi-head attention, ...
2. Add-In: $[\mathbf{x}_t]$ NEED DETAILS
3. Layer Normalization: Suppose a layer in a network outputs a vector $\mathbf{z}$ whose size is the number of units in the layer. This vector is then normalized by subtracting the mean and dividing by the standard deviation. In SCALATION, this is provided by the `standardize` method in `VectorD` or the more robust `standardize2` method (takes a Normal random variable to a Standard Normal random variable). See exercise about pre-layer and post-layer normalization.
4. Feed-Forward Neural Network: A basic configuration is to have a Linear Layer (no activation function), followed by a `reLU/geLU` layer (see the Percepton section), followed by another linear layer, and finally a dropout layer.
5. Add-In: $[\mathbf{z}_t]$ NEED DETAILS
6. Layer Normalization: The addition in the last step may throw off normalization, so it needs to be done again.

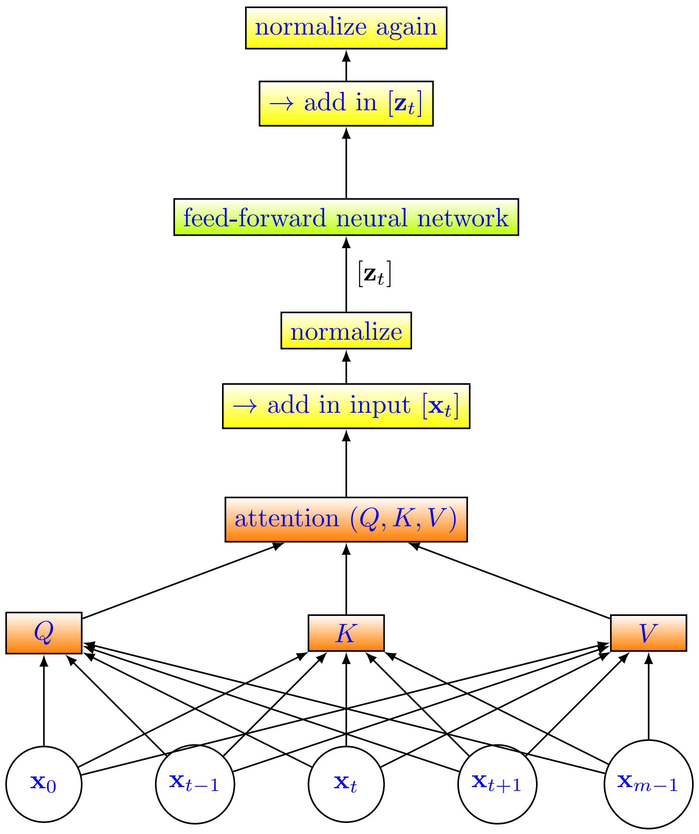


Figure 12.9: Transformer First Encoder Layer for a Single Head

Decoder

A decoder layer includes the modules from a encoder with the following changes and additions. The self-attention is modified by using using masking to prevent the decoder (i.e., the forecaster) from seeing the future. In addition, the decoder takes the output from its corresponding encoder, and processes it with another multi-head self-attention module. In other words, it contains three modules: (1) multi-head self-attention module applied to input, (2) multi-head self-attention module applied to encoder output, and (3) a feed-forward neural network to produce its output. The final decoder layer produces the forecasts.

NEED DETAILED FIGURE (using tikz)

12.10.4 Exercises

1. The Scaled Dot Product of vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\mathbf{x} \text{ sdot } \mathbf{y} \;=\; \frac{\mathbf{x} \cdot \mathbf{y}}{\sqrt{n}} \tag{12.62}$$

   where n is the dimensionality (number of elements) of each vector. Explain why dividing by the square root of dimensionality improves the stability of gradient calculations.

2. What are advantages of pre-layer over post-layer normalization?

12.10.5 Further Reading

- "Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch," Sebastian Raschka, `https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html`

Chapter 13

Dimensionality Reduction

When data matrices are very large with high dimensionality, analytics becomes difficult. In addition, there is likely to be co-linearity between vectors, making the computation of inverses or pseudo-inverses problematic. In such cases, it is useful to reduce the dimensionality of the data.

13.1 Reducer

The `Reducer` trait provides a common framework for several data reduction algorithms.

Trait Methods:

```
    trait Reducer

    def reduce (): MatrixD
    def recover (): MatrixD
```

13.2 Principal Component Analysis (PCA)

The `PrincipalComponents` class computes the Principal Components (PCs) for data matrix X with the following dimensions

$$X \in \mathbb{R}^{m \times n} \tag{13.1}$$

where the number of rows m is the number of instances/samples and the number of columns n is the number of predictor variables.

Principal Component Analysis (PCA) can be used to reduce the dimensionality of the data matrix. Using the `PrincipalComponents` class, first find the PCs by calling 'findPCs' and then call 'reduce' to reduce the data (i.e., reduce matrix X to a lower dimensionality matrix).

13.2.1 Representation

PCA will replace the data matrix $X \in \mathbb{R}^{m \times n}$ with a lower-dimensional reduced matrix $Z \in \mathbb{R}^{m \times k}$ for $k \le n$.

$$Z \;=\; XE_k \tag{13.2}$$

The matrix E is the full eigen-decomposition of the covariance matrix of X [1].

$$\mathbb{C}\left[X\right] = \frac{X_c^\intercal X_c}{m-1} \tag{13.3}$$

where $X_c = X - \boldsymbol{\mu}_X$ is the centered data matrix.

$$E = \text{eigendecomp}\left[\frac{X_c^\intercal X_c}{m-1}\right] \tag{13.4}$$

The matrix E_k is the first k columns of E where the columns of E are ordered by the magnitude of the eigenvalues. The columns with the largest eigenvalues will explain most of the covariance. See the exercises to understand why the covariance matrix is used as the basis for the dimensionality reduction.

For a given vector $\mathbf{x}$ of dimension n, its reduced representation $\mathbf{z}$ will have dimension k.

$$\mathbf{z} \;=\; E_k\mathbf{x} \tag{13.5}$$

Note, that each element of vector $\mathbf{z}$ is a linear combination of elements in the original vector $\mathbf{x}$.

Example Problem:

Class Methods:

```
@param x  the data matrix to reduce, stored column-wise

class PrincipalComponents (x: MatrixD)

def meanCenter (): VectorD =
def computeCov (): MatrixD =
def computeEigenVectors (eVal: VectorD): MatrixD =
def findPCs (k: Int): MatrixD =
def reduceData (): MatrixD =
def recover (): MatrixD = reducedMat * featureMat.t + mu
def solve (i: Int): (VectorD, VectorD) =
```

13.2.2 Exercises

1.

13.3 Autoencoder (AE)

An Autoencoder (AE) is a fully-connected neural network that contains a middle layer of lower dimensionality than the input layer. The output layer has the same dimensionality as the input layer, as shown in figure 13.1. The loss function then simply measures the difference between the input vectors and output vectors. If the loss is small, then the middle layer may be used as a lower-dimensional representation of the input.

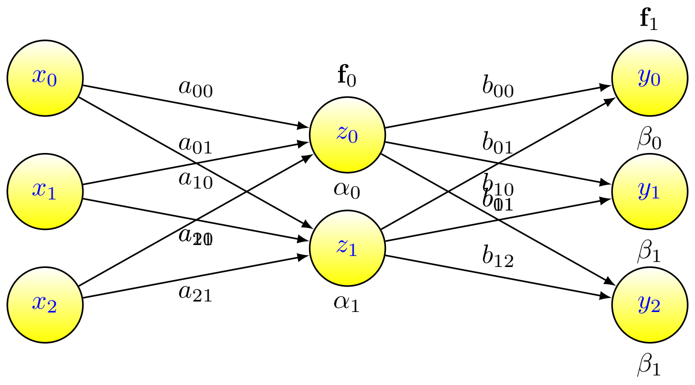


Figure 13.1: Three-Layer Autoencoder Neural Network

A common choice for the loss function is the sum of squared reconstruction errors,

$$sse = \|X - \hat{Y}\|_F \tag{13.6}$$

where X is the input data matrix and $\hat{Y}$ is the reconstruction of X.

13.3.1 Representation

For a three layer autoencoder, the lower dimensional representation vector $\mathbf{z}$ corresponds to the values at the middle hidden layer and is given by the following equation,

$$\mathbf{z} \;=\; \mathbf{f}_0(A^{\intercal}\mathbf{x} + \boldsymbol{\beta}) \tag{13.7}$$

where $\mathbf{x}$ is the input vector, A is the parameter/weight matrix, $\boldsymbol{\beta}$ is the bias vector and $\mathbf{f}_0$ is the vectorized activation function.

Notice that if $\mathbf{f}_0$ is the identity function, then $\mathbf{z}$ becomes a linear transformation of $\mathbf{x}$, as is the case for PCA.

13.3.2 Denoising Autoencoder (DEA)

Chapter 14

Clustering

Clustering is related to classification, except that specific classes are not prescribed. Instead data points (vectors) are placed into clusters based on some similarity or distance metric (e.g., Euclidean or Manhattan distance). It is also related to prediction in the sense that a predictive model may be associated with each cluster. Points in a cluster, are according to some metric, closer to each other than to points not in their cluster. Closeness or similarity may be defined in terms of ℓ^p distance $\|\mathbf{x}-\mathbf{z}\|_p$, correlation $\rho(\mathbf{x}, \mathbf{z})$, or cosine $\cos(\mathbf{x}, \mathbf{z})$. Abstractly, we may represents any of these by distance $d(\mathbf{x}, \mathbf{z})$. In ScalaTion, the function `dist` in the `clustering` package computes the square of Euclidean distance between two vectors, but may easily be changed (e.g., `(x - z).norm1` for Manhattan distance).

```
def dist (x: VectorD, z: VectorD): Double = (x - z).normSq
```

Consider a general modeling equation, where the parameters $\mathbf{b}$ are estimated based on a dataset $(X, \mathbf{y})$.

$$y \ = \ f(\mathbf{x}; \mathbf{b}) + \epsilon$$

Rather than trying to approximate the function f over the whole data domain, one might think that given point $\mathbf{z}$, that points similar to (or close to) $\mathbf{z}$, might be more useful in making a prediction $f(\mathbf{z})$.

A simple way to do this would be to find the κ-nearest neighbors to point $\mathbf{z}$,

$$top_\kappa(\mathbf{z}) \ = \ \{\mathbf{x}_i \in X \,|\, \mathbf{x}_i \text{ is among the } \kappa \text{ closest points to } \mathbf{z}\}$$

and simply average the responses or y-values.

Instead of surveying the responses from the κ-nearest neighbors, one could instead survey an entire group of similar points and take their averaged response (or utilize a linear model where each cluster c has its own parameters $\mathbf{b}_c$). The groups may be pre-computed and the averages/parameters can be maintained for each group. The groups are made by clustering the points in the X matrix into say k groups.

Clustering will partition the m-points $\{\mathbf{x}_i \in X\}$ into k groups/clusters. Group membership is based on closeness or similarity between the points. Commonly, algorithms form groups by establishing a centroid (or center) for each group/cluster. Centroids may defined as means (or medians) of the points in the group. In this way the data matrix X is partitioned into k sub-matrices

$$\{X_c \,|\, c \in \{0, \ldots, k-1\}\}$$

each with centroid $\boldsymbol{\xi}_c = \mu(X_c)$. Typically, point $\mathbf{x}_i$ is in cluster c because it is closer to its centroid than any other centroid, i.e.,

$$\mathbf{x}_i \in X_c \implies d(\mathbf{x}_i, \boldsymbol{\xi}_c) \leq d(\mathbf{x}_i, \boldsymbol{\xi}_h)$$

Define the *cluster assignment function* ξ to take a point $\mathbf{x}_i$ and assign it to the cluster with the closest centroid $\boldsymbol{\xi}_c$, i.e.,

$$\xi(\mathbf{x}_i) \ = \ c$$

The goal becomes to find an optimal cluster assignment function by minimizing the following objective/cost function:

$$\min_\xi \ \sum_{i=0}^{m-1} d(\mathbf{x}_i, \boldsymbol{\xi}_{\xi(\mathbf{x}_i)})$$

If the distance d is $\|\mathbf{x}_i - \boldsymbol{\xi}_{\xi(\mathbf{x}_i)}\|_2^2$ (the default in ScalaTion), then above sum may be viewed as a form of sum of squared errors (sse).

If one knew the optimal centroids ahead of time, finding an optimal cluster assignment function ξ would be trivial and would take $O(kmn)$ time. Unfortunately, k centroids must be initially chosen, but then as assignments are made, the centroids will move, causing assignments to need re-evaluation. The details vary by clustering algorithm, but it is useful to know that finding an optimal cluster assignment function is $\mathcal{NP}$-hard [7].

Other factors that can be considering in forming clusters include, balancing the size of clusters and maximizing the distance between clusters.

14.1 KNN_Regression

Similar to the `KNN_Classifier` class, the `KNN_Regression` class makes predictions based on individual predictions of its κ-nearest neighbors. For prediction, its function is analogous to using clustering for prediction and will be compared in the exercises in later sections of this chapter.

Training in `KNN_Regression` is lazy and is done in the `predict` method, based on the following equation:

$$\hat{y} \;=\; \frac{1}{\kappa}\,\mathbf{1} \cdot \mathbf{y}(top_\kappa(\mathbf{z})) \tag{14.1}$$

Given point $\mathbf{z}$, find κ points that are the closest, sum there response values y, and return the average.

```
override def predict (z: VectorD): Double =
    kNearest (z)                                // set top-kappa to kappa nearest
    var sum = 0.0
    for i <- 0 until kappa do sum = y(topK(i)._1)  // sum the individual predictions
    sum / kappa                                 // divide to get average
end predict
```

The `kNearest` method is same as the one in `KNN_Classifier`.

14.1.1 `KNN_Regression` Class

Class Methods:

```
@param x       the vectors/points of predictor data stored as rows of a matrix
@param y       the response value for each vector in x
@param fname_  the names for all features/variables (defaults to null)
@param hparam  the number of nearest neighbors to consider

class KNN_Regression (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                      hparam: HyperParameter = KNN_Regression.hp)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def distance (x: VectorD, z: VectorD): Double = (x - z).normSq
def train (x_ : MatrixD = x, y_ : VectorD = y): Unit = {}
def test (x_ : MatrixD = x, y_ : VectorD = y): (VectorD, VectorD) =
def testNoSpy (xe: MatrixD = x, ye: VectorD = y, i_no: IndexedSeq [Int]): (VectorD, VectorD) =
override def predict (z: VectorD): Double =
def predictNoSpy (z: VectorD, i_no: IndexedSeq [Int]): Double =
override def validate (rando: Boolean = true, ratio: Double = 0.2)
             (idx : IndexedSeq [Int] =
              testIndices ((ratio * y.dim).toInt, rando)): VectorD =
override def buildModel (x_cols: MatrixD): KNN_Regression =
```

Note that the `train` method has nothing to do, so it need not be called.

14.1.2 Exercises

1. Apply KNN_Regression to the following combined data matrix.

```
//                         x1 x2  y
val xy = MatrixD ((10, 3), 1, 5, 1,       // joint data matrix
                            2, 4, 1,
                            3, 4, 1,
                            4, 4, 1,
                            5, 3, 0,
                            6, 3, 1,
                            7, 2, 0,
                            8, 2, 0,
                            9, 1, 0,
                           10, 1, 0)

val mod = KNN_Regression (xy)()
mod.trainNtest ()()
val yp = mod.predict (xy.not (?, 2))
new Plot (xy(?, 0), y, yp, lines = true)
```

14.2 Clusterer

The `Clusterer` trait provides a common framework for several clustering algorithms.

`Clusterer` **Trait**

Trait Methods:

```
trait Clusterer:

def name_ (nm: Array [String]): Unit = _name = nm
def name (c: Int): String =
def setStream (s: Int): Unit = stream = s
def train (): Unit
def cluster: Array [Int]
def csize: VectorI
def centroids: MatrixD
def initCentroids (): Boolean = false
def calcCentroids (x: MatrixD, to_c: Array [Int], sz: VectorI, cent: MatrixD): Unit =
def classify (z: VectorD): Int
def distance (u: VectorD, cn: MatrixD, kc_ : Int = -1): VectorD =
def sse (x: MatrixD, to_c: Array [Int]): Double =
def sse (x: MatrixD, c: Int, to_c: Array [Int]): Double =
def sst (x: MatrixD): Double =
def checkOpt (x: MatrixD, to_c: Array [Int], opt: Double): Boolean = sse (x, to_c) <= opt
```

For readability, names may be given to clusters (see `name` and `name_`). To obtain a new (and likely different) cluster assignment, `setStream` method may be called to change the random number stream. The `train` methods in the implementing classes will take a set of points (vectors) and apply iterative algorithms to find a "good" cluster assignment function. The `cluster` method may be called after `train` to see the cluster assignments. The centroids are returned as rows in a matrix by calling `centroids`, whose cluster sizes are given by `csize`. The `initCentroid` method initializes the centroids, while the `calcCentroids` calculates the centroids based in the points contained in the each cluster.

```
def calcCentroids (x: MatrixD, to_c: Array [Int], sz: VectorI, cent: MatrixD): Unit =
    cent.setAll (0.0)                                    // set cent matrix to all zeros
    for i <- x.indices do
        val c   = to_c(i)                                // x_i currently assigned to cluster c
        cent(c) = cent(c) + x(i)                         // add the next vector in cluster
    end for
    for c <- cent.indices do cent(c) = cent(c) / sz(c)   // divide to get averages/means
end calcCentroids
```

Given a new point/vector `z`, the `classify` method will indicate which cluster it belongs to (in the range 0 to k-1). The distances between a point and the centroids is computed by the `distance` method. The objective/cost function is defined to be the sum of squared errors (`sse`). If the cost of an optimal solution is known, `checkOpt` will return true if the cluster assignment is optimal.

14.3 K-Means Clustering

The `KMeansClustering` class clusters several vectors/points using k-means clustering. The user selects the number of clusters desired (k). The algorithm will partition the points in X into k clusters. Each cluster has a centroid (mean) and each data point $\mathbf{x}_i \in X$ is placed in the cluster whose centroid it is nearest to.

14.3.1 Initial Assignment

There are two ways to initialize the algorithm: Either (1) randomly assign points to k clusters or (2) randomly pick k points as initial centroids. Technique (1) tends to work better and is the primary technique used in ScalaTion. Using the primary technique, the first step is to randomly assign each point $\mathbf{x}_i$ to a cluster.

$$\xi(\mathbf{x}_i) \;=\; \text{random integer from } \{0, \ldots, k-1\}$$

In ScalaTion, this is carried out by the `assign` method, that uses the `Randi` random integer generator. It also uses multiple counters for determining the size `sz` of each cluster (i.e., the number of points in each cluster).

```
protected def assign (): Unit =
    val ran = new Randi (0, k-1, s)          // for random integers: 0, ..., k-1
    for i <- x.indices do
        to_c(i) = ran.igen                   // randomly assign x(i) to a cluster
        sz(to_c(i)) += 1                     // increment size of that cluster
    end for
end assign
```

See the exercises for more details on the second technique for initializing clusters/centroids.

Handling Empty Clusters

If any cluster turns out to be empty, move a point from another cluster. In ScalaTion this is done by removing a point from the largest cluster and adding it to the empty cluster. This is performed by the `fixEmptyClusters` method.

```
protected def fixEmptyClusters (): Unit =
```

After the `assign` and `fixEmptyClusters` methods have been called, the data matrix X will be logically partitioned into k non-empty sub-matrices X_c with cluster c having n_c (`sz(c)`) points/rows.

Calculating Centroids

The next step is to calculate the centroids using the `calcCentroids` method. For cluster c, the centroid is the vector mean of the rows in submatrix X_c.

$$\boldsymbol{\xi}_c \;=\; \frac{1}{n_c} \sum_{\mathbf{x}_i \in X_c} \mathbf{x}_i$$

ScalaTion iterates over all points and based on their cluster assignment adds them to one of the k centroids (stored in the `cent` matrix). After the loop, these sums are divided by the cluster sizes `sz` to get means. The `calcCentroids` method is defined in the base trait `Clusterer`.

14.3.2 Reassignment of Points to Closest Clusters

After initialization, the algorithm iteratively reassigns each point to the cluster containing the closest centroid. The algorithm stops when there are no changes to the cluster assignments. For each iteration, each point $\mathbf{x}_i$ needs to be re-evaluated and moved (if need be) to the cluster with the closest centroid. Reassignment is based on taking the argmin of all the distances to the centroids with ties going to the current cluster.

$$\xi(\mathbf{x}_i) \;=\; \text{argmin}_c\, d(\mathbf{x}_i, \boldsymbol{\xi}_c)$$

In ScalaTion, this is done by the `reassign` method which iterates over each $\mathbf{x}_i \in X$ computing the distance to each of k centroids. The cluster (`c2`) with the closest centroid is found using the `argmin` method. The distance to `c2`'s centroid is then compared to the distance to its current cluster `c1`'s centroid, and if the distance to `c2`'s centroid is less, $\mathbf{x}_i$ will be moved and a `done` flag will be set to `false`, indicating that during this reassignment phase at least one change was made.

```
protected def reassign (): Boolean =
    var done = true                              // done indicates no changes
    for i <- x.indices do                        // standard order for index i
        val c1 = to_c(i)                         // c1 = current cluster for point x_i
        if sz(c1) > 1 then                       // if size of c1 > 1
            val d  = distance (x(i), cent)       // distances to all centroid
            val c2 = d.argmin ()                 // c2 = cluster with closest centroid to x_i
            if d(c2) < d(c1) then                // if closest closer than current
                sz(c1) -= 1                      // decrement size of current cluster
                sz(c2) += 1                      // increment size of new cluster
                to_c(i) = c2                     // reassign point x_i to cluster c2
                done    = false                  // changed clusters => not done
                if immediate then return false   // optionally return after first change
            end if
        end if
    end for
    done
end reassign
```

The exercises explore a change to this algorithm by having it return after the first change.

14.3.3 Training

The `train` method simply uses these methods until the `reassign` method returns true (internally the `done` flag is true). The method is set up to work for this and derived classes. It assigns points to clusters and then either initializes/picks centroids or calculates centroids from the first cluster assignment. Inside the loop, `reassign` and `calcCentroid` are called until there is no change to the cluster assignment. After the loop, an exception is thrown if there are any empty clusters (a useful safe-guard since this method is used by derived classes). Finally, if post-processing is to be performed (`post = true`), then the `swap` method is called. This method will swap two points in different clusters, if the swap results in a lower sum of squared error (*sse*).

```
def train (): Unit =
    sz.set (0)                                                 // cluster sizes initialized to zero
    raniv = PermutedVecI (VectorI.range (0, x.dim), stream)    // for randomizing index order
    assign ()                                                  // randomly assign points to clusters
    fixEmptyClusters ()                                        // move points into empty clusters
    if ! initCentroids () then calcCentroids (x, to_c, sz, cent)   // pick points for initial centroids
    breakable {
        for l <- 1 to MAX_IT do
            if reassign () then break ()                       // reassign points (no change => break)
            calcCentroids (x, to_c, sz, cent)                  // re-calculate the centroids
        end for
    } // breakable
    val ce = sz.indexOf (0)                                    // check for empty clusters
    if ce != -1 then throw new Exception (s"Empty cluster c = $ce")
    if post then swap ()                                       // swap points to improve sse
end train
```

14.3.4 KMeansClusterer Class

The `KMeansClusterer` class and its derived classes take a data matrix, the desired number clusters and an array of flags as input parameters. The array of flags are used to make adjustments to the algorithms. For this class, there are two: `flags(0)` or `post` indicates whether to use post-processing, and `flags(1)` or `immediate` indicates whether return upon the first change in the `reassign` method.

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the array of flags used to adjust the algorithm
                  default: no post processing, no immediate return upon change

class KMeansClusterer (x: MatrixD, k: Int, val flags: Array [Boolean] = Array (false, false))
      extends Clusterer:

def train (): Unit =
def cluster: Array [Int] = to_c
def centroids: MatrixD = cent
def csize: VectorI = sz
protected def assign (): Unit =
protected def fixEmptyClusters (): Unit =
protected def reassign (): Boolean =
protected def swap (): Unit =
def classify (z: VectorD): Int = distance (z, cent).argmin ()
def show (l: Int): Unit = println (s"($l) to_c = ${to_c.deep} \n($l) cent = $cent")
```

14.3.5 Exercises

1. Plot the following points.

```
//                          x0   x1
val x = MatrixD ((6, 2), 1.0, 2.0,
                         2.0, 1.0,
                         4.0, 5.0,
                         5.0, 4.0,
                         8.0, 9.0,
                         9.0, 8.0)

new Plot (x(?, 0), x(?, 1), null, "x0 vs. x1")
```

For `k = 3`, determine the optimal cluster assignment ξ. What is the sum of squared errors *sse* for this assignment?

2. Using the data from the previous exercise, apply the K-Means Clustering Algorithm by hand to complete the following cluster assignment function table. Let the number of clusters k be 3 (clusters 0, 1 and 2). The ξ^0 column is the initial random cluster assignment, while the next two columns represent the cluster assignments for the next two iterations.

Table 14.1: Cluster Assignment Function Table

point	(x_0, x_1)	ξ^0	ξ^1	ξ^2
0	(1, 2)	0	?	?
1	(2, 1)	2	?	?
2	(4, 5)	0	?	?
3	(5, 4)	1	?	?
4	(8, 9)	1	?	?
5	(9, 8)	2	?	?

3. The `test` function in the `Clusterer` object is used test various configurations of classes extending `Clusterer`, such as the `KMeansClusterer` class.

```
@param x    the data matrix holding the points/vectors
@param fls  the array of flags
@param alg  the clustering algorithm to test
@param opt  the known optimum for see (ignore if not known)

def test (x: MatrixD, fls: Array [Boolean], alg: Clusterer, opt: Double = -1.0): Unit =
```

Explain the meaning of each of the flags: `post` and `immediate`. Call the `test` function, passing in `x` and `k` from the last exercise. Also, let the value `opt` be the value determined in the last exercise. The `test` method will give the number of test cases out of `NTESTS` that are correct in terms of achieving the minimum *sse*.

4. The `primary` versus `secondary` techniques for initializing the clusters/centroids are provided by the `KMeansClusterer` class and the `KMeansClusterer2` class, respectively. Test the quality of these two techniques.

5. Show that the time complexity of the `reassign` method is $O(kmn)$. The time complexity of K-Means Clustering using Lloyd's Algorithm [113] is simply the complexity of the `reassign` method times the number of iterations. In practice, the number of iterations tends to be small, but in the worst case only upper and lower bounds are known, see [9] for details.

6. Consider the objective/cost function given in ISL equation 10.11 in [85]. What does it measure and how does it compare to sse used in this book?

14.4 K-Means Clustering - Hartigan-Wong

An alternative to the Lloyd algorithm that often produces more tightly packed clusters is the Hartigan-Wong algorithm [69]. Improvement is seen in the fraction of times that optimal clusters are formed as well as the reduction in sum of squared errors (sse). The change to the code is minimal in that only the `reassign` method needs to be overridden.

The basic difference is that rather than simply reassigning each point to the cluster with the closest centroid (the Lloyd algorithm), the Hartigan-Wong algorithm weights the distance by the relative changes in the number of points in a cluster. For example, if a point is to be moved into a cluster with 10 points currently, the weight would be 10/11. If the point is to stay in its present cluster with 10 points currently, the loss in removing it would be weighted as 10/9. The weighting scheme has two effects: First it makes it more likely to move a point out of its current cluster. Second it makes it more likely to join a small cluster.

Mathematically, the weighted distance d' to cluster c when the point $\mathbf{x}_i \notin X_c$ is given by

$$d'(\mathbf{x}_i, \boldsymbol{\xi}_c) = \frac{n_c}{n_c + 1} d(\mathbf{x}_i, \boldsymbol{\xi}_c) \tag{14.2}$$

When the point $\mathbf{x}_i \in X_c$, the weighted distance d' to cluster c is given by

$$d'(\mathbf{x}_i, \boldsymbol{\xi}_c) = \frac{n_c}{n_c - 1} d(\mathbf{x}_i, \boldsymbol{\xi}_c) \tag{14.3}$$

The code for the `reassign` method is similar to the one in `KMeansClusterer`, except that the private method `closestByR2` calculates weighted distances to return the closest centroid.

```
protected override def reassign (): Boolean =
    var done = true                                // done indicates no changes
    for i <- raniv.igen do                         // randomize order of index i
        val c1 = to_c(i)                           // c1 = current cluster for point x_i
        if sz(c1) > 1 then                         // if size of c1 > 1
            val d  = distance2 (x(i), cent, c1)    // adjusted distances to all centroid
            val c2 = d.argmin ()                   // c2 = cluster with closest centroid to x_i
            if d(c2) < d(c1) then                  // if closest closer than current
                sz(c1) -= 1                        // decrement the size of cluster c1
                sz(c2) += 1                        // increment size of cluster c2
                to_c(i) = c2                       // reassign point x_i to cluster c2
                done = false                       // changed clusters => not done
                if immediate then return false     // optionally return after first change
            end if
        end if
    end for
    done
end reassign
```

Besides switching from distance d to weighted distance d', the code also randomizes the index order and has the option of returning immediately after a change is made.

14.4.1 Adjusted Distance

The `distance2` method computes the adjusted distance of point `u` to all of the centroids `cent`, where `cc` is the current centroid that `u` is assigned to. Notice the inflation of distance when `c == cc`, and its deflation, otherwise.

```
def distance2 (u: VectorD, cent: MatrixD, cc: Int): VectorD =
    val d = new VectorD (cent.dim)
    for c <- 0 until k do
        d(c) = if c == cc then (sz(c) * dist (u, cent(c))) / (sz(c) - 1)
               else (sz(c) * dist (u, cent(c))) / (sz(c) + 1)
    end for
    d
end distance2
```

14.4.2 `KMeansClusteringHW` Class

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the flags used to adjust the algorithm

class KMeansClustererHW (x: MatrixD, k: Int, flags: Array [Boolean] = Array (false, false))
      extends KMeansClusterer (x, k, flags)

protected override def reassign (): Boolean =
def distance2 (u: VectorD, cent: MatrixD, cc: Int): VectorD =
```

14.4.3 Exercises

1. Compare `KMeansClustererHW` with `KMeansClusterer` for a variety of datasets, starting with the six points given in the last section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the *sse* over the `NTESTS` test cases.

14.5 K-Means++ Clustering

The KMeansClustererPP class clusters several vectors/points using a k-means++ clustering algorithm [10]. The class may be derived from a K-Means clustering algorithm and in ScalaTion it is derived from the Hartigan-Wong algorithm (KMeansClustererHW). The innovation for KMeansClustererPP is to pick the initial centroids wisely, yet randomly. The wise part is to make sure points are well separated. The random part involves making a probability mass function (pmf) where points farther away from the current centroids are more likely to be selected as the next centroid. Picking the initial centroids entirely randomly leads to KMeansClusterer2 which typically does not perform as well KMeansClusterer. However, maintaining randomness while giving preference to more distant points becoming the next centroid has been shown to work well.

14.5.1 Picking Initial Centroids

In order to pick k initial centroids, the first one, cent(0), is chosen entirely randomly, using the ranI random variate generator object. The method call ranI.igen will pick one of the m = x.dim points as the first centroid.

```
val ranI = new Randi (0, x.dim-1, stream)        // uniform random integer generator
cent(0)  = x(ranI.igen)                          // pick first centroid uniformly at random
```

The rest of the centroids are chosen following a distance-derived discrete distribution, using the ranD random variate generator object. The probability mass function (pmf) for this discrete distribution is produced so that the probability of a point being selected as the next centroid is proportional to its distance to the closest existing centroid.

```
for c <- 1 until k do                            // pick remaining centroids
    val ranD = update_pmf (c)                    // update distance derived pmf
    cent(c)  = x(ranD.igen)                      // pick next centroid according to pmf
end for
```

Each time a new centroid is chosen, the pmf must be updated as it is likely to be the closest centroid for some of the remaining as yet unchosen points. Given that the next centroid to selected is the c^{th} centroid, the update_pmf method will update the pmf and return a new distance-derived discrete distribution.

```
def update_pmf (c: Int): Discrete =
    for i <- x.indicea do pmf(i) = distance (x(i), cent, c).min    // shortest distances
    pmf /= pmf.sum                                                 // divide by sum
    Discrete (pmf, stream = (stream + c) % NSTREAMS)               // distance-derived generator
end update_pmf
```

The pmf vector initially records the shortest distance from each point $\mathbf{x}_i$ to any of the existing already selected centroids $\{0, \ldots, c-1\}$. These distances are turned into probabilities by dividing by their sum. The pmf vector then defines a new distance-derived random generator that is returned.

14.5.2 `KMeansClustererPP` Class

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the flags used to adjust the algorithm

class KMeansClustererPP (x: MatrixD, k: Int, flags: Array [Boolean] = Array (false, false))
      extends KMeansClustererHW (x, k, flags)

override def initCentroids (): Boolean =
def update_pmf (c: Int): Discrete =
```

14.5.3 Exercises

1. Compare `KMeansClustererPP` with `KMeansClustererHW` and `KMeansClusterer` for a variety of datasets, starting with the six points given in the `KMeansClusterer` section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the *sse* over the `NTESTS` test cases.

14.6 Clustering Predictor

The `ClusteringPredictor` class is used to predict a response value for new vector **z**. It works by finding the cluster that the point **z** would belong to. The recorded response value for y is then given as the predicted response. The per cluster recorded response value is the consensus (e.g., average) of the response values y_i for each member of the cluster. Training involves clustering the points in data matrix X and then computing each cluster's response. Assuming the closest centroid to **z** is $\boldsymbol{\xi}_c$, the predicted value $\hat{y}$ is

$$\hat{y} \;=\; \frac{1}{n_c} \sum_{\xi(\mathbf{x}_i)=c} y_i \tag{14.4}$$

where n_c is the number points in cluster c and $\xi(\mathbf{x}_i) = c$ means that the i^{th} point is assigned to cluster c.

14.6.1 Training

The `train` method first clusters the points/rows in data matrix X by calling the `train` method of a clustering algorithms (e.g., `clust = KMeansClusterer (...)`). It then calls the `assignResponse` method to assign a consensus (average) response value for each cluster.

```
def train (xx: MatrixD = x, yy: VectorD = y): Unit =
    clust.train ()
    val clustr = clust.cluster
    assignResponse (clustr)
end train
```

The computed consensus values are stored in `yclus`, so that the `predict` method may simply use the underlying clustering algorithm to classify a point **z** to indicate which cluster it belongs to. This is then used to index into the `yclus` vector.

```
override def predict (z: VectorD): Double = yclus (clust.classify (z))
```

14.6.2 `ClusteringPredictor` Class

Class Methods:

```
@param x       the vectors/points of predictor data stored as rows of a matrix
@param y       the response value for each vector in x
@param fname_  the names for all features/variables (defaults to null)
@param hparam  the number of nearest neighbors to consider

class ClusteringPredictor (x: MatrixD, y: VectorD, fname_ : Array [String] = null,
                           hparam: HyperParameter = ClusteringPredictor.hp)
      extends Predictor (x, y, fname_, hparam)
         with Fit (dfm = x.dim2 - 1, df = x.dim - x.dim2):

def train (xx: MatrixD = x, yy: VectorD = y): Unit =
private def assignResponse (clustr: Array [Int]): Unit =
override def test (xx: MatrixD = x, yy: VectorD = y): (VectorD, VectorD) =
def classify (z: VectorD): Int = clust.classify (z)
override def predict (z: VectorD): Double = yclus (clust.classify (z))
def reset (): Unit =
override def buildModel (x_cols: MatrixD): Predictor =
```

14.6.3 Exercises

1. Apply `ClusteringPredictor` to the following combined data matrix.

```
//                        x0 x1  y
val xy = MatrixD ((10, 3), 1, 5, 1,       // joint data matrix
                           2, 4, 1,
                           3, 4, 1,
                           4, 4, 1,
                           5, 3, 0,
                           6, 3, 1,
                           7, 2, 0,
                           8, 2, 0,
                           9, 1, 0,
                          10, 1, 0)

val cp = ClusteringPredictor (xy)()
cp.trainNtest ()()
val (x, y) = (xy.not(?, 2), xy(?, 2))
val yp = cp.predict (x)
new Plot (xy(?, 0), y, yp, lines = true)
```

   Compare its results to that of `KNN_Regression`.

2. Compare `Regression`, `KNN_Regression` and `ClusteringPredictor` on the `AutoMPG` dataset.

14.7 Hierarchical Clustering

One critique of K-Means Clustering is that the user chooses the desired number of clusters (k) beforehand. With modern computing power, several values for k may be tried, so this is less of an issue now. There is, however, a clustering technique called Hierarchical Clustering [87] where this a non-issue.

In ScalaTion the `HierClusterer` class starts with each point in the data matrix X forming its own cluster (m clusters). For each iteration, the algorithm will merge two clusters into a one larger cluster, thereby reducing the number of clusters by one. The two clusters that are closest to each other are chosen as the clusters to merge. The `train` method is shown below.

```
def train (): Unit =
    sz.set (0)                                      // initialize cluster sizes to zero
    initClusters ()                                 // make a cluster for each point

    for kk <- x.dim until k by -1 do
        val (si, sj) = bestMerge (kk)               // find the 2 closest clusters
        clust += si | sj                            // add the union of sets i and j
        clust -= si                                 // remove set i
        clust -= sj                                 // remove set j
    end for

    finalClusters ()                                // make final cluster assignments
    calcCentroids (x, to_c, sz, cent)               // calculate centroids for clusters
end train
```

After reducing the number of clusters to the desired number k (which defaults to 2), final cluster assignments are made and centroids are calculated. Intermediate clustering results are available making it easier for the user to pick the desired number of clusters after the fact. The algorithm can be rerun with this value for k.

14.7.1 `HierClusterer` Class

Class Methods:

```
@param x  the vectors/points to be clustered stored as rows of a matrix
@param k  stop when the number of clusters equals k

class HierClusterer (x: MatrixD, k: Int = 2)
      extends Clusterer:

def train (): Unit =
def cluster: Array [Int] = to_c
def centroids: MatrixD = cent
def csize: VectorI = sz
def classify (z: VectorD): Int = distance (z, cent).argmin ()
```

14.7.2 Exercises

1. Compare `HierClusterer` with `KMeansClustererHW` and `KMeansClusterer` for a variety of datasets, starting with the six points given in the `KMeansClusterer` section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the sse over the `NTESTS` test cases.

2. K-Means Clustering techniques often tend to produce better clusters (e.g., lower *sse*) than Hierarchical Clustering techniques. For what types of datasets might Hierarchical Clustering be preferred?

3. What is the relationship between Hierarchical Clustering and Dendrograms?

14.8 Markov Clustering

The `MarkovClusterer` class implements a Markov Clustering Algorithm (MCL) and is used to cluster nodes in a graph. The graph is represented as an edge-weighted adjacency matrix (a non-zero cell indicates nodes i and j are connected).

The primary constructor takes either a graph (adjacency matrix) or a Markov transition matrix as input. If a graph is passed in, the normalize method must be called to convert it into a Markov transition matrix. Before normalizing, it may be helpful to add self loops to the graph. The matrix (graph or transition) may be either dense or sparse. See the MarkovClusteringTest object at the bottom of the file for examples.

14.8.1 `MarkovClusterer` Class

Class Methods:

```
@param t  either an adjacency matrix of a graph or a Markov transition matrix
@param k  the strength of expansion
@param r  the strength of inflation

class MarkovClusterer (t: MatrixD, k: Int = 2, r: Double = 2.0)
      extends Clusterer:

def train (): Unit =
def cluster: Array [Int] = clustr
def centroids: MatrixD = throw new UnsupportedOperationException ("not applicable")
def csize: VectorI = throw new UnsupportedOperationException ("not applicable")
def addSelfLoops (weight: Double = 1.0): Unit =
def normalize (): Unit =
private def expand (): Unit = t~^^k
private def inflate (): Boolean =
def processMatrix (): MatrixD =
def classify (y: VectorD): Int = throw new UnsupportedOperationException ()
```

14.8.2 Exercises

1. Draw the directed graph obtained from the following adjacency matrix, where `g(i, j) == 1.0` means that a directed edge exists from node *i* to node *j*.

```
val g = MatrixD ((12, 12),
    0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0,  0.0,
    0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0,  0.0,
    1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,  1.0,
    1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0,  1.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,  0.0)
```

Apply the MCL Algorithm to this graph and explain the significance of the resulting clusters.

```
val mg = new MarkovClusterer (g)
mg.addSelfLoops ()
mg.normalize ()
println ("result  = " + mg.processMatrix ())
mg.train ()
println ("cluster = " + mg.cluster.deep)
```

Part III

Simulation

Chapter 15

Simulation Foundations

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics. The focus of this chapter is simulation modeling. Viewed as a black-box, a simple *model* maps an input vector $\mathbf{x}$ and a scalar time t to an output/response vector $\mathbf{y}$,

$$\mathbf{y} \;=\; \mathbf{f}(\mathbf{x}, t) + \boldsymbol{\epsilon} \tag{15.1}$$

where $\boldsymbol{\epsilon}$ is the error/residual vector.

A *simulation model* typically adds to these the notion of *state*, represented by a vector-valued function of time $\mathbf{x}(t)$. External input (e.g., an external driving force) is now renamed to $\mathbf{u}(t)$ and the error vector is replaced with a noise process $\mathbf{w}(t)$ (e.g., a Gaussian while noise process). Commonly, simulations model systems that evolve over time.

$$\boxed{\mathbf{y}(t) \;=\; \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) + \mathbf{w}(t)} \tag{15.2}$$

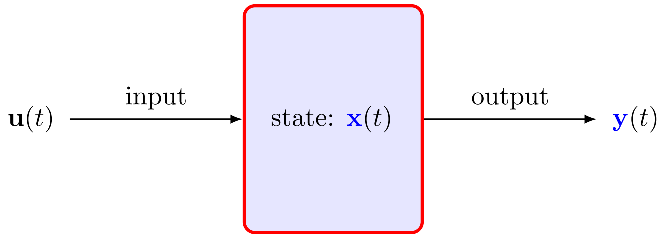


Figure 15.1: Conceptual Framework for Simulation Models

Knowledge about a system or process is used to define state as well as how state can change over time. Theoretically, this should make such models more accurate, more robust, and have more explanatory power. Ultimately, we may still be interested in how inputs affect outputs, but to increase the realism of the model with the hope of improving its accuracy, much attention must be directed in the modeling effort to state and state transitions. This is true to a degree with most simulation modeling paradigms or world views.

Once a simulation model has been validated, one of its strengths is that it can be used to address *what-if questions*. What if we add another lane to an interstate highway. Will this lead to reduced traffic congestion and reduced travel times? Such capabilities allow simulation models to play a larger role in *perspective analytics*. This can be taken farther with *simulation optimization*, which can seek improvements to systems.

This chapter focuses on foundations necessary for creating simulation models. as well as some simulation modeling techniques that can be performed without substantial software.

The following textbooks on Discrete-Event Simulation are recommended:

1. *Discrete-Event System Simulation*, 5th Edition, J. Banks, J. Carson, B. Nelson and D. Nicol, 2010 [12].

2. *Simulation Modeling and Analysis*, 5th Edition, A. Law, 2015 [105].

15.1 Basic Concepts

The following basic concepts are common to many types of simulation models.

- **Simulation Model**: A simulation model consists of collection of entities that interact with each other. The model may be view as simplified version of a system (existing or imagined). The model should be useful for description, prediction, and/or prescription. For improved explainability, the it is often desirable that the model mimics the behavior the real system.
- **Entity**: An entity is an identifiable object in a simulation model, e.g., a customer entering a bank, or a vehicle traveling on a road. One may think of an entity having a trajectory in time and space.
- **Attribute**: An attribute is a property of an entity that is relevant for the model, e.g., the speed and weight of the vehicle.
- **State**: The current values for all the variable in the model (or attribute values for all entities). These may be collected into a state vector $\mathbf{x}(t)$ that evolves over time. The *Restorable State* of the model may be thought of a sufficient recording of the execution of the model so far. This would allow a snapshot to be saved and later restored for continued execution. For some models, recent history needs to be maintained along with the current state. For a **Markov Models**, the state only depends on current values, i.e., the future conditioned on the present is independent of the past. For such models, the two notions of state are identical.
- **Event**: An event is an *instantaneous occurrence* that has the potential to change the state of the system being modeled, e.g., the arrival of customer at a bank. It may also trigger other events to occur in the future.
- **Simulation Clock**: To keep track of the advancement of time, a simulation clock is maintained. For *continuous-time simulation*, time smoothly advances in small increments. For *discrete-time simulation*, time advances by one time-unit (e.g., set to 1) for each tick of the clock. For *discrete-event simulation*, time jumps from the "event time of the current event" to the "event time of the next event" (any intermediate time is skipped).
- **Activity**: Entities in a model undergo activities that start on one event, have a duration described by a random variable, and end on another event, e.g., a customer being served by a bank teller.
- **Indefinite Delay**: An entity must wait for a server or other resource to become available. As this delay depends on other entities, the delay in not definite as it is for activities, e.g., waiting time in a queue.

15.2 Types of Models

While many modeling techniques, such as Regression, focus on predicting expected values,

$$\hat{y} = \mathbb{E}[y|\mathbf{x}] \tag{15.3}$$

simulation models generate data. Then techniques discussed in this text, can be used to analyze the data instances produced as output of the simulation model.

15.2.1 Example: Modeling an M/M/1 Queue

A single server queue is characterized by an arrival process and a service distribution. For an M/M/1 Queue the inter-arrival time distribution and service distribution are both Exponential,

$$ia_j \sim \text{Exponential}(\lambda) \tag{15.4}$$

$$s_j \sim \text{Exponential}(\mu) \tag{15.5}$$

where λ is the arrival rate and μ is the service rate (not to be confused the mean).

The queue can be modeled as a Continuous-Time Markov Chain and the expected waiting time in the queue can be determined. Unfortunately, as queuing systems become more complex, analytic solution may not be available.

A more general, although less efficient approach is to simulate customers making their way through the queue. They arrive a certain time t_a, begin service at time t_s and depart at time t_d. Averages for m customers may then be used to estimate expected waiting times T_q and service times T_s.

The construction of such simulation models is straightforward. One approach is based on the observation that the number of customers in a queueing system remained constant, except at special points in time, where an event occurs. Changes of the state of systems (e.g., number of customers) can only occur at events. The simulation therefore consists of programming logic that indicates what happens when an event occurs. This is the first major paradigm for simulation modeling and is referred to as *event-scheduling*.

For complex simulations, the logic may become fragmented, so an alternative is to track active entities in the systems and give the logic that they follow. For example, one may think of a customer entering a bank as an actor following a script. The programming logic for the simulation model then becomes writing scripts of each type of actor. This is the second major paradigm for simulation modeling and is referred to as *process-interaction*.

There are additional paradigms for simulation modeling that will be discussed later.

15.3 Random Number Generation

Let us imagine you are one of the actors following the script that you were given and you notice that all actors are doing exactly the same thing, following the same steps, take the same routes through the system and experiencing the same service times. Clearly, such simulations would not *mimic* reality. There should be uncertainty or variability in behavior. This is accomplished by introducing *randomness*.

The question becomes how to introduce randomness in a deterministic digital computer. In particular, one wants to generate a sequence of random numbers of the following form.

$$r_i \sim UID(0, 1) \tag{15.6}$$

The random numbers should be Uniformly and Independently Distributed. A Random Number Generator (RNG) may be used to produce such numbers.

15.3.1 Example RNG: Random0

Random number generators work by producing a long sequence of integers (`Int` or `Long`) that do not repeat. An obvious way to do this is to pick a large modulus `M`, increment an integer `x` and take the mod. Such a sequence of numbers will not repeat until M are generated. The `Random0` class implements this approach.

```
@param stream  the random number stream index

case class Random0 (stream: Int = 0)
     extends RNG (stream):

    private val M    = 2147483647          // modulus for a popular 32-bit generator (2^31 - 1)
    private val NORM = 1.0 / M.toDouble    // normalization to (0, 1)
    private var x    = stream              // set stream value to its seed = stream index here

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Return the modulus used by this random number generator.
     */
    def getM: Double = M.toDouble

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Return the next random number as a 'Double' in the interval (0, 1).
     *  Compute x_i = (x_i-1 + 1) % m using x = (x + 1) % m
     */
    inline def gen: Double = { x = (x + 1) % M; x * NORM }

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Return the next stream value as a 'Int' in the set {1, 2, ... , m-1}.
     *  Compute x_i = (x_i-1 + 1) % m using x = (x + 1) % m
     */
    inline def igen: Int = { x = (x + 1) % M; x }

end Random0
```

The `Random0` class will produce numbers in the interval [0, 1) when the `gen` method is called. The *period* of the generator (before it repeats itself) is equal to `M`. So far, so good. Before using this generator, a battery of tests should be applied to check its suitability.

15.3.2 Testing Random Number Generators

Means Test

The Means Test simply computes several means by averaging sub-sequences of the random number stream. Suppose the test has a sample of n means from sub-sequences of length m.

$$\mu_i = \frac{1}{m}\sum_{j=0}^{m-1} r_{im+j} \qquad \text{for} \quad i = 0, \ldots n-1 \tag{15.7}$$

Due to the Central Limit Theory, the means should be Normally distributed and the sample should have the following expected value and variance.

$$\mathbb{E}[\mu_i] = \frac{1}{m} m \mathbb{E}[r_i] = 0.5 \tag{15.8}$$

$$\mathbb{V}[\mu_i] = \frac{1}{m^2} m \mathbb{V}[r_i] = \frac{1}{12m} \tag{15.9}$$

Distribution Test

The Distribution Test determines how well sub-streams of the generated random numbers are distributed over the unit interval. Are they uniformly spread out over the interval or concentrated in certain regions. This can be assessed by a Goodness-of-Fit Test, e.g., the Chi-square Goodness-of-Fit Test or the Kolmogorov-Smirnov Goodness-of-Fit Test (see the exercises).

The Chi-square Goodness-of-Fit Test checks how well a histogram from a subsequence of length m matches the density function of the Uniform(0, 1) distribution. Each interval, say I_j, in the histogram will have an *observed* o_j and *expected* e_j number of generated random numbers within interval I_j.

$$\begin{aligned} o_j &= \nu\{r_i \in I_j : i \in \{0, \ldots m-1\}\} \\ e_j &= m \cdot f(\text{middle}(I_j)) \end{aligned}$$

f is the probability density function (pdf) and in this case it is the pdf for the `Uniform (0, 1)` distribution. Suppose there are n intervals, then $e_j = m/n$, while o_j is determined by a counter that is incremented whenever a random number r_i is generated that is within interval I_j. The Chi-square test statistic is then

$$\chi^2 = \sum_{j=0}^{n-1} \frac{(o_j - e_j)^2}{e_j} \tag{15.10}$$

When $\chi^2 > \chi^2_{\alpha,n-1}$, the sample suggests the distribution is not Uniform, where α is the significance level (e.g., .95).

Auto-Correlation Test

The are many tests related to checking correlation. As with time series, the auto-correlation may be examined by looking at a Correlogram that indicates the Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF) for increasing lags. The k-lag auto-correlation ρ_k is given by

$$\rho_k \;=\; \frac{\mathbb{C}\,[r_j, r_{j+k}]}{\mathbb{V}\,[r_j]} \qquad \text{for any} \quad j \tag{15.11}$$

Ideally, $\rho_0 = 1$ and the rest are close to zero, indicating lack of correlation. Note, the above fomula for ρ_k assumes stationarity (see the chapter on time series).

The `Random0` class **fails** all three tests, see the exercises.

15.3.3 Example RNG: Random3

Rather than adding one each time before using the mod operator (%), it will work better to multiply the stream value by a constant.

```
def gen: Double = { x = A * x % M; x * NORM }
```

Extensive testing has been used to find good values for the constant `A`. One example is `A` = 16807. Random number generators of this form are know as Multiplicative Linear Congruential Generators (MLCG). This value for `A` is an example of primitive-element modulo `M`, allowing the generator to exhibit a full period, *period* = `M-1`. In other words, the generator will produce all stream values from 1 to `M-1`, inclusive, before repeating itself. In ScalaTion, this generator is available in the `Random3` class.

$$x_i \;=\; Ax_{i-1} \,\%\, M \qquad \text{stream value} \tag{15.12}$$

$$r_i \;=\; x_i / M \qquad \text{random number} \tag{15.13}$$

where $A = 7^5 = 16807$ and $M = 2^{31} - 1 = 2147483647$.

In general, a Linear Congruential Generators (LCG) take the following form,

$$x_i \;=\; (Ax_{i-1} + C) \,\%\, M \qquad \text{stream value} \tag{15.14}$$

$$r_i \;=\; x_i / M \qquad \text{random number} \tag{15.15}$$

The next step up (longer period and better properties) is a Multiple Recursive Generator (MRG). These can be combined for further improvement into a Combined Multiple Recursive Generator (CMRG). ScalaTion's `Random` class is an example of an CMRG developed by L'Ecuyer and Touzin [107].

15.3.4 Exercises

1. Apply the above three tests to the `Random0`, `Random` and `Random3` random number generators. See the `RNGTester` object in the `scalation.random` package for the test methods: `meansTest`, `distributionTest`, and `correlationTest`.

2. Discuss additional tests that are applied to assess the quality of a random number generator.

3. Explain how Multiple Recursive Generators (MRG) work and what advantages they may have over Linear Congruential Generators (LCG).
4. Explain how Kolmogorov-Smirnov Goodness-of-Fit tests work.
5. Finish the implementation of the `distributionTest_KS` function to implement the Kolmogorov-Smirnov Goodness-of-Fit test.

15.4 Random Variate Generation

With a good quality random number generator as a foundation, Random Variate Generators (RVG) for a variety of probability distributions can be created.

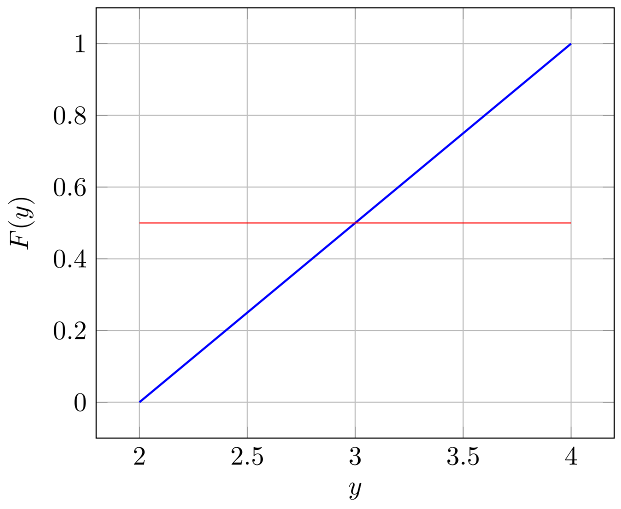
15.4.1 Inverse Transform Method

One of the simplest methods for converting random numbers into random variates following some distribution F is the inverse transform method that uses the iCDF F^{-1} to transform the random number.

General Uniform Distribution

Suppose $y \sim F$ and $r \sim \text{Uniform}(0, 1)$. Now let F be a general uniform distribution, $\text{Uniform}(a, b)$.

CDF for Uniform Distribution on [2, 4]



Conceptually, the Inverse Transform Method generates a random number r and use it select the height of a horizontal line in the diagram. Drop a vertical line from where it intersects the Cumulative Distribution Function (CDF) F. The position in the horizontal axis is the value for the random variate y. For example, if $r = 0.5$, then $y = 3.0$.

Mathematically, the relationship is

$$F(y) = r \tag{15.16}$$

$$y = F^{-1}(r) \tag{15.17}$$

The CDF for the general uniform distribution is $F(y) = \dfrac{y-a}{b-a}$, so

$$y = a + (b-a)r \tag{15.18}$$

Exponential Distribution

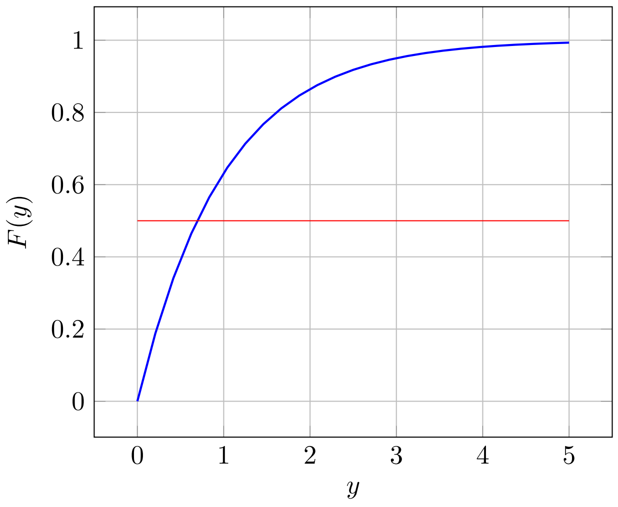
The Inverse Transform Method also works well for the *Exponential*(λ) distribution where the CDF $F(y) = 1 - e^{-\lambda y}$. Setting $F(y) = r$ and solving for the inverse yields,

$$\begin{aligned}
F(y) \;=\; 1 - e^{-\lambda y} &= r \\
e^{-\lambda y} &= 1 - r \\
e^{-\lambda y} &= r \\
-\lambda y &= \ln(r) \\
y &= -\frac{\ln(r)}{\lambda}
\end{aligned}$$

The third step relies on the fact that $1 - r$ and r have the same distribution.

To illustare its use, for example with $\lambda = 1$, if the generated random number $r = 0.5$, then generated random variate $y = 0.693$.

CDF for Exponential Distribution with $\lambda = 1$ on [0, 5]



There are numerous Random Variate Generators in SCALATION that extend the `Variate` abstract class, including the `Exponential` class.

```
@param mu      the mean
@param stream  the random number stream

case class Exponential (mu: Double = 1.0, stream: Int = 0)
     extends Variate (stream):

    if mu <= 0.0 then flaw ("constructor", "parameter mu must be positive")

    private val l = 1.0 / mu              // lambda, the rate parameter

    val mean = mu
    def pf (z: Double): Double = if z >= 0 then l * exp (-l*z) else 0.0

    def gen: Double = -mu * log (r.gen)

    def gen1 (z: Double): Double = -z * log (r.gen)

end Exponential
```

Note, since the mean is the reciprocal of the rate, $\mu = \frac{1}{\lambda}$, the simple `gen` method `-mu * log (r.gen)` produces exponentially distributed random variates.

15.4.2 Convolution Method

Another simple method for generating random random variates is the convolution method that involves the summation of multiple simpler random variates.

For example, a `Binomial` random variate may be created by adding n `Bernoulli` random variates. The Bernoulli(p) distributions models the flip of coin where the probability of head (success) is p (and $q = 1-p$). The probability mass function (pmf) is shown below.

$$p_x(x) \;=\; p^x q^{1-x} \qquad \text{where} \;\; x \in \{0, 1\}$$

The Binomial(p, n) distribution models the number of heads (successes) in flipping n coins ($y = x_1 + \cdots + x_n$).

$$p_y(y, n) \;=\; \binom{n}{y} p^y q^{n-y} \qquad \text{where} \;\; y \in \{0, 1, \dots n\}$$

The `gen` method in the `Binomial` class therefore simply adds up the results of n coin flips where a head is 1 and a tail is 0.

```
@param p       the probability of success
@param n       the number of independent trials
@param stream  the random number stream

case class Binomial (p: Double = .5, n: Int = 10, stream: Int = 0)
     extends Variate (stream):

    if p < 0.0 || p > 1.0 then flaw ("constructor", "parameter p must be in [0, 1]")
    if n <= 0 then             flaw ("constructor", "parameter n must be positive")
    _discrete = true
    private val q    = 1.0 - p                  // probability of failure
    private val p_q  = p / q                    // the ratio p divided by q
    private val coin = Bernoulli (p, stream)    // coin with prob of success of p

    val mean = p * n

    def pf (z: Double): Double = { val k = z.toInt; if z == k then pf (k) else 0.0 }

    def pf (k: Int): Double = if k in (0, n) then choose (n, k) * p~^k * q~^(n-k) else 0.0
    override def pmf (k: Int): Array [Double] =
        val d = Array.ofDim [Double] (n+1)      // array to hold pmf distribution
        d(0)  = q~^n
        for k <- 1 to n do d(k) = d(k-1) * p_q * (n-k+1) / k.toDouble
        d
    end pmf

    def gen: Double = summation (n)(coin.gen)

    def gen1 (z: Double): Double = summation (z.toInt)(coin.gen)

end Binomial
```

Note, the `summation` top-level function is defined in `Util.scala`.

```
inline def summation (n: Int)(formula: => Double): Double =
    var sum = 0.0
    for i <- 0 until n do sum += formula
    sum
end summation
```

It evaluates the given `formula` a total of `n` times and returns the sum.

15.4.3 Acceptance-Rejection Method

The Acceptance-Rejection method for generating random variates may be used for complex distributions. Given the density function $f(y)$ for a complex distribution, find a simpler (or rather easier to generate) distribution with density function $g(y)$, such that

$$f(y) \;\leq\; cg(y) \qquad \text{for } y \in D_y \tag{15.19}$$

where the reciprocal of the constant $c \geq 1$ indicates the probability of acceptance. The procedure is to generate a random value y from the simple distribution and depending on the following ratio

$$\frac{f(y)}{cg(y)} \tag{15.20}$$

randomly keep it the closer the ratio is to 1, i.e.,

$$\text{if } \; \frac{f(y)}{cg(y)} \;\geq\; r \;\; \text{then} \;\; \text{accept} \;\; \text{else} \;\; \text{reject} \tag{15.21}$$

where r is a random number. Rejection means to keep trying.

15.4.4 Exercises

1. Consider a distribution where the density linearly increases on the interval $[0, b]$. The pdf for this distribution is the following:

$$f(y) = \frac{2y}{b^2} \qquad \text{on} \quad [0, b]$$

Use the Inverse Transform Method (ITM) to generate random variates following this distribution.

(a) Determine the Cumulative Distribution Function (CDF) $F(y)$.

(b) Determine the inverse Cumulative Distribution Function (iCDF) $F^{-1}(r)$.

(c) Write code for the `gen` method.

(d) Create a `case class` to contain the `gen` method and produce a `Histogram` that shows how the generated random variates are distributed.

2. Use the convolution method to generate random variates following the Erlang(λ, k) distribution, where λ is the rate parameter and k is the number of events. The random variable can be used to measure the time for k events to occur. When $k = 1$, it reduces to the Exponential distribution. Since an Erlang random variable is the sum of k independent exponential random variables, the convolution method may be applied. The pdf for the Erlang distribution is shown below.

$$f(y) = \frac{\lambda^k y^{k-1} e^{-\lambda y}}{(k-1)!} \qquad \text{on} \quad [0, \infty)$$

3. Test the Convolution Method for generating Binomial random variates for $p = .5$ and $n = 4$. Generate 10,000 random variates and show the histogram.

4. Consider the Standard Normal distribution for positive values of y. Its density function can be bounded using c times an Exponental density function. Flipping a coin allows the generation of negative values. Apply the Acceptance-Rejection method to generate Standard Normal random variates.

   Hint: see `http://www.columbia.edu/~ks20/4703-Sigman/4703-07-Notes-ARM.pdf` [171].

5. Consider a distribution with density on the interval $[0, 2]$. Let the probability density function (pdf) for this distribution be the following:

$$f_y(y) = \frac{y}{2} \qquad \text{on} \quad [0, 2]$$

Use the Inverse Transform Method (ITM) to generate random variates following this distribution.

(i) Determine the inverse Cumulative Distribution Function (iCDF) $F_y^{-1}(r)$. Recall r denotes a random nymber.

(ii) Write code for the `gen` method for its Random Variate Generator (RVG).

(iii) Draw the CDF $F_y(y)$ vs. y and illustrate how the ITM works in this case.

15.5 Poisson Process

In this section, the relationships between three random variables are examined. Consider a system in which events (e.g., arrivals) occur randomly, but at a constant rate λ. It is further assumed that the time to the next arrival is independent of the previous arrival.

- Inter-arrival Time. T = the time interval between subsequent arrivals/events. As the time duration Δt becomes arbitrarily small, the probability of an arrival equals the arrival rate multiplied by the time span.

$$F_T(\Delta t) \;=\; P(T \leq \Delta t) \;=\; \lambda \Delta t \tag{15.22}$$

- Time of n^{th} Arrival. S_n = the time span for n arrivals/events to occur.

$$S_n \;=\; \sum_{i=1}^{n} T_i \tag{15.23}$$

- Counting Process. $N(t)$ = the counter of the number events (e.g., arrivals) by time t.

$$N(t) < n \;\text{ iff }\; S_n > t \tag{15.24}$$

$$N(t) = 0 \;\text{ iff }\; T > t \tag{15.25}$$

Define the complementary CDF (cCDF) as follows:

$$\bar{F}_T(t) \;=\; P(T > t) \;=\; 1 - F_T(t) \tag{15.26}$$

Due to the independence assumption, the probability of no arrivals by time $t + \Delta t$, is the product of the following two probabilities.

$$\bar{F}_T(t + \Delta t) \;=\; \bar{F}_T(t)\bar{F}_T(\Delta t) \tag{15.27}$$

Since $\bar{F}_T(\Delta t) = 1 - \lambda \Delta t$, the following holds.

$$\bar{F}_T(t + \Delta t) \;=\; \bar{F}_T(t)(1 - \lambda \Delta t) \tag{15.28}$$

Multiplying out the rhs and rearranging gives,

$$\bar{F}_T(t + \Delta t) - \bar{F}_T(t) \;=\; -\lambda \Delta t \bar{F}_T(t) \tag{15.29}$$

Dividing both sides by Δt results in

$$\frac{\bar{F}_T(t + \Delta t) - \bar{F}_T(t)}{\Delta t} \;=\; -\lambda \bar{F}_T(t) \tag{15.30}$$

A derivative is produced by taking the limit as $\Delta t \to 0$

$$\frac{d}{dt}\bar{F}_T(t) \;=\; -\lambda \bar{F}_T(t) \tag{15.31}$$

Both sides may now be integrated.

$$\int \frac{d\bar{F}_T(t)}{\bar{F}_T(t)} \;=\; -\int \lambda dt \tag{15.32}$$

The integral of a reciprocal introduces a natural logarithm.

$$\ln \bar{F}_T(t) \;=\; -\lambda t \tag{15.33}$$

Taking the exp function of both sides yields,

$$\bar{F}_T(t) \;=\; e^{-\lambda t} \tag{15.34}$$

See thr exercises for more details. Switching back to the regular CDF, shows that inter-arrival time follows the Exponential distribution.

$$F_T(t) \;=\; 1 - e^{-\lambda t} \tag{15.35}$$

Consequently, S_n follows the Erlang distribution (see the exercises from the last section).

The counting process, $N(t)$, is a Poisson Process with the following pmf:

$$p_{N(t)}(n) \;=\; \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{15.36}$$

The probability of no arrivals by time t, which is

$$p_{N(t)}(0) \;=\; e^{-\lambda t} \;=\; \bar{F}_T(t) \tag{15.37}$$

corresponds to the probability that the inter-arrival time is greater time t.

15.5.1 Generating a Poisson Process

In ScalaTion, the `PoissonProcess` class is used to simulate a Poisson Process. For such processes, there is often interest in generating three things: arrival times, the counting process itself, and flow per time interval.

```
@param t        the terminal time
@param lambda   the arrival rate
@param stream   the random number stream to use

class PoissonProcess (t: Double, lambda: Double = 1.0, stream: Int = 0)
      extends VariateVec (stream):

def mean: VectorD = VectorD.fill (1)(lambda * t)      // mean of N(t)
def pf (z: VectorD): Double = ???
def igen: VectorI = gen.toInt
def gen: VectorD =
def num (tt: Double): Int =
def flow (t_span: Double): VectorI =
```

First is the event/arrival times, e.g., the times that vehicles pass a road sensor. The `gen` method will produce the arrival times from time zero up to the terminal/end time of the simulation t. These arrival times are returned in a vector. As the inter-arrival time `t_ia` distribution is `Exponential (mu, stream)` where `mu = 1.0 / lambda`, it is used to generate each time increment.

```
def gen: VectorD =
    val atime = ArrayBuffer [Double] ()
    var now   = 0.0
    while now <= t do
        now   += t_ia.gen
        atime += now
    end while
    t_a = VectorD (atime)
    t_a
end gen
```

Note that arrival time `t_a(k)` will be distributed as Erlang(λ, k).

Second is the counting process as a function of time $N(t)$, e.g., the number of vehicles passing a sensor since the beginning of the simulation. Given that the arrival times have been generated, the `num` method returns the index value where the next arrival time exceeds the specified time `tt`.

```
def num (tt: Double): Int =
    if t_a == null then gen
    for i <- t_a.indices if t_a(i) > tt do return i
    t_a.dim
end num
```

Note that count `num(tt)` follows a PoissonProcess(t, λ).

Third is the incremental counts per an interval of time, e.g., the number of vehicles passing a sensor over a 5 minute interval of time. The `flow` method counts the number arrivals for each time interval (of length `t_span`) by taking the difference between the count at the end of interval `n2` and the beginning of the interval `n1`.

```
def flow (t_span: Double): VectorI =
    if t_a == null then gen
    val flw = ArrayBuffer [Int] ()
    var now = 0.0
    var n1  = 0
    while now <= t do
        val n2 = num (now)
        flw   += n2 - n1
        now   += t_span
        n1     = n2
    end while
    VectorI (flw)
end flow
```

The distribution of the flow is left as an exercise.

15.5.2 Generating a Non-Homogeneous Poisson Process

A simulation using a constant arrival rate to model traffic flow will be of low fidelity. Vehicle arrival rates vary dramatically over a day, with low vehicle counts at night and high spikes during morning and late afternoon rush hours.

To create more realistic, or higher fidelity, simulation models, a Non-Homogeneous Poisson Process (NHPP) may be used. The extension can be accomplished by converting the constant λ to a function of time $\lambda(t)$. In ScalaTion, the `NH_PoissonProcess` can be used to generate arrivals where the arrival rate is given by the `lambdaf` function.

```
 *  @param t        the terminal time
 *  @param lambdaf  the arrival rate function, lambda(t)
 *  @param stream   the random number stream to use
 */
class NH_PoissonProcess (t: Double, lambdaf: FunctionS2S, stream: Int = 0)
      extends PoissonProcess (t, 1.0, stream):

    override def mean: VectorD = VectorD.fill (1)(lambdaBar * t)   // mean of N(t)
    override def pf (z: VectorD): Double = ???
    override def gen: VectorD =
```

The `gen` must be overridden to adjust the time jump based on the current arrival rate. Fortunately, this can be done dividing an Exponential (1) random variate by the current arrival rate.

```
   override def gen: VectorD =
        val atime = ArrayBuffer [Double] ()
        var now   = 0.0
        while now <= t do
            val lamb = lambdaf (now)                    // current value of the lambda function
            now     += t_ia.gen / lamb                  // adjust by dividing current lambda
            atime   += now
        end while
        t_a = VectorD (atime)
        t_a
    end gen
```

15.5.3 Exercises

1. Plot the pdf of the Erlang$(1, k)$ distribution, for $k = 2$ and $k = 3$.

2. Compare the above pdf with a histogram of the time for the second arrival ($k = 2$) and the third arrival ($k = 3$). See the `Histogram` class in the `mathstat` package.

3. Call the `num` method for several time points and plot it ($N(t)$) versus time t.

4. For the vehicle arrival process simulation, collect data on the number of arrivals every 5 minutes. Create a histogram to show the distribution.

5. One way to create a time-dependent lambda function `lambdaf` is to take a data file, say consisting of traffic counts, e.g., `travelTime.csv` in the `data` directory. Then create a Polynomial Regression model using `PolyRegression` from the `modeling` package. Finally, define a `lambdaf` that calls the Polynomial Regression model `predict` method.

6. The following equation states that

$$\frac{d}{dt}\bar{F}_T(t) \;=\; -\lambda\bar{F}_T(t)$$

the rate of decrease in the cCDF is proportional to its value. This is analogous to the phenomena of radioactive decay, where the rate of decay is proportional to the amount of radioactive material. Both are described to the above Ordinary Differential Equation (ODE). The following steps may be followed to solve the ODE.

Step 1: Separation of Variables

$$\frac{d\bar{F}_T}{\bar{F}_T} \;=\; -\lambda dt$$

Step 2: Integrate Both Sides

$$\int \frac{d\bar{F}_T}{\bar{F}_T} \;=\; \int -\lambda dt$$

Step 3: Use the Fact that the Derivative of $\ln x = \dfrac{1}{x}$

$$\ln \bar{F}_T + C \;=\; -\lambda t$$

Step 4: Determine Constant of Integration C using the Initial Condition (IC): $\bar{F}_T(0) = 1$

$$\ln 1 + C \;=\; 0$$

Since C = 0, we have,

$$\ln \bar{F}_T \;=\; -\lambda t$$

Step 5: Take the *exp* Function on Both Sides

$$\bar{F}_T \;=\; e^{-\lambda t}$$

Verify that the solution is correct by showing that it satisfies both the ODE and the IC.

15.6 Monte Carlo Simulation

Many problems can be addressed by drawing a *sample* and determining whether it satisfies some criterion. For example, draw five cards and determine whether the hand is a full house (three-of-a-kind and pair/two-of-a-kind). If this process is *repeated* enough times, an estimate for the probability of a full house may be obtained.

15.6.1 Simulation of Card Games

Many card games such as Poker and Blackjack can be simulated to determine probabilities or expected earning/losses. A deck of 52 playing cards can be simulated using an array of cards. Initially, card i will have the number i associated with it. The card ordinal number (0 to 51) is mapped to the (face-value, suit) using the `value` method.

- The face-value is 1 (Ace), 2, 3, 4, 5, 6, 7, 8, 9, Jack, Queen, or King. The rank (low to high) usually moves Ace to high end of the list.
- The suit is ordered (low to high) in some games like bridge: Clubs ($\clubsuit$), Diamonds ($\diamondsuit$), Hearts ($\heartsuit$), and Spades ($\spadesuit$).

The deck will be shuffled using the `shuffle` method to randomize the positions of cards in the deck. The `draw` method pulls the next card from the top of the deck. Calling it five times yields a poker hand. A counter can be incremented in case the hand is a full house. The ratio of this counter to the number of hands dealt becomes and estimate for the probability of a full-house.

```
class Cards:

    private val NUM_CARDS = 52                              // number of cards in deck
    private val card      = Array.range (0, NUM_CARDS)      // the cards themselves
    private val rn        = Randi (0, NUM_CARDS - 1)        // random number generator
    private var top       = 0                               // index of top card
    private val suit      = Array ('C', 'D', 'H', 'S')      // Clubs, Diamonds, Hearts, Spades

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Draw the top card from the deck and return it.  Return -1 if no cards are
     *  left in the deck.
     */
    def draw (): Int = if top == NUM_CARDS then -1
                       else { val c = card(top); top += 1; c }

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Shuffle the deck of cards.
     */
    def shuffle (): Unit =
        for i <- card.indices do swap (card, i, rn.igen)
        top = 0
    end shuffle
    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Convert the card number c (0 to 51) to the face value (1 to 13) and
     *  suit (0(C), 1(D), 2(H), 3(S)).
     *  @param c  the card ordinal number
     */
    def value (c : Int): (Int, Char) = (c % 13 + 1, suit (c / 13))

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Convert the card deck to a string.
     */
    override def toString: String = "Cards ( " + stringOf (for c <- card yield value (c)) + " )"

end Cards
```

Probability of Drawing a Particular Hand

The estimate of the probability of a full house can be checked against probability theory. The total number of hands consisting of five cards is

$$\binom{52}{5} \;=\; \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} \;=\; 2,598,960$$

The number of ways n_{ways} for get a full house may be determined as follows: (a) choose 1 of 13 face values for the three-of-a-kind, (b) choose 1 of 12 remaining face values for the pair, (c) choose 3 of 4 suits for the three-of-a-kind, and (d) choose 2 of 4 suits for the pair, i.e.,

$$\binom{13}{1}\binom{12}{1}\binom{4}{3}\binom{4}{2} \;=\; [13]\,[12]\left[\frac{4 \cdot 3 \cdot 2}{1 \cdot 2 \cdot 3}\right]\left[\frac{4 \cdot 3}{1 \cdot 2}\right] \;=\; 78 \cdot 4 \cdot 6 \;=\; 3,744$$

Letting $\mathbf{y}$ be the random draw of five cards, the probability of a full house is simply the ratio of the two numbers.

$$P(\mathbf{y} = \text{full house}) \;=\; \frac{3,744}{2,598,960} \;=\; 0.00144$$

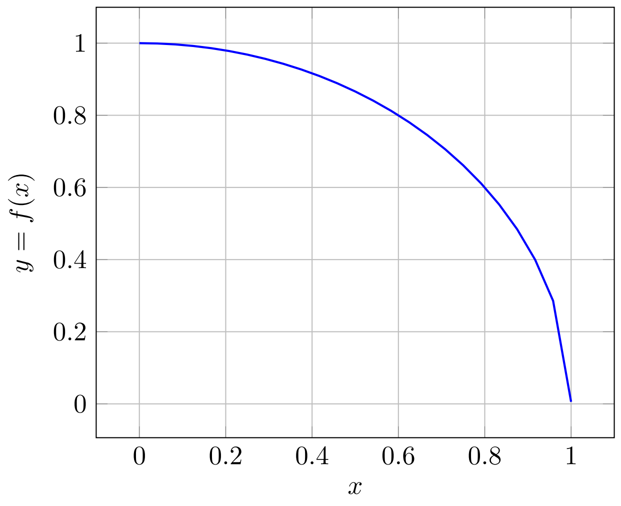
15.6.2 Integral of a Complex Function

As discussed earlier, there is no closed-form formula for the Cumulative Distribution Function (CDF) for the Normal distribution. As the integral of the probability density function (pdf), the Numerical Integration can be used to compute the CDF. Monte Carlo integration offers one approach for doing this, which is practically effective for multiple integrals in higher dimensions.

To illustrate, consider the following one dimensional function defined on the domain $[0, 1]$.

$$y \;=\; f(x) \;=\; \sqrt{(1 - x^2)}$$

Integration: Area Under the Curve



The integral is the area under the curve. This is the same as the mean height of the curve times the length/size of the domain (1 in this case). The mean height of a function may be estimated by computing the height at many randomly selected points and taking the average.

$$\bar{y} \;=\; \frac{1}{m}\sum_{i=0}^{m-1} f(x_i) \;=\; \frac{1}{m}\sum_{i=0}^{m-1} \sqrt{(1 - x_i^2)}$$

In ScalaTion, this capability is provided by the `MonteCarloIntegration.integrate` method.

```
    def integrate (f: FunctionS2S, a: Double, b: Double, m: Int, s: Int = 0): Double =
        val length = b - a
        val x    = Uniform (a, b, s)
        var sum = 0.0
        for it <- 0 until m do sum += f(x.gen)
        sum * length / m
    end integrate
```

The particular function f is passed into `integrate` in the code below. As the function traces the unit circle in the first quadrant, multiplying by 4 allows π to be approximated.

```
@main def monteCarloIntegrationTest (): Unit =

    import MonteCarloIntegration.integrate

    def f(x: Double): Double = sqrt (1 - x~^2)

    for k <- 1 to 9; s <- 0 to 1 do
        val pi = 4 * integrate (f, 0, 1, 10~^k, s)
        println (s"for k = $k, s = $s: pi = $pi")
    end for

end monteCarloIntegrationTest
```

The case class `Uniform` is a random variate generator from the `random` package. `Uniform (a, b, s)` generates (via the `gen` method) uniformly distributed random numbers in the interval `[a, b]` using random number stream `s`.

15.6.3 Grain Dropping Experiment

The `RandomVec` class may be used to produce random vectors. As a simpler cousin to the Buffon Needle experiment, the Grain Dropping experiment provides a very simple way to compute π. The first paramater for the `RandomVec` class is the dimensionality of the space (so use 2 here). The idea is to generate many grains and drop them falling at x-y coordinates inside a square centered at the origin having sides of length 2. The grains at a distance of one or less from the origin will also be inside the unit circle (radius = 1) centered at the origin.

```
@param stream  the random number stream to use

class GrainDrop (stream: Int):

    private val grain = RandomVecD (2, max = 1, min = -1, stream = stream)

    //::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Return the fraction of grains found inside the unit circle.
     *  @param n  the number of grains to generate
     */
    def fraction (n: Int): Double =
        var count = 0
        for i <- 0 until n do if grain.gen.normSq <= 1.0 then count += 1
        count / n.toDouble
    end fraction

end GrainDrop
```

The `fraction` method counts the number of generated grains that that are inside the unit circle and divides that count by the total number of grains generated. For a good random number generator, this fraction should correspond to the ratio of the areas for the unit circle versus the bounding square $\{(x, y) : x \in [-1, 1], y \in [-1, 1]\}$. The area of the square is 4 so the fraction times 4 should provide an estimate for π.

```
@main def grainDropTest (): Unit =

    for stream <- 0 to 5 do
        val bn = new GrainDrop (stream)
        banner (s"Grain Drop for stream = $stream")
        for k <- 1 to 8 do
            val n = 10~^k
            println (s"for n = $n: pi = ${4 * bn.fraction (n)}")
        end for
    end for

end grainDropTest
```

15.6.4 Simulation of the Monty Hall Problem

Imagine you are a contestant on the *Let's Make a Deal* game show and host, Monty Hall, asks you to select door number 0, 1 or 2, behind which are two worthless prizes and one luxury car. Whatever door you pick, he randomly opens one of the other non-car doors and asked if you want to stay with you initial choice or switch to the remaining door. What are the probabilities of winning if you (a) stay with your initial choice, or (b) switch to the other door? Finish the code below to validate your results.

```
@main def montyHall (): Unit =

    val stream    = 0                              // random number stream (0 to 999)
    val rg        = Randi (0, 2, stream)           // door selection (0, 1 or 2) random generator
    val coin      = Bernoulli (stream + 1)         // coin flip generator
    val stream    = 0                              // random number stream, try up to 999
    var winStay   = 0                              // count wins with stay strategy
    var winSwitch = 0                              // count wins with switch strategy

    for it <- 1 to 100000 do                       // test the strategies 100,000 times
        // car randomly placed behind this door
        // contestant randomly picks a door
        // Monty Hall shows other non-car door (if choice, make randomly)
        if pick == car then winStay   += 1         // stay with initial pick
        else                winSwitch += 1         // switch to the other door
    end for

    println (s"winStay   = $winStay")
    println (s"winSwitch = $winSwitch")

end MontyHall
```

Note, since the opened door never has the car behind it, the car must be behind either the originally picked door (stay) or the remaining door (switch). Hence, the form of the above `if then else` statement.

15.6.5 Exercises

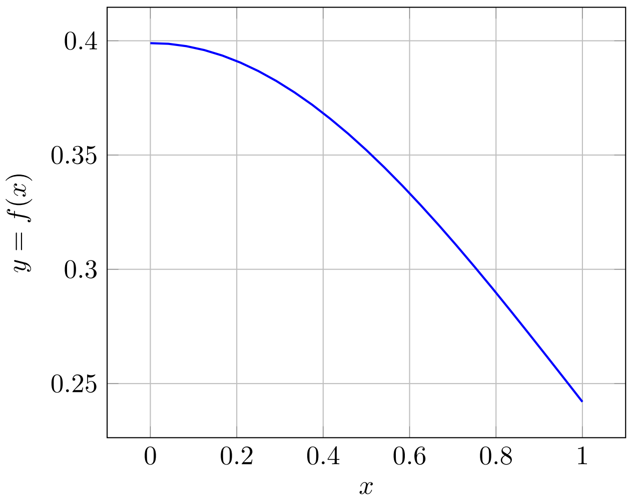
1. The hands in Five-Card Draw Poker are the following: (1) high card, (2) pair, (3) two pair, (4) three-of-a-kind, (5) straight, (6) flush, (7) full house, (8) four-of-a-kind, (9) straight flush, and (10) royal flush.

   Use probability theory to determine the probabilities of each Poker hand. Do the same thing using Monte Carlo simulation and compare the results. Let the number of repetitions (hands drawn) increase until the estimates stabilize. Hint: use a large number of samples.

2. Use Monte Carlo simulation to integrate the CDF for the Standard Normal Distribution at 1, $F_y(1)$. Note, the distribution is symmetric around zero, so the following integral may be computed.

$$\text{area} \;=\; \int_0^1 \frac{1}{\sqrt{2\pi}} e^{-y^2/2}$$

Integration: Area Under the Curve



The solution for $F_y(1)$ will be $\frac{1}{2} + area$. Check your answers by calling `CDF.normalCDF (1)` in the `random` package.

3. Finish the coding of the Simulation of the Monte Hall Problem. Determine the winning percentages for the Stay and Switch strategies as the number of game simulations increases. Do they converge? Explain why one strategy is better than the other.

4. Estimate the probability distribution for rolling three 6-sided dice and taking their sum

$$y = x_1 + x_2 + x_3$$

where $x_i \sim$ `Randi(1, 6)`, i.e., $p_{x_i}(k) = 1/6$ for $k = 1, 2, 3, 4, 5, 6$. The probability mass function (pmf) for y has a range from 3 to 18. Use Monte Carlo Simulation to estimate $p_y(k) =?$ for $k = 3, 4, \ldots, 17, 18$ Draw/plot the pmf $p_y(k)$ vs. k.

5. When rolling n_d six-sided dice, how many ways can four be rolled, for $n_d = 1, 2, 3, 4$ dice.

Table 15.1: Counting the Number of Ways to Roll a Four (sum of n_d dice)

n_d	list of ways	n_{ways}	configurations	ratio	probability $p_y(k)$
1	[4]	1	6	1/6	.1667
2	[1,3], [2,2], [3,1]	3	36	1/12	.08333
3	[1,1,2], [1,2,1], [2,1,1]	3	216	1/72	.01389
4	[1,1,1,1]	1	1296	1/1296	.007880

Use Monte Carlo simulation to estimate the number of n_{ways} and the probability mass function $p_y(k)$ for $n_d = 1, 2, 3, 4$ dice.

6. The number of ways n_{ways} can also be solved using the following recursive function for the sum s from n_d to $6n_d$.

$$n_{ways}(n_d, s) \;=\; \sum_{k=1}^{6} n_{ways}(n_d - 1, s - k)$$

The base case for the recursion is when $n_d = 1$, in which case $n_{ways}(1, s) = 1$ for $s = 1, \ldots, 6$.

Write a program to calculate n_{ways} for $n_d = 1, 2, 3, 4$ dice. Note, the recursive function may be computed more efficiently using *dynamic programming*.

7. **Question 1**: Develop a Monte Carlo simulation to estimate the volume of a unit sphere (radius eqaul to one). What is your estimate? Also, provide your code. Hint: a point is inside the sphere when $x^2 + y^2 + z^2 \leq 1$.

15.7 Hand Simulation

Before using or developing software to perform simulation, it is instructive to carry out a simple simulation by hand.

This can be done as follows: Generate random variates for inter-arrival and service times for $m = 10$ customers. Fill in these two columns in Table 15.2. Use *Exponential*$(1/\lambda)$ for inter-arrival times and *Exponential*$(1/\mu)$ for service times. Let $\lambda = 10$ and $\mu = 12$ per hour, giving means of 6 and 5 minutes, respectively. Also, fill in the zeroth row, for a non-existent customer, with all zeros.

Table 15.2: Hand Simulation of M/M/1 Queue

customer	iarrival t	**arrival** t	begin-service t	waiting t	service t	**departure** t	system t
0	0	0	0	0	0	0	0
1	6	.	.	.	5	.	.
2	3	.	.	.	6	.	.
3	5	.	.	.	4	.	.
4	4	.	.	.	3	.	.
5	3	.	.	.	5	.	.
6	8	.	.	.	7	.	.
7	5	.	.	.	2	.	.
8	7	.	.	.	4	.	.
9	9	.	.	.	5	.	.
10	6	.	.	.	8	.	.
total	56	.	.	.	49	.	.
mean	5.6	.	.	.	4.9	.	.

Notice that the sample mean inter-arrival time and sample mean service time shown in the last row should correspond the theoretical means of 6 and 5 (keeping the mind the inaccuracy of small samples)

The hand simulation part now begins. Filling in the table row-by-row requires the event logic to be followed. A customer cannot begin service until the previous customer has departed. They must wait in the queue until the server is available. The time between arrival and beginning of service is the wait time. The inter-arrival time indicates the time gap for the next arriving customer.

Given the inter-arrival (iarrival) times (ι_i) and service times (s_i), the equations below may be applied to provide values for the empty columns.

$$a_i = a_{i-1} + \iota_i \qquad \text{arrival time} \qquad (15.38)$$

$$b_i = \max(a_i, d_{i-1}) \qquad \text{begin service} \qquad (15.39)$$

$$w_i = b_i - a_i \qquad \text{waiting time} \qquad (15.40)$$

$$d_i = b_i + s_i \qquad \text{departure time} \qquad (15.41)$$

$$t_i = d_i - a_i \qquad \text{time in system} \qquad (15.42)$$

Note, the columns in **bold** correspond to events. Before looking at the completed table, try to fill in the previous one.

Table 15.3: Completed Hand Simulation of M/M/1 Queue

customer	iarrival t	**arrival** t	begin-service t	waiting t	service t	**departure** t	system t
0	0	0	0	0	0	0	0
1	6	6	6	0	5	11	5
2	3	9	11	2	6	17	8
3	5	14	17	3	4	21	7
4	4	18	21	3	3	24	6
5	3	21	24	3	5	29	8
6	8	29	29	0	7	36	7
7	5	34	36	2	2	38	4
8	7	41	41	0	4	45	4
9	9	50	50	0	5	55	5
10	6	56	56	0	8	64	8
total	56	278	291	13	49	340	62
mean	5.6	27.8	29.1	1.3	4.9	34.0	6.2

Define Σ_q, Σ_s and Σ_y to be the sums of the waiting, service and system times, repectively. As shown in the table, the average times in the Queue, Service and sYstem are given by the formulas below.

$$T_q = \Sigma_q/m = 1.3 \qquad \text{Number in Queue} \qquad (15.43)$$

$$T_s = \Sigma_s/m = 4.9 \qquad \text{Number in Service} \qquad (15.44)$$

$$T_y = \Sigma_y/m = 6.2 \qquad \text{Number in sYstem} \qquad (15.45)$$

15.7.1 Little's Law

The start of simulation is 0.0, while the end of simulation is 64.0 (for a duration (τ) of 64.0 time units/minutes). During this time $m = 10$ customers arrived, giving an *effective arrival rate* of

$$\lambda_e = \frac{m}{\tau} = \frac{10}{64.0} = 0.1563 \text{ per minute} = 9.375 \text{ per hour} \qquad (15.46)$$

Little's Law relates time averages to occupancy averages. As waiting queues get longer, one would expect expect the waiting time to increase. Using Little's Law (see the section on Markov Chains for more details), the occupancy (number in the system) L_y is proportional to the time in the system T_y (and the same is true for sub-components).

$$L_y = \lambda_e T_y \qquad (15.47)$$

The proportionality constant is λ_e. Little's Law allows the following summary results for our M/M/1 Queue simulation to be collected into Table 15.4.

Table 15.4: Formulas for M/M/1 Queueing Models

part	**length**	**result**	**time**	**result**
Queue	$L_q =$	0.203	$T_q =$	1.3 minutes
Service	$L_s =$	0.766	$T_s =$	4.9 minutes
System	$L_y =$	0.969	$T_y =$	6.2 minutes

15.7.2 Event Times

Rather than computing the number in the Queue (q), Service (s) or System (y) using Little's Law, they may be computed directly as the areas under the curves of $L_q(t)$, $L_s(t)$ and $L_y(t)$, where for example,

$$L_y \;=\; \frac{1}{\tau}\int_0^{\tau} L_y(t)dt \tag{15.48}$$

Notice that the function $L_y(t)$ can only change at *event times* (the state of the system can only change at these times). The event times are the following:

```
VectorD(0.0, 6.0, 9.0, 11.0, 14.0, 17.0, 18.0, 21.0, 24.0, 29.0,
        34.0, 36.0, 38.0, 41.0, 45.0, 50.0, 55.0, 56.0, 64.0)
```

Event time 0.0 has the start simulation event, times 6.0 and 9.0 have arrival events, time 11.0 has a departure event, times 21.0 and 29.0 have both arrival and departure events, time 64.0 has the last departure event. As there is 1 start simulation event, 8 arrival events, 8 departure events and 2 dual events, there should be a total of 19 event times.

For the above simulation, $L_y(t)$ takes on the values 0, 1 or 2. When $L_y(t) = 0$ the server is idle, $L_y(t) = 1$ there is one customer in service and none waiting, and $L_y(t) = 2$ there is one customer in service and one waiting.

Start with $L_y(0) = 0$, then ":" means no change, + means add 1, - means subtract 1. In this way the value of $L_y(t)$ can be determined for all event times.

```
 0:,  6+,  9+, 11-, 14+, 17-, 18+, 21:, 24-, 29:,
34+, 36-, 38-, 41+, 45-, 50+, 55-, 56+, 64-)
```

Check that the number of arrivals (+) equals the number of departures (-). Tracing through the list, one can deduce the occupancy for the time intervals between the events.

- $L_y(t) = 0$: [0, 6], [38, 41], [45, 50], [55, 56]
- $L_y(t) = 1$: [6, 9], [11, 14], [17, 18], [24, 34], [36, 38], [41, 45], [50, 55], [56, 64]
- $L_y(t) = 2$: [9, 11], [14, 17], [18, 24], [34, 36],

Plot $L_y(t)$ vs. t from 0.0 to 64.0 and use it to determine the area under the curve. The subtotals for each are 15, 36, 13 the integral sums to $0\cdot 15 + 36\cdot 1 + 13\cdot 2 = 62$. Similar calculations yield results for L_q and L_s.

$$\begin{aligned}
L_q &= \Sigma_q/\tau = 13/64 = 0.203 \\
L_s &= \Sigma_s/\tau = 49/64 = 0.766 \\
L_y &= \Sigma_y/\tau = 62/64 = 0.969
\end{aligned}$$

Note, for 15 minutes there were no customers in the system, so the server *idle time* is 15 minutes, while the server *busy time* is 64 - 15 = 49 minutes.

15.7.3 Spreadsheet Simulation

To allow a longer simulation with more customers, essentially the same approach can be carried out using spreadsheet software (*spreadsheet simulation*). To reproduce the above hand simulation, copy the first table into a spreadsheet. This will have the `iarrival-t` and `service-t` columns filled in. The top row (row 0) will contain all zeros. Use spreadsheet formulas for filling in all the rest of columns for row 1. Now use copy-paste to fill in the rest of the m rows. Compute all of the column sums. Below the column sums row, compute all the sample averages (divide by m). Make a row below the sample averages row for time averages (divide by τ). Your spreadsheet should look like the hand simulation table, with one more row for the time averages.

Spreadsheet software provides Random Number Generators (RNGs) such as RAND in Excel. Several Random Variate Generators (RVGs) can easily be produced using the Inverse Transform Method (ITM). Try creating Exponential random variates using ITM and then use these to fill in the pre-filled columns (`iarrival-t` and `service-t`). These column can now be filled in one row at a time. Letting the arrival rate $\lambda = 10$ per hour and the service rate $\mu = 12$ per hour, run the speadsheet simulation for $m = 10$, 20 and 100 customers/entities.

A slightly more automated approach is provided by the ScalaTion's `tableau` package. To handle more complex event logic, the event-scheduling paradigm should be followed. Process-interaction and agent-based simulation are higher-level, more resource intensive alternative ways to develop simulation models.

15.7.4 Exercises

1. Use a spreadsheet to plot $L_q(t)$, $L_s(t)$ and $L_y(t)$ versus time from 0.0 to τ for the M/M/1 Queue. Directly compute the area under the curve and then the average height.

2. Do the same plot for the M/M/1 Queue using `PlotM` from ScalaTion's `mathstat` package.

3. Suppose there are now two severs and the arrival rate $\lambda = 20$ per hour. Redo all the calculations for this M/M/2 Queue.

4. Redo the plots for the M/M/2 Queue simulation.

5. Write code that takes the arrival and departure columns and produces all the event times in time order. Hint: set up an `i`-cursors for the arrival list and `j`-cursor for the departure list, if the values at `i` and `j` are the same, copy that value into the event list and advance both cursors, otherwise copy the smaller value and advance its cursor. Do this in a `while` loop.

6. Recall that $\lambda_e = m/\tau$. For the definitions given in this section for $L_q, L_s, L_y,$ and T_q, T_s, T_y, show the following forms of Little's Law hold.

$$L_q = \lambda_e T_q \tag{15.49}$$

$$L_s = \lambda_e T_s \tag{15.50}$$

$$L_y = \lambda_e T_y \tag{15.51}$$

7. Spreadsheet Simulation: A **Small Fast Food Restaurant** has two severs and enough space for three customers to wait (at most five customers total at any given time). For the case of a single queue, perform a spreadsheet simulation for $m = 20$ customer arrivals. Assume each server can process $\mu = 30$ customers per hour and that the customer arrival rate $\lambda = 75$ customers per hour (assume Exponential distributions). Each completed order gives a net profit (before paying the servers) of 2.00 dollars. Each server makes 11.00 dollars per hour. Should the restaurant hire a third server? Explain in terms of profit (after paying the servers) per hour. Give the simulation table and summary results produced by the spreadsheet.

15.8 Tableau-Oriented Simulation

In tableau-oriented simulation models, each simulation entity's event times are recorded in a row of a matrix/tableau. For example in a Bank simulation, each row would store information about a particular customer, e.g., when they arrived, how long they waited, their service time duration, etc. If 20 customers are simulated, the matrix will have 20 rows (actually 24 since there are always 4 special rows). Average waiting and service times can be easily calculated by summing columns and dividing by the number of customers. This approach is similar to, but not as flexible as Spreadsheet simulation. The complete code for this example may be found in `Ex_Bank.scala` in the `scalation.simulation.tableau` package.

```
@main def runEx_Bank (): Unit =

    val stream     = 0                                    // random number stream (0 to 999)
    val lambda     = 6.0                                  // customer arrival rate (per hour)
    val mu         = 7.5                                  // customer service rate (per hour)
    val maxCusts   = 20                                   // stopping rule: simulate maxCusts

    val iArrivalRV = Exponential (HOUR / lambda, stream)
    val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)

    // Run the simulation of the model Bank.

    val mm1 = new Model ("Bank", maxCusts, Array (iArrivalRV, serviceRV))
    mm1.simulate ()
    mm1.report ()

end runEx_Bank
```

Imports: `scalation.random.Exponential`, `scalation.random.RandomSeeds.N_STREAMS`. From outside ScalaTion also import: `scalation._`, `scalation.simulation.tableau._`.

Note that it is important that the various random variate generators use "different random number streams" to keep them independent. The `stream` number (0 to 999) specifies the combinations of seeds to use for the random number generator. In this model, `iArrivalRV` uses `stream`, while `serviceRV` uses `stream + 1`.

15.8.1 Iterating through Tableau Equations

The logic of the hand simulation is coded into equations for computing the value of each column. The tableau/matrix called `tab` is built up row-by-row.

- The zeroth row is a placeholder for the previous non-existent entity (the values are all zero).
- Rows 1 to m are for the entity timings, i.e., row i records times for the i^{th} entity.
- The last three rows hold the column sums, sample averages and time averages, respectively.

The `simulate` method is used to evaluate the equations, row-by-row. A basic set of equations is provided in the `Model` class that works for a collection of simple, related models. Other models will require the `simulate` method to be overridden.

```
    for i <- 1 to m do tab(i, 0) = i                          // ID-0

    def simulate (startTime: Double = 0.0): Unit =
        for i <- 1 to m do
            tab(i, 1) = rv(0).gen                             // IArrival-1
            tab(i, 2) = tab(i-1, 2) + tab(i, 1)               // Arrival-2
            tab(i, 3) = tab(i, 2) max tab(i-1, 6)             // Begin-3
            tab(i, 4) = tab(i, 3) - tab(i, 2)                 // Wait-4
            tab(i, 5) = rv(1).gen                             // Service-5
            tab(i, 6) = tab(i, 3) + tab(i, 5)                 // Departure-6
            tab(i, 7) = tab(i, 6) - tab(i, 2)                 // Total-7
        end for
    end simulate
```

The columns are the same as those given in the Hand Simulation section.

15.8.2 Reproducing the Hand Simulation

The example from the hand simulation example is replicated by the following `tableau` model.

```
@main def runQueue_MM1 (): Unit =

    val iArrivalArr = Array [Double] (6, 3, 5, 4, 3, 8, 5, 7, 9, 6)
    val serviceArr  = Array [Double] (5, 6, 4, 3, 5, 7, 2, 4, 5, 8)

    val maxCusts   = 10                                       // stopping rule: at maxCusts
    val iArrivalRV = Known (iArrivalArr)                      // inter-arrival time random variate
    val serviceRV  = Known (serviceArr)                       // service time random variate

    // Run the simulation of the model Queue_MM1.

    val mm1 = new Model ("Queue_MM1", maxCusts, Array (iArrivalRV, serviceRV))
    mm1.simulate ()
    mm1.report ()                                             // show the table/matrix
    mm1.summary ()                                            // show summary performance statistics

end runQueue_MM1
```

The *Known* Random Variate Generator (RVG) simply repeats the given sequence of numbers.

15.8.3 Customized Logic/Equations

For models where the event logic is different, as mentioned, the `simulate` method will need to **overridden**, For example, in a `Ex_CallCenter` model with a single server (tele-service representative) and no call-waiting, the logic now requires an `if` statement to check if the phone line is busy. It requires the departure time (call hang up) of the last completed call (call `l`) to be less than or equal to the arrival time of the current call (call `i`).

```
override def simulate (startTime: Double): Unit =
    var l = 0                                            // last established call
    for i <- 1 to m do
        tab(i, 1)  = rv(0).gen                           // IArrival-1
        tab(i, 2)  = tab(i-1, 2) + tab(i, 1)             // Arrival-2
        if tab(l, 6) <= tab(i, 2) then                   // call established
            tab(i, 3) = tab(i, 2); l = i                 // Begin-3
            tab(i, 4) = tab(i, 3) - tab(i, 2)            // Wait-4
            tab(i, 5) = rv(1).gen                        // Service-5
            tab(i, 6) = tab(i, 3) + tab(i, 5)            // Departure-6
            tab(i, 7) = tab(i, 6) - tab(i, 2)            // Total-7
        end if
    end for
end simulate
```

Model developers may wish to copy the base `simulate` method code from `Model` and make minimal modifications to the equations. The only changes above are the introduce of the variable `l`, the addition of the `if` statement, and modification to the `Begin-3` equation.

15.8.4 Tableau.scala

The `Model` class support tableau-oriented simulation models in which each simulation entity's events are recorded in tabular form (in a matrix). This is analogous to Spreadsheet Simulation (http://www.informs-sim.org/wsc06papers/002.pdf).

Class Methods:

```
@param name    the name of simulation model
@param m       the number entities to process before stopping
@param rv      the random variate generators to use
@param label_  the column labels for the matrix

class Model (name: String, m: Int, rv: Array [Variate], label_ : Array [String])
      extends Modelable:

def simulate (startTime: Double = 0.0): Unit =
def report (): Unit =
def summary (): Unit =
def timeLine (): (VectorD, VectorD) =
def save (): Unit =
```

The `report` method displays the `tab` matrix and should be called after `simulate`. The last two rows display useful averages such as average inter-arrival, waiting, service and system times. The `summary` method displays averages for lengths of queues and waiting times, etc. It summarizes the number and time in the queue (q), service (s) and system (y), i.e., `L_q, L_s, L_y, T_q, T_s, T_y`. The `Model.occupancy` (`mm1.timeLine ()`) line gives the event times and corresponding values for $L_y(t)$. The `save` method saves the matrix in a `.csv` file that may be loaded into a spreadsheet for further processing.

The `report` method for the hand simulation problem outputs the following table where the last three rows show the sums, sample averages and time averages.

```
-----------------------------------------------------------------------------------------
     ID-0 IArrival-1  Arrival-2    Begin-3     Wait-4  Service-5 Departure-6    Total-7
-----------------------------------------------------------------------------------------
    0.000      0.000      0.000      0.000      0.000      0.000      0.000      0.000
-----------------------------------------------------------------------------------------
    1.000      6.000      6.000      6.000      0.000      5.000     11.000      5.000
    2.000      3.000      9.000     11.000      2.000      6.000     17.000      8.000
    3.000      5.000     14.000     17.000      3.000      4.000     21.000      7.000
    4.000      4.000     18.000     21.000      3.000      3.000     24.000      6.000
    5.000      3.000     21.000     24.000      3.000      5.000     29.000      8.000
    6.000      8.000     29.000     29.000      0.000      7.000     36.000      7.000
    7.000      5.000     34.000     36.000      2.000      2.000     38.000      4.000
    8.000      7.000     41.000     41.000      0.000      4.000     45.000      4.000
    9.000      9.000     50.000     50.000      0.000      5.000     55.000      5.000
   10.000      6.000     56.000     56.000      0.000      8.000     64.000      8.000
-----------------------------------------------------------------------------------------
   55.000     56.000    278.000    291.000     13.000     49.000    340.000     62.000
    5.500      5.600     27.800     29.100      1.300      4.900     34.000      6.200
    0.859      0.875      4.344      4.547      0.203      0.766      5.313      0.969
-----------------------------------------------------------------------------------------
     ID-0 IArrival-1  Arrival-2    Begin-3     Wait-4  Service-5 Departure-6    Total-7
-----------------------------------------------------------------------------------------
```

15.8.5 Exercises

1. Run the bank simulation for many more customers and see if the averages begin to stablize.

2. Suppose there are now two severs and the arrival rate $\lambda = 20$ per hour. Override the `simulate` method for this M/M/2 Queue. The logic will need to handle the fact that there are now two statistically identical servers. What do the new results indicate?

3. Now suppose the second server works 20% faster than the first server. What do the new results indicate?

4. Rework the previous exercise using a spreadsheet.

5. An M/M/1/1 Queue has one server and a system capacity of one (no space for waiting). Develop and run a Tableau simulation for $\lambda = 10$ per hour and $\mu = 12$ per hour. Redo for an M/M/2/2 Queue and $\lambda = 20$ per hour.

Chapter 16

State Space Models

In dynamic models, the state of a system may be described by a state vector $\mathbf{x}(t)$. For example, a particle may be tracked over time. In two-dimensional space, one might be interested in the height and down range distance of the particle over time,

$$\mathbf{x}(t) \;=\; [x_0(t), x_1(t)] \tag{16.1}$$

where $x_0(t)$ is the height of the particle and $x_1(t)$ is the horizontal distance travelled at time t.

The dynamics of the particle may be described by a n-dimensional vector-valued function of time.

$$\mathbf{f} : \mathbb{R}^+ \to \mathbb{R}^n \tag{16.2}$$

Such a function may be developed using physical laws that are expressed as a system of Ordinary Differential Equations (ODEs) consisting of first-order time derivatives. When the system is linear, the time derivative $\dot{\mathbf{x}}(t)$ equals an affine transformation of the current state $\mathbf{x}(t)$, i.e., the sum of a linear transformation of the current state and a constant vector.

$$\dot{\mathbf{x}}(t) \;=\; F\mathbf{x}(t) + \mathbf{c} \tag{16.3}$$

where the dot signifies a time derivative ($\frac{d}{dt}$), and $F \in \mathbb{R}^{n\times n}$ is a matrix of coefficients and $\mathbf{c} \in \mathbb{R}^n$ is a vector of constants.

16.1 Example: Trajectory of a Ball in One-Dimensional Space

Consider the application of Newton's Laws of Motions to determine the trajectory of a ball in one dimension. Suppose someone hits a golf ball with a driver straight up in the air and wishes to know how high it will go and how long it will be in the air. Let $\mathbf{x}(t) = [y(t), v(t)]$ be the height of the ball $y(t)$ and its velocity $v(t)$ at time t. Also, let the initial conditions be $[0\,\text{m}, 60\,\text{m/s}]$, corresponding to almost hitting the ball from the ground with initial upward velocity of 60 mps (approximately 134.2 mph).

16.1.1 Ordinary Differential Equations

Newton's Second Law of Motion (force = mass times acceleration), $F = ma$) and the Law of Gravity (force = mass times minus the gravitational constant, $F_G = -mg$) yield a system of Ordinary Differential Equations (ODEs),

$$\begin{aligned} \dot{y}(t) &= v(t) && \text{time derivative of position} \\ \dot{v}(t) &= -g && \text{time derivative of velocity} \end{aligned}$$

where the gravity of Earth $g = 9.807\,\text{m/s}^2$ and $\dot{v}(t) = a$ (constant acceleration).

The system of differential equations may be written using vector and matrix notation, with all the columns being treated as column vectors.

$$\begin{bmatrix} \dot{y}(t) \\ \dot{v}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ -g \end{bmatrix}$$

This system of differential equations may be solved by integration. Integrating both sides of the second equation/row produces,

$$v(t) - v(0) = \int_0^t -g\, d\tau = -gt$$

Thus, $\dot{y}(t) = v(0) - gt$, so

$$\begin{aligned} y(t) &= y(0) + v(0)t - \frac{1}{2}gt^2 \\ v(t) &= v(0) - gt \end{aligned}$$

Substituting in the initial conditions, $\mathbf{x}(0) = [y(0), v(0)] = [0\,\text{m}, 60\,\text{m/s}]$, gives

$$\begin{aligned} y(t) &= -\frac{1}{2}gt^2 + 60t \\ v(t) &= -gt + 60 \end{aligned}$$

16.1.2 Discretization

Typically, it is not so easy to solve a system of ODEs, so iterative algorithms (called integrators, e.g., Euler, Verlet, Runge-Kutta, Dormand-Prince) are used to provide approximate solutions. Using the Verlet Method [198, 67, 165] with a time gap of Δt, the ODEs can discretized to the following form,

$$\begin{aligned} y(t) &= y(t-\Delta t) + v(t-\Delta t)\Delta t - \frac{1}{2}g(\Delta t)^2 \\ v(t) &= v(t-\Delta t) - g\Delta t \end{aligned}$$

where the current value is computed from the previous value. This also allows a continuous-time system to treated as a discrete-time system. Here to maintain notational compatibility with the notation previously used for time series analysis, t in $\mathbf{x}_t$ has two interpretation: (1) time index and (2) the actual discrete time/timestamp. Therefore, the above equation can written as follows.

$$\begin{aligned} y_t &= y_{t-1} + v_{t-1}\Delta t - \frac{1}{2}g(\Delta t)^2 \\ v_t &= v_{t-1} - g\Delta t \end{aligned}$$

16.1.3 Trajectory Simulation

The trajectory of the ball can be simulated using ScalaTion's `DynamicEq` class in the `scalation.dynamics` package.

```
val g   = 9.807
val mps = 60.0
def f (t: Double): VectorD = VectorD (-0.5 * g * t~^2 + mps * t, -g * t + mps)

val dyn = new DynamicEq (f)
val n   = 100
val te  = 12.0
val t   = VectorD.range (0, n) * (te / n.toDouble)
val trj = dyn.trajectory (0.0, te, n)
new Plot (t, trj(?, 0), trj(?, 1), "Plot (x_t0, x_t1) vs. t")
```

To determine the maximum height, simply set the velocity to zero ($x_1(t) = 0$), solve for the time τ when velocity becomes zero, and calculate the height $x_0(t)$ at time τ.

$$\begin{aligned} \tau &= 60/g && = 6.118 \text{ s} \\ x_0(t) &= -0.5 * g * 6.118^2 + 60 * 6.118 && = 183.5 \text{ m} \end{aligned}$$

Can a golf ball hit with the swing speed of an average golfer really go so high? To seek the answer a theorist and an experimentalist may be consulted. The theorist explains that the model ignores other forces (e.g., drag due to air resistance) and therefore, the state equation needs to have an error term. The experimentalist explains that measurements (e.g., using RADAR/LIDAR) need to be taken and of course there will be measurement errors (another error term). These errors can be modeled as noise and, as such, turn the deterministic state vector into a random one $\mathbf{x}(t)$. Furthermore, since measurements are not made continuously, discrete time is introduced. The measurements may be recorded as time series data $\mathbf{y}_t$.

Now there are two stochastic processes: $\{\mathbf{x}(t) | t \in [0, t_e]\}$, the actual process, and $\{\mathbf{y}_t | t \in \{0, t_e\}\}$ the observed process. Simplicity argues for merging the two stochastic processes. Unfortunately, it is commonly the case that the actual process is only partially observable (some of the state variables are not directly measurable). Therefore, merging the two may result in less accurate and/or less explainable models.

Dynamic models consisting of state equations and observation/measurement equations come in several varieties:

Table 16.1: Types State-Observation Models

state	**state time**	**model type**
continuous	continuous	Kalman-Bucy Filter
continuous	discrete	Kalman Filter
discrete	continuous	CT Hidden Markov Chain
discrete	discrete	Hidden Markov Model

Note: Kalman-Bucy Filter has continuous-time for the state and discrete time for the observations. Similarly, CT Hidden Markov Model has continuous-time for the state and discrete time for the observations. The next section discusses Dynamic Linear Models as Kalman Filters where the forcing/control vector is missing. Kalman filters are models that are particularly useful for dealing with/filtering out noise.

16.1.4 Exercises

1. The diameter and mass of modern golf balls are approximately 4.268 cm and 45.93 grams, respectively. Combine Newton's Second Law of Motion ($F = ma$) and the Law of Gravity ($F_G = -mg$) to deduce the following equation:

$$\dot{v}(t) \;=\; -\,g$$

2. Consider the effect of wind/air resistance (drag) as another force, drag force F_D (in addition to gravity F_G)

$$F_D \;=\; \frac{\rho\, C_D A}{2} v(t)^2$$

where ρ = density of air (1.225 kg/m^3), C_D = drag coefficient (0.4), and A = cross sectional area of the golf ball (14.3 cm^2).

Recompute the time in the air and maximum height considering the effects of both F_G and F_D.

16.2 Markov Chains

A Markov Chain [173] is a simple type of Markov Model where one is interested in tracking the state of a system over time. Consider the following Markov Chain with $n = 6$ states, corresponding to the number of dollars one currently has. At each discrete time point, flip a coin, heads (with probability p) gives a dollar, while tails (with probability $q = 1 - p$) takes a dollar. There are two terminal states, 0 (lose) and 5 (win). One pays $x_0 \in \{1, 2, 3, 4\}$ dollars to start the game. The Markov Chain that models this game is shown in Figure 16.1.

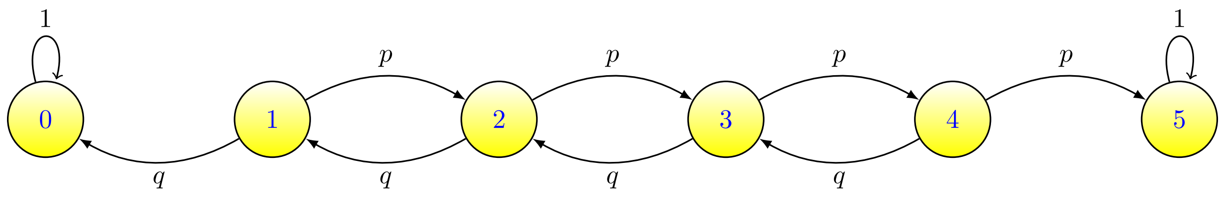


Figure 16.1: State Transition Diagram for Six-State Markov Chain

Consider a discrete-valued, discrete-time stochastic processes that represents the state of a system over time t.

$$\{x_t : t \in \{0, 1, \dots\}\} \qquad \text{where} \quad x_t \in S \tag{16.4}$$

The state space S is discrete (finite with n states or countable inifinite).

Since x_t is a stochastic process, its trajectory needs to be described probabilistically. For tractability and because it often suffices, the assumption is made that the state x_t is only significantly influenced by its previous state x_{t-1}.

$$P(x_t | x_{t-1}, x_{t-2}, \dots x_0) \;=\; P(x_t | x_{t-1}) \tag{16.5}$$

This is the *Markov Property*: the transitions from state to state are governed by a *discrete-time Markov chain* and characterized by a *state-transition probability matrix* $A = [a_{ij}] \in [0, 1]^{n \times n}$, where

$$a_{ij} \;=\; P(x_t = j | x_{t-1} = i) \tag{16.6}$$

The transition from state i to state j is depicted in Figure 16.2.

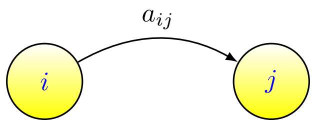


Figure 16.2: Transition from State i to State j

The Markov Property can be generally stated as the future given the present is conditionally independent of the past.

16.2.1 Probability Mass Function

The probability mass function (pmf) for the state at time t as a vector may be given as follows:

$$\boldsymbol{\pi}_t \;=\; [P(x_t = 0), P(x_t = 1), \dots P(x_t = n-1)] \tag{16.7}$$

Due to the Markov Property, the next state vector (as a row vector) may be computed by vector-matrix multiplication.

$$\boxed{\boldsymbol{\pi}_t \;=\; \boldsymbol{\pi}_{t-1} A} \tag{16.8}$$

Suppose $p = .6$ and the initial state is 3 (3 dollars to enter the game). Probabilistically the initial state is given by $\boldsymbol{\pi}_0 = [0, 0, 0, 1, 0, 0]$, so the RHS of the above equation is

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ .4 & 0 & .6 & 0 & 0 & 0 \\ 0 & .4 & 0 & .6 & 0 & 0 \\ 0 & 0 & .4 & 0 & .6 & 0 \\ 0 & 0 & 0 & .4 & 0 & .6 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Then probabilistically the next state is $\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 A = [0, 0, .4, 0, .6, 0]$. The dot product of the row vector $\boldsymbol{\pi}_0$ with each column of A is used to compute $\boldsymbol{\pi}_1$.

In ScalaTion, advancing to the next state is carried out by the `next` method.

```
def next (pi: VectorD): VectorD = pi *: a
```

where the right-associative `*:` operator is for vector-matrix multiplication (and complements the left associate `*` operator for matrix-vector multiplication).

This recurrence can be unfolded to yield the following equation,

$$\begin{aligned} \boldsymbol{\pi}_t &= \boldsymbol{\pi}_{t-1} A \\ &= \boldsymbol{\pi}_{t-2} A^2 \\ &= \boldsymbol{\pi}_0 A^t \end{aligned}$$

i.e., $\boldsymbol{\pi}_0$ times A to the t^{th} power.

The only choice in optimizing one's game is to determine the number of dollars to spend to start the game to, for example, maximize expected earnings, depending on the weight (value of p) of the coin. If $p = 1$, always start with one dollar, but if $p = 0$, do not play the game. In the first case, the expected earning are 4 dollars per game, while in the second playing the game will ensure some level of loss, so 0 dollars of expected earnings is optimal.

For other values of p, a simple way to estimate the expected earnings is to simulate playing the game many times for a given start state, and determining the average earnings. See `MarkovChainTest2` and `MarkovChainTest3` in the `scalation.simulation.state` package for two ways to estimate the earnings.

16.2.2 Reducible Markov Chains

State j is *reachable* from state i if

$$a_{ij}^{(t)} > 0 \tag{16.9}$$

for some discrete time t, where $a_{ij}^{(t)}$ is the i, j element in A^t.

Based on *reachability* a Markov Chain may decomposed in into multiple subschains, where all the states in each subchain are reachable from each other. If states i and j are mutually reachable, they are said to *communicate*. Since communication forms an equivalence class (reflexive, symmetric and transitive), each subchain is a equivalence class. The six-state Markov Chain shown in the figure has three equivalence classes.

1. $S_1 \;=\; \{0\}$ Lost
2. $S_1 \;=\; \{1, 2, 3, 4\}$ Still Playing the Game
3. $S_1 \;=\; \{5\}$ Won

States 0 and 5 are absorbing states, since once in such a state, it will never be left. A Markov Chain that has just one communication/equivalence class is called an *Irreducible* Markov Chain.

A state i if said the *recurrent* if the probability of returning some time in the future is one, otherwise it said to be *transient* (may never return). A recurrent state is either positive (finite return time) or null recurrent (infinite return time).

16.2.3 Limiting/Steady-State Distribution

For an Irreducible, Positive Recurrent Markov Chain, $\boldsymbol{\pi}_t$ may converge to a limiting probability distribution,

$$\boldsymbol{\pi} \;=\; \lim_{t \to \infty} \boldsymbol{\pi}_t \tag{16.10}$$

After convergence, $\boldsymbol{\pi}$ may be substituted for both $\boldsymbol{\pi}_t$ and $\boldsymbol{\pi}_{t-1}$, so the previous boxed equations becomes,

$$\boxed{\boldsymbol{\pi} \;=\; \boldsymbol{\pi} A} \tag{16.11}$$

where $\boldsymbol{\pi}$ is the probability vector (non-negative and sums to 1) and A is the transition probability matrix. When this equation has a solution, $\boldsymbol{\pi}$ is the limiting/steady-state probability vector.

Interpretation

If the Markov Chain is

- *aperiodic*, $\boldsymbol{\pi}$ can be viewed as a long-term probability for t large enough, otherwise, if it is
- *periodic*, $\boldsymbol{\pi}$ is interpreted as as an average over time.

For example, a Markov Chain that oscillates between two states, the long-term probabilities will depend on the initial conditions.

Solving for the Limiting Probabilities

Subtracting $\boldsymbol{\pi}$ from the above boxed equation ($\boldsymbol{\pi} = \boldsymbol{\pi}A$) yields,

$$\boldsymbol{\pi}A - \boldsymbol{\pi} \; = \; \mathbf{0} \tag{16.12}$$

with $\mathbf{0}$ as a row vector. Using an identity matrix I, this may be rewritten,

$$\boldsymbol{\pi}(A - I) \; = \; \mathbf{0} \tag{16.13}$$

Taking the transpose produces

$$(A - I)^{\intercal} \boldsymbol{\pi}^{\intercal} \; = \; \mathbf{0}^{\intercal} \tag{16.14}$$

The vector $\boldsymbol{\pi}$ is the eigenvector solution to the left eigenvalue problem for eigenvalue $\lambda = 1$ (see the chapter on Linear Algebra). This can be solved by computing the nullspace of $(A - I)^{\intercal}$. In ScalaTion, the limiting distribution can be found using QR Factorization (see the exercises).

```
def limit: VectorD =
    val fac = new Fac_QR ((a - eye (a.dim, a.dim)).$\mathcal{T}$, true)
    fac.nullspace (a.dim-1)(?, 0).toProbability
end limit
```

The solution for $\boldsymbol{\pi}$ also may be found by first computing $(A - I)^{\intercal}$ and then solving for $\boldsymbol{\pi}$ using for example LU factorization. One of the equations will turn out to be redundant and can be replaced with the normalization equation (probabilities sum to 1). The example below illustrates three ways to solve for $\boldsymbol{\pi}$.

Example: Computing the Limiting Probabilities

Consider the example problem from *Introduction to Probability Models*, 3rd Ed., Ross, p. 146 [160, 161] having the following transition probability matrix A. Solve for the stationary (steady-state) distribution three ways:

1. Start with state probability vector $\boldsymbol{\pi} = [\pi_0, \pi_1, \pi_2] = [.5, .5, 0]$ and repeatedly compute $\boldsymbol{\pi}A$.

2. Solve the eigenvector problem $\boldsymbol{\pi} = \boldsymbol{\pi}A$, i.e.,

$$[\pi_0, \pi_1 \pi_2] \; = \; [\pi_0, \pi_1 \pi_2] \begin{bmatrix} .5 & .4 & .1 \\ .3 & .4 & .3 \\ .2 & .3 & .5 \end{bmatrix}$$

Since the A matrix is stochastic, one of the equations is redundant and may be replaced with the normalization equation $\|\boldsymbol{\pi}\|_1 = 1$.

$$\begin{aligned} \pi_0 &= .5\pi_0 + .3\pi_1 + .2\pi_2 \\ \pi_1 &= .4\pi_0 + .4\pi_1 + .3\pi_2 \\ \pi_2 &= 1 - \pi_0 - \pi_1 \end{aligned}$$

3. This may be rewritten as an augmented matrix and solved using LU Factorization.

$$\left[\begin{array}{ccc|c} -.5 & .3 & .2 & 0 \\ .4 & -.6 & .3 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right]$$

   Note, the matrix above is simply $(A - I)^{\intercal}$ with the last row replaced with 1s from the normalization equation.

Software Solution

The following ScalaTion code uses all three solution techniques.

```
/** The ‘markovChainTest4‘ function tests the ‘MarkovChain‘ class.
 *  @see Introduction to Probability Models, 3rd Ed., Ross, p. 146.
 *  > runMain scalation.simulation.state.markovChainTest4
 */
@main def markovChainTest4 (): Unit =

    val a  = MatrixD ((3, 3), .5, .4, .1,         // 3-by-3 matrix
                              .3, .4, .3,
                              .2, .3, .5)

    val mc = new MarkovChain (a)
    println ("Discrete-Time Markov Chain mc = " + mc + "\n")
    banner ("Discrete-Time Markov Chain: transient solution:")

    var pi  = VectorD (.5, .5, 0)
    println ("on epoch 0,\tpi = " + pi)
    for k <- 1 to 10 do
        pi = mc.next (pi)
        println (s"on epoch $k,\tpi = $pi")
    end for

    banner ("eigenvector solution for steady-state \tpi = " + mc.limit)

    val aa = MatrixD ((3, 3), -.5,  .3, .2,
                               .4, -.6, .3,
                               1,   1,  1 )
    val b  = VectorD (0, 0, 1)
    val lu = new Fac_LU (aa).factor ()
    banner ("lu factorization for steady-state \tpi = " + lu.solve (b))

end markovChainTest4
```

The `MarkovChain` class in the `scalation.simulation.state` package provides both transient (via the `next` method) and steady-state (via the `limit` method) solutions.

16.2.4 `MarkovChain` Class

Class Methods:

```
@param a  the transition probability matrix

class MarkovChain (a: MatrixD):

def next (pi: VectorD): VectorD = pi *: a
def next (pi: VectorD, k: Int): VectorD =
def limit: VectorD =
def simulate (i0: Int, endTime: Int): Unit =
def animate (): Unit =
def isStochastic: Boolean =
override def toString: String = s"MarkoveChain($a)"
```

16.2.5 Continuous-Time Markov Chains

Continuous-Time Markov Chains are analogous to their discrete-time counterparts, except that transitions can occur at any time. This makes the notion of transition probability hard to define, so it is replaced with the notion of *transition rate*. Therefore, the transition probability matrix A (denoted P in some textbooks) is replaced with a transition rate matrix $Q = [q_{ij}] \in (\mathbb{R}^+)^{n \times n}$.

Consider a Markov model for a single-server queue with room for four customers. Arrivals are assumed to come at a rate of λ customers per unit time and can be serviced at a rate of μ customers per unit time. If the system is empty, the server is idle, while if it is full of customers, the new arrival is turned away. This can be modeled as an $n =$ five state Continuous-Time Markov Chain, as shown in Figure 16.3.

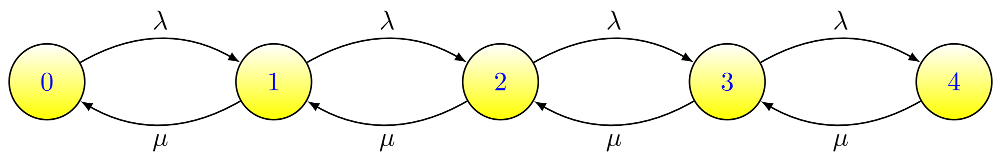


Figure 16.3: State Transition Diagram for Five-State Continuous-Time Markov Chain

Self-loops are not included, since the state is unchanged until there is an out transition. One may view this as an event that changes the state (here there are arrival and service completion events).

The transient solution for Continuous-Time Markov Chains can be found by solving Kolmogorov differential equations, see Ross, Chapter 6 [160].

16.2.6 Limiting/Steady-State Distribution

The limiting/steady-state solution can be given by,

$$\boldsymbol{\pi} Q = \mathbf{0} \tag{16.15}$$

See `https://mast.queensu.ca/~stat455/lecturenotes/set5.pdf` for a derivation (where Q is called the generator matrix G).

When the system (single-server queue) opens, there will be no customers, but they will arrive at rate λ until the system closes. The server can process customers at rate of μ per unit time. The stationary/steady-state solution can be found by solving the following equations.

$$\begin{bmatrix} \pi_0 & \pi_1 & \pi_2 & \pi_3 & \pi_4 \end{bmatrix} \begin{bmatrix} -\lambda & \lambda & 0 & 0 & 0 \\ \mu & -(\lambda+\mu) & \lambda & 0 & 0 \\ 0 & \mu & -(\lambda+\mu) & \lambda & 0 \\ 0 & 0 & \mu & -(\lambda+\mu) & \lambda \\ 0 & 0 & 0 & \mu & -\mu \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The diagonal elements are set so that the rows of matrix Q sum to 0 (inflow = output).

Total Balance Equations

Multiplying the vector $\boldsymbol{\pi}$ by the matrix Q produces the following five equations.

$$\begin{aligned} -\lambda\pi_0 + \mu\pi_1 &= 0 \\ \lambda\pi_{j-1} - (\lambda+\mu)\pi_j + \mu\pi_{j+1} &= 0 \qquad \text{for } \ j = 1,2,3 \\ \lambda\pi_3 - \mu\pi_4 &= 0 \end{aligned}$$

These equations are referred to as the total balance equations.

Partial Balance Equations

In this case a simpler approach is possible, as the solution can be developed using partial balance equations that equates up-flow with down-flow, so

$$\mu\pi_j = \lambda\pi_{j-1} \qquad \text{for } \ j = 0,1,2,3,4 \tag{16.16}$$

Therefore, $\pi_1 = \frac{\lambda}{\mu}\pi_0$, $\pi_2 = \frac{\lambda}{\mu}\pi_1$, $\pi_3 = \frac{\lambda}{\mu}\pi_2$, and $\pi_4 = \frac{\lambda}{\mu}\pi_3$.

Traffic Intensity: The traffic intensity is the ratio of the arrival rate to the service rate. The higher the traffic intensity, the more the chain is pushed to the right (toward more congestion).

$$\rho = \frac{\lambda}{\mu} \tag{16.17}$$

The partial balance equations can be expressed with ρ replacing λ and μ.

$$\pi_j = \rho\pi_{j-1} \qquad \text{for } \ j = 0, \ldots 4 \tag{16.18}$$

This recursive equation can be unfolded to give,

$$\pi_j \;=\; \rho^j \pi_0 \tag{16.19}$$

Furthermore, the probabilities must add to one (normalization equation), so

$$[1 + \rho + \rho^2 + \rho^3 + \rho^4]\pi_0 \;=\; 1 \tag{16.20}$$

Solving for π_0 (see the exercises) yields

$$\pi_0 \;=\; \frac{1-\rho}{1-\rho^n} \tag{16.21}$$

Finally, the state-probabilities may be determined.

$$\pi_j \;=\; \frac{1-\rho}{1-\rho^n}\,\rho^j \tag{16.22}$$

This is the solution for an M/M/1/K Queue (with $K = 4$ and $n = K + 1 = 5$). The notation M/M/1/K means the arrival process is Markovian (Poisson or Exponential inter-arrival times), the service distribution is Exponential, the number of servers is 1 and customer capacity is K (one in service and the rest waiting).

The `MarkovChainCT` class in the `scalation.simulation.state` package provides both transient (via the `next` method) and steady-state (via the `limit` method) solutions. Currently, the transient solution has not been implemented (hence `= ???`).

16.2.7 `MarkovChainCT` Class

Class Methods:

```
@param tr  the transition rate matrix

class MarkovCT (tr: MatrixD):

def next (p: VectorD, t: Double = 1.0): VectorD = ???
def limit: VectorD =
def simulate (i0: Int, endTime: Double): Unit =
def animate (): Unit =
override def toString: String = s"MarkovCT($tr)"
```

16.2.8 Queueing Models

The solution for an M/M/1/K Queue my be written

$$\boxed{\pi_j \;=\; \frac{1-\rho}{1-\rho^{K+1}}\,\rho^j} \tag{16.23}$$

This solution works when $\rho = 0$ ($\pi_0 = 1$), $\rho \in (0, 1)$, $\rho = 1$ (via L'Hospital's Rule) and when $\rho > 1$.

As the waiting capacity K (and therefore the number of states n) goes to infinity, stability requires traffic intensity $\rho < 1$. In which case ρ^{K+1} goes to zero. Therefore, the solution for an M/M/1 Queue can be obtained.

$$\boxed{\pi_j \;=\; (1-\rho)\rho^j} \tag{16.24}$$

Note that $\pi_0 = 1 - \rho$ is the probability the server is *idle*.

Expected Number in the System

Given that $\pi_j = P(x = j)$ indicates the probability there are j customers in the system, the expected number in the system is given as follows.

$$\mathbb{E}[x] \;=\; \sum_{j=0}^{\infty} j\,\pi_j \;=\; (1-\rho)\sum_{j=0}^{\infty} j\rho^j \tag{16.25}$$

As indicated in the exercises, the result becomes,

$$L \;=\; \mathbb{E}[x] \;=\; \frac{\rho}{1-\rho} \tag{16.26}$$

The number in service L_s corresponds to probability the server is $busy = 1 - \pi_0 = \rho$. Therefore, the expected length of the queue is

$$L_q \;=\; L - L_s \;=\; \frac{\rho}{1-\rho} - \rho \;=\; \frac{\rho^2}{1-\rho} \tag{16.27}$$

Application of Little's Law

The expected time in service $T_s = \frac{1}{\mu}$ is one over the service rate, so

$$L_s \;=\; \lambda T_s \;=\; \frac{\lambda}{\mu} \;=\; \rho \tag{16.28}$$

This relationship between length (number in) and time carries over to the queue

$$L_q \;=\; \lambda T_q \tag{16.29}$$

and the system.

$$L \;=\; \lambda T \tag{16.30}$$

Summary of Results

In summary, the formulas for an M/M/1 Queue are collected into Table 16.2.
The `MMc_Queue` class in the `scalation.simulation.queueingnet` package produces steady-state solutions for M/M/1 and M/M/c queues, where c is the number of servers.

Table 16.2: Formulas for M/M/1 Queueing Models

part	**length**	**formula**	**time**	**formula**
Queue	$L_q =$	$\frac{\rho^2}{1-\rho}$	$T_q =$	$\frac{\rho/\mu}{1-\rho}$
Service	$L_s =$	ρ	$T_s =$	$\frac{1}{\mu}$
System	$L =$	$\frac{\rho}{1-\rho}$	$T =$	$\frac{1/\mu}{1-\rho}$

16.2.9 `MMc_Queue` Class

Class Methods:

```
@param lambda  the arrival rate
@param mu      the service rate
@param c       the number of servers

class MMc_Queue (lambda: Double, mu: Double, c: Int = 1):

def prob_0: Double =
def t_wait: Double = (prob_0 * rho * rhoc / _1_a~^2) / lambda
def view (): Unit =
def report (): Unit =
```

The `MMcK_Queue` class produces steady-state solutions for M/M/1/K and M/M/c/K queues, where c is the number of servers and K is the system capacity.

16.2.10 `MMcK_Queue` Class

Class Methods:

```
@param lambda  the arrival rate
@param mu      the service rate
@param c       the number of servers
@param k       the capacity of the queue

class MMck_Queue (lambda: Double, mu: Double, c: Int = 1, k: Int = 1):

def prob_0: Double =
def prob_k: Double = pr_0 * rho~^k / (c~^k_c * c_fac)
def view (): Unit =
def report (): Unit =
```

16.2.11 Exercises

1. For the given six-state discrete-time Markov Chain, advance the state probability $\boldsymbol{\pi}_t$ over the next 20 time points.

2. For the six-state Markov Chain, what happens to the probabilities of states 1 to 4 as time increases.

3. Consider the middle subchain, states 1 to 4. Do they form a irreducible Markov Chain and admit a stationary/steady state solution? Compute the value for this solution $\boldsymbol{\pi}$.

4. A square matrix is stochastic if all its elements are non-negative and all the columns sums equal 1. Show that a stochastic matrix has an eigenvalue equal to 1. Hint: see `https://textbooks.math.gatech.edu/ila/stochastic-matrices.html`

5. For discrete-time Markov chains, explain how the `limit` method works for computing $\boldsymbol{\pi}$.

6. Show the following formula holds.

$$s_n = \sum_{j=n}^{\infty} \rho^j \;=\; \frac{\rho^n}{1-\rho}$$

   Use this to deduce that

$$[1+\rho+\rho^2+\rho^3+\rho^4]\pi_0 \;=\; [s_0 - s_5]\pi_0 \;=\; 1$$

   and finally that

$$\pi_0 \;=\; \frac{1-\rho}{1-\rho^5}$$

7. Develop the steady-state solution π_j for an M/M/c/K Queue where c is the number of servers and K is the system capacity.

8. Derive the formula for the expected number in the system L for an M/M/1 Queue. Hint: $\frac{d}{d\rho}\rho^j = j\rho^{j-1}$.

9. The relationship $L = \lambda T$ is called *Little's Law.* Sketch Stidham's proof of the law. See [174]

10. Use the `MMc_Queue` class to address the one line versus two line question. Let $\lambda = 20$ and $\mu = 12$ per hour. What is the mean time in the queue T_q for an M/M/2 queue? Compare this with T_q for two M/M/1 queues, where customers are randomly split between the two lines/servers, i.e., the arrival rate to each is $\lambda/2$. Note, if customers join the shorter line, the analysis of this problem becomes difficult, but simulation is still straightforward.

11. In the above problem, let λ take on all integer values from 4 to 23 and plot T_q (the mean waiting time) over these values for both the one line and two line solutions. Note, for $\lambda = 24$ or higher the queues will be unstable.

12. Explain what the Kolmogorov backward equations are and how they can be used to solve for transient solutions to Continuous-Time Markov Chains [172].

13. One simple way to model a epidemic such as the COVID-19 Pandemic is to use a Discrete-Time Markov Chain (DTMC). One could start with an SEIR compartmental model and relate subpopulations of individuals to probabilities of being in a given state. Consider the discrete-time Markov Chain model shown in Figure 16.4.

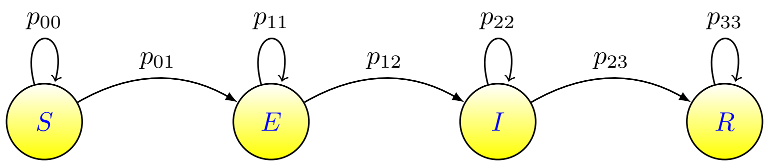


Figure 16.4: State Transition Diagram for SEIR Markov Chain

Assume the population of the state Georgia that is susceptible to COVID-19 is $N = 10,000,000$. The basic SEIR model assumes there are four subpopulations of individuals.

$$\begin{aligned}
S &= N\pi_0 && \text{Susceptible} \\
E &= N\pi_1 && \text{Exposed} \\
I &= N\pi_2 && \text{Infected} \\
R &= N\pi_3 && \text{Recovered}
\end{aligned}$$

Further assume that on average it takes 20 days to transition from state S to state E, 8 days from E to I, and 10 days from I to R. Let the transition probabilities correspond to the reciprocals of the days. Remember the probabilities in each row of the transition probability matrix must add to 1. The discrete time unit is one day. Each day, an individual may transition to the next state (e.g., S to E) or remain in the same state (e.g., S to S). Again the probabilities must add to one.

(a) Construct the transition probability matrix A.

(b) Let the initial probability vector $\boldsymbol{\pi}_0 = [0.99, 0.0, 0.01, 0.0]$, i.e., 99% in state S and 1% in state I. Compute $\boldsymbol{\pi}_t$ for the next two weeks (14 days). Show $\boldsymbol{\pi}_t$ for each of these days.

14. The above DTMC model is too simple to exhibit high accuracy in forecasting COVID-19. Discuss a more accurate simulation/modeling technique for COVID-19.

15. **Question 2**: Consider a CTMC for the M/M/2 queue (i.e., `Exponential` inter-arrival times with rate λ, `Exponential` service times with rate μ, and two service units). The rate μ is for each server. The traffic intensity $\rho = \frac{\lambda}{2\mu}$.

(a) Solve for the steady-state probabilities π_j, using the partial balance equations.

$$\begin{aligned} \lambda \pi_{j-1} &= 2\mu \pi_j && \text{for } j \geq 2 \\ \lambda \pi_0 &= \mu \pi_1 \end{aligned}$$

Hint:

$$\pi_0 = \frac{1-\rho}{1+\rho}, \qquad \pi_j = ? \quad \text{for } j \geq 1$$

(b) Use this result to solve for the expected number in the system.

$$L = \mathbb{E}[x] = \sum_{j=0}^{\infty} j\, \pi_j$$

(c) Using the formula for L, logic and Little's Law, create a formula summary table for the M/M/2 queue having six formulas $(L_q, L_s, L, T_q, T_s, T)$. The summary will have the form of Table 15.2.

Table 16.3: Formulas for M/M/2 Queueing Models

part	**length**	**formula**	**time**	**formula**
Queue	$L_q =$	?	$T_q =$	?
Service	$L_s =$	?	$T_s =$	$\frac{1}{\mu}$
System	$L =$	?	$T =$	?

(d) Suppose $\lambda = 12$ per hour (overall arrival rate) and $\mu = 7.5$ per hour (per server service rate); compute values for π_0 and the six formulas (for times in minutes).

16.3 Dynamic Linear Models

As with a Hidden Markov Model (HMM), a Dynamic Linear Model (DLM) may be used to represent a system in terms of two stochastic processes, the state of the system at time t, $\mathbf{x}_t$ and the observed values from measurements of the system at time t, $\mathbf{y}_t$. The main difference from an HMM is that the state and its observation are treated as continuous quantities. For time series analysis, it is natural to treat time as discrete values.

As background, consider the following system of homogeneous ODEs that only includes a linear transformation (no constant term). The transition matrix $F^{(c)}$ has been renamed to emphasize that it is for the continuous time problem.

$$\dot{\mathbf{x}}(t) \;=\; F^{(c)}\mathbf{x}(t) \tag{16.31}$$

The corresponding discrete-time system is defined as

$$\mathbf{x}_t \;=\; e^{\Delta t\, F^{(c)}}\mathbf{x}_{t-\Delta t} \tag{16.32}$$

where Δt is time gap between consecutive time points. It is assumed here that the time gaps are uniform. The matrix exponential ($e^X = \sum \frac{1}{k!}X^k$) can be calculated using the Al-Mohy & Higham algorithm [4]. We define matrix F as follows:

$$F \;=\; e^{\Delta t\, F^{(c)}} \tag{16.33}$$

Substituting in the matrix F gives,

$$\mathbf{x}_t \;=\; F\mathbf{x}_{t-\Delta t} \tag{16.34}$$

Again to maintain notational compatibility with the notation previously used for time series analysis, t in $\mathbf{x}_t$ has two interpretation: (1) time index and (2) the actual discrete time/timestamp. Therefore, the above equation can written as follows.

$$\mathbf{x}_t \;=\; F\mathbf{x}_{t-1} \tag{16.35}$$

To deal with uncertainty in the system a noise term may be added. In addition, the state and observation equations distinguished.

For a basic DLM, the dynamics of the system are described by two equations: The *State Equation* indicates how the next state vector $\mathbf{x}_t$ is dependent on the previous state vector $\mathbf{x}_{t-1}$ and a process noise vector $\mathbf{w}_t \sim \text{Normal}(0, Q)$

$$\mathbf{x}_t \;=\; F\mathbf{x}_{t-1} + \mathbf{w}_t \tag{16.36}$$

where Q is the covariance matrix for the process noise. If the dynamics are deterministic, then the covariance matrix is zero, otherwise it can capture uncertainty in the relationships between the state variables (e.g., simple models of the flight of a golf ball often ignore the effects due to the spin on the golf ball).

The *Observation/Measurement Equation* indicates how at time t, the observation vector $\mathbf{y}_t$ is dependent on the current state $\mathbf{x}_t$ and a measurement noise vector $\mathbf{v}_t \sim \text{Normal}(0, R)$

$$\mathbf{y}_t \;=\; H\mathbf{x}_t + \mathbf{v}_t \tag{16.37}$$

where R is the covariance matrix for the measurement noise/error. The process noise and measurement noise are assumed to be independent of each other. The state transition matrix F indicates the linear relationships between the state variables, while the H matrix establishes linear relationships between the state of system and its observations/measurements.

16.3.1 Example: Traffic Sensor

Consider the operation of a road sensor that records traffic flow (vehicles per 15 minutes) and average speed (km per hour). Let $\mathbf{x_t} = [x_{t0}, x_{t1}]$ be the flow of vehicles x_{t0} and their average speed x_{t1} at time t. Assume that the flow is high enough that it can be treated as a continuous quantity and that the covariance matrices are diagonal (uncertainty of flow and speed are independent). The dynamics of the system then may be described by the following state equations:

$$\begin{aligned} x_{t0} &= f_{00}x_{t-1,0} + f_{01}x_{t-1,1} + w_{t0} \\ x_{t1} &= f_{10}x_{t-1,0} + f_{11}x_{t-1,1} + w_{t1} \end{aligned}$$

The sensor tries to capture the dynamics of the system, but depending on the quality of the sensor there will be measurement errors. The observation/measurement variables $\mathbf{y}_t = [y_{t0}, y_{t1}]$ may correspond to the state variables in a one-to-one correspondence or by some linear relationship. The observation of the system then may be described by the following observation equations:

$$\begin{aligned} y_{t0} &= h_{00}x_{t0} + h_{01}x_{t1} + v_{t0} \\ y_{t1} &= h_{10}x_{t0} + h_{11}x_{t1} + v_{t1} \end{aligned}$$

Further assume that estimates for the F and H parameters of the model have been found (see the subsection on Training).

State Equations

$$\begin{aligned} x_{t0} &= 0.9x_{t-1,0} + 0.2x_{t-1,1} + w_{t0} \\ x_{t1} &= -0.4x_{t-1,0} + 0.8x_{t-1,1} + w_{t1} \end{aligned}$$

$$\mathbf{x}_t = \begin{bmatrix} x_{t0} \\ x_{t1} \end{bmatrix} = \begin{bmatrix} 0.9 & 0.2 \\ -0.4 & 0.8 \end{bmatrix} \mathbf{x}_{t-1} + \mathbf{w}_t$$

These state equations suggest that the flow will be a high percentage of the previous flow, but that higher speed suggests increasing flow. In addition, the speed is based on the previous speed, by higher flow suggests that speeds may be decreasing (e.g., due to congestion).

Observation/Measurement Equations

$$\begin{aligned} y_{t0} &= 1.0x_{t0} - 0.1x_{t1} + v_{t0} \\ y_{t1} &= -0.1x_{t0} + 1.0x_{t1} + v_{t1} \end{aligned}$$

$$\mathbf{y}_t \;=\; \begin{bmatrix} y_{t0} \\ y_{t1} \end{bmatrix} \;=\; \begin{bmatrix} 1.0 & -0.1 \\ -0.1 & 1.0 \end{bmatrix} \mathbf{x}_t \;+\; \mathbf{v}_t$$

These observation/measurement equations suggest that higher speed makes it more likely for a vehicle to pass the sensor without being counted and higher flow makes under-estimation of speed to be greater.

16.3.2 Exercises

1. For a DLM, consider the case where $m = n = 1$. The state equations and measurement equations become

$$\begin{aligned} x_t &= ax_{t-1} + w_t \\ y_t &= cx_t + v_t \end{aligned}$$

   where $w_t \sim \text{Normal}(0, \sigma_q^2)$ and $v_t \sim \text{Normal}(0, \sigma_r^2)$. Compare this model with an AR(1) model.

2. For the Traffic Sensor Example, let $Q = \sigma_q^2 I$ and $R = \sigma_r^2 I$. Develop a DLM model using ScalaTion and try low, medium and high values for the variances σ_q^2 and σ_r^2 (9 combinations). Let the initial state of the system be $x_{00} = 100.0$ vehicles per 15 minutes and $x_{01} = 100.0$ km per hour. How does the relative amount of process and measurement error affect the dynamics/observation of the system?

3. Consider the state and observation equations given in the Traffic Sensor Example and assume that the state equations are deterministic (no uncertainty in system, only in its observation). Reduce the DLM to a simpler type of time series model. Explain.

4. Use the Traffic Sensor Dataset (`traffic.csv`) to estimate values for the 2-by-2 covariance matrices Q and R.

5. Use the Traffic Sensor Dataset (`traffic.csv`) to estimate values for the parameters of a DLM model, i.e., for the F and H 2-by-2 matrices.

16.4 Kalman Filter

A Kalman Filter (KF) is a Dynamic Linear Model that incorporates an outside influence on the system. If a driving force or control is applied to the system, an additional term $G\mathbf{u}_t$ is added to the state equation [202, 154],

$$\mathbf{x}_t \;=\; F\mathbf{x}_{t-1} + G\mathbf{u}_t + \mathbf{w}_t \tag{16.38}$$

where $\mathbf{u}_t$ is the (assumed deterministic) force/control vector and the B matrix establishes as linear relationships between the force/control vector and the state vector. An example of a force/control is the constant vector that includes the force of gravity as discussed in the golf ball trajectory problem,

$$\mathbf{u}_t \;=\; [-g]$$

The observation/measurement equation remains the same.

$$\mathbf{y}_t \;=\; H\mathbf{x}_t + \mathbf{v}_t$$

The process noise $\mathbf{w}_t$ and the measurement noise $\mathbf{v}_t$ also remain the same. The Kalman Filter model, therefore includes five matrices.

Table 16.4: Matrices Used in Kalman Filter Model

matrix	**dimensions**	**description**
F	n-by-n	state transition matrix
G	n-by-l	state-control matrix
H	m-by-n	observation matrix
Q	n-by-n	process noise covariance matrix
R	m-by-m	measurement noise covariance matrix

The Kalman Filter model at time t includes five vectors:

Table 16.5: Vectors Used in Kalman Filter Model

vector	**dimension**	**description**
$\mathbf{x}_t$	n	state vector
$\mathbf{u}_t$	l	force/control vector
$\mathbf{y}_t$	m	measurement vector
$\mathbf{w}_t$	n	process noise vector
$\mathbf{v}_t$	m	measurement noise vector

16.4.1 Example: Golf Ball Trajectory

Returning to problem posed at the beginning of the chapter of modeling the trajectory of a golf ball hit straight up with an initial velocity of 60 meters per second (mps), the discretized form of the system of Ordinary Differential Equations is copied below.

$$\begin{aligned} y_t &= y_{t-1} + v_{t-1}\Delta t - \frac{1}{2}g(\Delta t)^2 \\ v_t &= v_{t-1} - g\Delta t \end{aligned}$$

In matrix-vector form, the equation for $\mathbf{x}_t$ may be written as

$$\mathbf{x}_t = \begin{bmatrix} y_t \\ v_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \mathbf{x}_{t-1} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} [-g]$$

When noise is added, the variables will become random variables (blue font). The discrete-time system then serves as the basis to formulate the Kalman filter state equations [154].

$$\mathbf{x}_t = \begin{bmatrix} y_t \\ v_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \mathbf{x}_{t-1} + \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} [-g] + \mathbf{w}_t$$

In this case, the force/control is the one-dimensional vector $\mathbf{u}_t = [-g]$. Again the initial conditions are $\mathbf{x}_0 = [0 \text{ m}, 60 \text{ m/s}]$. For simplicity, the process noise $\mathbf{w}_t$ is assumed to be on scale with the measurement noise (see below).

$$Q = \mathbb{C}[\mathbf{w}_t] = 0.05\, I = \begin{bmatrix} 0.05 & 0.0 \\ 0.0 & 0.05 \end{bmatrix}$$

The measurement equation indicate what variables are measured and how they relate to the state variables. When are the state variables are directly measurable and one if interested all the state variables, the observation matrix will be the 2-dimensional identity matrix $H = I$.

$$\mathbf{y}_t = I\mathbf{x}_t + \mathbf{v}_t$$

The last step is to determine the covariance R of the measurement noise $\mathbf{v}_t$

$$\mathbf{v}_t \sim \text{Normal}(0, R)$$

Modern golf ball tracking devices have standard deviations as low as $\sigma_0 = 0.25$ meters for position in a particular dimension and $\sigma_1 = 0.2$ meters per second for velocity. These are rough estimates based on sources such as [106]. These may be used to for the variances (diagonal elements) in the covariance matrix. Remaining is to determine the correlation ρ_{01} between v_{t0} and v_{t1} to get the covariance $\sigma_{01} = \rho_{01}\sigma_0\sigma_1$. Such information is hard to come by, but it makes sense that they would be positively correlated, so let $\rho_{01} = 0.5$.

$$R = \mathbb{C}[\mathbf{v}_t] = \begin{bmatrix} \sigma_0^2 & \sigma_{01} \\ \sigma_{01} & \sigma_1^2 \end{bmatrix} = \begin{bmatrix} 0.0625 & 0.00125 \\ 0.00125 & 0.04 \end{bmatrix}$$

See the next section for alternative approaches for estimating Q and R.

Note that if only the height is of interest, then observation matrix $H = [1, 0]$.

16.4.2 Training

The main goal of training is to minimize the error in estimating the state. At time t, a new measurement $\mathbf{y}_t$ becomes available. The errors before and after this event are the differences between the actual state $\mathbf{x}_t$ and the estimated state before $\hat{\mathbf{x}}_t^-$ (predicted) and after $\hat{\mathbf{x}}_t$ (corrected) [202].

$$\begin{aligned} \mathbf{e}_t^- &= \mathbf{x}_t - \hat{\mathbf{x}}_\mathbf{t}^- \\ \mathbf{e}_t &= \mathbf{x}_t - \hat{\mathbf{x}}_\mathbf{t} \end{aligned}$$

Since $\mathbf{w}_t$ has a zero mean, the covariance matrices (P_t^- and P_t) for the before and after state errors may be computed as expectations of their outer products.

$$P_t^- = \mathbb{C}\left[\mathbf{e}_t^-\right] = \mathbb{E}\left[\mathbf{e}_t^- \otimes \mathbf{e}_t^-\right] \qquad \text{before errors} \tag{16.39}$$

$$P_t = \mathbb{C}\left[\mathbf{e}_t\right] = \mathbb{E}\left[\mathbf{e}_t \otimes \mathbf{e}_t\right] \qquad \text{after errors} \tag{16.40}$$

The essential insight by Kalman was that the after estimate should be the before estimate adjusted by a weighted difference between the actual measured value $\mathbf{y}_t$ and its before estimate $H\hat{\mathbf{x}}_t^-$.

$$\hat{\mathbf{x}}_t^- = F\hat{\mathbf{x}}_{t-1} + G\mathbf{u}_t \qquad \text{predicted} \tag{16.41}$$

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + K_t[\mathbf{y}_t - H\hat{\mathbf{x}}_t^-] \qquad \text{corrected} \tag{16.42}$$

The n-by-m K_t matrix is called the *Kalman Gain* and the above equations may be referred to as the *Kalman state update equations*. If the actual measurement is very close to its predicted value, little adjustment to the predicted state value is needed. On the other hand, when there is a disagreement, the adjustment based upon the measurement should be tempered based upon the reliability of the measurement. A small gain will dampen the adjustment, while a high gain may result in large adjustments. The trick is to find the optimal gain K_t.

Using a Minimum Variance Unbiased Estimator (MVUE) for parameter estimation for a Kalman Filter means that the trace of the error covariance matrix should be minimized (see exercises for details).

$$\mathbb{V}\left[\|\mathbf{e}_t\|\right] = \mathbb{E}\left[\|\mathbf{e}_t\|^2\right] = \text{trace}\ \mathbb{C}\left[\mathbf{e}_t\right] \tag{16.43}$$

Plugging the Kalman Gain equation into the above equation gives the following optimization problem:

$$\min \text{trace}\ \mathbb{E}\left[(\hat{\mathbf{x}}_t^- + K_t[\mathbf{y}_t - H\hat{\mathbf{x}}_t^-) \otimes (\hat{\mathbf{x}}_t^- + K_t[\mathbf{y}_t - H\hat{\mathbf{x}}_t^-)\right] \tag{16.44}$$

This optimization will produce (see exercises) the following equation that can be used to update the Kalman Gain.

$$P_t^- = FP_{t-1}F^\intercal + Q \tag{16.45}$$

$$K_t = P_t^- H^\intercal [HP_t^- H^\intercal + R]^{-1} \tag{16.46}$$

$$P_t = [I - K_t H]P_t^- \tag{16.47}$$

16.4.3 Exercises

1. Suppose that fog negatively affects traffic and speed. Use the Traffic Sensor with Fog Dataset (`traffic_fog.csv`) to estimate values for the 2-by-2 covariance matrices Q and R.

2. Use the Traffic Sensor with Fog Dataset (`traffic_fog.csv`) to estimate values for the parameters of a Kalman Filter model, i.e., for the F, G and H 2-by-2 matrices.

3. Show that if $\hat{\mathbf{y}}$ is an unbiased estimator for $\mathbf{y}$ (i.e., $\mathbb{E}[\hat{\mathbf{y}}] = \mathbb{E}[\mathbf{y}]$) then the minimum error variance $\mathbb{V}[\|\mathbf{y} - \hat{\mathbf{y}}\|]$ is

$$\mathbb{E}\left[\|\mathbf{y} - \hat{\mathbf{y}}\|^2\right] \;=\; \text{trace}\, \mathbb{E}[(\mathbf{y} - \hat{\mathbf{y}}) \otimes (\mathbf{y} - \hat{\mathbf{y}})]$$

4. Explain why minimizing the trace of the covariance $\mathbb{C}[\mathbf{e}_t]$ leads to optimal Kalman Gain K.

16.5 Extended Kalman Filter

When some of the relationships between state variables are nonlinear, the simplest option is to use an Extended Kalman Filter (EKF). The linear combinations in the equations for Kalman Filters are now replaced with differentiable (nonlinear) vector functions **f** and **h**.

State Transition Function

The dynamics of the state are governed by a (nonlinear) state transition function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ and specified in the state equation,

$$\mathbf{x}_t = \mathbf{f}(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{w}_t \tag{16.48}$$

where $\mathbf{u}_t$ is the (assumed deterministic) control vector, if relevant.

Observation/Measurement Function

The observation/measurement equation may also include a (nonlinear) observation function $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^m$ of the state.

$$\mathbf{y}_t = \mathbf{h}(\mathbf{x}_t) + \mathbf{v}_t \tag{16.49}$$

The process noise $\mathbf{w}_t$ and the measurement noise $\mathbf{v}_t$ should be close to Gaussian (Normally distributed). They are also assumed to be additive, otherwise they need to be incorporated into the **f** and **h** functions.

Table 16.6: Functions Used in Extended Kalman Filter Model

function	**description**
$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$	state transition function
$\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^m$	observation function

For the example in the next subsection of a *SEIHRD* epidemic/pandemic model, the state vector is 6-dimensional ($n = 6$), while the measurement/observation vector is 4-dimensional ($m = 4$).

16.5.1 Training

Extended Kalman Filters operate much like Kalman filters [154]. The only change to the Kalman update equations is to use the **f** and **h** functions in place of the multiplications involving the F, G and H matrices (for simplicity in previous section, these were taken to be constant matrices, but in a more general treatment they would vary with time F_t, G_t and H_t).

The Kalman state update equations for EKF are as follows:

$$\hat{\mathbf{x}}_t^- = \mathbf{f}(\hat{\mathbf{x}}_{t-1}, \mathbf{u}_t) \qquad \text{predicted} \tag{16.50}$$

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + K_t[\mathbf{y}_t - \mathbf{h}(\hat{\mathbf{x}}_t^-)] \qquad \text{corrected} \tag{16.51}$$

The Kalman Gain for EKF may be updated as follows:

$$\begin{aligned} P_t^- &= F_{t-1} P_{t-1} F_{t-1}^\intercal + Q & (16.52) \\ K_t &= P_t^- H_t^\intercal [H_t P_t^- H_t^\intercal + R]^{-1} & (16.53) \\ P_t &= [I - K_t H_t] P_t^- & (16.54) \end{aligned}$$

where now the matrices F_t and H_t are slopes of the $\mathbf{f}$ and $\mathbf{h}$ functions, respectively.

$$\begin{aligned} F_t &= \frac{\partial \mathbf{f}}{\partial \mathbf{x}_t} & (16.55) \\ H_t &= \frac{\partial \mathbf{h}}{\partial \mathbf{x}_t} & (16.56) \end{aligned}$$

In other words, at each discrete time step t, the nonlinear vector-valued functions $\mathbf{f}$ and $\mathbf{h}$ are approximated by their local slopes the at predicted state $\hat{\mathbf{x}}_t^-$, computed as Jacobian matrices. Note, $F_t^\intercal$ and $H_t^\intercal$ are the transposes of F_t and H_t, respectively.

Recall that the Jacobian of a vector function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^l$, is an l-by-n matrix.

$$\mathbb{J}_\mathbf{f}(\mathbf{x}) = \left[\frac{\partial f_i}{\partial x_j}\right]_{0 \le i < l, 0 \le j < n} = \begin{bmatrix} \frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \dots & \frac{\partial f_0}{\partial x_{n-1}} \\ \frac{\partial f_1}{\partial x_0} & \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_{n-1}} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_{l-1}}{\partial x_0} & \frac{\partial f_{l-1}}{\partial x_1} & \dots & \frac{\partial f_{l-1}}{\partial x_{n-1}} \end{bmatrix}$$

16.5.2 Example: SEIHRD Model

This subsection presents a detailed example of how an Extended Kalman Filter (EKF) can be used to model epidemics and pandemics. An SEIHRD model as an extension of the SEIR model [110] can be used to forecast the number of infections, hospitalizations and deaths in epidemics.

The EKF model will derived from a system of Ordinary Differential Equations (ODEs). The time varying variables/functions are specified below. At this point, time is treated continuously.

1. The number of Susceptible individuals at time t is denoted by $S(t)$.
2. The number of Exposed individuals at time t is denoted by $E(t)$.
3. The number of Infected and not Hospitalized individuals at time t is denoted by $I(t)$.
4. The number of Infected and Hospitalized individuals at time t is denoted by $H(t)$.
5. The number of Recovered individuals at time t is denoted by $R(t)$.
6. The number of Deaths time t is denoted by $D(t)$.

To make this more clear, suppose the study is the spread of COVID-19 during the year 2020 in the United States. The population of individuals is given by $N = 330$ million.

State Transition Rates

The dynamics of the system are governed by the state transition rates. These will become parameters in the differential equations to be estimated from the data. Assuming the birth rate matches the COVID-19 death rate, the population not change over time (a short time span approximation).

$$N \;=\; S(t) + E(t) + I(t) + H(t) + R(t) \tag{16.57}$$

The model is begun when there is a minimal level of infection. Assuming that the whole population is susceptible, the initial conditions are $S(0) = N - I(0)$. For this model, time 0 is set to when United States has at least one infection. Each of the other variables at time 0 is 0.

The state transition rates are explained in the table below.

Table 16.7: State Transition Rates

Rate	**From**	**To**	**Description**
q_0	S	E	exposure rate
q_1	E	I	infection rate
q_2	I	H	hospitalization rate
q_3	I	R	recovery rate for mild cases
q_4	H	R	recovery rate for severe cases
q_5	H	D	death rate
.	D	S	birth = death rate = $\dot{D}(t)$

State Transition Diagram

The state transition diagram is shown in Figure 16.5. If there is a significant reinfection rate, an edge can be added from state R to state S.

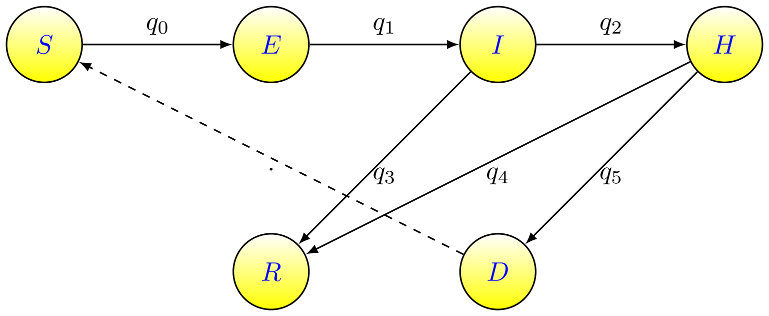


Figure 16.5: State Transition Diagram for SEIHRD Models

Ordinary Differential Equations

The six time variables are interrelated through ordinary differential equations, where a dot above the variable denotes a time derivative,

$$\begin{aligned}
\dot{S}(t) &= \dot{D}(t) - q_0 S(t) && \text{Susceptible} \\
\dot{E}(t) &= q_0 S(t) - q_1 E(t) && \text{Exposed} \\
\dot{I}(t) &= q_1 E(t) - (q_2 + q_3) I(t) && \text{only Infected} \\
\dot{H}(t) &= q_2 I(t) - (q_4 + q_5) H(t) && \text{Infected and Hospitalized} \\
\dot{R}(t) &= q_3 I(t) + q_4 H(t) && \text{Recovered} \\
\dot{D}(t) &= q_5 H(t) && \text{Died}
\end{aligned}$$

The LHS is the rate of change of the variable, while the RHS is the sum of incoming edges minus the sum of the outgoing edges. From epidemiology, the first transition rate q_0 can be further dissected. Exposure depends on members in $S(t)$ coming in contact with an infected individual in $I(t)$ or $H(t)$ in terms of their fraction of the population N and is proportional to the new parameter α.

$$q_0 = \alpha \frac{I(t) + H(t)}{N} \tag{16.58}$$

Replacing q_0 and $\dot{D}(t)$ in the differential equations results in

$$\begin{aligned}
\dot{S}(t) &= q_5 H(t) - \alpha \frac{I(t) + H(t)}{N} S(t) && \text{Susceptible} \\
\dot{E}(t) &= \alpha \frac{I(t) + H(t)}{N} S(t) - q_1 E(t) && \text{Exposed} \\
\dot{I}(t) &= q_1 E(t) - (q_2 + q_3) I(t) && \text{only Infected} \\
\dot{H}(t) &= q_2 I(t) - (q_4 + q_5) H(t) && \text{Infected and Hospitalized} \\
\dot{R}(t) &= q_3 I(t) + q_4 H(t) && \text{Recovered} \\
\dot{D}(t) &= q_5 H(t) && \text{Died}
\end{aligned}$$

In order for the six equations to be useful for forecasting, the six parameters, $\alpha, q_1, q_2, q_3, q_4, q_5, q_6$, must be estimated from data.

Notice that a further simplification is possible: One of the variables may be removed since all the variables sum up to N.

Discretization

As was done with the golf ball example, the ODEs may be discretized. In this case, since all the differential equations are first-order, the performance of the Euler Method should be acceptable, although more advanced numerical integration methods could be applied. (Note that Newton's Second Law is a second-order ODE (converted into two coupled first order equations) so the Verlet Method was superior to the Euler Method for the golf ball trajectory problem.) Recall that Δt is the time gap in the discrete-time system and time can now be used as subscript since it is discrete.

$$\begin{array}{rcll}
S_t &=& S_{t-1} + \Delta t \left[q_5 H_{t-1} - \alpha \dfrac{I_{t-1} + H_{t-1}}{N} S_{t-1} \right] & \text{Susceptible} \\
E_t &=& E_{t-1} + \Delta t \left[\alpha \dfrac{I_{t-1} + H_{t-1}}{N} S_{t-1} - q_1 E_{t-1} \right] & \text{Exposed} \\
I_t &=& I_{t-1} + \Delta t \left[q_1 E_{t-1} - (q_2 + q_3) I_{t-1} \right] & \text{only Infected} \\
H_t &=& H_{t-1} + \Delta t \left[q_2 I_{t-1} - (q_4 + q_5) H_{t-1} \right] & \text{Infected and Hospitalized} \\
R_t &=& R_{t-1} + \Delta t \left[q_3 I_{t-1} + q_4 H_{t-1} \right] & \text{Recovered} \\
D_t &=& D_{t-1} + \Delta t \left[q_5 H_{t-1} \right] & \text{Died}
\end{array}$$

Similar discretized system of ODEs are given in the literature for SEIR models and their extensions [27].

Formulation as an Extended Kalman Filter

The discretized system of ODEs may be now formulated as an Extended Kalman Filter model. For this model, there is no forcing/control vector, i.e., $\mathbf{u}_t$ is removed. Also, the observation function h is assumed to be linear. Therefore, the state and observation/measurement vector equations take the following form:

$$\begin{array}{rcl}
\mathbf{x}_t &=& \mathbf{f}(\mathbf{x}_{t-1}) + \mathbf{w}_t \\
\mathbf{y}_t &=& H\mathbf{x}_t + \mathbf{v}_t
\end{array}$$

State Equations

The state vector includes random variables for each of the variables in the discrete-time system of equations.

$$\mathbf{x}_t = [S_t, E_t, I_t, H_t, R_t, D_t]$$

For this model, the nonlinear function $\mathbf{f}$ is quadratic so it can be written in matrix form.

$$\begin{bmatrix} S_t \\ E_t \\ I_t \\ H_t \\ R_t \\ D_t \end{bmatrix} = \mathbf{x}^\intercal t-1 \begin{bmatrix} 0 & 0 & -\frac{\alpha}{N}\Delta t & -\frac{\alpha}{N}\Delta t & 0 & 0 \\ 0 & 0 & \frac{\alpha}{N}\Delta t & \frac{\alpha}{N}\Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{x}_{t-1} + \begin{bmatrix} 1 & 0 & 0 & q_5\Delta t & 0 & 0 \\ 0 & 1-q_1\Delta t & 0 & 0 & 0 & 0 \\ 0 & q_1\Delta t & 1-q_{23}\Delta t & 0 & 0 & 0 \\ 0 & 0 & q_2\Delta t & 1-q_{45}\Delta t & 0 & 0 \\ 0 & 0 & q_3\Delta t & q_4\Delta t & 1 & 0 \\ 0 & 0 & 0 & q_5\Delta t & 0 & 1 \end{bmatrix} \mathbf{x}_{t-1} + \mathbf{w}_t$$

where $\mathbf{x}^\intercal t-1$ is the transpose of $\mathbf{x}_{t-1}$ (making it a row vector), $q_{23} = q_2 + q_3$ and $q_{45} = q_4 + q_5$.

Observation/Measurement Equations

The observation/measurement vector includes random variables for the observable state variable, i.e., those variables for which time series data exists.

$$\mathbf{y}_t = [I^o_t, H^o_t, R^o_t, D^o_t]$$

For this model, there is a direct relationship between the state and measurement variables, so the observation/measurement vector is as follows:

$$\begin{bmatrix} I_t^o \\ H_t^o \\ R_t^o \\ D_t^o \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x}_t + \mathbf{v}_t$$

Jacobian Matrices

$F_t \in \mathbb{R}^{6\times 6}$ is given by the Jacobian matrix for the state transition function $\mathbf{f}$.

$$F_t = \frac{\partial \mathbf{f}}{\partial \mathbf{x}_t}$$

$$F_t = \begin{bmatrix} 1-\alpha\frac{I_{t-1}+H_{t-1}}{N}\Delta t & 0 & \frac{S_{t-1}}{N}\Delta t & (q_5+\frac{S_{t-1}}{N})\Delta t & 0 & 0 \\ \alpha\frac{I_{t-1}+H_{t-1}}{N}\Delta t & 1-q_1\Delta t & \frac{S_{t-1}}{N}\Delta t & \frac{S_{t-1}}{N}\Delta t & 0 & 0 \\ 0 & q_1\Delta t & 1-q_{23}\Delta t & 0 & 0 & 0 \\ 0 & 0 & q_2\Delta t & 1-q_{45}\Delta t & 0 & 0 \\ 0 & 0 & q_3\Delta t & q_4\Delta t & 1 & 0 \\ 0 & 0 & 0 & q_5\Delta t & 0 & 1 \end{bmatrix}$$

Similarly, $H_t \in \mathbb{R}^{4\times 6}$ is given by the Jacobian matrix for the observation function $\mathbf{h}$.

$$H_t = \frac{\partial \mathbf{h}}{\partial \mathbf{x}_t}$$

$$H_t = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Noise Covariance Matrices

The covariance of the process noise $Q \in \mathbb{R}^{n\times n}$ and the covariance of the measurement noise $R \in \mathbb{R}^{m\times m}$ can be challenging to determine. These covariance matrices feed into the calculation of the Kalman Gain K, so that reliance on process predictions versus measured quantities is partially based on how low their respective covariances are.

There are three basic approaches for assigning values to Q and R: (1) Use knowledge of the process for Q and of the measurement devices for R. The covariance of the process noise Q may be roughly determined based on inherent uncertainty in the model (unpredictably changing wind conditions) or missing model elements (e.g., the force of drag). The covariance of the measurement noise R may be roughly determined based on characteristics of the measurement device(s). (2) Use hyper-parameters and Hyper-Parameter Optimization (HPO) as discussed below. (3) Use Adaptive (Extended) Kalman Filters that adjust Q_t and R_t at each step based on calculated errors/residuals/innovations [3].

As approach (1) is very problem specific and approach (3) is complicated, approach (2) is a good alternative for a quick start. This alternative uses tunable hyper-parameters λ_Q and λ_R as multipliers of identity matrices.

$$\begin{aligned} Q &= \lambda_Q I \\ R &= \lambda_R I \end{aligned}$$

where I_Q an an n-by-n identity matrix and I_R an an m-by-m identity matrix. The hyper-parameters are problem specific, but $\lambda = 0.1$ or 1.0 are reasonable ballparks [179] for starting grid searches or more efficient HPO techniques.

Note that approach (1) is used in the golf ball trajectory problem given in the last section and approach (3) is explored in the exercises.

Initialization

Some initialization is needed before starting an Extended Kalman Filter. Due to lack of sensitivity, the initial covariance of the process error can be set to an n-by-n identity matrix, i.e., $P_0 = I$. The initial state $x_o = [329999999, 0, 1, 0, 0, 0]$. The time gap/increment $\Delta t = 1$ day.

16.5.3 Exercises

1. Use the ScalaTion COVID-19 datasets found at `https://github.com/scalation/data` to train an Extended Kalman Filter (EKF) and make four-week ahead forecasts. It is composed of data collected from multiple datasets stored at `https://github.com/CSSEGISandData/COVID-19/tree/master/csse_covid_19_data`. In particular, use the dataset that contains data about COVID-19 cases, hospitalizations, recoveries, and deaths in the United States for one full year since the first confirmed case on January 22, 2020. The dataset is in a 367 row by 6 column CSV file containing the data indicated in the table below. There are 367 rows since the first row contains column headers and the year 2020 was a leap year.

   Table 16.8: COVID-19 Dataset: United States

Column	Description
d	date: January 22, 2020 to January 21, 2021
t	time: 0 to 365
C_t	total (cumulative) number of confirmed cases by day t
H_t	total (cumulative) number of hospitalized individuals by day t
R_t	total (cumulative) number of recovered individuals by day t
D_t	total (cumulative) number of deaths by day t
I_t	deduced current number of infected individuals on day t (not counting hospitalizations)
I_t^H	deduced current number of hospitalized individuals on day t

   The data cover the last four states of the State Transition Diagram. The data also include the total number of confirmed cases C_t and the total number of hospitalizations H_t that are not directly part of the $SEIHRD$ model, but are used to compute I_t and H_t.

$$\begin{aligned} I_t &= C_t - H_t \\ H_t &= H_t - R_t - D_t \end{aligned}$$

2. **Early Stage Approximation**. In the early stage of an epidemic/pandemic, the number of Susceptible individuals changes slowly, so that $\frac{S_{t-1}}{N}$ is nearly a constant that can be rolled into α. This allows the first two difference equations to be rewritten.

$$\begin{aligned} S_t &= S_{t-1} + \Delta t \; [q_5 H_{t-1} - \alpha(I_{t-1} + H_{t-1})] && \text{Susceptible} \\ E_t &= E_{t-1} + \Delta t \; [\alpha(I_{t-1} + H_{t-1}) - q_1 E_{t-1}] && \text{Exposed} \end{aligned}$$

   The state equations are now linear (and the observation/measurement equations have been linear), so an ordinary Kalman Filter may be used. Use the ScalaTion COVID-19 dataset for the year 2020 to train a Kalman Filter (KF) and make four-week ahead forecasts. Compare with the results of using the EKF.

3. S_t and E_t are hard to measure. Use the following identity to eliminate S_t from the model.

$$S_t = N - E_t - I_t - H_t - R_t$$

   Use the ScalaTion COVID-19 dataset for the year 2020 to train a Kalman Filter (KF) and make four-week ahead forecasts. Compare with previous results.

4. Based on the last exercise, there is only one unobserved variables E_t. Use the early stage approximation to eliminate E_t. Now all the state variables are observable ($n = m = 4$). Create a new Kalman Filter where both F and H are 4-by-4 matrices. Use the ScalaTion COVID-19 dataset for the year 2020 to train this new Kalman Filter (KF) and make four-week ahead forecasts. Compare with previous results.

5. When all the state variables are observable and the state transition and observation functions are linear, a Vector Auto-Regressive VAR(p, n) model may be applied. Use the ScalaTion COVID-19 dataset for the year 2020 to train a VAR model and make four-week ahead forecasts. In particular, create a VAR(1, 4) model and a VAR(2, 4) model as shown below,

$$\mathbf{y}_t = \delta + \Phi^{(0)}\mathbf{y}_{t-2} + \Phi^{(1)}\mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t$$

   where $\mathbf{y}_t = [I_t, H_t, R_t, D_t]$, $p = 2$ and $n = 4$, with two parameter matrices: $\Phi_0 \in \mathbb{R}^{4\times 4}$ and $\Phi_1 \in \mathbb{R}^{4\times 4}$ Compare with previous results.

6. Create an AR$^*(1, 4)$ model and a AR$^*(2, 4)$ model. Recall that an AR$^*(p, n)$ model is a VAR(p, n) model where all the parameter matrices are diagnonal. Compare with previous results.

7. As the state transition function $\mathbf{f}$ or the observation function $\mathbf{h}$ are no longer well locally approximately by linear functions or the noise is no longer well approximated by Gaussian distributions, it is recommended to use an **Unscented Kalman Filter** [199]. Use the ScalaTion COVID-19 dataset for the year 2020 to train an Unscented Kalman Filter (UKF) and make four-week ahead forecasts. Compare with previous results.

8. As the dimensionality of the problem becomes very large, such as in Numerical Weather Prediction (NWP), it is recommended to use an **Ensemble Kalman Filter** [46, 91]. Use the ScalaTion COVID-19 dataset for the year 2020 to train an Ensemble Kalman Filter (EnKF) and make four-week ahead forecasts. Compare with previous results.

9. There are two techniques for estimating noise covariance matrices Q_t and R_t for Adaptive (Extended) Kalman Filters, one based innovations (errors before correction) the other based on residuals (errors after correction). Read the following two papers [24, 3] and write a short essay on how the two techniques work.

10. The discretization of the system of Ordinary Differential Equations (ODEs) for the *SEIHRD* model used the Euler Method. For a system of ODEs, the vector equation is of the form:

$$\dot{\mathbf{y}}(t) \; = \; \mathbf{f}(t, \mathbf{y}(t))$$

Discretization using the explicit, first-order Euler Method gives

$$\mathbf{y}_t \; = \; \mathbf{y}_{t-1} \, + \, \Delta t \, \mathbf{f}(t - \Delta t, \mathbf{y}_{t-1})$$

Try using an explicit, second-order Runge-Kutta Method (RK2) instead

$$\begin{aligned} \tilde{\mathbf{y}}_t \; &= \; \mathbf{y}_{t-1} \, + \, \Delta t \, \mathbf{f}(t - \Delta t, \mathbf{y}_{t-1}) && \text{predicted} \\ \mathbf{y}_t \; &= \; \mathbf{y}_{t-1} \, + \, \frac{\Delta t}{2} \left[\mathbf{f}(t - \Delta t, \mathbf{y}_{t-1}) \, + \, \mathbf{f}(t, \tilde{\mathbf{y}}_t) \right] && \text{corrected} \end{aligned}$$

11. **Mobility Aware *SEIHRD* Model**

This next more realistic, although more complex, model looks at regions (e.g., states, counties) and considers mobility/traffic between regions.

FIX

Mobility Aware Differential Equations

(a) The number of Susceptible individuals in region k at time t is denoted by S_{kt}.

(b) The number of Exposed individuals in region k at time t is denoted by E_{kt}.

(c) The number of Infected and not Hospitalized individuals in region k at time t is denoted by I_{kt}.

(d) The number of Infected and Hospitalized individuals in region k at time t is denoted by H_{kt}.

(e) The number of Recovered individuals in region k at time t is denoted by R_{kt}.

(f) The number of Deaths in region k at time t is denoted by D_{kt}.

Since infections in one region can cause infections in other regions, use of mobility data can improve forecasts. A simple approach is to add a connectivity/mobility matrix that indicates the amount of travel between regions.

$$C_{jk} = \text{number of individuals traveling from region } j \text{ to region } k \text{ in one time unit} \tag{16.59}$$

The diagonal in the matrix is minus the number of individuals leaving the region. The issue is to determine, for each state the number of individuals entering region k in one time unit. For example, for susceptibility the travel adjusted number is

$$\tau(S_{kt}) \;=\; S_{kt} + \sum_{j=0}^{l-1} \frac{S_{jt}}{n_j} C_{jk} \tag{16.60}$$

Therefore, the six space-time variables are now interrelated through a new set of ordinary differential equations,

$$\begin{aligned}
\dot{S}_{kt} &= q_5\tau(H_{kt}) - \alpha\frac{\tau(I_{kt}) + \tau(H_{kt})}{n_k}\tau(S_{kt}) && \text{Susceptible} \\
\dot{E}_{kt} &= \alpha\frac{\tau(I_{kt}) + \tau(H_{kt})}{n_k}\tau(S_{kt}) - q_1\tau(E_{kt}) && \text{Exposed} \\
\dot{I}_{kt} &= q_1\tau(E_{kt}) - (q_2 + q_3)\tau(I_{kt}) && \text{only Infected} \\
\dot{I}^H_{kt} &= q_2\tau(I_{kt}) - (q_4 + q_5)\tau(H_{kt}) && \text{Infected and Hospitalized} \\
\dot{R}_{kt} &= q_3\tau(I_{kt}) + q_4\tau(H_{kt}) && \text{Recovered} \\
\dot{D}_{kt} &= q_5\tau(H_{kt}) && \text{Died}
\end{aligned}$$

Parameter Estimation

A common way to estimate the parameters is to use Maximum Likelihood Estimation [109]. The likelihood function is ...

Basic Difference Equations Approximation

The basic six differential equation can be approximated using six difference equations, where for example $\Delta S_{kt} = S_{kt} - S_{k,t-1}$. One time unit will correspond to one day.

$$\begin{array}{rcll}
\Delta S_{kt} & = & q_5 H_{k,t-1} - \alpha \dfrac{I_{k,t-1} + H_{k,t-1}}{n_k} S_{k,t-1} & \text{Susceptible} \\
\Delta E_{kt} & = & \alpha \dfrac{I_{k,t-1} + H_{k,t-1}}{n_k} S_{k,t-1} - q_1 E_{k,t-1} & \text{Exposed} \\
\Delta I_{kt} & = & q_1 E_{k,t-1} - (q_2 + q_3) I_{k,t-1} & \text{only Infected} \\
\Delta H_{kt} & = & q_2 I_{k,t-1} - (q_4 + q_5) H_{k,t-1} & \text{Infected and Hospitalized} \\
\Delta R_{kt} & = & q_3 I_{k,t-1} + q_4 H_{kt} & \text{Recovered} \\
\Delta D_{kt} & = & q_5 H_{k,t-1} & \text{Died}
\end{array}$$

Again, the six parameters to estimate are $\alpha, q_1, q_2, q_3, q_4, q_5,$ and q_6. Each of the fifty states in the United States may be solve separately or pooled together.

Note that the approximation above only uses first lags, utilizing more lags may lead to better results.

The dataset will consist of three time series, I_{kt} infected cases, H_{kt} hospitalizations, and D_{kt} deaths.

Mobility Aware Difference Equations Approximation

TBD.

16.6 ODE Parameter Estimation

$$
\begin{aligned}
y &= \mathbf{x}(t) + \epsilon \\
\frac{d\mathbf{x}(t)}{dt} &= f(\mathbf{x}(\mathbf{t});\ \mathbf{b})
\end{aligned}
$$

Nonlinear Least Squares (NLS)

Least Squares Approximation (LSA)

Chapter 17

Event-Oriented Models

The simulation modeling techniques discusses so far center around specifying a set of equations. The structure or operation rules of actual systems may require logic that is hard to express in this fashion. A very flexible way of expressing such logic is to focus what happens that may affect the state of system. This is the approach that is followed by Event-Oriented Models. These are also called Event Scheduling Models that suggests how the simulation engine would need to work. The engine needs to provide an means for creating, scheduling and processing events over time.

Before discussing event oriented models in more detail, several simulation modeling paradigms will be highlighted.

17.1 A Taxonomy/Ontology for Simulation Modeling

The most recent version of the Discrete-event Modeling Ontology (DeMO) lists five simulation modeling paradigms or world-views for simulation (see the bullet items below). These paradigms are briefly discussed below and explained in detail in [175].

- **State-Oriented Models**. State-oriented models focus states and state transitions. State-oriented models include Markov Chains and their generalizations such as Generalized Semi-Markov Processes (GSMPs) A GSMP can be defined using three functions,

  - an activation function $\{e\} = a(\mathbf{x}(t))$,
  - a clock function $t' = c(\mathbf{x}(t), e)$,
  - a state-transition function $\mathbf{x}(t') = \mathbf{d}(\mathbf{x}(t), e)$.

  In simulation, advancing to the current state $\mathbf{x}(t)$ causes a set of events $\{e\}$ to be activated according to the activation function a. Events occur instantaneously and may affect both the clock and transition functions. The clock function c determines how time advances from t to t' and the state-transition function determines the next state $\mathbf{x}(t')$. One can also tie in the input and output vectors. The input vector $\mathbf{u}$ is used to initialize a state at some start time t_0 and the response vector $\mathbf{y}$ can be a function of the state sampled at multiple times during the execution of the simulation model.

- **Event-Oriented Models**. State-oriented models may become unwieldy when the state-space becomes very large. One option is to focus on state changes that occur by processing events in time order. An event may indicate what other events it causes as well as how it may change the state. Essentially, the activation and state transition functions are divided into several simpler functions, one for each event e:

  - $\{e\} = a_e(\mathbf{x}(t))$,
  - $\mathbf{x}(t') = \mathbf{d}_e(\mathbf{x}(t))$.

  Logic for each event type implements the a_e, what other event to trigger and d_e how to change (or transition) the state. Time advance is simplified to just setting the time t' to the time of the most imminent event on a Future Event List (activated events are placed on this list in time order).

- **Process-Oriented Models**. One of the motivations for process-oriented models is that event-oriented models provide a fragmented view of the system or phenomena. As combinations of low-level events determine behavior, it may be difficult to see the big picture or have an intuitive feel for the behavior. Process-oriented or process-interaction models aggregate events by putting them together to form a process. An example of a process is a customer in a store. As the simulated customer (as an active entity) carries out behavior it will conditionally execute multiple events over time. A simulation then consists of many simultaneously active entities and may be implemented using coroutines (or threads/actors as a more heavyweight alternative). Typically, there is one coroutine for each active entity. The overall state of a simulation then includes a combination of the states of each active entity (each coroutine/thread has its own stack). The global shared state also includes a variety of resources types.

- **Activity-Oriented Models**. There are many types of activity-oriented models including Petri-Nets and Activity-Cycle Diagrams. The main characteristics of such models is a focus on the notion of activity. An activity (e.g, customer checkout) corresponds to a distinct action that occurs over time and includes a start event and an end event. Activities may be started because time advances to its start time or a triggering condition becomes true. Activities typically involve one or more entities. State information is stored in activities, entities and the global shared state.

- **System Dynamics Models**. System dynamics models have been added to DeMO, since hybrid models that combine continuous and discrete aspects are becoming more popular. One may consider modeling the flight of a golf ball once struck by a golf club. Let the response vector $\mathbf{y} = [y_0 \; y_1]$ where y_0 indicates the horizontal distance traveled, while y_1 indicates the vertical height of the ball. Future positions of $\mathbf{y}$ depend on the current position and time t. Using Newton's Second Law of Motion, $\mathbf{y}$ can be estimated by solving a system of Ordinary Differential Equations (ODEs) such as

  $$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t), \; \mathbf{y}(0) = \mathbf{y}_0.$$

  The object uses the Dormand-Prince ODE solver to solve this problem. More accurate models for estimating how far a golf ball will carry when struck by a driver can be developed based on inputs/factors such as club head speed, spin rate, smash factor, launch angle, dimple patterns, ball compression characteristics, etc. There have been numerous studies of this problem, including [25].

In addition to these main modeling paradigms, ScalaTion support a simpler approach called **Tableau-Oriented Models** that can be thought of as an automated analog to *Spreadsheet Simulation*.

One may also classify models as supporting **Object-Oriented Simulation** where the advantages of object-oriented programming are utilized. For example, some forms of process-oriented simulation (e.g., GPSS) require each process to strictly follow a sequence of blocks. This makes it easier to create models (and they can be fully specified in a GUI), but reduces the flexibility with which simulation can be built (see [155] for details). In ScalaTion, all of the simulation modeling techniques take advantage of object-orientation, especially the process-interaction models. The importance of each actor running concurrently is the reason it falls in the process-oriented category. In the sense that it has active entities that are objects and concurrent, puts it in the lineage of Simula-67.

17.2 List Processing

Event-Oriented Models will require events and entities to be maintained in various types of lists or queues.

17.2.1 FCFS Queue

The most common type of queue is a First-Come, First-Serve (FCFS) Queue also known as a First-In, First-Out (FIFO) Queue. As a data structure, they may be implemented as an array or linked list. Efficient access to the front (head) and back (tail) of the queue are needed, as well as efficient methods for adding a item to the back of the queue (`enqueue`) and removing an item from the front of the queue (`dequeue`).

The `Queue` class in the `scala.collection.mutable` package provides these capabilities. It extends the `ArrayDeque` class from the same package. The `ArrayDeque` class is implemented using a resizable circular array that allows efficient operations to both ends (front and back) of the queue. `Deque` stands for double ended queue, see Figure 17.1.

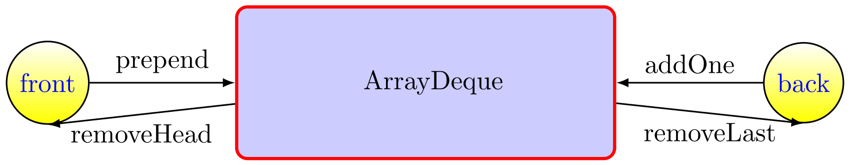


Figure 17.1: Efficient Access to the Front and Back of `ArrayDeque`

The following methods are provided by `ArrayDeque` for adding and removing items from the ends of a queue.

- `prepend` (alias `+:`) - add an item to the front of the queue.
- `addOne` (alias `+=`) - add an item to the back of the queue.
- `removeHead` - remove the item at the front of the queue.
- `removeLast` - remove the item at the back of the queue.

The `Queue` class implements its `enqueue` (with alias `+=`) and its `dequeue` methods as follows:

```
def enqueue (elem: A): this.type = this += elem        // join the back of the line (addOne)

def dequeue (): A = removeHead ()                      // remove from front
```

Note, specifying `this.type` means `q.enqueue (elem)` returns the type of `q` which could be `Queue` or a subclass of `Queue`, e.g., `MyQueue extends Queue`.
See `http://scalada.blogspot.com/2008/02/thistype-for-chaining-method-calls.html` for an example.

17.2.2 LCFS Queue

Another type of queue is a Last-Come, First-Serve (LCFS) Queue also known as a Last-In, First-Out (LIFO) Queue. As a data structure, they may also be implemented as an array or linked list. Efficient access to the front (top) of the queue is needed, as well as efficient methods for adding a item to the front of the queue (`push`) and removing an item from the front of the queue (`pop`).

The `Stack` class in the `scala.collection.mutable` package provides these capabilities. It slso extends the `ArrayDeque` class from the same package.

The `Stack` class implements its `push` and its `pop` methods as follows:

```
def push (elem: A): this.type = prepend (elem)         // join at the front

def pop (): A = removeHead ()                          // remove fron front
```

17.2.3 Priority Queue

Priority Queues are essential for simulation engines as events need to be efficiently placed in a data structure in time order. The time being when the event (e.g., next arrival or service completion) is to occur. How far in the future the event is to occur determines its placement in the priority queue. In this way events are processed in time order. The most imminent event is the one at the front of the queue. When it is removed, the *simulation clock* is advanced to the time of this event.

ScalaTion uses the `PriorityQueue` class in the `scala.collection.mutable` package for scheduling events. This class supports adding and removing items in logarithmic time by having an interval *heap* data structure. A heap may be implemented using a resizable array for which the heap order is maintained, i.e., `heap(i) <= heap(2*i)` and `heap(i) <= heap(2*i+1)`. New events are added to the `eventList` as follows:

```
eventList += anEvent
```

Note, `+=` is alias for the `addOne` method. The most imminent event is removed from the `eventList` as follows:

```
nextEvent = eventList.dequeue ()
```

17.2.4 Time Advance Mechanism

ScalaTion supports two types of event-oriented simulation modeling paradigms: Event Scheduling and its extension, called Event Graphs. For both paradigms, the state of the system only changes at discrete event times with the changes specified via event logic. Time is advanced (jumps forward) to the event time of the next event, so unlike discrete-time models, discrete-event models have non-constant time increments.

(1) Event Scheduling models encode the event logic in classes that extend the `Event` class. The event logic is written in each such class' customized `occur` method. A *scheduler* within the model will execute the events in time order. A time-ordered priority queue is used to hold the future events and is often referred to as a Future Event List (F.E.L.), see Figure 17.2.

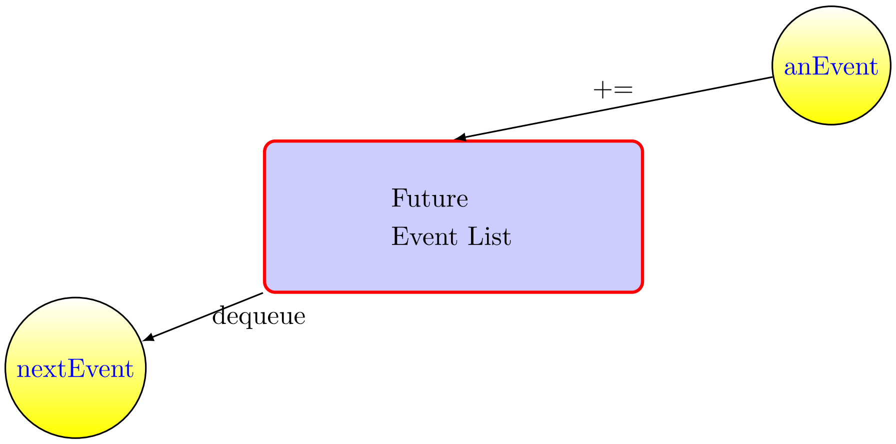


Figure 17.2: Time-Ordered Priority Queue: Future Event List (F.E.L.)

(2) Event Graph models capture the event logic related to triggering other events in causal links. In this way, Event Graph models are more declarative (less procedural) than Event Scheduling models. They also facilitate a graphical representation and animation. They can also serve as a design diagram for event scheduling as they depict the relationships between events.

17.3 Event Scheduling

A simple, yet practical way to develop a simulation engine to support discrete-event simulation is to implement event-scheduling. This involves creating the following three classes: `Event, Entity` and `Model`. An `Event` is defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation. An `Entity`, such as a customer in a bank, flows through the simulation. The `Model` serves as a container/controller for the whole simulation and carries out scheduling of event in time order. The centerpiece class for event scheduling and the one model developers will be most concerned with is the `Event` class.

17.3.1 Event Class

The abstract `Event` class provides a framework for defining types of simulation events. A subclass extending `Event` needs to be created for each type of event.

A subclass (e.g., `Arrival`) of `Event` must provide event-logic in the implementation of its `occur` method. The `Event` class also provides methods for comparing activation times (`actTimes` for events and converting an event to its string representation. Note: unique identification is mixed in via the `Identifiable` trait.

Class Methods:

```
@param entity    the entity involved in this event
@param director  the controller/scheduler that this event is a part of
@param delay     the time delay before this event's occurrence
@param stat      the object for collecting statistics about delay times
@param proto     the prototype (serves as node in animation) for this event

abstract class Event (val entity: Entity, director: Model, delay: Double = 0.0,
                      stat: Statistic = null, val proto: EventNode = null)
         extends Identifiable with Ordered [Event]:

def compare (ev: Event): Int = ev.actTime compare actTime
def cancel (): Unit = { _live = false }
def live: Boolean = _live
def occur (): Unit
override def toString: String = entity.toString + "\t" + me
```

An Event must be defined inside a `Model` referenced by `director` and have an `Entity` that is involved in this event.

An important field in the `Event` class is `actTime`, which indicates the activation/occurrence time for the event.

The methods in this class perform the following functions:

- The `compare` method compares the `actTime` of two events, thus allowing events to be placed in time order in the F.E.L. (`eventList`). Since Scala's `PrioityQueue` class is organized as Highest Priority First (HPF), the logic of the above `compare` method is reversed.

- The `cancel` method allows scheduled events to be cancelled by marking them as not live.
- The `live` method returns whether an event has been cancelled.
- The `occur` method must be implemented in each subclass and it captures the event logic for a particular type of event (e.g., `Arrival`). The method may (1) schedule other events and (2) specify state changes.
- The `toString` method converts internal information about an event into a string.

A frequently used method from the `Model` class is `schedule`. It will place the `event` on the `eventList` in time order. The `eventList` is managed by the `Model`.

```
def schedule (event: Event): Unit =
```

Much of what is needed to develop simply event scheduling models has now been covered.

17.3.2 Example: Bank Model

To create a simple bank simulation model, one could use the classes defined in the event-scheduling engine to create

- two subclasses of `Event`, called `Arrival` and `Departure`, and
- one subclass of `Model`, called `BankModel`.

The complete code for this Bank simulation example may be found in the `scalation.simulation.event` package in `Bank.scala`.

The event logic is coded in the `occur` method which in general triggers future events and updates the current state. It indicates what happens when the event occurs.

Arrival Class

The `Arrival` case class extends the abstract `Event` class. The class constructor takes two parameters: The entity involved in the event and the time delay (how far in the future this event is to occur). These parameters are passed into the base class (`Event`) along with the `Model` director (`this`) and a `Statistic` object for recording statistics on inter-arrival times `t_ia_stat`.

```
@param customer  the entity that arrives, in this case a bank customer
@param delay     the time delay for this event's occurrence

case class Arrival (customer: Entity, delay: Double)
     extends Event (customer, this, delay, t_ia_stat):

    def occur (): Unit = ???

end Arrival
```

Before implementing the logic of the `occur` methods, it is useful to create an Event Graph design diagram (event if Event Graphs are not used for the implementation). Each type of event is depicted as a node and the directed edges indicate event causality. For the Bank Model, two event types will suffice (`Arrival` and `Departure`). There are three causal links (directed edges): an arrival event triggers the next arrival and it may trigger a service completion/departure event, and a departure event may trigger the next service completion event. Typically, the directed edges have conditions on them (i.e., the event is only triggered when the condition is true). The events are also triggered to occur in the future based on the transition/delay times associated with edges. Figure 17.3 depicts the event graph to the Bank Model (the delays time are not placed in the graph, but in the caption to avoid clutter).

The first condition $nArr < nStop - 1$ will be true when the simulation stopping rule becomes true, the second condition $nIn = 0$ being true allows the arriving customer to go directly into service as the server is not busy, and the third condition allows a customer in the queue to begin service, by scheduling the end the service activity.

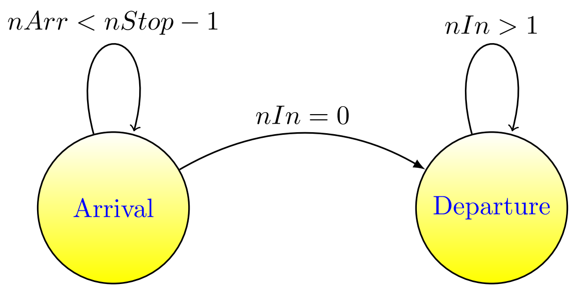


Figure 17.3: Bank Event Graph: with edge delay times of t_{ia}, t_s, and t_s, going left to right

The `occur` method will schedule the next arrival event (up to the limit), check to see if the teller is busy. If so, it will place itself in the Wait Queue (W.Q.), otherwise it schedules its own departure to correspond to its service completion time. Finally, it adjusts the state by incrementing both the number of arrivals (`nArr`) and the number in the system (`nIn`).

```
def occur (): Unit =
    if nArr < nStop-1 then
        val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel.this)
        schedule (Arrival (toArrive, toArrive.iArrivalT))
    end if
    if nIn == 0 then
        schedule (Departure (customer, customer.serviceT))
    else
        waitQueue.enqueue (customer)                  // collects time in Queue statistics
    end if
    nArr += 1                                         // update the current state
    nIn  += 1
end occur
```

Suppose `nStop = 3` and that three arrival events, A_0, A_1, A_2, with corresponding customers C_0, C_1, C_2 occur before any departure events D_0, D_1, D_2. The table below shows the situation at the end of each `occur` method.

Table 17.1: Example Event Execution

Event	**F.E.L.**	**W.Q.**	**State**
start	$[A_0]$	$[]$	$[0, 0]$
A_0.occur	$[A_1, D_0]$	$[]$	$[1, 1]$
A_1.occur	$[A_2, D_0]$	$[C_1]$	$[2, 2]$
A_2.occur	$[D_0]$	$[C_1, C_2]$	$[3, 3]$
D_0.occur	$[D_1]$	$[C_2]$	$[3, 2]$
D_1.occur	$[D_2]$	$[]$	$[3, 1]$
D_2.occur	$[]$	$[]$	$[3, 0]$

Departure Class

For the `Departure` class, the `occur` method will check to see if there is another customer waiting in the queue and if so, schedule that customer's departure. It will then signal its own departure by updating the state; in this case decrementing `nIn` and incrementing `nOut`.

```
 @param customer  the entity that departs, in this case a bank customer
 @param delay     the time delay for this event's occurrence

case class Departure (customer: Entity, delay: Double)
     extends Event (customer, this, delay, t_s_stat):

    def occur (): Unit =
        leave (customer)                                 // collects time in sYstem statistics
        if ! waitQueue.isEmpty then                      // nIn > 1
            val nextService = waitQueue.dequeue ()       // first customer in queue
            schedule (Departure (nextService, nextService.serviceT))
        end if
        nIn  -= 1                                        // update the current state
    end occur

end Departure
```

BankModel Class

The `BankModel` class defines a simple Event-Scheduling model of a Bank where service is provided by one teller and models an M/M/1 queue. The parameters to the constructor are the name of the simulation model, the number of independent replications to run (see the Simulation Output Analysis Chapter for details), the number of entities to create before stopping the simulation, and the base random number stream to use. Each random variate should use a difference stream for independence. The models need to initialize the model constants, create the random variates and state variables, specify the event logic in subclasses for `Event`, in this case, the `Arrival` and `Departure` events defined in the previous subsections, and finally, start the simulation after scheduling the first priming event. Once the simulation stops, reports and summarizes may be output.

```
@param name    the name of the simulation model
@param reps    the number of independent replications to run
@param nStop   the number arrivals before stopping
@param stream  the base random number stream (0 to 999)

class BankModel (name: String = "Bank", reps: Int = 1, nStop: Int = 100, stream: Int = 0)
      extends Model (name, reps):

    // Initialize Model Constants

    val lambda = 6.0                                 // customer arrival rate (per hr)
    val mu     = 7.5                                 // customer service rate (per hr)
```

```
        // Create Random Variables (RVs)

        val iArrivalRV = Exponential (HOUR / lambda, stream)
        val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)

        // Create State Variables

        var nArr      = 0.0                               // number of customers that have arrived
        var nIn       = 0.0                               // number of customers in the bank

        val t_ia_stat = new Statistic ("t_ia")            // time between Arrivals statistics
        val t_s_stat  = new Statistic ("t_s")             // time in Service statistics
        val waitQueue = WaitQueue (this)                  // waiting queue that collects stats
        addStats (t_ia_stat, t_s_stat)

        // Specify Logic for each Type of Simulation Event

        case class Departure ...

        case class Arrival ...

        // Start the simulation after scheduling the first priming event

        val firstArrival = Entity (iArrivalRV.gen, serviceRV.gen, this)
        schedule (Arrival (firstArrival, firstArrival.iArrivalT))     // first priming event
        simulate ()                                                   // start simulating

        report (("nArr", nArr), ("nIn", nIn))
        reportStats ()
        waitQueue.summary (nStop)

    end BankModel
```

Note, to aid with debugging, it may be useful to add the following state variable.

```
        var nOut      = 0.0                               // number of customers that departed
```

The three code segments may now be merged together and compiled. The following imports are required: `scalation.mathstat.Statistic`, `scalation.random.Exponential`, and `scalation.random.RandomSeeds.N_STREAMS`. Outside of SCALATION for example in `my_scalation`, the following import is also required: `scalation.simulation.event`.

Model Execution

The `BankModel` can be invoked by calling a main function, such as `runBank`.

```
    @main def runBank (): Unit = new BankModel ()
```

This function may be executed in `sbt` using `runMain`

```
    > runMain scalation.simulation.event.example_1.runBank
```

Defining Simulation Scenarios

Some of the model preamble (before the `Event` subclasses) may be moved out of the `BankModel` to allow trying multiple combinations of model parameters/constants.

```
@main def runBank2 (): Unit =
    val mu = 7.5
    for lambda <- 5 to 8 do new BankModel2 (lambda, mu)
end runBank2

class BankModel2 (lambda: Double = 6.0, mu: Double = 7.5,
                  name: String = "Bank", reps: Int = 1, nStop: Int = 100, stream: Int = 0)
      extends Model (name, reps):
```

The scenario specification can also be extended to include the random variates as well, allowing different arrival and service distributions to be readily tested.

Statistic Class

In order to collect statistical information, the constructor of the `Event` class calls the `tally` method from the `Statistic` class in the `scalation.mathstat` package to obtain statistics on

- the **t**ime in **q**ueue `t_q_stat` (T_q),
- the **t**ime in **s**ervice `t_s_stat` (T_s), and
- the **t**ime in s**y**stem `t_y_stat` (T_y).

```
if stat != null then stat.tally (delay)
```

Class Methods:

```
@param name      the name for this statistic (e.g., 'waitingTime')
@param unbiased  whether the estimators are restricted to be unbiased

class Statistic (val name: String = "stat", unbiased: Boolean = false):

def set (n_ : Int, sum_ : Double, sumAb_ : Double, sumSq_ : Double,
         minX_ : Double, maxX_ : Double): Unit =
def reset (): Unit =
def tally (x: Double): Unit =
inline def num: Int = n
inline def nd: Double = n.toDouble
inline def min: Double = if n == 0 then 0.0 else minX
inline def max: Double = maxX
def mean: Double = if n == 0 then 0.0 else sum / nd
def variance: Double =
def stdev: Double = sqrt (variance)
def ms: Double = sumSq / nd
def ma: Double = sumAb / nd
def rms: Double = sqrt (ms)
def interval (p: Double = .95): Double =
def interval_z (p: Double = .95): Double =
def show: String = s"Statistic: $n, $sum, $sumAb, $sumSq, $minX, $maxX"
def statRow: Array [Any] = Array (name, num, min, max, mean, stdev, interval ())
override def toString: String =
```

Monitor Class

The `Monitor` class is used to trace the key actions in the execution of a model. This class works for both event oriented and process oriented model. Information collected by calling the `trace` method is saved in log file.

Class Methods:

```
@param project  the project to be monitored

case class Monitor (project: String = "simulation"):

def toggle (): Unit = ew.toggle ()
def traceOff (): Unit = tracing = false
def traceOn (): Unit = tracing = true
def trace (who: Identifiable, what: String, whom: Identifiable, when: Double): Unit =
def finish (): Unit = ew.finish ()
```

Model developers may call `trace` anywhere in their code. The `Model` class calls `trace` mutilple times,

```
private [event] val log  = Monitor ("simulation")
log.trace (this, "starts", this, _clock)
log.trace (this, s"executes ${nextEvent.me} on ${nextEnt.eid}", nextEnt, _clock)
log.trace (this, "terminates", this, _clock)
```

The traces are are written into a file located in for example `log/simulation` directory. If this directory does not exist, it will need to be made for the model to run. Both `scalation` and `my_scalation` have this directory.

17.3.3 Example: Call Center Model

A simpler model is one for which there is no waiting queue. If a server is free/idle, service may begin, otherwise, it can be attempted later. A simple call center can be modeled in this way. The `CallCenter.scala` file contains event logic for the case of a call center with a staff of one. The Event Graph design diagram is similar the one for the Bank Model. The major difference is that upon departure, as there is no queue, the ending call cannot trigger the beginning of the next completed call, i.e., the `Departure` event does not trigger any other events, it just updates the state.

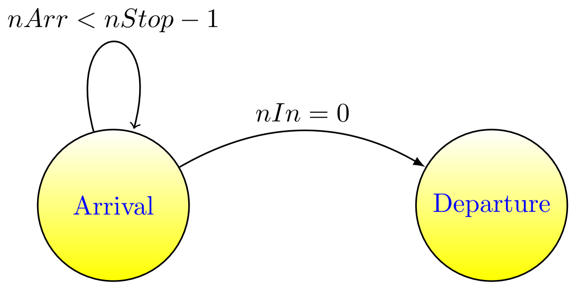


Figure 17.4: Call Center Event Graph: with edge delay times of t_{ia} and t_s, going left to right

Arrival Class

The `Arrival` class replaces waiting in a queue with incrementing a counter of the number of lost calls (`nLost`).

```
@param call   the entity that arrives, in this case a call
@param delay  the time delay for this event's occurrence

case class Arrival (call: Entity, delay: Double)
     extends Event (call, this, delay, t_a_stat):

    def occur (): Unit =
        if nArr < nStop-1 then
            val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, CallCenterModel.this)
            schedule (Arrival (toArrive, toArrive.iArrivalT))
        end if
        if nIn == 0 then
             schedule (Departure (call, call.serviceT))
        end if
        nArr += 1                                      // update the current state
        if nIn == 1 then nLost += 1 else nIn = 1
    end occur

end Arrival
```

Departure Class

The logic for the `Departure` class simply records information and updates counters.

```
@param call   the entity that departs, in this case a call
@param delay  the time delay for this event's occurrence

case class Departure (call: Entity, delay: Double)
     extends Event (call, this, delay, t_s_stat):

    def occur (): Unit =
        leave (call)                                     // collects time in sYstem statistics
        nIn   = 0                                        // update the current state
        nOut += 1
    end occur

end Departure
```

The remaining four of five classes used for creating simulation models following the Event Scheduling paradigm are discussed in the next four subsections.

17.3.4 Entity Class

Conceptually, entities are the dynamic objects that populate a simulation model, e.g., customers in a bank simulation or calls in a call center simulation. An instance of the `Entity` class represents a single simulation entity for event oriented simulation. For each instance, it maintains information about that entity's arrival time and next service time.

Class Methods:

```
@param iArrivalT  the time from the last arrival
@param serviceT   the amount of time required for the entity's next service
@param director   the controller/scheduler that this event is a part of

case class Entity (val iArrivalT: Double, var serviceT: Double, director: Model)
     extends Identifiable:

override def toString = "Entity-" + eid
```

Important fields in the `Entity` class are the entity id `eid`, the time at which the entity arrived `arrivalT = director.clock + iArrivalT`, and the time it begins waiting `startWait`, if applicable.

In the `BankModel`, a new entity (called `toArrive`) for a future arrival event is created before being passed to the event.

```
val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel.this)
schedule (Arrival (toArrive, toArrive.iArrivalT))
```

17.3.5 WaitQueue Class

When entities are unable to begin service immediately, they are often placed in a wait queue. The `WaitQueue` class provides a First-Come, First-Served (FCFS) queue and is implemented by extending Scala's `Queue` class. An entity is added to the back of the queue using the `enqueue` method and is removed from the front of the queue using the `dequeue` method. By default it has infinite capacity, but may be restricted by passing a value in the `cap` parameter. In this case, entities are barred from entering the queue when the queue is full. The number of times it is called is returned by the `barred` method.

```
override def enqueue (ent: Entity): WaitQueue.this.type =
    ent.startWait = director.clock
    if length <= cap then super.enqueue (ent) else _barred += 1
    this
end enqueue
```

The `dequeue` method collects both sample statistics (for T_q) and time-persistent statistics (for L_q) on queue occupancy, see the exercises.

```
override def dequeue (): Entity =
    val ent = super.dequeue ()
    val timeInQ = director.clock - ent.startWait
    waitTimes += timeInQ
    director.log.trace (director, s"records $timeInQ wait for ${ent.eid}", ent,
                        director.clock)
    t_q_stat.tally (timeInQ)
    l_q_stat.accum (length + 1, director.clock)
    ent
end dequeue
```

The `summary` method returns statistics about waiting times in the queue.

Class Methods:

```
@param director  the controller/scheduler that this event is a part of
@param ext       the extension to distinguish the wait queues
@param cap       the capacity of the queue (defaults to unbounded)

case class WaitQueue (director: Model, ext: String = "", cap: Int = Int.MaxValue)
     extends Queue [Entity]:

def barred: Int = _barred
def isFull: Boolean = length >= cap
override def enqueue (ent: Entity): WaitQueue.this.type =
override def dequeue (): Entity =
def summary (numEntities: Int): Unit =
```

17.3.6 WaitQueue_LCFS Class

Again, entities in the model that are unable to begin service immediately are often placed in a wait queue. The `WaitQueue_LCFS` class provides a Last-Come, First-Served (LCFS) queue and is implemented by extending Scala's `Stack` class. An entity is added to the front (top) of the queue using the `enqueue` method and is removed from the front (top) of the queue using the `dequeue` method. By default it has infinite capacity, but may be restricted by passing a value in the `cap` parameter. In this case, entities are barred from entering the queue, and need to stay where they are, go somewhere else or be lost to the simulation. The number of times entities are barred is returned by the `barred` method. The `summary` method returns statistics about waiting times in the LCFS queue.

Class Methods:

```
@param director  the controller/scheduler that this event is a part of
@param ext       the extension to distinguish the wait queues
@param cap       the capacity of the queue (defaults to unbounded)

case class WaitQueue_LCFS (director: Model, ext: String = "", cap: Int = Int.MaxValue)
     extends Queue [Stack]:

def barred: Int = _barred
def isFull: Boolean = length >= cap
def enqueue (ent: Entity): WaitQueue.this.type =
def dequeue (): Entity =
def summary (numEntities: Int): Unit =
```

See the exercises for how to add additional types of wait queues.

17.3.7 Model Class

As shown in the examples, the foundation class for event-oriented simulation models is the `Model` class. An application model will extend this class and create an instance called `director`. The `director` schedules events and implements the time advance mechanism for event-oriented simulation models. Two important methods in the `Model` class are the `schedule` and `simulate` methods. Scheduled events are placed in the Future Event List (F.E.L.) in time order. In ScalaTion, the F.E.L. is called the `eventList` and is implemented using `Scala`'s priority queue.

- The `schedule` method is used to add events to the `eventList` so they may be processed in time order. To preempt a scheduled event, the `cancel` method may be called.
- The `simulate` method repeatedly removes and processes events in the `eventList`. The `simulate` method will cause the main simulation loop to execute, which will remove the most imminent event from the `eventList` and invoke its `occur` method.

The main loop in the `Model` class is the following `while` with the line controlling animation removed.

```
    while simulating && ! eventList.isEmpty do
         nextEvent = eventList.dequeue ()
         if nextEvent.live then
             _clock  = nextEvent.actTime
             nextEnt = nextEvent.entity
             log.trace (this, s"executes ${nextEvent.me} on ${nextEnt.eid}", nextEnt, _clock)
             debug ("simulate", s"$nextEvent \t" + "%g".format (_clock))
             nextEvent.occur ()
         end if
     end while
```

The simulation will continue until a stopping rule evaluates to true (either the `simulating` flag becomes false or the `eventList` becomes empty). The `nextEvent` is removed from the front of the priority queue and checked to make sure it is still `live` (not cancelled). If it is `live`, the director's simulation clock is set to the this event's activation time `actTime`. For tracing purposes, this event's associated entity is referenced. Then this event's `occur` method is called that carries out the event logic based on the type of event it is.

Methods to `getStatistics` and `report` statistical results are also provided.

Class Methods:

```
@param name       the name of the model
@param animation  whether to animate the model (only for Event Graphs)

class Model (name: String, animating: Boolean = false)
      extends Modelable with Identifiable:

def addStats (stat: Statistic*): Unit = for st <- stat do stats += st
def schedule (event: Event): Unit =
def cancel (event: Event): Unit = event.cancel ()
def leave (entity: Entity): Unit = t_y_stat.tally (clock - entity.arrivalT)
def simulate (startTime: Double = 0.0): Unit =
def report (vars: (String, Double)*): Unit =
def reportStats (): Unit =
def getStatistics: ListBuffer [Statistic] = stats
def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape, at: Array [Double]): Unit =
def animate (who: Identifiable, what: CommandType, color: Color,
             shape: Shape, from: Event, to: Event, at: Array [Double] = Array ())
```

The `animate` methods are used with Event Graphs (see the next section).

As a starting point for developing simulation models following the event scheduling paradigm, a model developer may copy the `Ex_Template.scala` file in the `scalation.simualation.event` package.

17.3.8 Example: Machine Shop Model

Before ending this section, consider the following more complex model: Parts arrive at a two-stage machine shop with an arrival rate of $\lambda = 10 \text{ hr}^{-1}$. The first machine reshapes the part, while the second machine polishes the part. The service rate for the first machine is $\mu_1 = 12$ hr^{-1} and is $\mu_2 = 15$ hr^{-1} for the second machine. Both machines have space to store three parts waiting to be machined. When there is a backup of parts, flow from other departments must continue, so these parts are removed to be sold for scrap. The company wishes to conduct a simulation study to see which policy is better.

- **Policy One**: Block machine 1 when machine 2's queue is full.
- **Policy Two**: Do not block machine 1 when machine 2's queue is full, rather send the partially finished part to scrap.

The cost of a raw part is 100 dollars, the value of partially finished part is 50 dollars and value of finished part is 200 dollars. The operational cost of machine is 60 dollars an hour and is 30 dollars for machine 2.

The management has argued that machine 1 should be blocked since 50 dollars are lost whenever a partially finished part is sold for scrap, while if it a raw part it can be resold for 100 dollars. Others say stopping/forced idling of machine 1 is costly.

Figure 18.1 shows the flow of parts through the machine shop. Use simulation to estimate performance characteristics and determine the better policy. How sensitive is the decision to the relative service rates of the two machines.

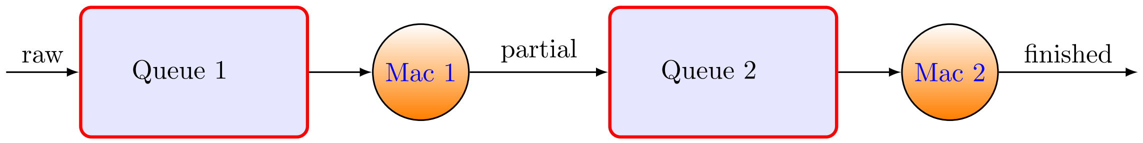


Figure 17.5: Machine Shop Schematic Diagram: Each Queue Has Capacity Three

The `Machine.scala` file in the `event.example_1` package contains a partial implementation of the machine shop model. The Event Graph design diagram now requires three event types: `Arrival`, `FinishMachine1`, and `FinishMachine2`. The Event Graph is depicted in Figure 17.6.

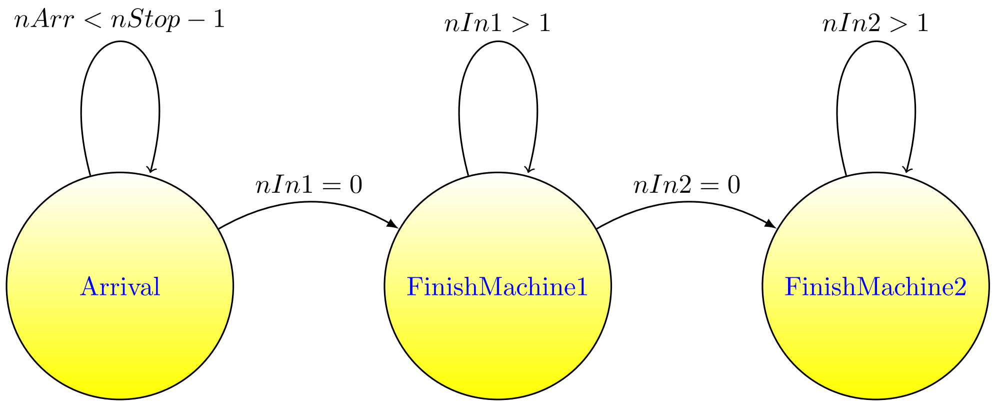


Figure 17.6: Machine Shop Event Graph

The state variable are the following:

- the number of part arrivals at machine 1, `nArr`,
- the current number of parts at machine 1 (queue + service), `nIn1`,
- the current number of parts at machine 2, `nIn2`,
- the number of completed parts, `nOut`,
- the number of scrapped raw parts, `nScrap1`
- the number of scrapped partially finished parts, `nScrap2`

Blocking. Note that one policy may require machine station 1 to be blocked due to no room in machine station 2. The blocking occurs when machine 1 finishes a part and it is ready to move onto machine station 2, but cannot. This part will be stuck at machine 1 and machine 1's operator will be idle, until machine 2 completes the part it is working on. An additional edge should be added to the Event Graph to indicate this causal connection. As the blocked part is not in a queue and not in the F.E.L., it needs to be held somewhere, e.g., in model variable called `heldAtMachine1`.

17.4 Event Graphs

Event Graphs operate in a fashion similar to Event Scheduling. Originally proposed as a graphical conceptual modeling technique (Schruben, 1983) for designing event oriented simulation models, modern programming languages now permit more direct support for this style of simulation modeling.

In ScalaTion, the simulation engine for Event Graphs consists of the following seven classes:
The first five classes are shared with Event Scheduling.

1. An `Entity`, such as a customer in a bank, flows through the simulation.
2. An `Event` represents an instantaneous occurrence that affects the simulation.
3. An `WaitQueue` allows entities to wait for service in a FCFS Queue.
4. An `WaitQueue_LCFS` allows entities to wait for service in an LCFS Queue.
5. The `Model` serves as a container/controller for the whole simulation.

The last two classes are specific to Event Graphs.

1. An `EventNode` (subclass of `Event`), defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation, is represented as a *node* in the event graph.
2. A `CausalLink` emanating from an event/node is represented as an outgoing directed *edge* in the event graph. It represents causality between events. One event can conditionally trigger another event to occur some time in the future.

17.4.1 Example: Bank Model

For example, to create a simple bank simulation, one could use the four classes provided by the Event Graph simulation engine to create subclasses of `EventNode`, called `Arrival` and `Departure`, and one subclass of `Model`, called `BankModel`. The complete code for this example may be found in `Bank2.scala`. In more complex situations, one would typically define a subclass of `Entity` to represent the customers in the bank.

```
class BankModel2 (name: String, nStop: Int, iarrivalRV: Variate, serviceRV: Variate)
      extends Model (name, true)                             // true => animation on
```

The Scala code below was made more declarative than typical code for event-scheduling to better mirror event graph specifications, where the causal links specify the conditions and time delays. For instance,

```
() => nArr < nStop-1
```

is a anonymous function/closure returning `Boolean` that will be executed when arrival events are handled. In this case, it represents a stopping rule; when the number of arrivals exceeds a threshold, the arrival event will no longer schedule the next arrival. The `serviceRV` is a random variate to be used for computing service times.

In the `BankModel` class, the logic in the `Event` classes is simplified somewhat due to the specification of `CauasalLinks`. Before the `Event` classes are specified, the following four types for definitions are required: statistical accumulators, event nodes, causal links/edges, and the state variables.

First the statistical accumulators are defined, one for inter-arrivals and one for service. A wait queue is given and it maintains its own statistics.

```
    val t_ia_stat = new Statistic ("t_ia")               // time between Arrivals statistics
    val t_s_stat  = new Statistic ("t_s")                // time in Service statistics
    addStats (t_ia_stat, t_s_stat)
    val waitQueue = WaitQueue (this)                     // waiting queue that collects stats
```

For animation of the event graph, a prototype for each type of event is created and displayed as a node. Locations are required for each `EventNode`.

```
    val aLoc   = Array (150.0, 200.0, 50.0, 50.0)        // Arrival event node location
    val dLoc   = Array (450.0, 200.0, 50.0, 50.0)        // Departure event node location
    val aProto = new EventNode (this, aLoc)              // prototype for all Arrival events
    val dProto = new EventNode (this, dLoc)              // prototype for all Departure events
```

The edges connecting these prototypes represent the casual links. The `aLink` array holds two causal links emanating from `Arrival`, the first a self link representing triggered arrivals and the second representing an arrival finding an idle server, so it can schedule its own departure. The `dLink` array holds one causal link emanating from `Departure`, a self link representing the departing customer causing the next customer in the waiting queue to enter service (i.e., have its departure scheduled).

```
    val aLink = Array (CausalLink ("l_A2A", this, () => nArr < nStop-1, aProto),
                       CausalLink ("l_A2D", this, () => nIn == 0,       dProto))
    val dLink = Array (CausalLink ("l_D2D", this, () => nIn > 1,        dProto))
    aProto.displayLinks (aLink)
    dProto.displayLinks (dLink)
```

The state variables, `nArr, nIn` and `nOut`, are defined as `vars` since they will change during the simulation.

```
    var nArr   = 0.0                                     // number of customers that have arrived
    var nIn    = 0.0                                     // number of customers in the bank
    var nOut   = 0.0                                     // number of customers that finished & left
```

An animation of the Event Graph consisting of two `EventNodes` `Arrival` and `Departure` and three `CausalLinks` is depicted in Figure 17.7.

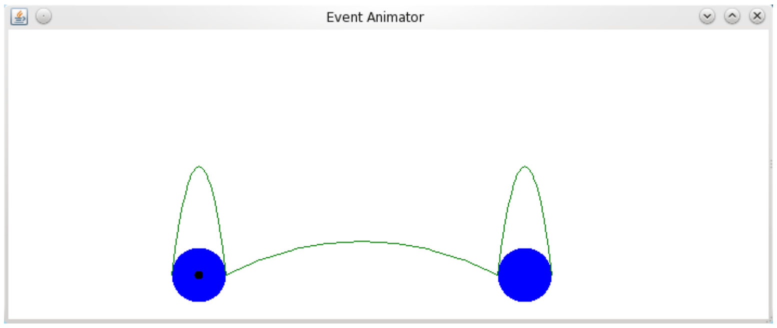


Figure 17.7: Event Graph Animation of a Bank.

The main thing to write within each subclass of Event is the occur method. To handle arrival events, the occur method of the Arrival class first checks the aLinks to see if it needs to trigger additional events. It then updates the current state by incrementing both the number of arrivals (nArr) and the number in the system (nIn).

```
case class Arrival (customer: Entity, delay: Double)
     extends Event (customer, this, delay, t_ia_stat, aProto):

    def occur (): Unit =
        if aLink(0).condition () then
            val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel2.this)
            schedule (Arrival (toArrive, toArrive.iArrivalT))
        end if
        if aLink(1).condition () then
            schedule (Departure (customer, customer.serviceT))
        else
            waitQueue.enqueue (customer)                     // collects time in Queue statistics
        end if
        nArr += 1                                            // update the current state
        nIn  += 1
    end occur

end Arrival
```

To handle departure events, the occur method of the Departure class first checks the dLink to see if it needs to trigger additional events. It then updates the state by decrementing the number in the system (nIn) and incrementing the number of departures (nOut).

```
case class Departure (customer: Entity, delay: Double)
     extends Event (customer, this, delay, t_s_stat, dProto):

    def occur (): Unit =
        leave (customer)                                     // collects time in sYstem statistics
        if dLink(0).condition () then
            val nextService = waitQueue.dequeue ()           // first customer in queue
            schedule (Departure (nextService, nextService.serviceT))
        end if
        nIn  -= 1                                            // update the current state
        nOut += 1
    end occur

end Departure
```

Four of the classes used for creating simulation models following the Event Scheduling paradigm can be used for Event Graphs, namely Entity, Event, Model, and WaitQueue. In addition, EventNode is also required as they form the nodes in the Event Graphs. An edge in the Event Graph is an instance of the CausalLink class. These two new classes (EventNode and CausalLink) are described in the subsections below.

17.4.2 EventNode Class

The `EventNode` class provides facilities for defining simulation events. Subclasses of `Event` provide event-logic in their implementation of the occur method. The main purpose of `EventNode` is to associate a type of event with a node in the event graph.

Class Methods:

```
@param director  the controller/scheduler that this event node is a part of
@param at        the location of this event node

class EventNode (director: Model, at: Array [Double] = Array ())
      extends Event (EventNode.makePrototype (director), director, -1.0, null):

def occur (): Unit = throw new NoSuchMethodException ("this occur should not be called")
def displayLinks (links: Array [CausalLink]): Unit =
```

17.4.3 CausalLink Class

The `CausalLink` class provides casual links between events. Before an event updates the state, it checks its causal links to schedule/cancel other events. Events graphs make the linkage between types of events more explicit, e.g., an arrival triggers the next arrival some time in the future. If a server is available, it can also trigger the entity's own service completion/departure event.

Class Methods:

```
@param label        the name/label of the causal link
@param director     the controller/scheduler that this causal link is a part of
@param condition    the condition under which the link is triggered
@param causedEvent  the event caused by this causal link

case class CausalLink (label: String, director: Model, val condition: () => Boolean,
                       causedEvent: Event)
      extends Identifiable:

def display (from: Event, to: Event): Unit =
override def toString: String = s"CausalLink(${getClass.getSimpleName ()}, $name)"
```

Events graphs support a more declarative means for specifying a simulation, allow the relationships between events to be seen visually and provide basic animation of simulation model execution.

17.5 Exercises

1. It is common practice to implement an `eventList` as a Heap-Based Priority Queue. The most imminent event is at the top of the heap data structure in a priority queue. Processing this event involves removing and returning this event, swapping in the event in the last position in the heap, and reestablishing the heap order. Draw pictures to illustrate what happens to the heap.

2. As mentioned, the `dequeue` in the `WaitQueue` class collects both sample and time-persistent statistics. The `Statistic` class in the `mathstat` packge is used to collect via `tally` sample statistics, while the `TimeStatistic` class is used to collect `accum` time-persistent statistics.

```
@param name        the name for this statistic (e.g., 'numberInQueue' or 'tellerQ')
@param _lastTime   the time of last observation
@param _startTime  the time observation began

class TimeStatistic (override val name: String = "p-stat",
                     private var _lastTime:  Double = 0.0,
                     private var _startTime: Double = 0.0)
      extends Statistic (name):
```

   How do the `tally` and `accum` methods differ? Why is it necessary to have two such methods?

3. Explain what each line of the main `while` loop does in the `simulate` method of the `Model` class.

4. Draw an event graph for the simulation of an M/M/c/K Queue. Let the arrival rate $\lambda = 7$ per hour and the service rate $\mu = 8$ per hour. Write and execute an event scheduling model for this Queue System. Compare the results with the theoretical results from Queueing Theory. Consider the following cases: $c = 1, K = 1$; $c = 1, K = \infty$; $c = 2, K = 10$; $c = 2, K = \infty$.

5. **Wendy's vs. McDonald's Lines**. Given two servers, is it better to have a line/queue for each server or one common line for both servers. Analyze the waiting times and standard deviation of the waiting times. Let $\lambda = 20$ hr^{-1} and $\mu = 12$ hr^{-1}.

   Hint: To compare the two waiting times, compute the adjusted waiting time. For example, with a single queue, suppose W_q is the average time in queue for the n_q customers that had to wait. The overall adjusted waiting time is then

$$T_q \quad = \quad \frac{n_q W_q + (m - n_q)0}{m}$$

   When there are two queues with waiting times W_{q_1} and W_{q_2} the formula becomes

$$T_q \quad = \quad \frac{n_{q_1} W_{q_1} + n_{q_2} W_{q_2} + (m - n_{q_1} - n_{q_2})0}{m}$$

6. **Machine Shop**. Explain the edge conditions given in the Event Graph design diagram for the Machine Shop. Specify the random variates for the transition/delay times for the edges in the Event Graph.

7. **Machine Shop**. Complete the implementation of the Machine Shop simulation and determine which policy is better.

8. **Two-Stage Queueing System Simulation**. Consider modeling a system with two stages of service: In stage one a patient registers, and in stage two the patient receives treatment. The line for registration is unbounded, while patients waiting for treatment must be within a room with a total capacity of K. The first stage has one server, while the second stage has two. Based on the cases in the previous exercise, simulate a two-stage service system where the first stage has a queue with $c = 1, K = \infty$ and the second stage has a queue with $c = 2, K = 10$. When the second queue is full, the server in the first stage will be blocked. (i.e, must be idle until space is available).

9. **Emergency Department Simulation**. Create and execute an event scheduling model for an emergency department/room based on the specifications given in the following paper, "Modeling and Improving Emergency Department Systems using Discrete Event Simulation," by Christine Duguay and Fatah Chetouane, `https://journals.sagepub.com/doi/10.1177/0037549707083111`.

10. ScalaTion uses Scala's Priority Queue class for its time ordered F.E.L. (`eventList`), but that class could also be used for priority based waiting queues. Add a new class to the `event` package called `WaitQueue_PQ` to provide this capability.

11. Use the new `WaitQueue_PQ` class along with `WaitQueue` and `WaitQueue_LCFS` to test various job scheduling algorithms: FCFS, LCFS, Shortest Job First (SJF) and Highest Priority First (HPF).

12. Event Scheduling (ES) Simulation: A **Small Fast Food Restaurant** has two severs and enough space for three customers to wait (at most five customers total at any given time). For the case of a single queue, perform an ES simulation for $m = 20$ customer arrivals. Assume each server can process $\mu = 30$ customers per hour and that the customer arrival rate $\lambda = 75$ customers per hour (assume Exponential distributions). Each completed order gives a net profit (before paying the servers) of 2.00 dollars. Each server makes 11.00 dollars per hour. Should the restaurant hire a third server? Explain in terms of profit (after paying the servers) per hour. **Note**: this simulation problem is posed in the Hand/Spreadsheet Simulation section of the Simulation Foundations Chapter.

    (i) Use ScalaTion's `Known` Random Variate Generator (RVG) to make ES reproduce the results of the Spreadsheet Simulation. Give the Event Graph, code for the `Event` subclasses including their `occur` methods, and the summary results.

    (ii) Replace the `Known` RVG with ScalaTion's `Exponential` RVG in order to run the simulation longer to obtain better results. Also, run multiple replications to produce multiple estimates for the final profit for having two servers vs. three servers. To make the replications independent, make sure each replication uses a different base random number `stream`.

    (iii) Explain how **Confidence Intervals** can be used to make more informed decisions.

    (iv) [**Bonus**] Create 95% Confidence Intervals for the two and three server simulations. Let the number of replications be 10 and use the Student's t Distribution.

13. **Question 3**: Simulation model design: Draw an Event Graph for a simulation model used to study a Bank with two tellers with `Exponential` inter-arrival and service time distributions with rates λ and μ, respectively. Having one line and two servers along with `Exponential` distributions makes this an M/M/2 queue. Explain the nodes and edges in your event graph.

Chapter 18

Process-Oriented Models

A process can be thought of an unit of execution. Conceptually, an active entity in a process-oriented simulation may execute as its own process.

Computers with multiple cores and hyper-threading may execute many processes at once. A system with 16 cores and hyper-threading (2 threads run per core), would allow 32 processes to run in parallel. Actually, since several processes will be waiting on input, the number of processes is much larger that this number would suggest. Still, process-oriented simulation typically will require many more processes, so traditional heavy-weight processes are out of the question. Typically, the choice is between threads and coroutines.

18.1 Base Traits and Classes for Process-Oriented Models

The `simulation` package contains several base traits and classes that can be used by several types of simulation models, and are especially useful for process-oriented simulation models.

18.1.1 Identifiable Trait

The `Identifiable` trait provides unique identification for simulation components, entities and events. Includes a mandatory id and an optional name.

```
trait Identifiable:

def name: String = _name
def name_= (name: String): Unit =
def simType: String = getClass.getSimpleName ()
def me: String = s"$simType.$_name.$id"
override def equals (that: Any): Boolean =
override def hashCode: Int = id
```

18.1.2 Locatable Trait

The `Locatable` trait provides location information/coordinates for objects in simulation models (e.g., `Components`).

```
trait Locatable:

def at: Array [Double] = _at
def at_= (at: Array [Double]): Unit =
```

18.1.3 Modelable Trait

The `Modelable` trait defines the notions of clock and simulate, common to many types of ScalaTion simulation models.

```
trait Modelable:

def clock = _clock
def simulate (startTime: Double): Unit
```

18.1.4 Temporal Trait

The `Temporal` trait adds time (actTime) and temporal ordering to the `Identifiable` trait.

```
trait Temporal
      extends Identifiable:

def compare (other: Temporal): Int = { actTime compare other.actTime }
override def toString: String = s"Temporal ($me, $actTime)"
```

18.2 Concurrent Processing of Actors

A simulation with multiple actors as active entities whose behaviors overlap in time are most naturally implemented using concurrent programming.

Traditionally, programming language support for concurrent programming has been limited and this makes providing support of process-oriented models more difficult. Typically, support for *coroutines* is sufficient for developing simulation engines of this type. Languages supporting coroutines include: Simula, Smalltalk, Modula-2, Ruby, Julia, Go, and Kotlin.

On the other hand, many languages support *threads*, notably Java and therefore all Java based languages including Scala. Although threads are capable of getting the job done, they introduce two problems: Unusual transfer of control between threads can lead to challenging bugs to eliminate. There is also more overhead (time and space) required to use threads rather than coroutines. To deal with the first problem, ScalaTion implements a `Coroutine` class using Java's `Runnable` interface and `Thread` class. Then users of ScalaTion can avoid the complexity and bugs associated with threads. Improvement on the problem of overhead will be provided by Java 18 in the form of `VirtualTheads`. These run in user space with reduced overheads and can allow may more actors to run concurrently.

18.2.1 Java's Thread Class

The `PingPong` Java class provides a simple example of how Java threads work. The class implements the `Runnable` interface allowing the `run` method of object instances of this class to be executed concurrently in multiple `Threads`. In this case, two instances are created: the first instance that writes "ping" and sleeps for 333 milliseconds, and the second instance that writes "PONG" and sleeps for 1000 milliseconds.

The following are some of the commonly used methods available in Java Threads: `currentThread` (the currently running thread), `interrupt` (receive an InterruptedException), `join` (waits for thread to terminate), `run` (execute the overridden method concurrently) `sleep` (temporarily cease execution for the given number of milliseconds), `start` (the thread begins execution and calls the `run` method), and `yield` (this threads offers to give another thread a chance to run). See the on-line Java API Documentation for more details.

```
import static java.lang.System.*;

public class PingPong implements Runnable
{
    private String word;
    private int    delay;

    /*************************************************************
     * Construct a ping or pong object.
     */
    public PingPong (String whatToSay, int delayTime)
    {
        word  = whatToSay;
        delay = delayTime;
    } // PingPong
    /*************************************************************
     * Run method for the ping/pong object.
     */
    public void run ()
    {
        for ( ; ; ) {
            out.println (word);
            try {
                Thread.sleep (delay);
            } catch (InterruptedException ex) {
                return;
            } // try
        } // for
    } // run

    /*************************************************************
     * Main method for invoking the application.
     * @param  args  Command-line arguments
     */
    public static void main (String [] args)
    {
        new Thread (new PingPong ("ping", 333)).start ();
        new Thread (new PingPong ("PONG", 1000)).start ();
    } // main

} // PingPong
```

This code can be compiled typing the following:

```
$ javac PingPong.java
$ java PingPong
```

Notice the interleaved execution. Had the `run` been called directly, rather than indirectly via calling the `start` method, no interleaving would be seen, since this would require the first thread to finish before the second one can begin. Try change "start" to "run" in the code above. Note, to terminate the program, type "Ctrl C".

18.2.2 ScalaTion's Coroutine Class

The `Coroutine` class supports (one-at-a-time) quasi-concurrent programming. A coroutine runs/acts until it yields control from 'this' to 'that' coroutine. When resumed, a coroutines continues its execution where it left off. As in the `PingPong` class, the `Coroutine` class implements (extends in Scala) `Runnable` interface. The `run` method keeps track of the number of coroutines started and terminated as well as delegates to the `act` method. This abstract method must be implemented by the simulation model developer. The `start` method causes the `run` method to be executed concurrently, which delegates to the `act` method (that contains the simulation logic).

```
@param label  the label for the class of coroutines to be created.
```

```
abstract class Coroutine (label: String = "cor")
     extends Runnable:

def counts: (Int, Int, Int) = (nCreated, nStarted, nTerminated)
def run (): Unit =
def act (): Unit
def yyield (that: Coroutine, quit: Boolean = false): Unit =
def start (): Future [_] =
def interrupt (): Unit =
```

The `Coroutine` also uses the following from the `java.util.concurrent` package: Executors, ExecutorService, Future, Semaphore, and ThreadPoolExecutor. See `Coroutine.scala` in the `scalation.simulation` package for details.

18.3 Process Interaction

Many discrete-event simulation models are written using the process-interaction world view, because the code tends to be concise and intuitively easy to understand. Take for example the process-interaction model of a bank (`BankModel` a subclass of `Model`) shown later. Following this world view, one simply constructs the simulation components and then provides a script for entities (`cimActors`) to follow while in the system. In this case, the `act` method for the customer class provides the script (what entities should do), i.e., enter the bank, if the tellers are busy wait in the queue, then receive service and finally leave the bank.

The development of a simulation engine for process-interaction models is complicated by the fact that concurrent (or at least quasi-concurrent) programming is required. Various language features/capabilities from lightweight to middleweight include continuations, coroutines, fibers, actors, virtual-threads and threads. Heavyweight concurrency via OS processes is infeasible, since simulations may require a very large number of concurrent entities. The main requirement is for a concurrent entity to be able to suspend its execution and be resumed where it left off (its state being maintained on a stack). Since preemption is not necessary, lightweight concurrency constructs are ideal. Presently, ScalaTion uses the `Coroutine` class.

ScalaTion includes several types of model components: `Gate, Junction, Resource, Route, Sink, Source, Transport, WaitQueue` and `WaitQueue_LCFS`. A model may be viewed as a directed graph with several types of nodes:

- `Gate`: a gate is used to control the flow of entities, they cannot pass when it is shut.
- `Junction`: a junction is used to connect two transports.
- `Resource`: a resource provides services to entities (typically resulting in some delay).
- `Sink`: a sink consumes entities.
- `Source`: a source produces entities.
- `WaitQueue`: a FCFS wait-queue provides a place for entities to wait, e.g., waiting for a resource to become available or a gate to open.
- `WaitQueue_LCFS`: an LCFS wait-queue provides a place for entities to wait, e.g., waiting for a resource to become available or a gate to open.

These nodes are linked together with directed edges (from, to) that model the flow entities from node to node. A `Source` node must have no incoming edges, while a `Sink` node must have no outgoing edges.

- `Route`: a route bundles multiple transports together (e.g., a two-lane, one-way street).
- `Transport`: a transport is used to move entities from one component node to the next.

The model graph includes coordinates for the component nodes to facilitate animation of the model. Coordinates for the component edges are calculated based on the coordinates of its from and to nodes. Small colored tokens move along edges and jump through nodes as the entities they represent flow through the system.

Formally, it is not required to be a graph since entities can move from node to node without going along an edge (e.g., `WaitQueue` to `Resource`). These graph-like structures are referred to as *Network Diagram*. ScalaTion's Agent-Based Simulation (see the next section), however, does require the model to be based on an underlying *Property Graph*.

18.3.1 Model Template

All process-interaction simulation models in ScalaTion are of the following basic form. The file called Ex_Template.scala in the scalation.simulation.process package may be used as a starting template for the development of specific process-interaction simulation models.

```
@param name       the name of the simulation model
@param reps       the number of independent replications to run
@param animating  whether to animate the model
@param aniRatio   the ratio of simulation speed vs. animation speed
@param nStop      the number arrivals before stopping
@param stream     the base random number stream (0 to 999)

class SOMEModel (name: String = "SOME", reps: Int = 1, animating: Boolean = true,
                 aniRatio: Double = 8.0, nStop: Int = 100, stream: Int = 0)
      extends Model (name, reps, animating, aniRatio):

    // Initialize Model Constants

    val lambda = 6.0                                  // customer arrival rate (per hour)

    // Create Random Variables (RVs)

    val iArrivalRV = Exponential (HOUR / lambda, stream)

    // Create Model Components

    val entry = Source ("entry", this, () => SOMEActor (), 0, nStop, iArrivalRV, (100, 290))
    val exit  = Sink ("exit", (600, 290))

    addComponent (entry, exit)

    // Specify Scripts for each Type of Simulation Actor

    case class SOMEActor () extends SimActor ("s", this):

        def act (): Unit =
            println ("SOMEActor: please write the script for this actor")
            exit.leave ()
        end act

    end SOMEActor

    simulate ()
    waitFinished ()
    Model.shutdown ()

end SOMEModel
```

Specification of a process-interaction involves the four steps: (1) Initialize Model Constants, (2) Create Random Variables (RVs), (3) Create Model Components, and (4) Specify Scripts for each Type of Simulation Actor. The `SOMEModel` class can be invoked as follows:

```
> runMain scalation.simulation.process.runSOME

@main def runSOME (): Unit = new SOMEModel ()
```

18.3.2 Component Trait

The basic structure of a simulation is given by the simulation components. The `Component` trait provides basic common features for simulation components. A component may function either as a node or edge. Entities/sim-actors interact with component nodes and move/jump along component edges. All components maintain sample/duration statistics (e.g., time in waiting queue) and all except `Gate`, `Source` and `Sink` maintain time-persistent statistics (e,g., number in waiting queue).

The most important component is the containing `Model`. For example, in `BankModel` discussed next, the instance of this class will serve as the *director*. Its role is similar to that in the `event` package, but rather than executing `occur` methods, it transfers control between the `SimActor`s.

Class Methods:

```
trait Component
      extends Identifiable with Locatable:

def initComponent (label: String, loc: Array [Double]): Unit =
def director: Model = _director
def director_= (director: Model): Unit =
def composite: Boolean = subpart.size > 0
protected def initStats (label: String): Unit =
def aggregate (): Unit =
def display (): Unit
def tally (duration: Double): Unit = _durationStat.tally (duration)
def accum (value: Double): Unit = _persistentStat.accum (value, _director.clock)
def durationStat: Statistic = _durationStat
def persistentStat: TimeStatistic = _persistentStat
```

18.3.3 Example: BankModel

Consider an simple bank simulation model where customers wish to utilize one of several tellers, but may need to wait their turn in a shared queue. Figure 18.1 depicts the flow of customers (`SimActors`) through the bank (`entry` to `door`/exit) and corresponds closely with the code required to implement the model in the process-interaction paradigm.

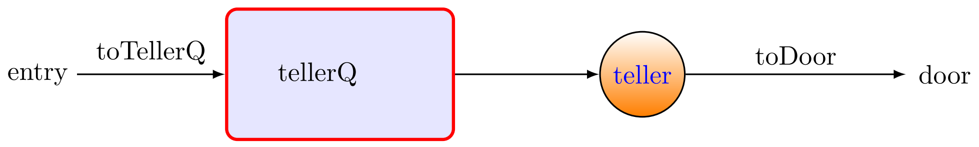


Figure 18.1: Bank Model Schematic Diagram

Such a BankModel (see the example_1 package) may be developed as follows:

- Initialize Model Constants: In this case customer arrival (lambda) and service (mu) rates need to be specified. In addition, the number of service units (nTellers) needs to be specified.
- Create Random Variables (RVs): This model will have three random variates: one for the inter-arrival times (iArrivalRV), one for service times (serviceRV), and one for movement (moveRV) along Transports.
- Create Model Components: A key step is to define the component nodes entry, tellerQ, teller, and door. Then two edge components, toTellerQ and toDoor, are defined. These six components are added to the BankModel using the addComponent method. Note, the endpoint nodes for an edge must be added before the edge itself.

```
class BankModel (name: String = "Bank", reps: Int = 1, animating: Boolean = true,
                 aniRatio: Double = 8.0, nStop: Int = 100, stream: Int = 0)
      extends Model (name, reps, animating, aniRatio):

    // Initialize Model Constants

    val lambda   = 6.0                                  // customer arrival rate (per hour)
    val mu       = 7.5                                  // customer service rate (per hour)
    val nTellers = 1                                    // the number of bank tellers (servers)

    // Create Random Variables (RVs)

    val iArrivalRV = Exponential (HOUR / lambda, stream)
    val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
    val moveRV     = Uniform (4 * MINUTE, 6 * MINUTE, (stream + 2) % N_STREAMS)

    // Create Model Components

    val entry     = Source ("entry", this, () => Customer (), 0, nStop, iArrivalRV, (100, 290))
    val tellerQ   = WaitQueue ("tellerQ", (330, 290))
    val teller    = Resource ("teller", tellerQ, nTellers, serviceRV, (350, 285))
    val door      = Sink ("door", (600, 290))
    val toTellerQ = Transport ("toTellerQ", entry, tellerQ, moveRV)
    val toDoor    = Transport ("toDoor", teller, door, moveRV)

    addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)
```

- Specify Scripts for each Type of Simulation Actor: Finally, an inner case class called `Customer` is defined where the `act` method specifies the script for bank customers to follow. The `act` method specifies the behavior of concurrent entities (`SimActor`) and is analogous to the `run` method for Java/Scala Threads.

```
    // Specify Scripts for each Type of Simulation Actor

    case class Customer () extends SimActor ("c", this):

        def act (): Unit =
            toTellerQ.move ()
            if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()
            teller.utilize ()
            teller.release ()
            toDoor.move ()
            door.leave ()
        end act

    end Customer

    simulate ()
    waitFinished ()
    Model.shutdown ()

end BankModel
```

The script for the actor consists of the following steps:

1. Upon creation by the `Source` at `entry`, the actor will move to the teller queue.

2. The actor will check whether all the tellers are busy and if all are busy, will wait in the queue `tellerQ` which is a `WaitQueue`. Note, the call to `noWait` is just for statistics collection.

3. The actor will utilize one of the tellers in the `teller` `Resource`, for a period of time corresponding to its service time.

4. After service is finished, the actor will then release the teller. This allows a waiting actor to begin service and triggers the collection of statistics.

5. The actor now moves to the door/exit.

6. Upon arrival at the `door`, a `Sink`, the actor will leave the bank/simulation and overall statistics will be collected.

The last three method calls will run the simulation using multiple threads/coroutines (`simulate`), wait until all the threads/coroutines are finished (`waitFinished`), and then safely shut down concurrent execution (`Model.shutdown`).

18.3.4 Executing the Bank Model

The BankModel class can be invoked as follows:

```
> runMain scalation.simulation.process.example_1.runBank

@main def runBank (): Unit = new BankModel ()
```

To make the animation easier to follow, try changing the aniRatio to 50.0.

18.3.5 Network Diagram

A Network Diagram for the Bank Model shown in Figure 18.2 displays four nodes (Source in green, WaitQueue in cyan, Resource in orange, and Sink in purple) and two edges (both Transports in blue).

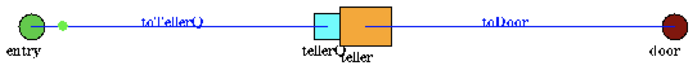


Figure 18.2: Animation of Network Diagram of Bank Model

An active entity (SimActor) representing a Customer is shown as token (small circle) frozen in its motion along the first Transport. Like an Event Graph, a Network Diagram can be used for both simulation model design and animation.

18.3.6 Comparison to Event Scheduling

In comparison, the Bank Model for event-scheduling did not include time delays and events for moving tokens along transports (although these could be added). The BankModel in the example_MIR package reduces the impact of transports by (1) using the transport's jump method rather than its move method and (2) reducing the time through the transport by an order of magnitude. The jump method has the tokens jumping directly to the middle of the transport, while the move method simulates smooth motion using many small hops.

The example_1 package provides several example process-interaction models based on the OSS method, including BankModel, CallCenterModel, EmerDeptModel, LoopModel, MachineModel, OneWayStreetModel, RoadModel, and TrafficModel.

The example_MIR and example_MBM packages include a subset of these models.

18.3.7 SimActor Class

The `SimActor` abstract class represents entities that are active in the model. The `act` abstract method, which specifies entity behavior, must be defined for each subclass. Each `SimActor` extends the `Coroutine` class and may be roughly thought of as running in its own thread. The script for entities/sim-actors to follow is specified in the `act` method of the subclass as was done for the `Customer` case class in the `BankModel`.

For example, a customer in the `BankModel` will enter the bank, move to a teller, if there is a line/queue will wait in the queue, be served by the teller and finally leave the bank. The `director` will transfer control to the actor (bank customer) which will execute code to get to the next step and transfer control back to the `director`. In this way the entity progresses through time processing multiple multiple events over its lifetime. A `SimActor` is created by a `Source` and terminated by a `Sink`. The `act` method encodes the logic of the actor's *script*.

Class Methods:

```
@param label     the label/name of the entity (‘SimActor‘)
@param director  the director controlling the model

abstract class SimActor (label: String, director: Model)
         extends Coroutine (label) with Temporal with Ordered [SimActor] with Locatable:

def trajectory: Double = _trajectory
def trajectory_= (trajectory: Double): Unit =
def compare (actor2: SimActor): Int = actor2.actTime compare actTime
def act (): Unit
def schedule (delay: Double): Unit =
def yieldToDirector (quit: Boolean = false): Unit =
override def toString: String = s"SimActor ($me at $actTime)"
```

Two of the key methods involved in transferring control between actors (via the director) are `schedule` and `yieldToDirector`.

```
def schedule (delay: Double): Unit =
    actTime = director.clock + delay
    director.reschedule (this)
end schedule
```

The `schedule` method places this actor in `agenda` (like a future event list) effectively specifying when the actor (a coroutine) will be reactivated. The `delay` parameter indicates how are into the future this will be.

```
def yieldToDirector (quit: Boolean = false): Unit =
    director.log.trace (this, "resumes", director, director.clock)
    yyield (director, quit)
end yieldToDirector
```

When the actor is has completed a step (conceptually like an embedded event) and either placed itself in a queue or the agenda, it is ready to let another actor to execute. It does this by yielding control to the director so the director can take the next action via the `yieldToDirector` method. The `quit` parameter is a flag indicating whether this actor has completed its last step.

18.3.8 Source Class

The `Source` class is used to periodically inject entities (`SimActors`) into a running simulation model (and a token into the animation). It may act as an arrival generator. A `Source` is both a simulation `Component` and a special `SimActor`, and therefore can run concurrently.

The `act` method loops over time through the creation of `units` entities. After making an entity using the `makeEntity` function, it schedules itself to run in the future by an amount of the given by `iArrivalTime.gen`, and it transfers control back to the `director` by calling `yieldToDirector`. A `Source` may create entities of different subtypes. For example, in the `TrafficModel` the `esubtype` indicates which of the four directions cars will traveling as they approach an intersection. The `act` method has an outer loop over replications and an inner scheduling loop shown below.

```
breakable {
     for i <- 1 to units do                                  // minor loop - make actors
         debug ("act", s"make $i SimActor")
         if director.stopped then break ()                   // terminate source, simulation ended
         val actor = makeEntity ()                           // make new actor
         actor.mySource = this                               // actor's source
         actor.subtype  = esubtype                           // set the entity subtype
         director.numActors += 1                             // number of actors created by all sources
         director.log.trace (this, "generates", actor, director.clock)
         director.animate (actor, CreateToken, randomColor (actor.id), Ellipse (),
                  Array (at(0) + at(2) + RAD / 2.0, at(1) + at(3) / 2.0 - RAD))
         actor.schedule (0.0)

         if i < units then
             val duration = iArrivalTime.gen
             tally (duration)
             schedule (duration)
             yieldToDirector ()                              // yield and wait duration time units
         end if
     end for
 } // breakable
```

For conciseness the `Source.group` method may be used to create multiple sources for a model.

```
def group (director: Model, makeEntity: () => SimActor, units: Int, xy: (Int, Int),
          src: (String, Int, Variate, (Int, Int))*): List [Source] =
    val sourceGroup = new ListBuffer [Source] ()
    for s <- src do sourceGroup += Source (s._1, director, makeEntity, s._2, units, s._3,
                                          (xy._1 + s._4._1, xy._2 + s._4._2))
    sourceGroup.toList
end group
```

For animation, the location of a `Source` node is specified by `loc`.

Class Methods:

```
@param name          the name of the source
@param director      the director controlling the model
@param makeEntity    the function to make entities of a specified type
@param esubtype      indicator of the subtype of the entities to be made
@param units         the number of entities to make
@param iArrivalTime  the inter-arrival time distribution
@param loc           the location of the source (x, y, w, h)

class Source (name: String, director: Model, makeEntity: () => SimActor,
              esubtype: Int, units: Int,
              iArrivalTime: Variate, loc: Array [Double])
      extends SimActor (name, director) with Component:

def this (name: String, director: Model, makeEntity: () => SimActor, esubtype: Int,
          units: Int, iArrivalTime: Variate, xy: (Double, Double)) =
def display (): Unit =
def act (): Unit =
```

18.3.9 Sink Class

The `Sink` class is used to terminate entities (`SimActors`) when they are finished. This class will remove the token from the animation and collect important statistics about the entity. A sink has a name and a location in the animation. Unlike a `Source`, a `Sink` is not active, rather is bundles logic to be executed by an actor when they are ready to leave the simulation. They do this by calling the `leave` method, which will tally the actors' time in the system.

```
tally (director.clock - actor.arrivalT)
```

Note, when a `SimActor` is no longer referenced (e.g., in director's `agenda` or in a wait queue) it becomes available for garbage collection (memory reclamation).

Class Methods:

```
@param name  the name of the sink
@param at    the location of the sink (x, y, w, h)
class Sink (name: String, at: Array [Double])
      extends Component:

def this (name: String, xy: (Double, Double)) =
def display (): Unit =
def leave (): Unit =
```

18.3.10 Transport Class

The `Transport` class provides a pathway between two other component nodes. The `Components` in a `Model` conceptually form a graph in which the edges are `Transport` objects and the nodes are other `Component` objects. An edge may be either a `Transport` or `Route`. A `Transport` is directional connecting a `from` component to a `to` component. When flow is required in both directions, two transports are required.

A `SimActor` may utilize a `Transport` by calling either the `move` or `jump` methods. The `move` method is intended for smooth animation, while the `jump` transports the entity quickly to the next component.

Class Methods:

```
@param name      the name of the transport
@param from      the first/starting component
@param to        the second/ending component
@param motion    the speed/trip-time to move down the transport
@param isSpeed   whether speed or trip-time is used for motion
@param bend      the bend or curvature of the transport (0 => line)
@param shift1    the x-y shift for the transport's first endpoint (from-side)
@param shift2    the x-y shift for the transport's second endpoint (to-side)

class Transport (name: String, val from: Component, val to: Component,
                 motion: Variate, isSpeed: Boolean = false, bend: Double = 0.0,
                 shift1: R2 = new R2 (0.0, 0.0), shift2: R2 = new R2 (0.0, 0.0))
      extends Component:

override def at: Array [Double] =
def display (): Unit =
def jump (): Unit =
def move (): Unit =
```

18.3.11 Resource Class

The `Resource` class provides services to entities (`SimActors`). The service provided by a resource typically delays the entity by an amount of time corresponding to its service time. The `Resource` may or may not have an associated waiting queue. It provides a number of service `units` (e.g., tellers) and a `busy` method to determine if all the servers are busy. A `SimActor` typically calls `utilize ()` to go into service for an amount time determined by `serviceTime.gen`. Alternatively, if the actor knows the length of its service time, it may call `utilize (duration)`. When finished, the actor should `release` the server. Finally, it is possible change the number service units available (e.g., a teller goes on break) by calling `changeUnits`.

Class Methods:

```
@param name         the name of the resource
@param line         the line/queue where entities wait
@param units        the number of service units (e.g., bank tellers)
@param serviceTime  the service time distribution
@param at           the location of the resource (x, y, w, h)

class Resource (name: String, line: WaitQueue, private var units: Int, serviceTime: Variate,
                at: Array [Double])
      extends Component:

def this (name: String, line: WaitQueue, units: Int, serviceTime: Variate,
          xy: (Double, Double))
def changeUnits (dUnits: Int): Unit =
def display (): Unit =
def busy: Boolean = inUse == units
def utilize (): Unit =
def utilize (duration: Double): Unit =
def release (): Unit =
```

18.3.12 WaitQueue Class

The `WaitQueue` class is a wrapper for Scala's Queue class, which supports FCSC Queues. It adds monitoring capabilities and optional capacity restrictions. If the queue is full, entities (`SimActors`) attempting to enter the queue are barred. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet). An entity on a `WaitQueue` is suspended for an indefinite wait. The actions of some other concurrent entity will cause the suspended entity to be resumed (e.g., when a bank customer finishes service and releases a teller).

When seeking service, an actor should check whether the servers are busy, and if so call `waitIn`. Although it is not necessary to call `noWait`, it is preferred to make this call to get the average waiting time for all actors, not just those that had to wait (of course, if that is the real interest of the study, the `noWait` call may be left out).

```
if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()
```

Class Methods:

```
@param name  the name of the wait-queue
@param at    the location of the wait-queue (x, y, w, h)
@param cap   the capacity of the queue (defaults to unbounded)

class WaitQueue (name: String, at: Array [Double], cap: Int = Int.MaxValue)
      extends Queue [SimActor] with Component:
```

```
    def this (name: String, xy: (Double, Double), cap: Int) =
    def isFull: Boolean = length >= cap
    def barred: Int = _barred
    def display (): Unit =
    def waitIn (): Boolean =
    def noWait (): Unit = tally (0.0)
```

18.3.13 WaitQueue_LCFS Class

The `WaitQueue_LCFS` class is a wrapper for Scala's `Stack` class, which supports Last-Come, First-Serve (LCFS) Queues. It adds monitoring capabilities and optional capacity restrictions. If the queue is full, entities `SimActor`'s) attempting to enter the queue are 'barred'. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet).

Class Methods:

```
    @param name  the name of the wait-queue
    @param at    the location of the wait-queue (x, y, w, h)
    @param cap   the capacity of the LCFS queue (defaults to unbounded)

    class WaitQueue_LCFS (name: String, at: Array [Double], cap: Int = Int.MaxValue)
          extends Stack [SimActor] with Component:

    def this (name: String, xy: (Double, Double), cap: Int) =
    def isFull: Boolean = length >= cap
    def barred: Int = _barred
    def display (): Unit =
    def waitIn (): Boolean =
    def noWait (): Unit = tally (0.0)
```

18.3.14 Junction Class

The `Junction` class provides a connector between two transports/routes. Since `Lines` and `QCurves` have limitation (e.g., hard to make a loop back), a junction may be needed, see the `LoopModel` for an example. Statistics are collected about entities at junctions.

For traffic simulations, junctions may also indicate locations for road sensors. The statistics collected at the junction can then be compared with the data recorded by a real-life sensor and used to calibrate/validate the simulation model. For example CalTrans records and makes available sensor data in its Performance Measure System (PeMS) `https://dot.ca.gov/programs/traffic-operations/mpr/pems-source`.

Class Methods:

```
@param name      the name of the junction
@param director  the director controlling the model
@param jTime     the jump-time through the junction
@param at        the location of the junction (x, y, w, h)

class Junction (name: String, director: Model, jTime: Variate, at: Array [Double])
      extends Component:

def this (name: String, director: Model, jTime: Variate, xy: (Double, Double)) =
def display (): Unit =
def jump (): Unit =
```

18.3.15 Gate Class

The `Gate` class models the operation of gates that can open and shut. When a gate is open, entities can flow through and when shut, they cannot. When shut, the entities may wait in a queue or go elsewhere. A gate can model a traffic light (green $\implies$ open, red $\implies$ shut). The `onTime` indicates how long the gate will be open (green light), while `offTime` indicated how long the gate will be shut (red light).

Class Methods:

```
@param name      the name of the gate
@param director  the model/container for this gate
@param line      the queue holding entities waiting for this gate to open
@param units     number of units/phases of operation
@param onTime    distribution of time that gate will be open
@param offTime   distribution of time that gate will be closed
@param loc       the location of the Gate (x, y, w, h)
@param shut0     'Boolean' indicating if the gate is initially opened or closed
@param cap       the maximum number of entities that will be released when the gate is opened

class Gate (name: String, director: Model, line: WaitQueue, units: Int,
            onTime: Variate, offTime: Variate,
            loc: Array [Double], shut0: Boolean = false, cap: Int = 10)
      extends SimActor (name, director) with Component:

def this (name: String, director: Model, line: WaitQueue, units: Int,
          onTime: Variate, offTime: Variate,
          xy: (Double, Double), shut0: Boolean, cap: Int) =
def shut: Boolean = _shut
def display (): Unit =
def release (): Unit =
def act (): Unit =
def gateColor: Color = if _shut then red else green
```

```
    def flip (): Unit = _shut = ! _shut
    def duration: Double = if _shut then offTime.gen else onTime.gen
```

18.3.16 Route Class

The Route class provides a multi-lane pathway between two other node components. The Components in a Model conceptually form a graph in which the edges are Transports/Routes and the nodes are other components. A route is a composite component that bundles several transports. The route will have k lanes that are stored in an array of Transports.

```
    val lane = Array.ofDim [Transport] (k)
```

See the RoadModel which uses two Route objects, one for West-bound traffic and the other for East-bound traffic. Each route has two lanes, making the road a four-lane road overall.

Class Methods:

```
    @param name      the name of the route
    @param k         the number of lanes/transports in the route
    @param from      the starting component
    @param to        the ending component
    @param motion    the speed/trip-time to move down the transports in the route
    @param isSpeed   whether speed or trip-time is used for motion
    @param angle     angle in radians of direction (0 => east, Pi/2 => north, Pi => west, 3Pi/2 => south)
    @param bend      the bend or curvature of the route (0 => line)

    class Route (name: String, k: Int, from: Component, to: Component,
                 motion: Variate, isSpeed: Boolean = false,
                 angle: Double = 0.0, bend: Double = 0.0)
          extends Component:

    def selector: Variate = lane(0).selector
    def selector_= (selector: Variate): Unit = lane(0).selector = selector
    override def at: Array [Double] = lane(0).at
    def display (): Unit =
```

18.3.17 Model Class

The Model class maintains a list of components making up the model and controls the flow of entities (SimActors) through the model, following the process-interaction world-view. A simulation model must extend the Model class and create an instance object that serves as the director. The director maintains a time-ordered priority queue called agenda to activate/re-activate each of the entities. The metaphor is that the director directs the actors in the play (i.e., simulation model). Each entity (SimActor) is implemented as a Coroutine and may be roughly thought of as running in its own thread. Control is transferred back and forth between the director and the actors in the play.

Class Methods:

```
@param name       the name of the simulation model
@param reps       the number of independent replications to run
@param animating  whether to animate the model
@param aniRatio   the ratio of simulation speed vs. animation speed
@param full       generate a full report with both sample and time-persistent statistics

class Model (name: String, val reps: Int = 1, animating: Boolean = true, aniRatio: Double = 1.0,
             val full: Boolean = true)
      extends Coroutine (name) with Completion with Modelable with Component:

def addComponent (_parts: Component*): Unit = for p <- _parts do parts += p
def addComponents (_parts: List [Component]*): Unit = for p <- _parts; q <- p do parts += q
def theActor: SimActor = _theActor
def stopped: Boolean = ! simulating
def reset (): Unit =
def resetStats (rep: Int, rmax: Int = reps): Unit =
def simulate (_startTime: Double = 0.0): Unit =
def cleanup (): Unit =
def reschedule (actor: SimActor): Unit = agenda += actor
def act (): Unit =
def getStatistics: ListBuffer [Statistic] =
def display (): Unit = for p <- parts do p.display ()
def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape,
             at: Array [Double]): Unit =
def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape,
             from: Component, to: Component, at: Array [Double] = Array ()): Unit =
protected def fini (rep: Int): Unit =
protected def report (): Unit =
protected def reportV (showMeans: Boolean = false): Unit =
protected def reportF (): Unit = new StatTable (s"$name statistics", getStatistics)
```

Process-Interaction Simulation Engine

The operation of the process-interaction simulation engine can be understood by looking at the *inner scheduling loop* within the act method of the Model class. The director takes the first actor in the agenda and marks it as _theActor. Then it advances its _clock to the activation time of _theActor. The director then transfer control by yielding to the _theActor. After executing the next step in their logic, _theActor will transfer control back to the director. This will continue until the agenda becomes empty or the director is instructed to stop simulating.

```
while simulating && ! agenda.isEmpty do            // INNER SCHEDULING LOOP
    _theActor = agenda.dequeue ()                  // next from priority queue
    _clock    = _theActor.actTime                  // advance the time
    log.trace (this, "resumes", _theActor, _clock)
    yyield (_theActor)                             // director yields to actor
end while
```

The outer loop in the `act` method is for replications.

```
for rep <- 1 to reps do                            // LOOP THROUGH REPLICATIONS
```

For One-Shot Simulation (OSS), `reps` should be one and for the Method of Independent of Replications (MIR) it should be say 10 or greater. See the Simulation Output Analysis Chapter for details. The `simulate` method is called by the class extending `Model` to initialize the component `parts` of the model. Its call to `start` will make a new thread that begins executing the director's `act` method.

```
def simulate (_startTime: Double = 0.0): Unit =
    startTime = _startTime
    _clock = startTime
    log.trace (this, "starts", this, _clock)

    for p <- parts do
        log.trace (this, s"establish x = ${p.at(0)}, y = ${p.at(1)}", p, _clock)
        p.director = this
        for q <- p.subpart do q.director = this
        if p.isInstanceOf [Source] then reschedule (p.asInstanceOf [Source])
    end for

    start ()                            // start the director thread/actor -> act ()
end simulate
```

18.3.18 Vehicle Traffic Model

A Network Diagram for a Vehicle Traffic simulation model is shown in Figure 18.3. It models an intersection of two multi-lane roads, one East-West and the other North-South. The intersection is controlled by traffic lights. Each road is divided into four parts: two directions and two segments (before and after the light).

There are five random variates used in this simulation.

```
val iArrivalRV = Uniform (iaTime, stream)
val onTimeRV   = Sharp (onTime, (stream + 1) % N_STREAMS)
val offTimeRV  = Sharp (offTime, (stream + 2) % N_STREAMS)
val moveRV     = Uniform (mvTime, (stream + 3) % N_STREAMS)
val laneRV     = Bernoulli ((stream + 4) % N_STREAMS)
```

The components in the simulation model are prescribed as follows:

At the beginning of each route, a `Source` will generate car arrivals. As traffic flow changes during each day, a Non-Homogeneous Process-Process (NHPP) may be used to generate cars (this example just uses `Uniform`). The time-dependent arrival rate $\lambda(t)$ may be fit to traffic flow data. For conciseness, the `group` method is used to create all four `Source`s. The coordinates `(800, 250)` form a reference point; the rest are relative to the reference point.

```
val source = Source.group (this, () => Car (), nStop, (800, 250),
                          ("s1N", 0, iArrivalRV, (0, 0)),           // from North
                          ("s1E", 1, iArrivalRV, (230, 200)),
                          ("s1S", 2, iArrivalRV, (30, 400)),
                          ("s1W", 3, iArrivalRV, (-200, 230)))
```

A place is needed for cars waiting for a stop light to change from red to green. Thus four `WaitQueue`s are needed.

```
val queue = WaitQueue.group ((800, 430), ("q1N", (0, 0)),           // before North light
                                         ("q1E", (50, 20)),
                                         ("q1S", (30, 70)),
                                         ("q1W", (-20, 50)))
```

The 4-way intersection requires four traffic lights to the control flow of cars. At any particular time, two of the lights should be red (closed `Gate`) and two should be green (open `Gate`). The `onTimeRV` gives the duration for the green light, while `offTimeRV` gives the duration of the red light. Both are `Sharp` distributions that give a constant value. The `group` method swaps the on and off times, based upon whether its number in the group is even or odd. Lights are positioned at the back of the intersection, e.g., the light for traffic coming for the North source `"s1N"` is the bottom left light in Figure 18.3.

```
val light = Gate.group (this, nStop, onTimeRV, offTimeRV, (800, 480),
                       ("l1N", queue(0), (0, 0)),                  // traffic from North
                       ("l1E", queue(1), (0, -30)),
                       ("l1S", queue(2), (30, -30)),
                       ("l1W", queue(3), (30, 0)))
```

After making it through the intersection, traffic continues to its designated `Sink`. For example, cars created by source `s1N` will be terminated by sink `k1S`.

```
val sink = Sink.group ((830, 250), ("k1N", (0, 0)),
                                   ("k1E", (200, 230)),
                                   ("k1S", (-30, 400)),            // end for North traffic
                                   ("k1W", (-230, 200)))
```

For each `Source`, two `Route`s are created: one from the `Source` to the `WaitQueue` and the other from the `Gate` to the `Sink`.

```
val road = ListBuffer [Route] ()
for i <- source.indices do
    road += Route ("ra" + i, 2, source(i), queue(i), moveRV)
end for
for i <- source.indices do
    road += Route ("rb" + i, 2, light(i),  sink((i + 2) % 4), moveRV)
end for

addComponents (source, queue, light, sink, road.toList)
```

In total, there are 16 nodes: four `Sources`, four `WaitQueues`, four `Gates` and four `Sinks`. In addition, there are 8 edges: all `Routes` with two lanes each (so underlying there are 16 `Transports`).

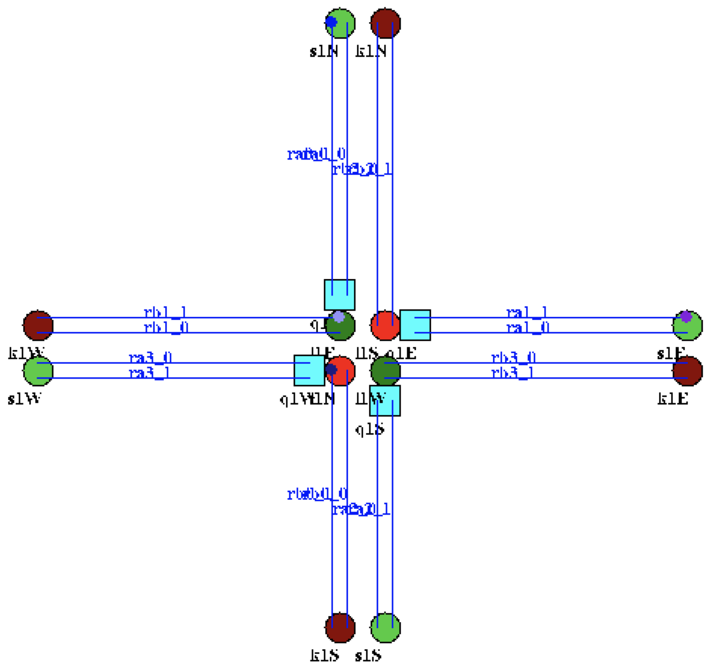


Figure 18.3: Animation of Network Diagram of Vehicle Traffic Model

For this simulation, `Cars` are the actors moving along the roads (may be thought of as autonomous vehicles or car-driver combinations). The behavior of a car depends on the direction it is traveling and is specified by its `subtype`.

```
case class Car () extends SimActor ("c", this):

    def act (): Unit =
        val i = subtype                       // from North (0), East (1), South (2), West (3)
        val l = laneRV.igen                   // randomly select lane l
        road(i).lane(l).move ()
        if light(i).shut then queue(i).waitIn ()
        road(i + 4).lane(l).move ()           // add 4 for next segment
        sink((i + 2) % 4).leave ()
    end act

end Car
```

The TrafficModel may be runs as follows:

```
> runMain scalation.simulation.process.example_1.runTraffic

@main def runTraffic (): Unit = new TrafficModel ()
```

18.3.19 Model_MBM Class

While the Model class works for both One-Shot Simulation (OSS) and the Method of Independent of Replications (MIR), the Model_MBM class is required for the Method of Batch Means (MBM). Although this method involves a single replication/simulation run, the single long run is divided into multiple batches in order to provide statistics. MIR and MBM provide complete statistics while OSS cannot, see the Simulation Output Analysis Chapter for details.

Class Methods:

```
 *  @param name       the name of the simulation model
 *  @param nBatch     the number of batches to run
 *  @param sizeB      the size of each batch
 *  @param animating  whether to animate the model
 *  @param aniRatio   the ratio of simulation speed vs. animation speed
 *  @param full       generate a full report with both sample and time-persistent statistics
 */
class Model_MBM (name: String, val nBatch: Int = 10, sizeB: Int = 100,
                 animating: Boolean = false, aniRatio: Double = 1.0, full: Boolean = true)
      extends Model (name, 1, animating, aniRatio, full):

    override def act (): Unit =
```

Notice that the Model_MBM class extends the Model class reusing all of methods, except its act method.

18.3.20 Exercises

1. Explain why the act method cannot be just a regular method/function call.

2. Explain what happens in the inner scheduling loop of the Model class. For the BankModel, suppose there are three coroutines/threads, the director, customer1, and customer2. Using three vertical time lines, one for each coroutine, with the director in the middle, show the control transfers between them. Assume customer2 arrives before customer1 finishes its service.

3. **Wendy's vs. McDonald's Lines**. Given two servers, is it better to have a line/queue for each server or one common line for both servers. Create a Network Diagram for Wendy's and another one for MacDonald's.

4. Implement a Process-Interaction Simulation and analyze the waiting times and standard deviation of the waiting times for Wendy's and MacDonald's. Let $\lambda = 20$ hr^{-1} and $\mu = 12$ hr^{-1}.

5. **Machine Shop**. Create a Network Diagram for the Machine Shop described in the Event-Oriented Simulation chapter.

6. Implement a Machine Shop simulation using Process-Interaction and determine which policy is better.

7. **Vehicle Traffic Simulation**. Create a Network Diagram for the stretch of US 101 at the Stanford exits from Willow Road to Oregon Expressway. See the Caltrans PeMS map.

8. Create a Process-Interaction Simulation of US 101 (Bayshore Freeway) at the Stanford exits. Data is recorded every five minutes at each of the sensors from Willow Road to Oregon Expressway giving 288 data points per day per sensor. Collect data for a portion the year 2021 for these sensors. Use it to calibrate and validate your models. Place `Source`s at the beginning of all road segments and on-ramps. Model the traffic inflow to the model using a Non-Homogeneous Poisson Process (NHPP) for each source that fits the data at that location. Place `Sink`s at the end of all road segments and off-ramps. Place `Junction`s at all road sensors. Use `Route`s for the road segments between sensors. There are typically four lanes Northbound and four lanes Southbound. Finally, use the data provided by Caltrans PeMS (traffic flow and speed) to to measure the accuracy (sMAPE) of your simulation model. See `https://getd.libs.uga.edu/pdfs/peng_hao_201908_phd.pdf` and `https://dot.ca.gov/programs/traffic-operations/mpr/pems-source`.

9. **Emergency Department Simulation**. Create and execute an Process-Interaction model for an emergency room/department based on the specifications given in the following paper, "Modeling and Improving Emergency Department Systems using Discrete Event Simulation," by Christine Duguay and Fatah Chetouane. `https://journals.sagepub.com/doi/10.1177/0037549707083111`. Model the patients as actors and the doctors and nurses as resources.

10. Compare the results of `BankModel` for the `event` package and `process package`. What happens as the number entities (customers) increases? What happens when the `move` method is replaced with the `jump` method. How do these results compare to those from Queueing Theory?

11. Rewrite the `Car` class for the `VehicleModel` from section 17.3.18 and put it in your answer sheet. Put this inside a new simulation model called `TrafficModelTurn` that has cars go straight with probability 0.75 and turn right with probability 0.25. Run the before `TrafficModel` and after `TrafficModelTurn` models and indicate the changes in travel time (cars going from `Source` to `Sink`).

```
runMain scalation.simulation.process.example_1.runTraffic
runMain scalation.simulation.process.example_1.runTrafficTurn
```

Give the mean travel times reported by each `Sink` for both models.

12. **Question 4**: Develop a process interaction simulation model for a Bank with two tellers (`nTellers = 2`). Let the inter-arrival time distribution be `Exponential` with rate $\lambda = 12$ per hour and the service time distribution be `Exponential` with rate $\mu = 7.5$ per hour. Simulate for 100 customers and 1 replication. Report the mean waiting time T_q in minutes. You may modify the `BankModel` class (`Bank.scala`) in the

```
scalation.simulation.process.example_1
```

package, if you like. Show all modifications you made to the code. Note, it is important to develop a correct model as this question is linked to the next question.

18.4 Agent-Based Simulation

Agent-Based Simulation (ABS) may be viewed as a cousin of Process-Interaction Simulation. An important enhancement provided by ABS is to provide a richer structure for actors to interact. For example, in a traffic simulation, it may be useful for a `Car` to know about the cars ahead and behind, as they may influence what the car does. Although this can done with process-interaction, it is up to model developer to create all the code to handle this. An ABS system should provide a framework that facilitates enriched interactions, reducing the burden of the simulation model developer. To reflect the enhanced capabilities of actors, including greater knowledge of its environment and other actors, they are typically named *agents*.

For simplicity, we focus on ABS as a form of time-based simulation (discrete-time or discrete-event) with event causality and a time-advance mechanism, but do not consider the more general Agent-Based Modeling (ABM) that may run multiple autonomous agents without controlled causal ordering of events, see [208] for a discussion.

In this context, the increased flexibility provided by Agent-Based Simulation partially derives from the capabilities/properties of agents. Desirable characteristics for agents include the following [115, 116, 114]:

1. An agent needs to be identifiable, *self-contained*, and active. Although actors in the process interaction paradigm share this, the event-scheduling paradigm does not as entities are passive and their logic/behavior specification is scattered among multiple event routines.

2. Agents have some level of *autonomy*. An agent should have the ability to sense its environment, make decisions and act accordingly. An example where this is not the case would be a `SimActor` whose script includes no parameterization or decision making (e.g., `if` statements). An agent should be able to Observe-Decide-Act [59].

3. Agents have the ability to *interact* with other agents. An example, in a Vehicle Traffic simulation, would be a car using a car-following rule/model which influences the driver's speed and gap to the car in front. The agent must be aware of its neighborhood to put a car-following rule/model into play. Some models may require move detailed interaction between agents, e.g., may require a communication protocol.

4. An agent is *situated* in an environment and interacts with its local environment. As agents can move around in their environment typically they will be given coordinates. Although coordinates were given for process-interaction, they were added for animation and providing a real interpretation of the coordinates is up to the model developer. An ABS system should directly support this capability.

5. It is also useful to provide support for agents to *learn*, for example, the model developer, could provide a set of possible rules for a car to change lanes. Support for for learning would mean that cars can collect information and analyze it to improve there decision making. This allows the agents to adapt as the simulation continues. Improvement implies goals, for example, the car would prefer less (not more) travel time.

6. *Resources* may exist in the environments or within agents. For example, a server is often thought of as part of the environment in process-interaction simulations, but a more realistic or detailed model could represent each teller as an agent that does other thinks besides just serving bank customers, e.g., they work a shift, have lunch and take breaks.

Further Reading

- "Introductory Tutorial: Agent-Based Modeling and Simulation," by Charles Macal, Michael North, *Proceedings of the 2014 Winter Simulation Conference*, `https://informs-sim.org/wsc14papers/includes/files/004.pdf`.

In ScalaTion agents may access information about the simulated world via a Knowledge Graph. In particular, a spatial Property Graph (`PGraph`) is used to set up the components in the simulated world. See the *Property Graph* section in the Data Management Chapter.

The vertices in the `PGraph` represent resources that agents can work with. An agent has a position in the simulated world with 2D or 3D coordinates. These coordinates may be transformed to screen coordinates for purposes of animation (see the next section). The edges in the `PGraph` represent one-way connections between vertices.

18.4.1 SimAgent

A `SimAgent` is a dynamic entity that moves to vertices and along edges as the simulation progresses. Its location is recorded in terms of its topological coordinates. An agent is thought to be at a vertex or on a edge. In theory, vertices are points, but ScalaTion allows them to take up space and measures distance as the distance from the center of the vertex. Distance along an edge is given by the distance from the beginning of the edge where it connects to the `from` vertex to the agent. In animation, an agent is represented as a token that moves along the graph. Due to the small size of vertices, tokens within may be represented collectively. Tokens on edges are represented as circles moving along edges. Topological coordinates are specified using the `Topological` trait.

```
@param elem  the element in the graph (at a vertex or on an edge)
@param dist  its distance along the segment

trait Topological (var elem: Element, var dist: Double)
      extends PartiallyOrdered [Topological]:

def tryCompareTo [B >: Topological: AsPartiallyOrdered] (other: B): Option [Int] =
def neighbors: VEC [Topological] = elem.tokens
def neighbors (d: Double): VEC [Topological] =
override def toString: String = s"Topological ($elem, $dist)"
```

While moving through the graph, an agent may interact with vertices as well as other agents. Agents moving along an edge may speed up, slow down or jump to a parallel edge (e.g., lane change in a vehicle traffic simulation).

```
val tokens = Set [Topological] ()                // topological objects/tokens at this edge
```

At a vertex, agents may passively wait (e.g., in wait queue), work with a server for a period of time, update their properties (e.g., the value of a part changes at each machine stage), wait for a traffic light to change color, choose which edge to follow upon leaving the vertex (e.g., the road to turn onto).

```
val tokens = Set [Topological] ()                // topological objects/tokens at this vertex
```

As expected, agents play a more central role in Agent-Based Simulation compared to Process-Interaction Simulation where much of the specification of behavior/decision making is often delegated to the components/blocks in the simulation.

A Property Graph (`PGraph`) consists of multiple vertex-types (`VertexType`) and multiple edge-types (`EdgeTypes`) as defined in the `scalation.database.graph` package.

18.4.2 Vertices

The vertices in a property graph are grouped into one or more vertex-types. A vertex is `Identifiable` and situated is space using `Spatial` coordinates.

```
@param _name  the name of this vertex ('name' from 'Identifiable')
@param prop   maps vertex property names into property values
@param _pos   the position (Euclidean coordinates) of this vertex ('pos' from 'Spatial')

class Vertex (_name: String, val prop: Property, _pos: VectorD = null)
      extends Identifiable (_name)
         with Spatial (_pos)
         with PartiallyOrdered [Vertex]
         with Serializable:
```

ScalaTion's ABS system defined in `scalation.simulation.agent_based` currently supports the following types of vertices:

- `Gate`: a gate is used to control the flow of agents, they cannot pass when it is shut.
- `Junction`: a junction is used to connect two transports.
- `Resource`: a resource provides services to agents (typically resulting in some delay).
- `Sink`: a sink consumes agents.
- `Source`: a source produces agents.
- `WaitQueue`: a FCFS wait-queue provides a place for agents to wait, e.g., waiting for a resource to become available or a gate to open.
- `WaitQueue_LCFS`: an LCFS wait-queue provides a place for agents to wait, e.g., waiting for a resource to become available or a gate to open.

These vertex-types correspond to the component nodes in process-interaction. Their constructors and methods are more oriented towards allowing more flexibility and specificity from the agents.

18.4.3 Edges

The edges in a property graph are grouped into one or more edge-types. An edge is `Identifiable` and situated is space using `Spatial` coordinates based one of the vertices it connects to.

```
@param _name  the name of this edge ('name' from 'Identifiable')
@param from   the source/from vertex of this edge
@param prop   maps edge property names into property values
@param to     the target/to vertex of this edge

class Edge (_name: String, val from: Vertex, val prop: Property, val to: Vertex)
      extends Identifiable (_name)
         with Spatial (if from == null then to.pos else from.pos)
         with Serializable:
```

ScalaTion's ABS system currently supports the following types of edges:

- `Link`: a link supports simple/quick movement of agents between closely connected vertices (e.g., a queue before a resource).
- `Route`: a route bundles multiple transports together (e.g., a two-lane, one-way street).
- `Transport`: a transport is used to move agents from one vertex to the next.

Note, closely connected nodes in the process-interaction were not required to have an edge between them, however, as ScalaTion's ABS system in built using `PGraph`, edges are required between vertices, hence the need for the `Link` class.

18.4.4 Bank Model

Specification of an Agent-Based Simulation model for a simple bank is not much different than for a Process-Interaction model.

Initialize Model Constants. As with any type of simulation model, constants or model parameters need to be specified, although in practice it is usually preferred to specify them in a scenario specification, e.g., in a `@main` top-level function.

```
val lambda   = 6.0                                  // customer arrival rate (per hour)
val mu       = 7.5                                  // customer service rate (per hour)
val nTellers = 1                                    // the number of bank tellers (servers)
```

Create Random Variates (RVs). A fourth random variate is added for jumping through the link between the wait queue and the resource containing the servers/tellers.

```
val iArrivalRV = Exponential (HOUR / lambda, stream)
val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
val moveRV     = Uniform (4 * MINUTE, 6 * MINUTE, (stream + 2) % N_STREAMS)
val jumpRV     = Uniform (0.4 * MINUTE, 0.6 * MINUTE, (stream + 3) % N_STREAMS)
```

Create the Graph Model. Specifying a graph model replaces the specification of the model components under process interaction. Notice the addition of the `Link` edge.

```
val entry_pos = Source.at (100, 290)
val cust_pos  = VectorD (110, 290, 10, 10)
val entry     = Source ("entry", this, 0.0, iArrivalRV, () => Customer (), nStop, pos = entry_pos)
val tellerQ   = WaitQueue ("tellerQ", this, pos = WaitQueue.at (330, 290))
val teller    = Resource ("teller", this, serviceRV, nTellers, pos = Resource.at (380, 285))
val door      = Sink ("door", this, pos = Sink.at (600, 290))
val toTellerQ = Transport ("toTellerQ", this, entry.vert, tellerQ, moveRV)
val toTeller  = Link ("to", this, tellerQ, teller, jumpRV)
val toDoor    = Transport ("toDoor", this, teller, door, moveRV)
```

Specify Scripts for each Type of Simulation Agent. The script is similar to the specification of actor scripts for process interaction. The service, move and jump times may also pass directly into the `work`, `move` and `jump` methods. Notice the increased specification available to `SimAgents`. For more complex models, this allows greater flexibility. In addition, an explicit `ping` method is required, as it is now not implicitly done by the resource, i.e., actions are now under control of the agent.

```
case class Customer () extends SimAgent ("c", director.clock, this, cust_pos.copy):

    def act (): Unit =
        toTellerQ.move (this)
        if teller.busy then tellerQ.waitIn (this) else tellerQ.noWait (this)
        toTeller.jump (this)
        teller.work (this)
        teller.release (this)
        tellerQ.ping ()
        toDoor.move (this)
        door.leave (this)
    end act

end Customer

simulate ()
waitFinished ()
Model.shutdown ()
```

18.4.5 Vehicle Traffic Model

The advantages of Agent-Based Simulation are not apparent in models as simple a Bank Simulation Model. Vehicle Traffic Models, on the other hand, are complex enough to illustrate some advantages.

Accurate simulation models of vehicle traffic often use a *car-following rule/model* to determine speed and braking actions of a car (an agent) based on the car ahead. This seems simple enough, just maintain a reference to the car ahead. Several complexities may ensue:

- The car ahead exits: the exiting car may then link the car behind (you) to the car ahead of itself, as in deletion from a doubly linked list.
- Your car wants to change lanes: where is the new car to follow, it needs to be efficiently found, by searching the local road segment.
- Your car wants to turn onto another road: the car needs to know where it is in the graph. The intersection is a vertex, the chosen road is an edge and the car starts on a particular road segment.

The following Car class includes an ability to change lanes. This requires a more flexible move method, where the second parameter indicates the fraction of the length of a Route to move along. Midway, a new lane l2 is determined and passed to the changeLane method. The car continues along lane l2 to the traffic light. If the light is green, then it continues through, otherwise it waits in the queue. After getting through the light it continues to its Sink.

```
case class Car () extends SimAgent ("c", director.clock, this):

    def act (): Unit =
        val i = subtype                             // from North (0), East (1), South (2), West (3)
        val l = laneRV.igen                         // randomly select lane l
        road(i).lane(l).move (this, 0.5)            // move half way down lane l
        val l2 = (l + 1) % 2                        // index of other lane
        road(i).changeLane (this, l, l2)            // change to lane l2
        road(i).lane(l2).move (this, 0.5)           // move the rest of the way down lane l2
        if light(i).shut then queue(i).waitIn (this)   // stop and wait for red light
        road(i + 4).lane(l2).move (this)            // add 4 for next segment
        sink((i + 2) % 4).leave (this)              // end at this sink
    end act

end Car
```

18.4.6 Hybrid Models

Hybrid simulation techniques, although more complex, allow the most appropriate modeling technique to be applied to parts of an overall simulation study. A comprehensive example is given in

"Application of Mixed Simulation Method to Modelling Port Traffic," by Ehiagwina Omoforma Augustine, 2021 [42]. https://researchonline.ljmu.ac.uk/id/eprint/15592/1/2021AugustineEhiagwinaPhD%20.pdf

18.4.7 Exercises

1. Compare the software available for Agent-Based Simulation.

2. Discuss the simulation world-views/paradigms used for modeling port traffic in the following dissertation, "Application of Mixed Simulation Method to Modelling Port Traffic."

3. What data structures and algorithms can used in the changeLane method to efficiently find the car ahead?

4. Design an Agent-Based Simulation for Vehicle Traffic Forecasting.

5. Develop an Agent-Based Simulation for **Vehicle Traffic Forecasting**. Add the capabilities discussed in this section to provide a more realistic simulation compared to Process-Interaction. See

6. Design the Emergency Department Simulation as an Agent-Based Simulation.

7. Develop the **Emergency Department Simulation** as an Agent-Based Simulation and model doctors and nurses as agents rather than resources as was indicated for the process-interaction approach.

Create and execute an model for an emergency department/room based on the specifications given in the following paper, "Modeling and Improving Emergency Department Systems using Discrete Event Simulation," by Christine Duguay and Fatah Chetouane, `https://journals.sagepub.com/doi/10.1177/0037549707083111`.

8. Design an Agent-Based Simulation for analyzing the COVID-19 Pandemic.

9. Develop an Agent-Based Simulation for analyzing the **COVID-19 Pandemic**. See "A realistic agent-based simulation model for COVID-19 based on a traffic simulation and mobile phone data," Sebastian A. Muller et al., `https://arxiv.org/pdf/2011.11453.pdf` as a starting point.

10. Develop an Agent-Based Simulation for **Military Applications**. See "Simulating Small Unit Military Operations with Agent-Based Models of Complex Adaptive Systems," by Victor Middleton, Proceedings of the 2010 Winter Simulation Conference, `https://www.informs-sim.org/wsc10papers/013.pdf` as a starting point.

11. Explain how Agent-Based Modeling and Simulation (ABMS) is used to study *emergent phenomena*. Give an example.

18.5 Animation

Process-Interaction and Agent-Based Simulation are ideally suited for animation. The environment can be given largely by displaying nodes and edges of a graph. The dynamics can be displayed by creating, moving and destroying tokens. The locations and actions of an actor or agent over its lifetime can be depicted in the animation.

18.5.1 2D Animation

In the JVM world (includes Java, Scala and several other languages), simple 2D animation can be accomplished using `awt` and `swing`. A richer graphics library is provided by `JavaFx`. Motion involves both space and time, so Java's `Thread` class and `Runnable` interface are also used for animation.

Basics of 2D Animation

Java's abstract window toolkit (`awt`) and `swing` packages support simple 2D animations. The example below illustrates this by drawing a large blue circle and having a small red ball continually trace the circle. The drawing `Canvas` is a class that extends `JPanel` that is an inner class within a `JFrame`. The `run` method (1) updates the ball coordinates, (2) sleeps for `tau` millisecond, e.g., 20 milliseconds corresponds to 50 frames per second (generally fast enough for humans to see motion as smooth), and (3) repaints the canvas by calling `repaint`.

```
class SimpleAnimator (title: String)
      extends JFrame (title) with Runnable:

    private val dim     = new Dimension (600, 500)                   // the size of the canvas
    private val tau     = 20                                         // operate at 50 Hz
    private val circle  = new Ellipse2D.Double (200, 200, 200, 200)  // the circle to traverse
    private val ballPos = new Point2D.Double (0, 300)                // ball position
    private val ball    = new Ellipse2D.Double ()                    // the moving ball

    getContentPane ().add (new Canvas ())
    setLocation (100, 100)
    setSize (dim)
    setVisible (true)
    setDefaultCloseOperation (EXIT_ON_CLOSE)
    new Thread (this).start ()

    class Canvas extends JPanel:
        override def paintComponent (gr: Graphics): Unit = ???
    end Canvas

    def run (): Unit =
        var theta = 0.0
        while true do
            theta    += 0.05
            ballPos.x = 300 + 100 * cos (theta)
            ballPos.y = 300 + 100 * sin (theta)
            println (s"ballPos = $ballPos")

            try Thread.sleep (tau)
            catch case ex: InterruptedException => println ("SimpleAnimator.run: sleep failed")
            end try
            repaint ()
        end while
    end run

end SimpleAnimator
```

The call to `repaint` will cause execution of the `paintComponent` method. Based on the new ball coordinates determined by the angle `theta`, the ball will move a few pixels each time the canvas is repainted. The `setFrame (x, y, w, h)` method is used to reset the ball coordinates: x-coordinate, y-coordinate, width, and height of the ball. Notice that `draw` draws the shape's boundary, while `fill` fills the shape in with color.

```
    override def paintComponent (gr: Graphics): Unit =
       super.paintComponent (gr)
       val gr2 = gr.asInstanceOf [Graphics2D]                    // use hi-res

       gr2.setPaint (Color.blue)                                  // blue circle
       gr2.draw (circle)

       gr2.setPaint (Color.red)                                   // read ball
       ball.setFrame (ballPos.x - 10, ballPos.y - 10, 20, 20)
       gr2.fill (ball)
    end paintComponent
```

The x and y coordinates specify the top-left position for the bounding box of the ball. As the `ballPos` is intended to be the center of the ball, half the width/height (10) must be subtracted to center the ball on the large red circle.

This example animation may be run as follows:

```
    @main def runSimpleAnimator (): Unit = new SimpleAnimator ("SimpleAnimator")
```

ScalaTion avoids the direct use of any graphics framework to facilitate changing frameworks as the technology evolves. Consequently, the `scala2d` package is used to insulate code from the particulars of any graphics framework. The SimpleAnimation2 class shows the minimal changes required to switch from direct use of Java graphics libraries to the use of the `scala2d` package.

2D Animation in ScalaTion

The `Model` class in the `event`, `process`, and `agent_based` packages import the following from the `scalation.animation` package: `AnimateCommand`, `CommandType`, and `DgAnimator`.

The `AnimateCommand` class provides a data structure for holding animation command specifications.

```
    @param action    the animation action to perform
    @param eid       the external id for the component acted upon
    @param shape     the shape of graph component (node, edge or token)
    @param label     the display label for the component
    @param primary   whether the component is primary (true) or secondary (false)
    @param color     the color of the component
    @param pts       the set points/dimensions giving the shapes location and size
    @param time      simulation time when the command is to be performed
    @param from_eid  the 'eid' of the origination node (only for edges)
    @param to_eid    the 'eid' of the destination node (only for edges)

    case class AnimateCommand (action: CommandType, eid: Int, shape: Shape, label: String,
                           primary: Boolean, color: Color, pts: Array [Double], time: Double,
                           from_eid: Int = -1, to_eid: Int = -1):

    def compare (command2: AnimateCommand) = time compare command2.time
    def show (array: Array [Double]) =
    override def toString =
```

The CommandType enumeration specifies the types of commands passed from a simulation engine to the animation engine.

```
enum CommandType (val name: String):

    case CreateNode      extends CommandType ("CreateNode")
    case CreateEdge      extends CommandType ("CreateEdge")
    case CreateToken     extends CommandType ("CreateToken")
    case DestroyNode     extends CommandType ("DestroyNode")
    case DestroyEdge     extends CommandType ("DestroyEdge")
    case DestroyToken    extends CommandType ("DestroyToken")
    case MoveNode        extends CommandType ("MoveNode")
    case MoveToken       extends CommandType ("MoveToken")
    case MoveToken2Node  extends CommandType ("MoveToken2Node")
    case MoveTokens2Node extends CommandType ("MoveTokens2Node")
    case MoveToken2Edge  extends CommandType ("MoveToken2Edge")
    case ScaleNode       extends CommandType ("ScaleNode")
    case ScaleToken      extends CommandType ("ScaleToken")
    case ScaleTokensAt   extends CommandType ("ScaleTokensAt")
    case SetPaintNode    extends CommandType ("SetPaintNode")
    case SetPaintEdge    extends CommandType ("SetPaintEdge")
    case SetPaintToken   extends CommandType ("SetPaintToken")
    case TimeDilation    extends CommandType ("TimeDilation")

end CommandType
```

The DgAnimator class is an animation engine for animating graphs. It contains a Panel that provides a paintComponent method for displaying the nodes, edges and tokens of the graph.

```
    @param _title    the title for the display frame
```

```
@param fgColor   the foreground color
@param bgColor   the background color
@param aniRatio  the ratio of simulation speed vs. animation speed

class DgAnimator (_title: String, fgColor: Color = black, bgColor: Color = white,
                  aniRatio: Double = 1.0)
      extends VizFrame (_title, null, 1200, 800) with Runnable:

def saveImage (fname: String): Unit = writeImage (fname, this)
def run (): Unit =
def animate (tStart: Double, tStop: Double): Unit =
def invokeNow (cmd: AnimateCommand): Unit =
def getCommandQueue: ConcurrentLinkedQueue [AnimateCommand] = cmdQ
```

The `run` method repeatedly executes animation commands, sleeps and repaints the drawing canvas/panel. The length of the sleep is determined by the time gap between commands based on their values from the simulation `clock` as well as the `aniRatio`.

18.5.2 3D Animation

C++ and C# are commonly languages used for 3D animations. There are several libraries available in the JVM world as well. `JavaFx` has limited support for 3D. Currently, `LWJGL` and `libGDX` are widely used by the JVM community.

Further Reading

- *Introduction to Computer Graphics and Animation*, by F. E. Ekpenyong, National Open University of Nigeria, CIT 371 Course Material, `https://nou.edu.ng/sites/default/files/2017-03/CIT371.pdf`

18.5.3 Exercises

1. Create an animation of the trajectory of a golf ball using the equations given in the State Space Models chapter. Add a tracer to see the path taken by the golf ball.
2. Redesign the `scala2d` package and call it `scala2df` to use `JavaFx` rather than `awt` and `swing`.
3. Design a `scala3d` package that uses `JavaFx` with its limited 3D capabilities.
4. Design a `scala3d` package that uses `LWJGL`.
5. Design a `scala3d` package that uses `LibGDX`.
6. Add the ability to zoom-in and zoom-out for animations.
7. List two ways animation is useful in simulation. Make a convincing argument for each.

Chapter 19

Simulation Output Analysis

Unlike some modeling techniques, a simulation run will produce one result, the next run another results. Individually these results may be misleading as results depend on the combination of random variates generated. Simulation true power comes from generating several results and then performing statistical analysis on these results. In other words, the simulation outputs need to be analyzed.

Many types of output may analyzed. For example, many simulation studies are interested in reducing waiting times. For such studies, one may define,

$$y_i \;=\; \text{waiting time of the } i^{th} \textit{ customer} \tag{19.1}$$

A goal of the simulation study would then be to produce accurate point and interval estimates of the customer waiting times.

19.1 Point and Interval Estimates

As discussed in the Probability Chapter, given a sample of size m, the *sample mean* ($\bar{\mu} = \hat{\mu}$) is computed as follows:

$$\bar{\mu} \;=\; \frac{\mathbf{1} \cdot \mathbf{y}}{m} \;=\; \frac{1}{m} \sum_{i=0}^{m-1} y_i \tag{19.2}$$

The *sample variance* is computed as follows:

$$\hat{\sigma}^2 \;=\; \frac{\|\mathbf{y} - \bar{\mu}\|^2}{m-1} \;=\; \frac{1}{m-1} \sum_{i=0}^{m-1} (y_i - \bar{\mu})^2 \tag{19.3}$$

Typically, simulations are used to study average behavior, so the focus in on the sample mean. To collect statistics on average behavior, several means need to collected and it is important that they be independent (or at least not highly correlated). There are two common methods used to achieve this, the Method of Independent Replications and the Method of Batch Means.

Both methods will produce n means: $\bar{\mu}_0, \bar{\mu}_1, \ldots \bar{\mu}_{n-1}$. From these a *grand mean* will be calculated.

$$\bar{\bar{\mu}} \;=\; \frac{1}{n} \sum_{i=0}^{n-1} \bar{\mu}_i \tag{19.4}$$

The grand mean can be used to estimate average behavior or performance characteristics, such as average waiting times. In addition to using the grand mean as a *point estimate*, it is common practice to obtain an *interval estimate* $[\bar{\bar{\mu}} - ihw, \bar{\bar{\mu}} + ihw]$, where ihw is the interval half width.

To create an interval estimate in the form of a confidence interval, we need to determine the variability or variance in the point estimate $\bar{\bar{\mu}}$.

$$\mathbb{V}\left[\bar{\bar{\mu}}\right] \;=\; \mathbb{V}\left[\frac{1}{n}\sum_{i=0}^{n-1}\bar{\mu}_i\right] \;=\; \frac{1}{n^2}\sum_{i=0}^{n-1}\mathbb{V}\left[\bar{\mu}_i\right] \;=\; \frac{\sigma^2_{\bar{\mu}}}{n} \tag{19.5}$$

Consequently, the following is an estimate for the variance of $\bar{\bar{\mu}}$.

$$\hat{\sigma}^2_{\bar{\bar{\mu}}} \;=\; \frac{\hat{\sigma}^2_{\bar{\mu}}}{n} \tag{19.6}$$

Therefore the interval half width is

$$ihw \;=\; t^* \frac{\hat{\sigma}_{\bar{\mu}}}{\sqrt{n}} \tag{19.7}$$

where t^* is a critical value from the Student's t Distribution. The interval estimate is then

$$\left[\bar{\bar{\mu}} - \frac{t^* \sigma_{\bar{\mu}}}{\sqrt{n}},\, \bar{\bar{\mu}} + \frac{t^* \sigma_{\bar{\mu}}}{\sqrt{n}}\right] \tag{19.8}$$

Suppose the true mean waiting time μ can be computed (e.g., from Queueing Theory), then the probability the confidence interval will contain it corresponds to the confidence level chosen (and t^* is determined by this and the Degrees of Freedom).

19.2 One-Shot Simulation

One-shot simulation happens when there a single simulation run and that run is treated as a whole (i.e., it is not broken into multiple batches). This is the simplest way to run a simulation and it is good idea to convince yourself that the model is running correctly in this mode before moving on a method more useful for simulation output analysis.

The `process` package contains several example models that are placed in sub-directories according how they are set up for output analysis and validation.

- `example_1`:
  One-Shot Simulation (OSS)

  During model development, OSS may be used since model execution takes less time and animation is on by default, thus giving the model developer quick feedback on the correctness of the model. The default settings are 1 replication, animation on, uses `move` method, extends `Model` (no batching). The `move` method gives smooth motion for better animation.

- `example_MIR`:
  Method of Independent Replications (MIR)

  OSS will report a mean, but not a standard deviation, since that requires at least 2 simulation runs (or replications). The default settings are 10 replications, animation off, uses `jump` method, and extends `Model` (no batching). The `jump` method gives less overhead in `Transport`s which for many models may provide more accurate statistics that correspond better those in the `event` package. The developer needs to decide depending on what they are modeling. For example, one should use `move` and not `jump` for traffic simulations, but use `jump` over `move` for tandem queues where the time to move from one queue to next is negligible. Note, using `move` with OSS would likely be preferred as it will make it easier to track the entities while viewing the animation.

- `example_MBM`:
  Method of Batch Means (MBM)

  Although MIR can provide more useful statistics, it tends to be less effective for studying long-term or steady-state behavior. MBM has one long simulation run (replication), but divides it up into several batches. Each batch produces a batch mean and since there are several of them, standard deviations can be computed. The default settings are 1 replication, animation off, uses `jump` method, extends `Model_MBM` (batching). The `Model_MBM` class extends the `Model` class with batching capabilities (see the section on MBM for details).

19.3 Simulation Model Validation

In data science in general, model validation is both important and challenging. As simulation models tend to have many human designed components that must work together, so the chance of making errors is high. Also, like all computer programs bugs need to be removed. With sorting algorithms, it is obvious when the output is incorrect. With simulation, it is often difficult to know whether the output is correct or not.

One option would be to collect data and see if the model output agrees with data. This begs to question of whether a program implementing a model and having bugs could agree with the data. As this is real possibility, one should be skeptical of a new model and first look to ways to falsify the model. Of course, there is the additional complexity of whether the problem is with program implementing model or the model, or likely both. To have a clean separation between the implementing program and the model, would require a formal (or executable) specification of the model. The process of showing that a program implementation agrees with the model is sometimes called verification (to be discussed later).

Focusing first on validation a model based on a program that implements it, there are many ways to check the simulation model's behavior or output [104, 166].

1. Make sure the model works for *Simple Scenarios.* For example, consider the Machine Shop with limited storage capacity for parts being processed by machines in sequence. This model can to difficult to build correctly and hard to know if its output is correct. Due to the limited queue sizes, parts may be sold for scrap or machines may be blocked from operating. However, when the part arrival rate is small, no parts will be sent to scrap and no machines will be blocked. A simpler model can be used (or built) that uses infinite capacity queues. The new machine shop model should agree with this simpler model.

2. Slowly *Increase the Complexity* of scenarios. For a machine shop with two machines in sequence, the first queue becoming full will introduce the first increment of complexity. What is expected to happen when the arrival rate increases enough to cause the first queue to become full? In case the second queue becomes full first, its capacity can be temporarily increased to focus on one problem at a time. After any bugs triggered by the first queue becoming have been removed, the arrival may be further increased to make the second queue to become full. Finally, after bugs associated with this have been removed, add the option of blocking the first machine when the second queue becomes full. Removing bugs with all three issues simultaneously may be very difficult. And remember, the Machine Shop model is a relatively simple model.

3. Use *Analytic Models* as beacons in the sea of complexity. They may be too simple for the problem being addressed, but may be close enough to be indicators of whether you are on the right track. For example, the infinite queue approximation for machines in sequence can be solved analytically using a Queueing Network model. Although the restrictions of unchanging parameters, to certain distributions and steady-state conditions may have taken Queueing Networks out as viable models for the original problem, they still can be very helpful in model validation. Time series Forecasting models such as SARIMAX and LSTM many be very helpful in validating Vehicle Traffic simulation models.

4. A collection of related, validated models in a specific domain along with explanations of when they are applicable and why they are at least an approximation to the phenomena/system under study could be termed a *Theory.* The theory would typically include constraints that could be used to assess the realism of a newly proposed (simulation) model. Models that are accurate may still be in discord with theory, suggesting either the model is magical or the theory needs revision. There are new initiatives

in data science and machine learning that go by various names such as theory-guided data science [90, 123] or physics-informed machine learning [89].

5. The above item focused on model output, but model inputs are important as well. *Input Analysis* can be used to choose between distributions, e.g., are service times Exponential, Erlang, Normal, Weibull, etc. If arrivals follow an Non-Homogeneous Process Process (NHPP), how is the function $\lambda(t)$ estimated correctly? Are probabilities for cars go straight, turning right, turning left estimated correctly. Plots and Goodness-of-Fit test can be useful for this.

6. Before it was argued that simpler models are helpful for validation. Ideally, the new model could be positioned between a simpler model and a more *Complex Model* that has been previously validated. Such a model may not exist given the need for your study, but if one does, it could be that your models is faster or more generalizable than the complex model. Alternatively, it may be more amenable to optimization or interpretation. In any case, the model complex model can be very helpful for validation.

7. Besides looking at the results/outputs of a simulation model, one should *Examine the Behavior* of the model. One way to do this is to examine a *trace* of the model execution (e.g., what happens with each event). This can be tedious and even mind-numbing. An alternative is to examine an *animation* of simulation. The animation should be watched to slow motion to see what happens to the entities (or actors or agents) in detail. For both tracing and animation, the number of entities should be reduced. The real simulation may requires thousand of entities, examining all of them will likely be fruitless.

8. Although humans can digest short traces, they may miss the part of the simulation exhibiting the incorrect behavior (e.g., machine 1 gets blocked, but a newly arriving entity begins service in the blocked queue), *Anomaly Detection in Simulation Traces* can be used. For example, do entities enter the sink out of order, or does an entity's wait time vary greatly form the entities going through the system at roughly the same time and taking the same pathway. Both machine learning and rule-based anomaly detection algorithms can be useful.

9. As a later step in the validation process, the output of the model (whether predictions, forecasts or classifications) need to be compared with data and Quality of Fit (QoF) measures should be examined. For example, R^2, MAE and sMAPE may be used to assess the quality of forecasts. The QoF measures should be compared to other models addressing the problem under study.

10. The model developers should also follow standard *Software Engineering Practices*, a topic too large to summarize here.

19.3.1 Model Calibration

At some point during or after model validation, model calibration can be performed. Typically, this involves adjusting the model parameters to improve some QoF measure for the model. If performed too early, calibrating a buggy model is a waste of effort. Remember one should always be skeptical of newly developed model.

Calibration becomes a search process in parameter space. As an example, suppose a compartmental model for a pandemic includes 10 rate parameters each with 10 feasible levels. *Grid Search* would then require run 10^{10} simulations in order to find the best combination. Actually, it is worse than that, since a single run is not reliable, i.e., multiple runs need to be made for each combination of parameters. See the exercises for better alternatives to Grid Search.

19.3.2 Model Verification

In the simulation literature, verification is used to show that the program implementation meets the intentions of a model specification. The task is given to the model developer. Tool support includes those from software engineering to facilitate debugging and systemic software testing (e.g., unit testing and coverage conditions). ScalaCheck and ScalaTest help automate the software testing process. Recently, progress has been made on automated verification systems, such as Stainless [101] that is used for verifying the correctness of Scala programs. The fact that Scala has a purely functional subset and strict static type checking, facilities program verification.

19.4 Method of Independent Replications (MIR)

Since simulation models produce stochastic outputs, One-Shot Simulation may not be reliable. Suppose one is interested in customer waiting time T_q (renamed w). Each customer will have their own waiting time w_j. One simulation run typically involves hundreds or thousands of entities/customers. One might think that taking an average or mean would provide a useful estimate. Unfortunately, the w_j's may be highly correlated, which tends to inflate variability. This can be evidenced in the following table where an M/M/1 Queue is simulated for $n_r = 10$ replications (where the only thing changing per run is the base random number stream).

For MIR, let w_{ij} be the waiting time for the j^{th} entity in the i^{th} run. The i^{th} *run mean* is given as follows:

$$\bar{w}_i \;=\; \frac{1}{n_s}\sum_{j=0}^{n_s-1} w_{ij} \;=\; \text{mean waiting time for } i^{th} \text{ replication} \tag{19.9}$$

The number of customers per run `nStop` $= n_s$ is set to 100 (and then 1000) in Table 19.1. In order to make the replications independent, it is essential to change the stream each time.

Table 19.1: M/M/1 Queue: 10 Replications

Stream	$\bar{w}_i$ (100)	$\bar{w}_i$ (1000)
0	15.282	40.975
1	21.969	34.289
2	25.769	28.267
3	29.512	20.762
4	19.829	29.164
5	26.259	29.266
6	12.316	37.362
7	96.311	21.234
8	39.947	34.554
9	12.915	31.867
mean	30.011	30.774
stdev	24.760	6.478

Notice the high variability in $\bar{w}_i$ particularly for `nStop = 100`, less so with `nStop = 1000`. These data are created by running the MIR version of `BankModel`.

```
runMain scalation.simulation.process.example_MIR.runBank
```

The changes to the code from the `example_1` version are the following: animation is turned off, `reps = 10`, `jump` replaces `move` and the time on the transports is greatly reduced.

```
val moveRV     = Sharp (SECOND, (stream + 2) % N_STREAMS)
```

The mean and standard deviation for each column can be used to compute confidence intervals.

19.4.1 Confidence Intervals

Case $n_r = 10, n_s = 100$

For the case where `reps = 10, nStop = 100`, the *grand mean* is simply the mean of $n_r = 10$ run means.

$$\bar{\bar{w}} \;=\; \frac{1}{n_r} \sum_{i=0}^{n_r - 1} \bar{w}_i \;=\; 30.011 \tag{19.10}$$

The standard deviation of the run means $\bar{w}_i$'s is

$$\hat{\sigma}_{\bar{w}} \;=\; \frac{1}{n_r - 1} \sum_{i=0}^{n_r - 1} (\bar{w}_i - \bar{\bar{w}})^2 \;=\; 24.760 \tag{19.11}$$

Thus, the interval half width (ihw) is

$$t^* \frac{\sigma_{\bar{w}}}{\sqrt{n_r}} \;=\; 2.262 \, \frac{24.760}{\sqrt{10}} \;=\; 17.712 \tag{19.12}$$

where t^* is the value for the Student's t distribution with $n_r - 1$ Degrees of Freedom, where the area/probability of being in either tail is 0.05 (95% confidence interval). Therefore, the interval estimate is the following:

$$\left[\bar{\bar{w}} - \frac{t^* \hat{\sigma}_{\bar{w}}}{\sqrt{n_r}}, \; \bar{\bar{w}} + \frac{t^* \hat{\sigma}_{\bar{w}}}{\sqrt{n_r}} \right] \tag{19.13}$$

Finally, the 95% confidence interval is

$$[30.011 - 17.712, 30.011 + 17.712] \;=\; [12.299, 47.723] \tag{19.14}$$

Case $n_r = 10, n_s = 1000$

For the case where `reps = 10, nStop = 1000`, the interval half width (ihw) is

$$t^* \frac{\sigma_{\bar{w}}}{\sqrt{n_r}} \;=\; 2.262 \, \frac{6.478}{\sqrt{10}} \;=\; 4.634 \tag{19.15}$$

and the 95% confidence interval is much tighter.

$$[30.774 - 4.634, 30.774 + 4.534] \;=\; [26.140, 35.408] \tag{19.16}$$

Case $n_r = 40, n_s = 1000$

For the case where `reps = 40, nStop = 1000`, the interval half width (ihw) is

$$t^* \frac{\sigma_{\bar{w}}}{\sqrt{n_r}} \;=\; 2.023 \, \frac{7.758}{\sqrt{40}} \;=\; 2.481 \tag{19.17}$$

and the 95% confidence interval is even tighter.

$$[30.489 - 2.481, 30.489 + 2.481] \;=\; [28.008, 32.970] \tag{19.18}$$

Case $n_r = 40, n_s = 10000$

For the case where `reps = 40, nStop = 10000`, the interval half width (ihw) is

$$t^* \frac{\sigma_{\bar{w}}}{\sqrt{n_r}} \;=\; 2.023\,\frac{2.598}{\sqrt{40}} \;=\; 0.831 \tag{19.19}$$

and the 95% confidence interval is reasonably tight.

$$[31.600 - 0.831, 31.600 + 0.831] \;=\; [30.769, 32.431] \tag{19.20}$$

With 10,000 entities per run, it is reasonable to compare this results with the result from queueing theory.

$$T_q \;=\; \frac{\rho/\mu}{1-\rho} \tag{19.21}$$

Since the traffic intensity $\rho = \dfrac{\lambda}{\mu} \;=\; \dfrac{6}{7.5} \;=\; 0.8$,

$$T_q \;=\; \frac{0.8/7.5}{1-0.8} \;=\; 0.533 \text{ hours} \;=\; 32 \text{ minutes} \tag{19.22}$$

Note, the theoretical value of 32 minutes is inside all the confidence intervals.

19.4.2 Example: MIR Version of `BankModel`

The source code for the MIR version of `BankModel` in the `scalation.simulation.process/example_MIR` package is shown below.

```
class BankModel (name: String = "Bank", reps: Int = 100, animating: Boolean = false,
                 aniRatio: Double = 8.0, nStop: Int = 1000, stream: Int = 0)
      extends Model (name, reps, animating, aniRatio):

    //--------------------------------------------------
    // Initialize Model Constants

    val lambda   = 6.0                               // customer arrival rate (per hour)
    val mu       = 7.5                               // customer service rate (per hour)
    val nTellers = 1                                 // the number of bank tellers (servers)

    //--------------------------------------------------
    // Create Random Variables (RVs)

    val iArrivalRV = Exponential (HOUR / lambda, stream)
    val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
    val moveRV     = Sharp (SECOND, (stream + 2) % N_STREAMS)

    //--------------------------------------------------
    // Create Model Components

    val entry     = Source ("entry", this, () => Customer (), 0, nStop, iArrivalRV, (100, 290))
    val tellerQ   = WaitQueue ("tellerQ", (330, 290))
    val teller    = Resource ("teller", tellerQ, nTellers, serviceRV, (350, 285))
    val door      = Sink ("door", (600, 290))
    val toTellerQ = Transport ("toTellerQ", entry, tellerQ, moveRV)
    val toDoor    = Transport ("toDoor", teller, door, moveRV)

    addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)

    //--------------------------------------------------
    // Specify Scripts for each Type of Simulation Actor

    case class Customer () extends SimActor ("c", this):

        def act (): Unit =
            toTellerQ.jump ()
            if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()
            teller.utilize ()
            teller.release ()
            toDoor.jump ()
            door.leave ()
        end act

    end Customer

    simulate ()
    waitFinished ()
    Model.shutdown ()

end BankModel
```

19.5 Method of Batch Means (MBM)

The Method of Batch Means (MBM) is intended to provide a more efficient and reliable way to analyze the steady-state, as opposed to what was done in the last section of making `nStop` large enough for the simulation to exhibit steady-state behavior. Each of the forty runs/replications had to go through a warm-up period or transient phase.

With MBM, the simulation only goes through one transient phase at the beginning. Rather than having a mean for each run, a mean is created for each batch. One long run is divided into multiple batches. The trick is to make the batches uncorrelated enough, so that the advantage of independent replications is not lost. If the batch means are highly correlated, the confidence intervals will not be reliable. Fortunately, the longer the batch `sizeB` (s_b), the smaller the correlation between the batch means. The other hyper-parameter is `nBatch` (n_b). These are analogs of n_s and n_r.

For MBM, let w_{ij} be the waiting time for the j^{th} entity in the i^{th} batch. The i^{th} *batch mean* is given as follows:

$$\bar{w}_i \;=\; \frac{1}{s_b}\sum_{j=0}^{s_b-1} w_{ij} \;=\; \text{mean waiting time for } i^{th} \text{ batch} \tag{19.23}$$

The grand mean is simply the mean of n_b batch means.

$$\bar{\bar{w}} \;=\; \frac{1}{n_b}\sum_{i=0}^{n_b-1} \bar{w}_i \tag{19.24}$$

The standard deviation of the batch means $\bar{w}_i$'s is

$$\hat{\sigma}_{\bar{w}} \;=\; \frac{1}{n_b-1}\sum_{i=0}^{n_b-1} (\bar{w}_i - \bar{\bar{w}})^2 \tag{19.25}$$

Thus, the interval half width (ihw) is

$$ihw \;=\; t^* \frac{\sigma_{\bar{w}}}{\sqrt{n_b}} \tag{19.26}$$

where t^* is the value for the Student's t distribution with $n_r - 1$ Degrees of Freedom, where the area/probability of being in either tail is 0.05 (95% cofidence interval).

19.5.1 Effect of Increasing the Number of Batches

Table 19.2 compares the grand means $\bar{\bar{w}}$ and interval half widths (ihw) of MBM versus MIR with the batch size s_b and the run length n_s both set to 1000. The number of batches n_b and the number of replications n_r are given in the first column of the table and range from 10 to 100.

The MBM grand means $\bar{\bar{w}}$ appear to be converging to 32, the answer from queueing theory (see the exercises). Such a tendency is not obvious for MIR grand means that appear to converging to a lower number than 32. This is because the data from the transient phase is in every replication, so its effect does not diminish. On the contrary, MBM arguably has only one batch affected by the transient or warm-up phase, so as the number of batches increases, the transient effect should steadily decrease.

Table 19.2: M/M/1 Queue: MBM vs. MIR

n_b or n_r	MBM $\bar{\bar{w}}$	MBM *ihw*	MIR $\bar{\bar{w}}$	MIR *ihw*
10	31.875	5.074	30.774	4.634
20	30.146	2.821	28.880	2.815
30	30.030	2.980	29.924	3.074
40	31.703	3.121	30.489	2.481
50	32.141	2.845	30.777	2.398
60	32.382	2.574	30.935	2.154
70	32.503	2.313	31.120	1.914
80	32.220	2.089	30.495	1.759
90	31.812	1.947	30.584	1.683
100	31.764	1.841	30.426	1.583

19.5.2 Effect on Batch Correlation of Increasing the Batch Size

As the waiting time for entity/customer i is likely to be highly correlated with that of customer $i-1$, a small batch size will likely result in highly correlated batch means. The model developer can check the autocorrelation of the batch means and increase the batch size as needed. The grand means shown Table 19.2 (where the batch size is 1000) are simply the means of *batch means* for MBM or the means of the *run means* for MIR. For example, for $n_b = 40$, the following are the batch means.

```
val batchMeans = VectorD (
    41.0430, 34.1852, 27.8945, 20.7674, 29.5358, 28.1378, 39.3704, 22.8593, 38.9610, 35.9986,
    30.0565, 29.4135, 37.4061, 31.1298, 26.1880, 23.1076, 28.7468, 21.2382, 29.1787, 27.7081,
    54.9413, 16.9537, 26.2421, 27.1526, 39.5139, 18.0606, 34.2399, 27.5326, 21.1054, 32.2369,
    18.5720, 56.1361, 37.0753, 29.6010, 50.7372, 29.5670, 52.8855, 21.3998, 39.4527, 31.7850)
println (s"batchMeans.acorr () = ${batchMeans.acorr ()}")
```

The `acorr` method in `VectorD` computes the lag-1 autocorrelation ρ_1 for the vector shown above to be -0.2767. Having larger batch sizes s_b is likely to reduce to the magnitude (absolute value) of the correlation. Assuming covariance stationarity (see the section on the Auto-Correlation Function in the time series chapter), the lag-1 autocorrelation may be computed as follows:

$$\rho_1 \;=\; \frac{\mathbb{C}\,[\bar{w}_i, \bar{w}_{i-1}]}{\mathbb{V}\,[\bar{w}_i]} \tag{19.27}$$

See the exercises to see how autocorrelation changes with increasing batch sizes.

19.5.3 MBM versus MIR

The Method of Batch Means (MBM) is intended for longer running or steady-state simulations, while the Method of Independent Replications (MIR) is intended for shorter duration or transient-phase simulations (e.g., a systems shuts down each day and does not sufficiently stabilize for a steady-state simulation to be meaningful).

The Method of Batch Means and the Method of Independent Replications may be depicted as shown below. The MIR simulation consists of $n_r = 10$ runs/replications, while the MBM simulation consists of one run, that is divided in $n_b = 10$ batches. Let '-' represent 10 entities/customers, and thus the length of a replication $n_s = 100$ entities, and similarly the size of each batch $s_b = 100$ entities.

```
MIR:
---------- w_0
---------- w_1
---------- w_2
---------- w_3
---------- w_4
---------- w_5
---------- w_6
---------- w_7
---------- w_8
---------- w_9
MBM:
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
    w_0        w_1        w_2        w_3        w_4        w_5        w_6        w_7        w_8        w_9
```

The $\bar{w}_i$ (`w_i`) are run means for MIR and batch means for MBM.

19.5.4 Relative Precision

It had been mentioned that having tight confidence intervals is an important goal in simulation. That begs the question, tight relative to what. Often it is sufficient to make it relative the estimated grand mean. One may define γ to be 1 - the *relative precision* and calculate it as the interval half width over the grand mean, i.e.,

$$\gamma \;=\; \frac{ihw}{\bar{\bar{w}}} \tag{19.28}$$

Suppose the goal is to achieve 90% relative precision (or $\gamma \leq .1$). For $n_b = 10$, $\gamma = 5.074/31.875 = .159$, indicating more batches are needed. For $n_b = 20$, $\gamma = 2.821/30.146 = .094$, indicating acceptable relative precision. Note, some simulation studies may prefer to with work *absolute precision* instead.

A basic procedure for MBM would be to increase the batch size until the correlation between batch means drops to a threshold (e.g., $\rho_1 \leq .2$ or .3) and 1 - relative precision drops to another threshold (e.g., $\gamma \leq .05$ or .1). There are several advanced procedures that are more efficient/effective (although more complex) for MBM, see [105, 12, 182, 5].

19.5.5 Example: MBM Version of `BankModel`

The source code for the MBM version of `BankModel` in the `scalation.simulation.process/example_MBM` package is shown below.

```
class BankModel (name: String = "Bank", nBatch: Int = 100, sizeB: Int = 1000,
                 animating: Boolean = false, aniRatio: Double = 8.0, stream: Int = 0)
      extends Model_MBM (name, nBatch, sizeB, animating, aniRatio):
```

```
val nStop = nBatch * sizeB                        // number arrivals before stopping the Source

//--------------------------------------------------
// Initialize Model Constants

val lambda   = 6.0                                // customer arrival rate (per hour)
val mu       = 7.5                                // customer service rate (per hour)
val nTellers = 1                                  // the number of bank tellers (servers)

//--------------------------------------------------
// Create Random Variables (RVs)

val iArrivalRV = Exponential (HOUR / lambda, stream)
val serviceRV  = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
val moveRV     = Sharp (SECOND, (stream + 2) % N_STREAMS)

//--------------------------------------------------
// Create Model Components

val entry     = Source ("entry", this, () => Customer (), 0, nStop, iArrivalRV, (100, 290))
val tellerQ   = WaitQueue ("tellerQ", (330, 290))
val teller    = Resource ("teller", tellerQ, nTellers, serviceRV, (350, 285))
val door      = Sink ("door", (600, 290))
val toTellerQ = Transport ("toTellerQ", entry, tellerQ, moveRV)
val toDoor    = Transport ("toDoor", teller, door, moveRV)

addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)

//--------------------------------------------------
// Specify Scripts for each Type of Simulation Actor

case class Customer () extends SimActor ("c", this):

    def act (): Unit =
        toTellerQ.jump ()
        if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()
        teller.utilize ()
        teller.release ()
        toDoor.jump ()
        door.leave ()
    end act

end Customer

simulate ()
waitFinished ()
Model.shutdown ()
```

```
end BankModel
```

19.6 Exercises

1. Consider how increasing the number of batches by 10 for both MBM and MIR up 100 batches/replications. effects the accuracy of the simulation. Using the data from Table 19.2, plot the grand means versus n (the number of batches/replications). Also plot the theory line (32) and discuss the converge.

2. Convergence for MIR is dependent upon the length of the runs/replications n_s. For $n_r = 40$, plot the MIR grand means as the run length n_s increase from 10 to 100,000 on a log scale (10, 100, 1000, 10,000, 100,000). Again plot the theory line (32) and discuss the converge.

3. Increasing the MBM batch size s_b can also be advantageous, in that the correlation between batches decreases. For $n_b = 40$, plot the MBM grand means as the run length s_b increase from 10 to 100,000 on a log scale (10, 100, 1000, 10,000, 100,000). On another plot, indicate the lag-1 autocorrelation between the batch means $\bar{w}_i$. What is the takeaway message?

4. Explain why the MIR run means should in independent.

5. Under what circumstances would MBM be an inappropriate method to use in a simulation study.

6. For a large number of calibration parameters or parameters having many levels, Grid Search becomes infeasible. Consult the literature for better alternatives.

7. **Question 5**: For the process-interaction simulation model of a Bank with two tellers (see Section 17.3.20: Exercise 12), determine the mean waiting time T_q three ways. For the simulations use $n_s = s_b = 1000$.

   (a) Analytic Model based on Queueing Theory. Give the formula and compute the value for T_q. Be sure to indicate the time units.

   (b) Simulation Model using the Method of Independent Replications (MIR). Give the Grand Mean and its Confidence Interval. Make sure "1 - relative precision" $\gamma \leq .1$ (may require more runs). Is the value for T_q from Queueing Theory inside this confidence interval?

   (c) Simulation Model using the Method of Batch Means (MBM). Assume the lag-1 autocorrelation ρ_1 is small enough. Give the Grand Mean and its Confidence Interval. Make sure "1 - relative precision" $\gamma \leq .1$ (may require more batches). Is the value for T_q from Queueing Theory inside this confidence interval?

Appendices

Appendix A

Optimization in Data Science

As discussed in earlier chapters, when matrix factorization cannot be applied for determining optimal values for parameters, an optimization algorithm will often need to be applied. This chapter provides a quick overview of optimization algorithms that are useful for data science. Note that the notation in the optimization field differs in that we now focus on optimizing the vector $\mathbf{x}$ rather than the parameter vector $\mathbf{b}$.

Many optimization problems may be formulated as restricted forms of the following,

$$\begin{aligned} \text{minimize } & f(\mathbf{x}) \\ \text{subject to } & \mathbf{g}(\mathbf{x}) \le \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned}$$

where $f(\mathbf{x})$ is the objective function, $\mathbf{g}(\mathbf{x}) \le \mathbf{0}$ are the inequality constraints, and $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are the equality constraints. Consider the example below.

$$\begin{aligned} \text{minimize } & f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2 \\ \text{subject to } & \mathbf{g}(\mathbf{x}) = [x_1 - 3, x_2 - 1] \le \mathbf{0} \\ & h(\mathbf{x}) = x_1 - x_2 = 0 \end{aligned}$$

If we ignore all the constraints, the optimal solution is $\mathbf{x} = [4, 2]$ where $f(\mathbf{x}) = 0$, while enforcing the inequality constraints makes this solution infeasible. The new optimal solution is $\mathbf{x} = [3, 1]$ where $f(\mathbf{x}) = 2$. Finally, the optimal solution when all constraints are enforced is $\mathbf{x} = [1, 1]$ where $f(\mathbf{x}) = 10$. Note, for this example there is just one equality constraint that forces $x_1 = x_2$.

A.1 Partial Derivatives and Gradients

These topics were introduced in the chapter on Linear Algebra, but are probed in more depth here.

DEFINITION: The *partial derivative* w.r.t. x_j of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ ($y = f(\mathbf{x})$) is defined as follows.

$$\boxed{\frac{\partial f}{\partial x_j} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_j + h, \ldots, x_n) - f(\mathbf{x})}{h}} \tag{A.1}$$

It indicates the rate of change in function f with small changes to the x_j coordinate in $\mathbb{R}^n$ space. All the other coordinates are held fixed.

A.1.1 Basic Rules

Many of the rules from univariate calculus carry over directly.

PROPOSITION: Addition Rule:

$$\frac{\partial}{\partial x_j}(f + g) = \frac{\partial f}{\partial x_j} + \frac{\partial g}{\partial x_j} \tag{A.2}$$

PROPOSITION: Subtraction Rule:

$$\frac{\partial}{\partial x_j}(f - g) = \frac{\partial f}{\partial x_j} - \frac{\partial g}{\partial x_j} \tag{A.3}$$

PROPOSITION: Product Rule:

$$\frac{\partial}{\partial x_j}(fg) = \frac{\partial f}{\partial x_j} g + f \frac{\partial g}{\partial x_j} \tag{A.4}$$

PROPOSITION: Quotient Rule:

$$\frac{\partial}{\partial x_j}(f/g) = \frac{\frac{\partial f}{\partial x_j} g - f \frac{\partial g}{\partial x_j}}{g^2} \tag{A.5}$$

A.1.2 Chain Rules

There are multiple chains rules involving partial derivatives of composite functions.

PROPOSITION: Let $h(\mathbf{x}) = f(g(\mathbf{x}))$ be the composition $h = f \circ g$ where $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}$, and $u = g(\mathbf{x})$ then

$$\frac{\partial h}{\partial x_j} = \frac{df}{du} \frac{\partial u}{\partial x_j} \tag{A.6}$$

PROPOSITION: Given $f(x(t), y(t))$ where x and y may be thought of functions of time t, the derivative is

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt} \tag{A.7}$$

These rules can be generalized to higher dimensions. Naturally, there are additional chain rules for more complex functional compositions.

A.1.3 Gradient

Definition: The *gradient* of multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ ($y = f(\mathbf{x})$) is defined as follows.

$$\nabla f \;=\; \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right] \tag{A.8}$$

It is the n-dimensional vector of partial derivatives. At each point in $\mathbb{R}^n$, it is orthogonal to the contour curves of the response variable y and points in the direction of steepest increase.

The gradient may be written more concisely as follows

$$\nabla f \;=\; [\,\partial_{x_1} f,\, \partial_{x_2} f,\, \ldots,\, \partial_{x_n} f\,] \tag{A.9}$$

where $\partial_{x_j} f$ is more concise notation for $\dfrac{\partial f}{\partial x_j}$. Furthermove, we may use this notation for gradients,

$$\partial_{\mathbf{x}} f = \nabla f \tag{A.10}$$

A.1.4 Generalized Chain Rules

Proposition: Given two multi-variate, vector-valued functions $\mathbf{f} : \mathbb{R}^p \to \mathbb{R}^q$ and $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^p$, the Jacobian chain rule is

$$\mathbb{J}_{\mathbf{f}\circ\mathbf{g}}(\mathbf{x}) \;=\; \mathbb{J}_{\mathbf{f}}(\mathbf{g}(\mathbf{x}))\, \mathbb{J}_{\mathbf{g}}(\mathbf{x}) \tag{A.11}$$

The two Jacobian matrices are multiplied together. Consider the case where $n = 4$, $p = 3$, $q = 2$ and $\mathbf{h} = [h_0, h_1] = \mathbf{f} \circ \mathbf{g}$

$$\begin{bmatrix} \partial_{x_0} h_0 & \partial_{x_1} h_0 & \partial_{x_2} h_0 & \partial_{x_3} h_0 \\ \partial_{x_0} h_1 & \partial_{x_1} h_1 & \partial_{x_2} h_1 & \partial_{x_3} h_1 \end{bmatrix} \;=\; \begin{bmatrix} \partial_{u_0} f_0 & \partial_{u_1} f_0 & \partial_{u_2} f_0 \\ \partial_{u_0} f_1 & \partial_{u_1} f_1 & \partial_{u_2} f_1 \end{bmatrix} \begin{bmatrix} \partial_{x_0} g_0 & \partial_{x_1} g_0 & \partial_{x_2} g_0 & \partial_{x_3} g_0 \\ \partial_{x_0} g_1 & \partial_{x_1} g_1 & \partial_{x_2} g_1 & \partial_{x_3} g_1 \\ \partial_{x_0} g_2 & \partial_{x_1} g_2 & \partial_{x_2} g_2 & \partial_{x_3} g_2 \end{bmatrix}$$

where $\mathbf{u} = [u_0, u_1, u_2] = \mathbf{g}(\mathbf{x})$. Now consider the case when $q = 1$, i.e., function f is scalar-valued, then

$$\partial_{\mathbf{x}} f \circ \mathbf{g}(\mathbf{x}) \;=\; \partial_{\mathbf{u}} f(\mathbf{g}(\mathbf{x}))\, \mathbb{J}_{\mathbf{g}}(\mathbf{x}) \tag{A.12}$$

In other words, the gradient of h w.r.t. $\mathbf{x}$ as a 1-by-n matrix (effectively a vector) is the product of the gradient of f as a 1-by-p matrix and the Jacobian of $\mathbf{g}$ as a p-by-n matrix. Further consider the case when $p = 1$ making g a scalar valued function. The gradient of h w.r.t. $\mathbf{x}$ as a 1-by-n matrix (vector) becomes the product of the gradient of f as a 1-by-1 matrix (scalar) and the gradient of g as a 1-by-n matrix (vector).

$$\partial_{\mathbf{x}} f \circ g(\mathbf{x}) \;=\; \partial_u f(g(\mathbf{x}))\, \partial_{\mathbf{x}} g(\mathbf{x}) \tag{A.13}$$

Note that the scalar $\partial_u f(g(\mathbf{x}))$ is an ordinary derivative. Focusing on one dimension in the vector $\mathbf{x}$, x_j, the j^{th} partial derivative is then

$$\partial_{x_j} h(\mathbf{x}) \;=\; \partial_{x_j} f \circ g(\mathbf{x}) \;=\; \partial_u f(g(\mathbf{x}))\, \partial_{x_j} g(\mathbf{x}) \tag{A.14}$$

When the arguments to the functions are understood (and hence dropped), this can be shortened to

$$\partial_{x_j} h \;=\; \partial_u f \, \partial_{x_j} g \tag{A.15}$$

These formulas are summarized in Table A.2 for $\mathbf{f} : \mathbb{R}^p \to \mathbb{R}^q$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^p$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{u} = \mathbf{g}(\mathbf{x}) \in \mathbb{R}^p$.

Table A.1: Chain Rules for $\mathbf{f} \circ \mathbf{g}(\mathbf{x})$ with argument $\mathbf{x}$ understood

Name	n	p	q	**LHS**	**Space**	**Chain Rule**	**Type**
CR1	multi	multi	multi	$\mathbb{J}_{\mathbf{f}\circ\mathbf{g}}$	$\mathbb{R}^{n\times q}$	$\mathbb{J}_{\mathbf{f}}(\mathbf{g})\, \mathbb{J}_{\mathbf{g}}$	Jacobian
CR2	multi	multi	single	$\partial_{\mathbf{x}} f \circ \mathbf{g}$	$\mathbb{R}^n$	$\partial_{\mathbf{u}} f(\mathbf{g})\, \mathbb{J}_{\mathbf{g}}$	Gradient
CR3	multi	single	single	$\partial_{\mathbf{x}} f \circ g$	$\mathbb{R}^n$	$\partial_u f(g)\, \partial_{\mathbf{x}} g$	Gradient
CR4	single	single	single	$\partial_x f \circ g$	$\mathbb{R}$	$\partial_u f(g)\, \partial_x g$	Derivative

When the function is univariate and scalar, the ∂ symbol is understood by context to be an ordinary derivative, rather than a partial derivative.

The first two rules can be re-written as shown below by extending the concise notation to include Jacobians, i.e., $\partial_{\mathbf{x}}\mathbf{g} = \mathbb{J}_{\mathbf{g}}$ where both $\mathbf{x}$ and $\mathbf{g}$ are vector-valued.

Table A.2: Revised Chain Rules

Name	n	p	q	**LHS**	**Space**	**Chain Rule**	**Type**
CR1	multi	multi	multi	$\partial_{\mathbf{x}}\mathbf{f} \circ \mathbf{g}$	$\mathbb{R}^{n\times q}$	$\partial_{\mathbf{u}}\mathbf{f}(\mathbf{g})\, \partial_{\mathbf{x}}\mathbf{g}$	Jacobian
CR2	multi	multi	single	$\partial_{\mathbf{x}} f \circ \mathbf{g}$	$\mathbb{R}^n$	$\partial_{\mathbf{u}} f(\mathbf{g})\, \partial_{\mathbf{x}}\mathbf{g}$	Gradient

A.1.5 `calculus` Package

Calculus operations on functions are provided in the `calculus` package, including the `Differential` object.

`Differential` Object

```
object Differential:

    def resetH (step: Double): Unit = h = step; h2 = h + h; hh = h * h; hh4 = 4.0 * hh
    def resetHR (largeStep: Double): Unit = hl = largeStep
    def fdifference (f: FunctionS2S, x: Double): Double = (f(x + hl) - f(x)) / hl
    def derivative1 (f: FunctionS2S, x: Double): Double = (f(x + h) - f(x)) / h
    def derivative (f: FunctionS2S, x: Double): Double = (f(x + h) - f(x - h)) / h2
    def partial (i: Int)(f: FunctionV2S, x: VectorD): Double =
        (f(x + (i, h)) - f(x - (i, h))) / h2
    def grad (f: FunctionV2S, x: VectorD): VectorD =
    def slope (f: FunctionV2S, x: VectorD, n: Int = 0): VectorD =
    def jacobian (f: Array [FunctionV2S], x: VectorD): MatrixD =
    def eval (f: Array [FunctionV2S], x: VectorD): VectorD =
    def derivative2 (f: FunctionS2S, x: Double): Double =
        (f(x + h) - 2.0*f(x) + f(x - h)) / hh
    def partial2 (i: Int, j: Int)(f: FunctionV2S, x: VectorD): Double =
    def hessian (f: FunctionV2S, x: VectorD): MatrixD =
    def laplacian (f: FunctionV2S, x: VectorD): Double =

end Differential
```

Most of these methods also have math-like Unicode equivalents (see code for details). The most common first order methods are shown below.

```
    @param f  the function whose derivative is sought
    @param x  the point (scalar) at which to estimate the derivative

    def derivative (f: FunctionS2S, x: Double): Double = (f(x + h) - f(x - h)) / h2

    @param i  the dimension to compute the partial derivative on
    @param f  the function whose partial derivative is sought
    @param x  the point (vector) at which to estimate the partial derivative

    def partial (i: Int)(f: FunctionV2S, x: VectorD): Double =
        (f(x + (i, h)) - f(x - (i, h))) / h2

    @param f  the function whose gradient is sought
    @param x  the point (vector) at which to estimate the gradient

    def grad (f: FunctionV2S, x: VectorD): VectorD =
        VectorD (for i <- x.indices yield (f(x + (i, h)) - f(x - (i, h))) / h2)

    @param f  the array of functions whose Jacobian is sought
    @param x  the point (vector) at which to estimate the Jacobian

    def jacobian (f: Array [FunctionV2S], x: VectorD): MatrixD =
        MatrixD (for i <- f.indices yield grad (f(i), x))
```

A.2 Automatic Differentiation

As we learned Neural Networks work because of back-propagation, but this requires manual development of partial derivatives. Notice that for a Gated Recurrent Unit (GRU), getting the partial derivatives correct is not so easy, and with newer architectures, it is even harder.

Automatic Differentiation [63, 142, 13] allows one to specify the equations for forward propagation and have the system automatically handle the backward propagation (and even generalize it).

This has opened up a new research area are called differential programming [200] where model developers can specify parameterized equations and the system can automatically fit the parameters (based on data) using first or second order optimizers.

A.2.1 Forward Propagation

To keep things simple, suppose we make a forward pass in a perceptron using a single instance $\mathbf{x} \in X$ and $y \in \mathbf{y}$, i.e., $\mathbf{x} \in \mathbb{R}^n$, and $y \in \mathbb{R}$. Then the main purpose of the forward pass to estimate the prediction vector $\hat{y}$. For further simplification, let us assume the bias is handled by having $x_0 = 1$.

$$\hat{y} \;=\; f(\mathbf{x} \cdot \mathbf{b}) \tag{A.16}$$

Now the weight vector (not matrix) is $\mathbf{b} \in \mathbb{R}^n$. In addition, the forward pass calculates the loss function.

$$\mathcal{L} \;=\; \frac{1}{2}(y - \hat{y})^2 \;=\; \frac{1}{2}(y - f(\mathbf{x} \cdot \mathbf{b}))^2 \tag{A.17}$$

A.2.2 Reverse Mode Backward Propagation

The purpose of the backward pass is to calculate partial derivatives using chain rules w.r.t. the parameters $\mathbf{b}$. The chain rule applied in this case is CR3 with $u = f(\mathbf{x} \cdot \mathbf{b})$.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} \;=\; \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial \mathbf{b}} \tag{A.18}$$

The chain rule applied can be applied again with $v = \mathbf{x} \cdot \mathbf{b}$ to obtain.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} \;=\; \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial \mathbf{b}} \tag{A.19}$$

$$\partial_{\mathbf{b}}\mathcal{L} \;=\; \partial_u \mathcal{L}\; \partial_v f(\mathbf{x} \cdot \mathbf{b})\; \partial_{\mathbf{b}}(\mathbf{x} \cdot \mathbf{b}) \tag{A.20}$$

The formulas for the forward and backward passes are shown on the left and right, respectively.

$$\mathcal{L}(u) \;=\; \frac{1}{2}(y - u)^2 \;\;\rightarrow\;\; \partial_u \mathcal{L} \;=\; -(y - u) \tag{A.21}$$

$$u \;=\; f(\mathbf{x} \cdot \mathbf{b}) \;\;\rightarrow\;\; \partial_v f(\mathbf{x} \cdot \mathbf{b}) \;=\; f'(v) \tag{A.22}$$

$$v \;=\; \mathbf{x} \cdot \mathbf{b} \;\;\rightarrow\;\; \partial_{\mathbf{b}}(\mathbf{x} \cdot \mathbf{b}) \;=\; \mathbf{x} \tag{A.23}$$

The calculations may be depicted in a computation graph as shown in Figure A.1. Forward calculations (left-to-right) are shown above the nodes, while backward calculations (right-to-left) are shown below the nodes.

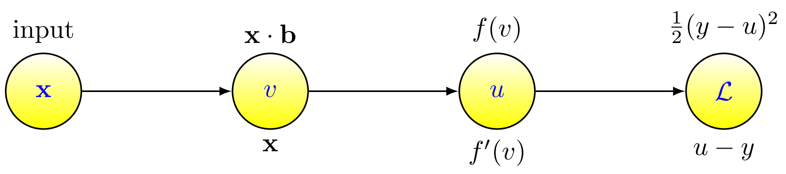


Figure A.1: Computation Graph for Perceptron

A.2.3 Example Calculation for Perceptron

Consider the following example calculation described in the Perceptron section (Neural Networks chapter) with the parameters initialized to $\mathbf{b} = [.1, .2., .1]$, the inputs to $\mathbf{x} = [1, .5, 1]$ corresponding to the sixth row in exercise 7, and the output to $y = .3$. The sigmoid function is used for activation (see the Perceptron section for its function f and derivative f'). Figure A.2 shows the calculated values for the forward and backward passes.

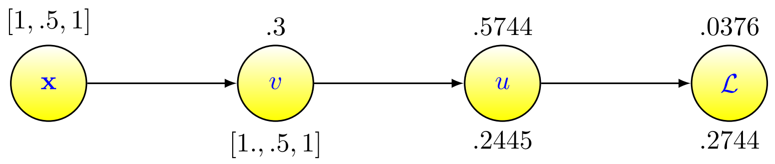


Figure A.2: Computation Graph Results for Perceptron

Label each node v_i, and then the partial derivative calculations simply accumulate the multiplications left-to-right according to following formula

$$\bar{v}_i = \bar{v}_{i+1} \frac{\partial v_{i+1}}{\partial v_i} \tag{A.24}$$

where $\bar{v}_i = \dfrac{\partial \mathcal{L}}{\partial v_i}$ and $\bar{v}_4 = 1$. This value $\bar{v}_i$ is called the *adjoint*, a term used in adjoint methods that provide more efficient ways of calculating derivatives [54].

The automatic differentiation calculations are summarized in Tables A.3 and A.4 symbolically and numerically.

Table A.3: Forward and Backward Symbolic Calculations for Perceptron Example

Node i	**Value** v_i	**Adjoint** $\bar{v}_i$	**Comment** (left, right)
0	$\mathbf{x}$	-	input, NA
1	$v = \mathbf{x} \cdot \mathbf{b}$	$\mathbf{x}\, \epsilon\, f'(v)$	pre-activation, gradient $\partial_{\mathbf{b}} \mathcal{L}$
2	$u = f(v)$	$\epsilon\, f'(v)$	prediction $\hat{y}$, delta δ
3	$\frac{1}{2}(y-u)^2$	$\epsilon = u - y$	loss, negative error
4	-	1	NA, multiplicative identity

Table A.4: Forward and Backward Calculations for Perceptron Example

Node i	**Value** v_i	**Adjoint** $\bar{v}_i$	**Comment** (left, right)
0	[1, .5, 1]	-	input, NA
1	.3	[.0671, .0335, .0671]	pre-activation, gradient $\partial_{\mathbf{b}}\mathcal{L}$
2	.5744	.0671	prediction $\hat{y}$, delta δ
3	.0376	.2744	loss, negative error
4	-	1	NA, multiplicative identity

The gradient $\partial_{\mathbf{b}}\mathcal{L}$ is then used for making parameter updates.

$$\mathbf{b} \;=\; \mathbf{b} \,-\, \partial_{\mathbf{b}}\mathcal{L}\,\eta \tag{A.25}$$

With the learning rate $\eta = 1$, the new parameter values will be $\mathbf{b} = [-.1666, .0778, -.1666]$. Notice that this update differs from the one given in the Perceptron section as that one is based on all the rows in data matrix X, whereas this one only uses row six.

Computation on Full Input

The previous example used a single row (the sixth) $\mathbf{x} \in X$ to perform the computation, as would be done with pure (single instance) Stochastic Gradient Descent. On the other hand, Gradient Descent would use the full input data/training matrix X. The updated two step chain rule becomes the following:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial \mathbf{b}} \tag{A.26}$$

This states that the gradient of the loss function $\mathcal{L}$ w.r.t. the parameters $\mathbf{b}$ is the product of the gradient of the loss function $\mathcal{L}$ w.r.t. $\mathbf{u}$ times the Jacobian of $\mathbf{u}$ w.r.t. $\mathbf{v}$ times the Jacobian of $\mathbf{v}$ w.r.t. $\mathbf{b}$. The two Jacobian matrices are diagonal because the activation function maps individual elements, i.e.,

$$u_i \;=\; f(v_i) \quad\Longrightarrow\quad \partial_{v_j} u_i \;=\; 0 \;\; \text{when} \;\; i \neq j \tag{A.27}$$

Thus, the matrix multipication becomes the element-wise vector product of three vectors.

$$\partial_{\mathbf{u}}\mathcal{L} * \text{diag}(\partial_{\mathbf{v}}\mathbf{u}) * \text{diag}(\partial_{\mathbf{b}}\mathbf{v}) \tag{A.28}$$

The full input calculations are depicted in the computation graph shown in Figure A.3.

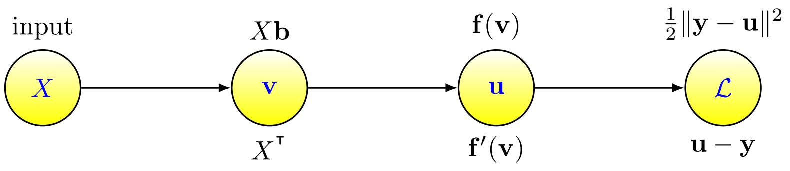


Figure A.3: Computation Graph for Perceptron - Input X

Calculations for the full input case are summarized in Table A.5. Where there is a vector of length nine, the previous table's numbers correspond to the sixth element. The overall loss and gradient should not agree as in the first table they are based in one row, while in this table they are based on all nine rows.

Table A.5: Forward and Backward Calculations for Perceptron Input X Example

Node i	**Value** v_i	**Adjoint** $\bar{v}_i$
0	X	-
1	[.1, .15, .2, .2, .25, .3, .3, .35, .4]	[.040, -.122, .189]
2	[.525,.537,.540,.540,.562,.574,.574,.587,.599]	[.006, .059, .087, -.062, .015, .067, -.104, -.052, .024]
3	.279	[.025, .237, .350, -.250, .0622, .274, -.426, -.213, .099]
4	-	1

A.2.4 Example for Three-Layer Neural Network

Automatic Differentiation simply requires the specification of the prediction equation and the loss function. The prediction equation for a Three-Layer (one hidden) Neural Network is given below.

$$\hat{Y} \;=\; \mathbf{f}_1(\mathbf{f}_0(XA+\boldsymbol{\alpha})B+\boldsymbol{\beta}) \tag{A.29}$$

Using one half *sse* as the loss function, it may be expressed as one half the Frobenius norm squared.

$$\mathcal{L} \;=\; \frac{1}{2}\|Y-\mathbf{f}_1(\mathbf{f}_0(XA+\boldsymbol{\alpha})B+\boldsymbol{\beta})\|_F^2 \tag{A.30}$$

Partial derivatives are now needed for all weight matrices and bias vectors:

$$\partial_{a_{jh}}\mathcal{L},\; \partial_{\boldsymbol{\alpha}_h}\mathcal{L},\; \partial_{b_{hk}}\mathcal{L},\; \partial_{\boldsymbol{\beta}_k}\mathcal{L} \tag{A.31}$$

Or in aggregated form.

$$\partial_A\mathcal{L},\; \partial_{\boldsymbol{\alpha}}\mathcal{L},\; \partial_B\mathcal{L},\; \partial_{\boldsymbol{\beta}}\mathcal{L} \tag{A.32}$$

Ignoring biases (or using the *Bias Trick* to incorporate them into weight matrices), the prediction equation becomes the following:

$$\hat{Y} \;=\; \mathbf{f}_1(\mathbf{f}_0(XA)B) \tag{A.33}$$

During back-propagation, calculation of $\dfrac{\partial\mathcal{L}}{\partial B}$ needed to update weight matrix B occurs first, followed by the calculation of $\dfrac{\partial\mathcal{L}}{\partial A}$ needed to update weight matrix A.

A.2.5 Partial Derivatives w.r.t. B

The partial derivative of the loss function $\mathcal{L}$ w.r.t. the second weight matrix B

$$\partial_B\mathcal{L} \;=\; \frac{\partial\mathcal{L}}{\partial B} \;=\; \left[\frac{\partial\mathcal{L}}{\partial b_{hk}}\right] \tag{A.34}$$

is an n_z-by-n_y matrix and can be decomposed using the following matrix calculus chain rule.

$$\frac{\partial \mathcal{L}}{\partial B} = \frac{\partial \mathcal{L}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial U} \frac{\partial U}{\partial B} \tag{A.35}$$

Figure A.4 represents a computation graph for three layer neural network relevant to $\dfrac{\partial \mathcal{L}}{\partial B}$.

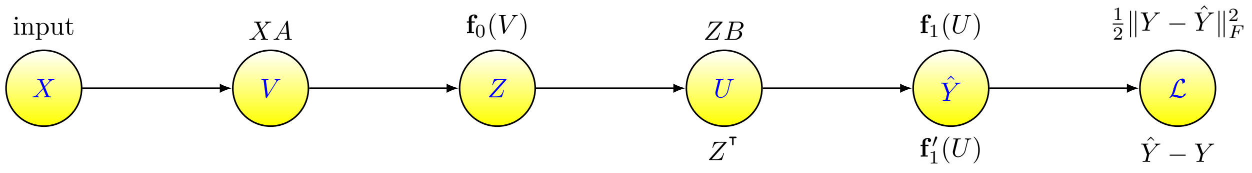


Figure A.4: Computation Graph for Three Layer Neural Network

To better illustrate how the partial derivative products accumulate right-to-left, Table A.6 shows the forward and backward calculations relevant to weight matrix B for three layer neural networks symbolically.

Table A.6: Forward and Backward Matrix Calculations Relevant to B for Neural Network Example

Node i	**Value** v_i	**Adjoint** $\bar{v}_i$	**Comment** (left, right)
0	X	-	input, NA
1	$V = XA$	-	hidden layer pre-activation, gradient
2	$Z = \mathbf{f}_0(V)$	-	hidden layer values, delta 0
3	$U = ZB$	$Z^\intercal \Delta^1$	output layer pre-activation, gradient
4	$\hat{Y} = \mathbf{f}_1(U)$	$\Delta^1 = E \odot \mathbf{f}_1'(U)$	prediction, delta 1
5	$\mathcal{L} = \frac{1}{2}\|Y - \hat{Y}\|_F^2$	$E = \hat{Y} - Y$	loss, negative error
6	-	1	NA, multiplicative identity

The parameter update for weight matrix B is then

$$B = B - Z^\intercal \Delta^1 \eta \tag{A.36}$$

i.e., move in the direction opposite the gradient ($Z^\intercal \Delta^1$) moderated by the learning rate η.

A.2.6 Partial Derivatives w.r.t. A

The partial derivative of the loss function $\mathcal{L}$ w.r.t. the first weight matrix A

$$\partial_A \mathcal{L} = \frac{\partial \mathcal{L}}{\partial A} = \left[\frac{\partial \mathcal{L}}{\partial a_{jh}}\right] \tag{A.37}$$

is an n-by-n_z matrix and can be decomposed using the following matrix calculus chain rule.

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial U} \frac{\partial U}{\partial Z} \frac{\partial Z}{\partial V} \frac{\partial V}{\partial A} \tag{A.38}$$

Figure A.5 represents a computation graph for three layer neural network relevant to $\dfrac{\partial \mathcal{L}}{\partial A}$.

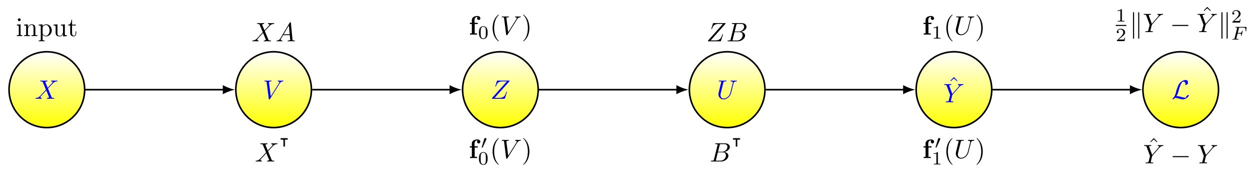


Figure A.5: Computation Graph for Three Layer Neural Network

Notice that the forward pass for the two computation graphs are identical, while the backward pass is the same until computing the partial derivative for node U. Table A.7 shows the forward and backward calculations relevant to weight matrix A for three layer neural networks symbolically.

Table A.7: Forward and Backward Matrix Calculations Relevant to A for Neural Network Example

Node i	**Value** v_i	**Adjoint** $\bar{v}_i$	**Comment** (left, right)
0	X	-	input, NA
1	$V = XA$	$X^\intercal\Delta^0$	hidden layer pre-activation, gradient
2	$Z = \mathbf{f}_0(V)$	$\Delta^0 = \Delta^1 B^\intercal \odot \mathbf{f}_0'(V)$	hidden layer values, delta 0
3	$U = ZB$	$\Delta^1 B^\intercal$	output layer pre-activation, gradient
4	$\hat{Y} = \mathbf{f}_1(U)$	$\Delta^1 = E \odot \mathbf{f}_1'(U)$	prediction, delta 1
5	$\mathcal{L} = \frac{1}{2}\|Y-\hat{Y}\|_F^2$	$E = \hat{Y} - Y$	loss, negative error
6	-	1	NA, multiplicative identity

The parameter update for weight matrix A is then

$$A \;=\; A - X^\intercal\Delta^0\,\eta \tag{A.39}$$

i.e., move in the direction opposite the gradient $(X^\intercal\Delta^0)$ moderated by the learning rate η.

A.3 Gradient Descent

One the simplest algorithms for unconstrained optimization is Gradient Descent (GD). Imagine you are in a mountain range at some point $\mathbf{x}$ with elevation $f(\mathbf{x})$. Your goal is the find the valley (or ideally the lowest valley). Look around (assume you cannot see very far) and determine the direction and magnitude of steepest ascent. This is the gradient.

Using the objective/cost function from the beginning of the chapter,

$$\text{minimize } f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2$$

the gradient of the objective function $\nabla f(\mathbf{x})$ is the vector formed by the partial derivatives $\left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right]$

$$\nabla f(\mathbf{x}) \; = \; [2(x_1 - 4), 2(x_2 - 2)]$$

In its most elemental form the algorithm simply moves in the direction that is opposite to the gradient $-\nabla f(\mathbf{x})$ and a distance determined by the magnitude of the gradient. Unfortunately, at some points in the search space the magnitude of the gradient may be very large and moving that distance may result in divergence (you keep getting farther away from the valley). One solution is to temper the gradient by multiplying it by a learning rate η (tunable hyper-parameter typically smaller than one). Using a tuned learning rate, update your current location $\mathbf{x}$ as follows:

$$\mathbf{x} \; = \; \mathbf{x} \; - \; \eta \nabla f(\mathbf{x}) \qquad \text{GD Update Equation} \tag{A.40}$$

Repeat this process until a stopping rule signals sufficient convergence. Examples of stopping rules include stop when the change to $\mathbf{x}$ or $f(\mathbf{x})$ becomes small or after the objective function has increased for too many consecutive iterations/steps.

A.3.1 Line Search

Notice that the gradient is re-evaluated at every iteration/step and that it is unclear how far to move in the direction opposite the gradient (hence the need/annoyance of tuning the learning rate). Adding a line search may help with these issues. The idea is that the gradient gives you a direction to follow that may work well for awhile. (Note, line search may be used with other optimization algorithms, such as Quasi-Newton methods.) Using a line search, you may move in that direction (straight line) so long as it productive. The line search induces a one dimensional function that reproduces the value of the original objective function along the given line.

One approach is to move along the line so long as there is sufficient decrease. Once this stops, re-evaluate the gradient and start another major iteration. An example of such an algorithm is the Wolfe Line Search. An alternative when you are confident of the extent of line search (upper limit on the range to be considered) is to use Golden Section Search that iteratively narrows down the search from the original extent.

The problem of learning rate is still there to some degree as the line search algorithms have step size as hyper-parameter. Of course, more complex variants may utilize adaptive learning rates or step sizes.

Wolfe Line Search

The parent optimization algorithm will choose a search direction $\mathbf{p}$ that is moving downward, while the gradient at the current location $\mathbf{x}$ will be in the direction of steepest increase. The basic idea is to increase/advance the displacement in the search direction α, so long as it is productive to do so. In general a line search algorithm may work in two phases: a bracketing phases and a selection phase. The bracketing phase finds an interval of acceptable step lengths that satisfy certain conditions, e.g., Wolfe condition 1 for an upper bound and Wolfe condition 2 for a lower bound. The selection phase looks for a reasonably good solution within the interval. As line search is called frequently, one typically wants to use minimal effort in phase 2. Also, the next iteration of the parent algorithm may in a way undo some of the work done in the last line search. One simple selection algorithm is bisection search, although interpolative search may be used [136].

The Wolfe condition 1 (or Armijo condition) will be satisfied when the function at new point $f(\mathbf{x}+\alpha\mathbf{p})$ is sufficiently less than its starting value ($\alpha = 0$).

$$f(\mathbf{x}+\alpha\mathbf{p}) \;\le\; f(\mathbf{x}) \,+\, c_1\,\alpha\,\nabla f(\mathbf{x})\cdot\mathbf{p} \tag{A.41}$$

To see the Sufficient Decrease Condition (SDC) clearly, one may consider gradient descent, which will set the search direction $\mathbf{p} = -\nabla f(\mathbf{x})$, so that the above equation becomes the following:

$$f(\mathbf{x}+\alpha\mathbf{p}) \;\le\; f(\mathbf{x}) \,-\, c_1\,\alpha\,\|\nabla f(\mathbf{x})\|^2 \tag{A.42}$$

Although various optimization algorithms deflect away from the direction of steepest descent, the dot product $\nabla f(\mathbf{x})\cdot\mathbf{p}$ will be negative. For small values of c_1 (e.g., .0001), this condition will allow the new point (due to the line search) to be on a line with small negative slope emanating from $(\alpha = 0, f(\mathbf{x}))$.

The Wolfe condition 2 has Weak and Strong versions. The goal of the Weak version is to have the dot product of the gradient and search direction become less negative.

$$\nabla f(\mathbf{x}+\alpha\mathbf{p})\cdot\mathbf{p} \;\ge\; c_2\,\nabla f(\mathbf{x})\cdot\mathbf{p} \tag{A.43}$$

The idea of this Curvature Condition (CC) is that as one approaches a minimal point, this dot product will approach zero from negative values. One use of the CC condition is to make sure to advance/increase α while the decent is still very steep, i.e., some fraction (e.g., $c_2 = .9$) of the original rate of descent.

The following method returns whether Wolfe condition 1, the Sufficient Decrease Condition (SDC) is satisfied.

```
    @param fx   the functional value of the original point
    @param fy   the functional value of the new point y = x + p * a
    @param a    the displacement in the search direction
    @param gxp  the dot product of the gradient vector g(x) and the search vector p

    inline def wolfe1 (fx: Double, fy: Double, a: Double, gxp: Double): Boolean =
        fy <= fx + c1 * a * gxp
```

The next method returns whether Wolfe condition 2, the Curvature Condition (CC) is satisfied.

```
    @param p    the search direction vector
    @param gy   the gradient at new point y = x + p * a
    @param gxp  the dot product of the gradient vector g(x) and the search vector p

    inline def wolfe2 (p: VectorD, gy: VectorD, gxp: Double): Boolean =
        (gy dot p) >= c2 * gxp
```

A.3.2 Application to Data Science

The gradient descent algorithm may be applied to data science, simply by defining an appropriate objective/cost function. Since the goal is often to minimize the sum of squared errors *sse* or some similar Quality of Fit (QoF) measure, it may be used for the objective function. For a `Perceptron`, the equation has been developed for $\mathcal{L}(\mathbf{b})$

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\,(\mathbf{y} - f(X\mathbf{b})\cdot(\mathbf{y} - f(X\mathbf{b})$$

in which case the gradient is

$$\nabla\mathcal{L}(\mathbf{b}) \;=\; -\,X^{\intercal}[f'(X\mathbf{b})\,\boldsymbol{\epsilon}] \tag{A.44}$$

where $\boldsymbol{\epsilon} = \mathbf{y} - f(X\mathbf{b})$.

For each epoch, the parameter vector $\mathbf{b}$ is updated according to the GD Update Equation.

$$\mathbf{b} \;=\; \mathbf{b} \,-\, \eta\,\nabla\mathcal{L}(\mathbf{b}) \tag{A.45}$$

A.3.3 Exercises

1. Write a ScalaTion program to solve the example problem given above.

```
    // function to optimize
    def f(x: VectorD): Double = (x(0) - 4)~^2 + (x(1) - 2)~^2

    // gradient of objective function
    def grad (x: VectorD): VectorD = VectorD (?, ?)

    val x   = new VectorD (2)                // vector to optimize
    val eta = 0.1                            // learning rate

    for k <- 1 to MAX_IT do
        x -= grad (x) * eta
        println (s"$k: x = $x, f(x) = ${f(x)}, lg(x) = ${lg(x)}, p = $p, l = $l")
    end for
```

2. Add code to collect the trajectory of vector `x` in a matrix `z` and plot the two columns in the `z` matrix.

```
    val z = new MatrixD (MAX_IT, 2)        // store x's trajectory
    z(k-1) = x.copy
    new Plot (z(?, 0), z(?, 1)
```

3. The strong Wolfe condition 2 is the following:

$$|\nabla f(\mathbf{x} + \alpha\mathbf{p})\cdot\mathbf{p}| \;\le\; c_2\,|\nabla f(\mathbf{x})\cdot\mathbf{p}| \tag{A.46}$$

How does it differ from the weak condition and when is it useful?

A.4 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a foundational algorithm for Data Science. In fits in the class of stochastic optimization algorithms since the objective/loss function is noisy, as it is based on functions of randomly selected mini-batches of data instances. Such algorithms need to be robust and need not be the best optimization algorithm in a deterministic setting.

Several improved variants Stochastic Gradient Descent are incorporated into popular neural network software packages.

The `GradientDescent_NoLS` class provides methods to optimize a loss/objective function f that may be stochastic. It tries to find a value for vector $\mathbf{x}$ that minimizes the loss function. In the context of Neural Networks, $\mathbf{x}$ may be thought of the parameter vector (weights and biases). This optimizer implements a Gradient Descent with no contained Line-Search optimizer.

This algorithm is rather simple: (1) Compute the gradient vector $\mathbf{g}$ using the `grad` function. (2) Multiply it by the step-size/learning rate α and subtract this from the previous value of the parameter vector $\mathbf{x}$.

$$\mathbf{g} \;=\; \nabla f(\mathbf{x}) \qquad \text{gradient} \tag{A.47}$$

$$\mathbf{x} \;-=\; \alpha[\mathbf{g}] \qquad \text{parameter update} \tag{A.48}$$

Note, the exact relationship between the learning rate α and the hyper-parameter `eta` (η) is code dependent.

Of course the core logic shown above must be embedded in an iterative optimization algorithm (see the `solve` method in the `GradientDescent_NoLS` class.

```
@param f        the vector-to-scalar (V2S) objective/loss function
@param grad     the vector-to-vector (V2V) gradient function, grad f
@param hparam   the hyper-parameters

class GradientDescent_NoLS (f: FunctionV2S, grad: FunctionV2V,
                            hparam: HyperParameter = hp)
      extends Minimize
         with StoppingRule (hp("upLimit").toInt):      // limit on increasing loss
```

The `solve` method iterates over several time-steps or learning epochs. It computes the gradient, multiplies it by learning rate α and subtracts this product from the vector $\mathbf{x}$. A stopping rule is checked and if progress is stalled, the algorithm terminates early. This method returns the best solution found for the loss function f and the vector $\mathbf{x}$.

```
@param x0   he starting point
@param α    the step-size/learning rate

def solve (x0: VectorD, α: Double = eta): FuncVec =
    var x    = x0                                 // start parameters at initial guess
    var f_x  = -0.0                               // loss function, value indefined
    var best = (f_x, x)                           // start with best = initial

    var (go, it) = (true, 1)
    cfor (go && it <= MAX_IT, it += 1) {          // iterate over each epoch/timestep
        val g = grad (x)                          // get gradient loss function
        x  -= g * α                               // update parameters x
        f_x = f(x)                                // compute new loss function value

```

```
            best = stopWhen (f_x, x)
            if best._2 != null then go = false          // early termination, return best
        } // cfor
        if go then getBest                              // best solution found
        else best
    end solve
```

A.4.1 Using SGD to Train Neural Networks

For simplicity, this discussion will focus on the `Perceptron` and `NeuralNet_2L` classes, starting with the loss function $\mathcal{L}(\mathbf{b})$,

$$\mathcal{L}(\mathbf{b}) \;=\; \frac{1}{2}\,(\mathbf{y} - f(X\mathbf{b})) \cdot (\mathbf{y} - f(X\mathbf{b}))$$

an estimate of the gradient is computed for a limited number of instances (a *mini-batch*). Several non-overlapping mini-batches are created simultaneous by taking a random permutation of the row indices of data/input matrix X. The permutation is split into n_B mini-batches. Letting i_B be the indices for the i^{th} mini-batch and $X[i_B]$ be the projection of matrix X onto the rows in i_B, the estimate for the gradient is simply

$$\nabla\mathcal{L}(\mathbf{b}) \;=\; -\,X[i_B]^{\intercal}\,[f'(X[i_B]\mathbf{b})\,\boldsymbol{\epsilon}] \tag{A.49}$$

where $\boldsymbol{\epsilon} = \mathbf{y}[i_B] - f(X[i_B]\mathbf{b})$. Using the definition of the delta vector

$$\boldsymbol{\delta} \;=\; -\,f'(X[i_B]\mathbf{b})\,\boldsymbol{\epsilon}$$

the gradient becomes

$$\nabla\mathcal{L}(\mathbf{b}) \;=\; X[i_B]^{\intercal}\boldsymbol{\delta}$$

For each epoch, n_B mini-batches are created. For each mini-batch, the parameter vector $\mathbf{b}$ is updated according to this equation, using that mini-batch's estimate for the gradient.

$$\mathbf{b} \;=\; \mathbf{b} \,-\, \eta\nabla\mathcal{L}(\mathbf{b}) \;=\; \mathbf{b} \,-\, X[i_B]^{\intercal}\boldsymbol{\delta}\eta \tag{A.50}$$

At a high level, the `optimize2` method for the Optimize_SGD class shown in the `NeuralNet_2L` section works as follows:

1. The outermost loop makes many complete passes through the training set portion of the dataset. Each pass may be thought of a learning epoch.

2. The inner loop will iterate through all the mini-batches, updating the parameters (weights and biases) based on delta corrections (combination of errors and slopes).

3. The `updateWeight` method simply encodes the boxed equations from the `Perceptron` section: computing predicted output, the negative of the error vector, the delta vector, and a mini-batch size normalized learning rate, and finally, returning the parameter vector `b` update.

4. The last part of the outer loop computes the new loss function and applies a stopping rule for early termination. The parameter settings with the lowest loss function are recorded and returned along with number of epochs.

5. The final line of `optimize` simply returns the value of the loss function and number epochs used by the algorithm, when there is no early termination.

The `optimize3` and `optimize` methods in the `Optimize_SGD` class are for `NeuralNet_3L` and `NeuralNet_XL`, respectively.

A.5 Stochastic Gradient Descent with Momentum

To better handle situations where the gradient becomes small or erratic, previous values of the gradient can be weighed in with the current gradient. Their contributions can be exponentially decayed, so that recent gradients have greater influence. The decay rate $\beta \in [0, 1]$ indicates how fast prior gradients are discounted. If $\beta = 0$, momentum is not used. In ScalaTion, the hyper-parameter `beta` (β) is set to 0.9, but can easily be changed.

These contributions may be collected via the gradient-based parameter updates to the parameter vector $\mathbf{x}$ [55]. First the gradient vector $\mathbf{g}$ is computed using the `grad` function. The gradient-based momentum vector $\mathbf{p}$ is calculated as the weighted average of the current gradient and the previous momentum. Finally, the parameter vector $\mathbf{x}$ is updated as a weighted average of the current gradient and the momentum, moderated by the learning rate $\alpha > 0$ (derived from the hyper-parameter eta (η) in some code).

$$\mathbf{g} \;=\; \nabla f(\mathbf{x}) \qquad \text{gradient} \qquad (A.51)$$

$$\mathbf{p} \;=\; (1 - \beta)\, \mathbf{g} + \beta\, \mathbf{p} \qquad \text{momentum} \qquad (A.52)$$

$$\mathbf{x} \;-\!=\; \alpha[(1 - \nu)\, \mathbf{g} + \nu\, \mathbf{p}] \qquad \text{parameter update} \qquad (A.53)$$

If β is zero, that algorithm behaves the same as Stochastic Gradient Descent. At the other extreme, if β is 1, there is no decay and all previous gradients will weigh in, so eventually the new gradient value will have little impact and the algorithm will become oblivious to its local environment.

The third hyper-parameter ν determines how much weight to place on the current gradient versus the momentum when updating the parameter vector $\mathbf{x}$. There are two special cases to consider.

1. When $\nu = 0$, the parameter update becomes

$$\mathbf{x} \;-\!=\; \alpha[\mathbf{g}] \qquad \text{parameter update} \qquad (A.54)$$

   so the algorithm becomes Stochastic Gradient Descent.

2. When $\nu = 1$, the parameter update becomes

$$\mathbf{x} \;-\!=\; \alpha[\mathbf{p}] \qquad \text{parameter update} \qquad (A.55)$$

   so the algorithm becomes Stochastic Gradient Descent using only Momentum.

See [55] for a more nuanced discussion on the variety of stochastic gradient descent algorithms that use momentum.

```
    @param f        the vector-to-scalar (V2S) objective/loss function
    @param grad     the vector-to-vector (V2V) gradient function, grad f
    @param hparam   the hyper-parameters

    class GradientDescent_Mo (f: FunctionV2S, grad: FunctionV2V,
                              hparam: HyperParameter = hp)
          extends Minimize
             with StoppingRule (hparam("upLimit").toInt):  // limit on increasing loss
```

The solve method iterates over several time-steps or learning epochs. It computes the gradient and uses it to recompute the momentum as the weighted average of this gradient and the previous momentum. The update to the vector $\mathbf{x}$ is another weighted average of this gradient and the updated momentum. This update is then multiplied by learning rate α and subtracted from the vector $\mathbf{x}$. A stopping rule is checked and if progress is stalled, the algorithm terminates early. This method returns the best solution found for the loss function f and the vector $\mathbf{x}$.

```
@param x0   the starting point
@param α    the step-size/learning rate

def solve (x0: VectorD, α: Double = eta): FuncVec =
    var p    = new VectorD (x0.dim)                // momentum-based aggregated gradient
    var x    = x0                                  // start parameters at initial guess
    var f_x  = -0.0                                // loss function, value indefined
    var best = (f_x, x)                            // start with best = initial

    var (go, it) = (true, 1)
    cfor (go && t <= MAX_IT, it += 1) {            // iterate over each epoch/timestep
        val g = grad (x)                           // get gradient of loss function
        p   = g * (1 - β) + p * β                  // update momentum-based agg. gradient
        x  -= (g * (1 - ν) + p * ν) * α            // update parameters
        f_x = f(x)                                 // compute new loss function value

        best = stopWhen (f_x, x)
        if best._2 != null then go = false         // early termination, return best
    } // cfor
    if go then getBest                             // best solution found
    else best
end solve
```

A.5.1 Using SGDM to Train Neural Networks

To utilize this algorithm for training Neural Networks (e.g., for NeuralNet_2L) the loss function $\mathcal{L}(\mathbf{b})$ and its gradient $\nabla\mathcal{L}(\mathbf{b})$ are required.

$$\mathcal{L}(\mathbf{b}) = \|y - f(XB)\|_F^2 \tag{A.56}$$

$$\nabla\mathcal{L}(\mathbf{b}) = X[i_B]^\intercal \boldsymbol{\delta} \tag{A.57}$$

where $\boldsymbol{\delta} = f'(XB) \odot \epsilon$.

The Optimizer_SGDM class has three optimization methods supporting Stochastic Gradient Descent with Momentum. The optimize2 method is for NeuralNet_2L. It takes the input x and output y matrices, the initial guess for parameters bb, the initial learning rate eta, and the activation function ff.

```
@param x     the m-by-n input matrix (training data consisting of m input vectors)
@param y     the m-by-ny output matrix (training data consisting of m output vectors)
@param bb    the array of parameters (weights & biases) between every two adjacent layers
@param eta   the initial learning/convergence rate
@param ff    the array of activation function family for every two adjacent layers

def optimize2 (x: MatrixD, y: MatrixD,
                bb: NetParams, eta: Double, ff: Array [AFF]): (Double, Int) =
```

The method initializes several constants and variables, mainly related to the hyper-parameters. The permutation generator is use to create random mini-batches.

```
        val permGen   = permGenerator (x.dim)             // permutation vector generator
        val b         = bb(0)                              // net-params: weights and biases
        val f         = ff(0)                              // activation function
        val bSize     = min (hp("bSize").toInt, x.dim)    // batch size
        val maxEpochs = hp("maxEpochs").toInt             // maximum number of epochs
        val upLimit   = hp("upLimit").toInt               // limit on increasing lose
        val β         = hp("beta").toDouble               // momentum hyper-parameter
        val ν         = hp("nu").toDouble                 // 0 => SGD, 1 => (normalized) SHB
        var η         = eta                               // set initial learning rate
        val nB        = x.dim / bSize                     // the number of batches
        var p         = new MatrixD (b.w.dim, b.w.dim2)   // momentum matrix
```

The code below shows the double loop (over `epoch` and `ib`). The parameter vector `b` is updated for each batch by calling `updateWeight`. The rest of the outer loop simply looks for early termination based on a stopping rule and records the best solution for `b` found so far. The square of the Frobenius norm is used to compute the sse over all outputs. The final part of the outer loop, increases the learning rate η at the end of each adjustment period (as the algorithm get closer to an optimal solution, gradients shrink and may slow down the algorithm).

```
        var sse_best_   = -0.0
        var (go, epoch) = (true, 1)
        cfor (go && epoch <= maxEpochs, epoch += 1) {      // iterate over each epoch
            val batches = permGen.igen.chop (nB)           // permute & split into batches

            for ib <- batches do b -= updateWeight (x(ib), y(ib))  // iteratively update b

            val sse = (y - f.fM (b * x)).normFSq           // recompute sum of squared errors
            collectLoss (sse)                              // collect loss per epoch
            val (b_best, sse_best) = stopWhen (Array (b), sse)
            if b_best != null then
                b.set (b_best (0))
                sse_best_ = sse_best                       // save best in sse_best_
                go = false
            else
                if epoch % ADJUST_PERIOD == 0 then η *= ADJUST_FACTOR   // adjust learn rate
            end if
        } // cfor
```

The core of algorithm is the `updateWeight` method that (1) computes the predicted outputs, (2) computes the error matrix (actually minus the error matrix for convenience), (3) computes the delta correction matrix as the Hadamard product of the slope matrix and the error matrix, (4) adjusts the learning rate by dividing by the batch size, (5) computes the change to parameters ignoring momentum from previous calls, (6) adds some of the momentum to this parameter change, and (7) returns the update matrix.

```
        @param x  the input matrix for the current batch
        @param y  the output matrix for the current batch

        inline def updateWeight (x: MatrixD, y: MatrixD): MatrixD =
            val α  = η / x.dim                             // eta over the current batch size
            val yp = f.fM (b * x)                          // prediction: Yp = f(XB)
            val ϵ  = yp - y                               // negative of error matrix
            val δ  = f.dM (yp) ⊙ ϵ                         // delta matrix for y
            val g  = x.𝒯 * δ                               // gradient matrix

            p = g * (1 - β) + p * β             // update momentum-based aggregated gradient
            (g * (1 - ν) + p * ν) * α           // parameter update amount (to be subtracted)
        end updateWeight
```

The final part of the optimize2 method returns the value of the loss function and the number of iterations when there is no early termination.

```
        if go then ((y - f.fM (b * x)).normFSq, maxEpochs)    // return sse and # epochs
        else       (sse_best_, epoch - upLimit)
    end optimize2
```

Note: the ScalaTion code also uses the ⊙ operator for Hadamard product (the alternative is $*\sim$) and a T-like Unicode symbol that looks similar to $\mathcal{T}$ for `transpose`.

The `optimize3` and `optimize` methods in the `Optimize_SGDM` class are for `NeuralNet_3L` and `NeuralNet_XL`, respectively.

A.5.2 Exercises

1. Plot the loss function versus the time-step/epoch for the AutoMPG for the `NeuralNet_2L`, `NeuralNet_3L` and `NeuralNet_XL`, comparing the SGD and SGDM.

   Hint: see the `MonitorLoss` trait in the `modeling` package.

2. Repeat the above exercise on a larger dataset.

A.6 SGD with ADAptive Moment Estimation

The ADAptive Moment estimation (Adam) Optimizer extends the optimizers that use first moments of momentum (means) of the gradients by including second moments (uncentered variances) [96]. The algorithm computes the gradient $\mathbf{g}$ and the momentum $\mathbf{p}$ just like in the previous section.

$$\mathbf{g} = \nabla f(\mathbf{x}) \qquad \text{gradient} \tag{A.58}$$

$$\mathbf{p} = (1-\beta_1)\,\mathbf{g} + \beta_1\,\mathbf{p} \qquad \text{momentum mean} \tag{A.59}$$

$$\mathbf{v} = (1-\beta_2)\,\mathbf{g}^2 + \beta_2\,\mathbf{v} \qquad \text{momentum uncentered variance} \tag{A.60}$$

$$\hat{\mathbf{p}} = \mathbf{p}/(1-\beta_1^t) \qquad \text{corrected mean} \tag{A.61}$$

$$\hat{\mathbf{v}} = \mathbf{v}/(1-\beta_2^t) \qquad \text{corrected uncentered variance} \tag{A.62}$$

$$\mathbf{x} \mathrel{-=} \alpha \left[\frac{\hat{\mathbf{p}}}{\sqrt{\hat{\mathbf{v}}} + eps}\right] \qquad \text{parameter update} \tag{A.63}$$

The new element is to include the uncentered variance (second raw moment) $\mathbf{v}$ into the calculation. It is included to normalize the gradient, dividing it by $\sqrt{\hat{\mathbf{v}}}$ (analog of dividing a random variable by its standard deviation). A very small value *eps* is added to it to avoid division by zero.

The calculated momentum mean $\mathbf{p}$ and momentum uncentered variance $\mathbf{v}$ are not used directly. This is because these two vectors are initialized to zero, so they will be under-estimates. These values are inflated by dividing them $(1-\beta_1^t)$ and $(1-\beta_2^t)$, respectively. For example, if $\beta_1 = 0.9$ and $t = 1$, $\mathbf{p}$ will be inflated by a factor of 10. Raising β_1 (same for β_2) to the t^{th} power reduces this effect as the time-steps increase.

The `GradientDescent_Adam` class provides an implementation of this algorithm.

```
@param f        the vector-to-scalar (V2S) objective/loss function
@param grad     the vector-to-vector (V2V) gradient function, grad f
@param hparam   the hyper-parameters

class GradientDescent_Adam (f: FunctionV2S, grad: FunctionV2V,
                             hparam: HyperParameter = hp)
      extends Minimize
         with StoppingRule (hparam("upLimit").toInt):   // limit on increasing loss
```

The `solve` method iteratively applies the above logic looking for a minimal solution. Again the code will terminate early due to lack of progress.

```
@param x0      the starting point
@param α       the step-size/learning rate

def solve (x0: VectorD, α: Double = eta): FuncVec =
    var p    = new VectorD (x0.dim)                // first moment of momentum
    var v    = new VectorD (x0.dim)                // second raw moment of momentum
    var ph   = VectorD.nullv                       // bias-corrected 1st moment
    var vh   = VectorD.nullv                       // bias-corrected 2nd raw moment
    var x    = x0                                  // start at initial guess
    var f_x  = -0.0                                // loss function, value undefined
    var best = (f_x, x)                            // start with best = initial

    var (go, it) = (true, 1)
    cfor (go && it <= MAX_IT, it += 1) {           // iterate over epochs/timesteps
```

```
            val g = grad (x)                                  // get gradient of loss function
            p  = p * β1 + g * (1 - β1)                        // update biased 1st moment
            v  = v * β2 + g~^2 * (1 - β2)                     // update biased 2nd raw moment
            ph = p / (1 - β1~^it)                             // compute bias-corrected 1st moment
            vh = v / (1 - β2~^it)                             // compute bias-corrected 2nd raw mo.
//          x  -= ph * α                                      // update parameters (1st moment)
            x  -= (ph / (vh~^0.5 + EPS)) * α                  // update parameters (both moments)
            f_x = f(x)                                        // compute new loss function value

            best = stopWhen (f_x, x)
            if best._2 != null then go = false                // early termination, return best
        } // cfor
        if go then getBest                                    // best solution found
        else best
    end solve
```

A.6.1 Exercises

1. Apply the Adam Optimizer to NeuralNet_2L, NeuralNet_3L, and NeuralNet_XL, i.e., finish the coding of Optimizer_Adam.

2. Test reducing the inflation of p and v by starting the time t at a larger value than 1 and discuss the effects, if any.

3. Report on the tuning of the hyper-parameters eta (η), beta (β_1) and beta2 (β_2).

```
object Minimize

    /** hyper-parameters for tuning the optimization algorithms - user tuning
     */
    val hp = new HyperParameter
    hp += ("eta", 0.5, 0.5)                          // initial learning/convergence rate
//  hp += ("eta", 0.01, 0.01)                        // initial learning/convergence rate
    hp += ("maxEpochs", 400, 400)                    // maximum number of epochs/iterations
    hp += ("upLimit", 4, 4)                          // up-limit for stopping rule
    hp += ("eps", 1E-8, 1E-8)                        // epilson, value close to zero
    hp += ("beta", 0.9, 0.9)                         // momentum decay hyper-parameter
    hp += ("beta2", 0.999, 0.999)                    // second momentum decay hyper-parameter
    hp += ("nu", 0.9, 0.9)                           // 0 => SGD, 1 => (normalized) SHB

end Minimize
```

4. Compare the loss curves, Quality of Fit (QoF), and run-times of Optimizer_SGD, Optimizer_SGDM, and Optimizer_Adam in ScalaTion, analogously in Keras and PyTorch.

A.7 Coordinate Descent

Rather than moving in the opposite direction of the gradient, the coordinate descent algorithm picks a coordinate direction and tries moving forward (1) or backward (-1) parallel to the coordinate axis. The next coordinate to try may be picked by a selection rule or cyclically as done by the code below.

Upon selecting a direction, a Line Search algorithm is applied to move down in that direction. The code can perform an exact (e.g., `GoldenSectionLS`) or inexact (e.g., `WolfeLS` line search. The Line Search algorithm looks in direction `dir` and returns the distance to move in that direction.

```
@param x      the current point
@param dir    the direction to move in
@param step   the initial step size

def lineSearch (x: VectorD, dir: VectorD, step: Double = STEP): Double =
    def f_1D (z: Double): Double = f(x + dir * z)          // create a 1D function
    val ls = if exactLS then new GoldenSectionLS (f_1D)   // Golden Section line search
             else new WolfeLS (f_1D, .0001, .1)            // Wolfe LS (c1 = .0001, c2 = .1)
    ls.search (step)                                       // perform a line search
end lineSearch
```

The `solve` method cyclically picks a coordinate axis and tries moving in both the forward and backward directions. The algorithm terminates when the distance moved on the last step drops below a tolerance or when a maximum `MAX_IT` number of iterations is exceeded.

```
@param x0     the starting point
@param step   the initial step size
@param toler  the tolerance

def solve (x0: VectorD, step: Double = STEP, toler: Double = EPSILON): FuncVec =
    val n     = x0.dim
    var x     = x0                                         // current point
    var fx    = f(x)                                       // obj. function at current point
    var y     = VectorD.nullv                              // next point
    var fy    = 0.0                                        // obj. function at next point
    val dir   = new VectorD (n)                            // set dir. by cycling coordinates
    var dist = 1.0                                         // distance current to next point
    var down = true                                        // moving down flag

    var it = 1
    cfor (it <= MAX_IT && down && dist > toler, it += 1) {

        for fb <- 1 to -1 by -2; j <- 0 until n do         // cycle coordinates get direction

            if j > 0 then dir(j-1) = 0.0
            dir(j) = fb                                    // set dir: forward/backward by fb
            y      = x + dir * lineSearch (x, dir, step)   // determine the next point
            fy     = f(y)                                  // objective value for next point

            debug ("solve", s"it = $it, y = $y, fy = $fy, dir = $dir")

            dist = (x - y).normSq                          // distance current to next point
            down = fy < fx                                 // still moving down?
            if down then { x = y; fx = fy }                // make next point current point
        end for
    } // cfor
    (fx, x)                                               // return functional value and point
end solve
```

A.8 Conjugate Gradient

The Conjugate Gradient (or Conjugarte Gradient Descent) algorithm like SGDM combines prior gradient (or search direction) information in with the current gradient (or search direction).

$$\mathbf{gr}^{(t)} = \nabla f(\mathbf{x}^{(t)}) \tag{A.64}$$

$$\mathbf{dir}^{(t)} = -\mathbf{gr}^{(t)} + \beta\,\mathbf{dir}^{(t-1)} \tag{A.65}$$

The new search direction $\mathbf{dir}^{(t)}$ is opposite the current gradient $\mathbf{gr}^{(t)}$ plus a correction term $\beta\,\mathbf{dir}^{(t-1)}$ proportional to the previous direction. This proportion β for the Fletcher-Reeves (FR) formula is the ratio of dot products of the search directions.

$$\beta = -\frac{\mathbf{dir}^{(t)} \cdot \mathbf{dir}^{(t)}}{\mathbf{dir}^{(t-1)} \cdot \mathbf{dir}^{(t-1)}} \tag{A.66}$$

For the Polak-Ribiere (PR) formula β is the ratio of dot products where the numerator takes the difference between subsequent search directions.

$$\beta = -\frac{\mathbf{dir}^{(t)} \cdot \left[\mathbf{dir}^{(t)} - \mathbf{dir}^{(t-1)}\right]}{\mathbf{dir}^{(t-1)} \cdot \mathbf{dir}^{(t-1)}} \tag{A.67}$$

ScalaTion uses the PR formula (adding `EPSILON` and taking a `max`).

```
@param sd1  the search direction at the previous point
@param sd2  the search direction at the current point

private inline def beta (sd1: VectorD, sd2: VectorD): Double =
    max (0.0, (sd2 dot (sd2 - sd1)) / (sd1.normSq + EPSILON))   // PR-CG (Polak-Ribiere)
```

The `ConjugateGradient` class supports finding minimal values for an objective/loss function f. It also supports having constraints and utilizes line-search.

```
@param f         the objective function/loss to be minimized
@param g         the constraint function to be satisfied, if any
@param ineq      whether the constraint function must satisfy inequality or equality
@param exactLS   whether to use exact (e.g., `GoldenLS`)
                          or inexact (e.g., `WolfeLS`) Line Search

class ConjugateGradient (f: FunctionV2S, g: FunctionV2S = null,
                         ineq: Boolean = true, exactLS: Boolean = true)
      extends Minimizer:
```

The `solve` method provides the iterative search engine and is set up for deterministic functions, but could be adapted to a stochastic setting.

```
@param x0     the starting point
@param step   the initial step-size
@param toler  the tolerance

def solve (x0: VectorD, step: Double = STEP, toler: Double = EPSILON): FuncVec =
    var x    = x0                                      // current point
    var f_x  = fg(x)                                   // objective function at current point
    var y    = VectorD.nullv                           // next point
    var f_y  = 0.0                                    // objective function at next point
    var dir  = - grad (fg, x)                         // initial direction is -gradient
    var dir0 = VectorD.nullv                          // keep the previous direction
    var dist = 1.0                                    // distance current to next point
    var down = true                                   // moving down flag

    for t <- 1 to MAX_IT if down && dist > toler && dir.normSq > toler do
        y    = x + dir * lineSearch (x, dir, step)    // determine the next point
        f_y  = fg(y)                                  // obj. function value for next point
        dir0 = dir                                    // save the current direction
        dir  = - grad (fg, y)                         // next search dir. via Gradient Desc.
        if t > 1 then dir += dir0 * beta (dir0, dir)  // modify search direction using PR-CG

        dist = (x - y).normSq                         // calc distance current to next point
        down = f_y < f_x                              // still moving down?
        if down then { x = y; f_x = f_y }             // make next point, the current point
    end for
    (f_x, x)                                          // return functional value & point
end solve
```

When formulas are available for the partial derivatives making up the gradient, they should be used since they are faster and more accurate than numerical methods.

```
@param partials  the vector of partial derivative functions

def setDerivatives (partials: FunctionV2V): Unit =
    if g != null then flaw ("setDerivatives", "only works for unconstrained problems")
    gr = partials                           // use given functions for partial derivatives
end setDerivatives
```

ScalaTion provides Gradient Descent and Conjugate Gradient Descent algorithms in two versions, one with Line Search (LS) and one without (NoLS) as shown in Table A.8.

Table A.8: Gradient Descent and Conjugate Gradient Algorithms

Algorithm	**Class** LS	**Class** NoLS	**Description**
Gradient Descent	`GradientDescent`	`GradientDescent_NoLS`	move opposite the gradient
Conjugate Gradient	`ConjugateGradient`	`ConjugateGradient_NoLS`	gradient with composite of previuos ones

A.8.1 Exercises

1. Adapt the above code for a stochastic setting.

2. Report on the literature concerning the use of or influence of the Conjugate Gradient algorithm in Machine Learning.

3. Consult the literature on the effectiveness of the Conjugate Gradient algorithm in the presence of constraints.

A.9 Quasi-Newton Methods

A.9.1 Newton-Raphson Method

In one dimension, the Newton (or Newton-Raphson) Method simply optimizes (minimizes) a function f by moving in the direction opposite the gradient (first derivative) divided by the Hessian (second derivative). The step size is moderated by the learning rate η (`eta`).

$$x_{i+1} \;=\; x_i \,-\, \eta\, \frac{\partial_x f}{\partial_x^2 f} \tag{A.68}$$

The `NewtonRaphson` class provides `solve` methods for finding roots (places where the function evaluates to zero) and a method for finding optimal values. The `optimize` method finds a local optima close to the starting point/guess `x0`. It applies the logic of the above equation and numerically approximates the first and second derivatives.

```
@param x0  the starting point/guess

def optimize (x0: Double): (Double, Double) =
    var x    = x0                                     // current point
    var f_x  = f(x)                                   // function value at x
    var df_x = 1.0                                    // first derivative value at x

    var it = 1
    cfor (it < MAX_IT && abs (df_x) > EPS, it += 1) {
        df_x  = maxmag (D (f)(x), EPS)                // make sure 1st der. isn't too small
        d2f_x = maxmag (DD (f)(x), EPS)               // make sure 2nd der. isn't too small
        x   -= df_x / d2f_x * eta                     // subtract the ratio
        f_x = f(x)
    } // cfor

    printf ("optimal solution x = %10.5f, f = %10.5f\n", x, f(x))
    (f_x, x)
end optimize
```

A.9.2 Newton Method

The same idea can be applied when f is takes vectors rather than scalars, i.e., $f : \mathbb{R}^n \to \mathbb{R}$, although the details are much more involved.

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i \,-\, \eta\, \partial_{\mathbf{x}} f\, [\partial_{\mathbf{x}}^2 f]^{-1} \tag{A.69}$$

Now $\partial_{\mathbf{x}} f$ is a gradient vector and $\partial_{\mathbf{x}}^2 f$ is a Hessian matrix. As division is not supported, matrix inversion is used. Using alternate notion, it can be written as follows:

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i \,-\, \eta\, \nabla_{\mathbf{x}} f\, [\mathbb{H}_{\mathbf{x}} f]^{-1} \tag{A.70}$$

This represents a vector-matrix multiplication, although it can be easily switched to a matrix-vector multiplication since the Hessian matrix is symmetric (and switching from row to column vectors for the gradient).

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i \,-\, \eta\, [\mathbb{H}_{\mathbf{x}} f]^{-1}\, \nabla_{\mathbf{x}} f \tag{A.71}$$

One may view the multiplication by the Hessian as modifying (or deflecting) the gradient due to the function's curvature. Define this to be the direction vector **d**.

$$\mathbf{d} \;=\; [\mathbb{H}_{\mathbf{x}} f]^{-1} \, \nabla_{\mathbf{x}} f \tag{A.72}$$

Rather than taking the inverse, it will be faster and more numerically stable to used matrix factorization (as was done for regression).

$$[\mathbb{H}_{\mathbf{x}} f] \, \mathbf{d} \;=\; \nabla_{\mathbf{x}} f \tag{A.73}$$

The `solve` method in the `Newton` class finds local optima close to the starting point/guess `x0`. This version numerically approximates the first and second derivatives. It uses factorization (`Fac_LU`) to solve for the direction vector **d**.

```
@param x0  the starting point/guess
@param α   the current learning rate

def solve (x0: VectorD), α: Double = eta: FuncVec =
    var x     = x0                                     // current point
    var f_x   = f(x)                                   // function value at x
    var df_x  = VectorD.one (x.dim)                    // initial dummy value for gradient

    var it = 1                                         // iteration counter
    cfor (it < MAX_IT && df_x.norm > EPS, it += 1) {
        df_x      = ∇ (f, x)                            // compute gradient, numerically
        val d2f_x = H (f, x)                           // compute Hessian, numerically

        val d = if gradDesc then df_x                  // direction = gradient
//              else Fac_LU.inverse (d2f_x)() * df_x   // deflected via inverse Hessian
                else Fac_LU.solve_ (d2f_x, df_x)       // deflected via factorized Hessian

        x  -= d * α                                    // subtract direction * α
        f_x = f(x)                                     // functional value
    } // cfor

    println (s"optimal solution x = §x, f = §{f(x)}")
    (f_x, x)
end solve
```

Cholesky and QR factorizations may be used as well.

A.9.3 BFGS Method

The idea of Qausi-Newton methods is to replace the cubic time complexity of the Newton update for computing the inverse Hessian with a quadratic complexity approximation. The quasi methods may require more iterations, but each iteration is faster. In addition, quasi Newton methods may also work better when the Hessian is a nearly singular matrix.

Address: f must be twice differential and H must be positive definite

Letting vector **s** be the latest change in position (i.e., step) and **y** be the latest change in the gradient, the `aHi_inc` method computes the change to the approximate Hessian inverse H^{-1} (`aHi`) matrix using the

Sherman–Morrison formula (see `https://mdav.ece.gatech.edu/ece-6270-spring2021/notes/09-bfgs.pdf`).

$$\frac{(\mathbf{s} \otimes \mathbf{s})\,(s_y + \mathbf{y} \cdot \mathbf{a_y})}{s_y^2} - \frac{\mathbf{a_y} \otimes \mathbf{s} + \mathbf{s} \otimes \mathbf{a_y}}{s_y} \tag{A.74}$$

where scalar $s_y = \mathbf{s} \cdot \mathbf{y}$ and vector $\mathbf{a_y} = H^{-1}\mathbf{y}$. Recall that $\otimes$ is the symbol for outer product (takes two vectors and produces a rank 2 matrix). The corresponding ScalaTion code is shown below.

```
@param aHi  the current value of the approximate Hessian inverse (aHi)
@param s    the step vector (next point - current point)
@param y    the difference in the gradients (next - current)

def aHi_inc (aHi: MatrixD, s: VectorD, y: VectorD): MatrixD =
    var sy = maxmag (s dot y, eps)
    val ay = aHi * y
    (⊗ (s, s) * (sy + (y dot ay))) / sy~^2 - (⊗ (ay, s) + ⊗ (s, ay)) / sy
end aHi_inc
```

Using this update method, the inverse Hessian is never computed, only efficiently approximated. This particular approximation yield the Broyden [23], Fletcher [48], Goldfarb [56], Shanno [170] (BFGS) algorithm. See [141] for a historical review of second-order optimization algorithms.

```
@param x0   the starting point/guess
@param α    the current learning rate

def solve (x0: VectorD, α: Double = eta): FuncVec =
    var x      = x0                                        // current point
    var f_x    = f(x)                                      // function value at x
    var df_x   = ∇ (f, x)                                   // compute gradient, numerically
    var aHi    = MatrixD.eye (x.dim, x.dim)                // approximate Hessian inverse
                                                           //   start with identity matrix
    var it = 1                                             // iteration counter
    cfor (it < MAX_IT && df_x.norm > EPS, it += 1) {
        val d = if gradDesc then df_x                      // direction = gradient
                else aHi * df_x                            // use approximate Hessian inverse

        val s      = d * -α                                // compute step vector
        x         += s                                     // update new x
        val df_x_  = df_x                                  // save previous gradient
        df_x       = ∇ (f, x)                               // compute new gradient, numerically
        aHi += aHi_inc (aHi, s, df_x - df_x_)              // update approximate Hessian inverse
        f_x   = f(x)                                       // functional value
    } // cfor

    println (s"optimal solution x = $x, f = ${f(x)}")
    (f_x, x)
end solve
```

A.9.4 Limited Memory-BFGS Method

The Limited Memory BFGS (L-BFGS) Method [112] does not maintain an approximate Hessian inverse, but rather maintains the last m $\mathbf{s}$ and $\mathbf{y}$ vectors. These are applied using two loops as shown below. The `findDir` method computes the deflected gradient by passing in the current gradient and using the last m

steps or changes in **x**-position vectors **s** and changes in gradient vectors **y**. There are stored using the `Ring` class that efficiently maintains the last `cap` $= m$ additions.

```
@param g  the current gradient
@param k  the k-th iteration

def findDir (g: VectorD, k: Int): VectorD =
    var q = g                                              // start with current gradient
    for i <- k-1 to k-m by -1 do
        a(i) = (s(i) dot q) * p(i)
        q -= y(i) * a(i)
    val ga = (s(k-1) dot y(k-1)) / y(k-1).normSq           // gamma
    var z = q * ga
    for i <- k-m until k do
        val b = (y(i) dot z) * p(i)
        z += s(i) * (a(i) - b)
    z                                                      // return direction = deflected grad.
end findDir
```

The `solve` method for L-BFGS only needs slight modification from the BFGS algorithm.

```
@param x0  the starting point/guess
@param α   the current learning rate

def solve (x0: VectorD, α: Double = eta): FuncVec =
    var x     = x0                                         // current point
    var f_x   = f(x)                                       // function value at x
    var df_x  = ∇ (f, x)                                   // compute gradient, numerically

    var it = 0                                             // iteration counter
    cfor (it < MAX_IT && df_x.norm > EPS, it += 1) {
        debug ("solve", s"it = $it: f($x) = $f_x, df_x = $df_x")

        val d = if it == 0 then df_x                       // direction = gradient
                else findDir (df_x, it)                    // find deflected gradient

        val s_    = d * -α                                 // compute step (- => opposite grad.)
        x        += s_                                     // update new x
        val df_x_ = df_x                                   // save previous gradient
        df_x      = ∇ (f, x)                               // compute new gradient, numerically
        val y_    = df_x - df_x_                           // difference in gradients
        f_x       = f(x)                                   // functional value
        s.add (s_); s.add (y_)                             // add s_ and y_ to their rings (last
 m)
    } // cfor

    println (s"optimal solution x = $x\, f = $s{f(x$)}")
    (f_x, x)
end solve
```

A.9.5 Summary

The three algorithms trade off efficiency of the approximate Hessian inverse update for more iterations/ability to find optima. The following study [178] shows these trade off for several example problems. The time complexity of update is Newton $O(n^3)$, BFGS $O(n^2)$, and L-BFGS $O(mn)$.

ScalaTion provides each of the three algorithms in two versions, one with Line Search (`LS`) and one without (`NoLS`). The classes implementing these algorithms are shown in Table A.9.

Table A.9: Newton and Quasi-Newton Optimization Algorithms

Algorithm	**Class** LS	**Class** NoLS	**Description**
Newton Method	`Newton`	`Newton_NoLS`	use inverse Hessian to deflect gradient
BFGS Method	`BFGS`	`BFGS_NoLS`	use approximate inverse Hessian
L-BFGS Method	`L_BFGS`	`L_BFGS_NoLS`	use last m steps and gradient changes

These classes implement both `solve` and `solve2` nethods.

```
def solve (x0: VectorD, α: Double = eta): FuncVec =
def solve2 (x0: VectorD, grad: FunctionV2V, α: Double = eta): FuncVec =
```

The first numerically computes the gradient, while the second one requires the user/application to pass in the gradient, typically as a function mapping vectors to vectors (`FunctionV2V`). As the Newton Method requires computing the Hessian as well it needs a function for each partial derivative so it it can perform an more efficient Jacobian calculation for the Hessian. In this case the type needs to be `Array [FunctionV2S]`.

A.9.6 Exercises

1. Compare the success rate, number of iterations and execution times of each of the three algorithms for their Line Search (LS) versions. Use the some of 30 benchmark problems given in Appendix A of [103].

   `https://arxiv.org/pdf/2204.05297.pdf`

2. Compare the success rate, number of iterations and execution times of each of the three algorithms for their No Line Search (NoLS) versions. Use the same benchmark.

3. Compare the best of the Newton/Quasi-Newton methods with Gradient Descent and Conjugate Gradient methods.

A.10 Method of Lagrange Multipliers

The Method of Lagrange Multipliers (or Lagrangian Method) provides a means for solving constrained optimizations problems. For optimization problems involving only one equality constraint, one may introduce a Lagrange multiplier λ. At optimality, the gradient of f should be orthogonal to the surface defined by the constraint $\mathcal{L}(\mathbf{x}) = 0$, otherwise, moving along the surface in the opposite direction to the gradient ($-\nabla f(\mathbf{x})$ for minimization) would improve the solution. Since the gradient of h, $\nabla h(\mathbf{x})$, is orthogonal to the surface as well, this implies that the two gradients should only differ by a constant multiplier λ.

$$-\nabla f(\mathbf{x}) \;=\; \lambda \nabla h(\mathbf{x}) \tag{A.75}$$

In general, such problems can solved by defining the *Lagrangian*

$$L(\mathbf{x}, \boldsymbol{\lambda}) \;=\; f(\mathbf{x}) - \boldsymbol{\lambda} \cdot \mathbf{h}(\mathbf{x}) \tag{A.76}$$

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers. When there is a single equality constraint, this becomes

$$L(\mathbf{x}, \lambda) \;=\; f(\mathbf{x}) - \lambda\, h(\mathbf{x})$$

Taking the gradient of the Lagrangian w.r.t. $\mathbf{x}$ and λ yields a vector of dimension $n + 1$.

$$\nabla L(\mathbf{x}, \lambda) \;=\; [\, \nabla f(\mathbf{x}) - \lambda \nabla h(\mathbf{x}), h(\mathbf{x})\,]$$

Now we may try setting the gradient to zero and solving a system of equations.

A.10.1 Example Problem

The Lagrangian for the problem given at the beginning of the chapter is

$$L(\mathbf{x}, \lambda) \;=\; (x_1 - 4)^2 + (x_2 - 2)^2 - \lambda\,(x_1 - x_2)$$

Computation of the gradient $[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial \lambda}]$ of the Lagrangian yields the following three equations,

$$\begin{aligned} -2(x_1 - 4) &= \lambda \\ -2(x_2 - 2) &= -\lambda \\ x_1 - x_2 &= 0 \end{aligned}$$

The first two equations are from the gradient w.r.t. $\mathbf{x}$, while the third equation is simply the constraint itself $h(\mathbf{x}) = 0$. The equations may be rewritten in the following form.

$$\begin{aligned} 2x_1 + \lambda &= 8 \\ 2x_2 - \lambda &= 4 \\ x_1 - x_2 &= 0 \end{aligned}$$

This is a linear system of equations with 3 variables $[x_1, x_2, \lambda]$ and 3 equations that may be solved, for example, by LU Factorization. In this case, the last equation gives $x_1 = x_2$, so adding equations 1 and 2 yields $4x_1 = 12$. Therefore, the optimal value is $\mathbf{x} = [3, 3]$ with $\lambda = 2$ where $f(\mathbf{x}) = 2$.

Adding an equality constraint is addressed by adding another Lagrange multiplier, e.g., 4 variables $[x_1, x_2, \lambda_1, \lambda_2]$ and 4 equations, two from the gradient w.r.t. $\mathbf{x}$ and one for each of the two constraints.

Linear systems of equations are generated when the objective function is at most quadratic and the constraints are linear. If this is not the case, a nonlinear system of equations may be generated.

A.11 Karush-Kuhn-Tucker Conditions

Introducing inequality constraints makes the situation is a little more complicated. A generalization of the Method of Lagrange Multipliers based on the Karush-Kuhn-Tucker (KKT) conditions is needed. For minimization, the KKT conditions are as follows:

$$-\nabla f(\mathbf{x}) \;=\; \boldsymbol{\alpha} \cdot \nabla \mathbf{g}(\mathbf{x}) + \boldsymbol{\lambda} \cdot \nabla \mathbf{h}(\mathbf{x}) \tag{A.77}$$

The original constraints must also hold.

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0} \quad \text{and} \quad \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

Furthermore, the Lagrange multipliers for the inequality constraints $\boldsymbol{\alpha}$ are themselves constrained to be non-negative.

$$\boldsymbol{\alpha} \geq \mathbf{0}$$

When the objective function is at most quadratic and the constraints are linear, the problem of finding an optimal value for $\mathbf{x}$ is referred to a Quadratic Programming. Many estimation/learning problems in data science are of this form. Beyond Quadratic Programming lies problems in Nonlinear Programming. Linear Programming (linear objective function and linear constraints) typically finds less use (e.g., Quantile Regression) in estimation/learning, so it will not be covered in this Chapter, although it is provided by ScalaTion.

A.11.1 Active and Inactive Constraints

A.12 Quadratic Programming

The `QuadraticSimplex` class solves Quadratic Programming (QP) problems using the Quadratic Simplex Algorithm. Given a constraint matrix A, constant vector $\mathbf{b}$, cost matrix Q and cost vector $\mathbf{c}$, find values for the solution/decision vector $\mathbf{x}$ that minimize the objective function $f(\mathbf{x})$, while satisfying all of the constraints, i.e.,

$$\begin{aligned} \text{minimize} \;\; f(\mathbf{x}) &= \frac{1}{2}\mathbf{x} \cdot Q\mathbf{x} + \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} \;\; \mathbf{g}(\mathbf{x}) &= A\mathbf{x} - \mathbf{b} \le \mathbf{0} \end{aligned}$$

Before considering the type of optimization algorithm to use, we may simplify the problem by applying the KKT conditions.

$$-\nabla f(\mathbf{x}) \;=\; Q\mathbf{x} + \mathbf{c} \;=\; \boldsymbol{\alpha} \cdot \nabla g(\mathbf{x}) \;=\; \boldsymbol{\alpha} \cdot A$$

Adding the constraints gives the following n equations and $2m$ constraints:

$$\begin{aligned} Q\mathbf{x} + \mathbf{c} &= \boldsymbol{\alpha} \cdot A \\ A\mathbf{x} - \mathbf{b} &\le \mathbf{0} \\ \boldsymbol{\alpha} &\ge \mathbf{0} \end{aligned}$$

These equations have two unknown vectors, $\mathbf{x}$ of dimension n and $\boldsymbol{\alpha}$ of dimension m.

The algorithm creates an simplex tableau. This implementation is restricted to linear constraints $A\mathbf{x} \le \mathbf{b}$ and Q being a positive semi-definite matrix. Pivoting must now also handle nonlinear complementary slackness.

Class Methods:

```
    @param a    the M-by-N constraint matrix
    @param b    the M-length constant/limit vector
    @param q    the N-by-N cost/revenue matrix (second order component)
    @param c    the N-length cost/revenue vector (first order component)
    @param x_B  the initial basis (set of indices where x_i is in the basis)

    class QuadraticSimplex (a: MatrixD, b: VectorD, q: MatrixD, c: VectorD,
                            var x_B: Array [Int] = null)

    def set (mat: MatrixD, i: Int, u: VectorD, j: Int = 0): Unit =
    def setBasis (j: Int = N, l: Int = M): Array [Int] =
    def entering (): Int =
    def comple (l: Int): Int =
    def leaving (l: Int): Int =
    def pivot (k: Int, l: Int): Unit =
    def solve (): (VectorD, Double) =
    def tableau: MatrixD = t
    def primal: VectorD =
    def dual: VectorD = null
    def objValue (x: VectorD): Double = (x dot (q * x)) * .5 + (c dot x)
    def showTableau (): Unit =
```

A.13 Augmented Lagrangian Method

The Augmented Lagrangian Method (also known as the Method of Multipliers) takes a constrained optimization problem with equality constraints and solves it as a series of unconstrained optimization problems.

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) \\ &\text{subject to } \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned}$$

where $f(\mathbf{x})$ is the objective function and $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are the equality constraints.

In penalty form, the constrained optimization problem becomes.

$$\text{minimize } f(\mathbf{x}) + \frac{\rho_k}{2} \|\mathbf{h}(\mathbf{x})\|_2^2$$

where k is the iteration counter. The square of the Euclidean norm indicates to what degree the equality constraints are violated. Replacing the square of the Euclidean norm with the dot product gives.

$$\text{minimize } f(\mathbf{x}) + \frac{\rho_k}{2} \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{x})$$

The value of the penalty parameter ρ_k increases (e.g., linearly) with k and thereby enforces the equality constraints more strongly with each iteration.

An alternative to minimizing $f(\mathbf{x})$ with a quadratic penalty is to minimize using the Augmented Lagrangian $L_{\rho_k}(\mathbf{x}, \boldsymbol{\lambda})$.

$$L_{\rho_k}(\mathbf{x}, \boldsymbol{\lambda}) \;=\; f(\mathbf{x}) + \frac{\rho_k}{2} \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{x}) - \boldsymbol{\lambda} \cdot \mathbf{h}(\mathbf{x}) \tag{A.78}$$

where $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers. After each iteration, the Lagrange multipliers are updated.

$$\boldsymbol{\lambda} = \boldsymbol{\lambda} - \rho_k \, \mathbf{h}(\mathbf{x})$$

This method allows for quicker convergence without the need for the penalty ρ_k to become as large (see the exercises for a comparison of the Augmented Lagrangian Method with the Penalty Method). This method may be combined with an algorithm for solving unconstrained optimization problems (see the exercises for how it can be combined with the Gradient Descent algorithm). The method also can be extended to work inequality constraints.

A.13.1 Example Problem

Consider the problem given at the beginning of the chapter with the inequality constraint left out.

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2 \\ &\text{subject to } h(\mathbf{x}) = x_1 - x_2 = 0 \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^2$, f is the objective function and h is the single equality constraint. The Augmented Lagrangian for this problem is

$$L_{\rho_k}(\mathbf{x}, \lambda) \;=\; (x_1 - 4)^2 + (x_2 - 2)^2 + \frac{\rho_k}{2}\,(x_1 - x_2)^2 - \lambda(x_1 - x_2) \tag{A.79}$$

The gradient of the Augmented Lagrangian $\nabla L_{\rho_k}(\mathbf{x}, \lambda)$ is made up of the following two partial derivatives.

$$\begin{aligned}\partial/\partial x_1 &= 2(x_1 - 4) + \frac{\rho_k}{2}\,2(x_1 - x_2) - \lambda \\ \partial/\partial x_2 &= 2(x_2 - 2) - \frac{\rho_k}{2}\,2(x_1 - x_2) + \lambda\end{aligned}$$

The Lagrange multiplier updates becomes

$$\lambda = \lambda - \rho_k\,(x_1 - x_2)$$

The code in the exercises tightly integrates the Gradient Descent algorithm with the Augmented Lagrangian method by updating the penalty and Lagrange multiplier during each iteration.

A.13.2 Exercises

1. Write a ScalaTion program to solve the example problem given above.

```
    // function to optimize
    def f(x: VectorD): Double = (x(0) - 4)~^2 + (x(1) - 2)~^2

    // equality constraint to maintain
    def h(x: VectorD): Double = x(0) - x(1)

    // augmented Lagrangian
    def lg (x: VectorD): Double = f(x) + (p/2) * h(x)~^2 - l * h(x)

    // gradient of Augmented Lagrangian
    def grad (x: VectorD): VectorD =  VectorD (?, ?)

    val x   = new VectorD (2)                   // vector to optimize
    val eta = 0.1                               // learning rate
    val p0  = 0.25; var p = p0                  // initial penalty (p = p0)
    var l   = 0.0                               // initial value for Lagrange multiplier

    for k <- 1 to MAX_IT do
        l -= p * h(x)                           // comment out for Penalty Method
        x -= grad (x) * eta
        println (s"$k: x = $x, f(x) = ${f(x)}, lg(x) = ${lg(x)}, p = $p, l = $l")
        p += p0
    end for
```

2. Add code to collect the trajectory of vector x in a matrix z and plot the two columns in the z matrix.

```
    val z = new MatrixD (MAX_IT, 2)        // store x's trajectory
    z(k-1) = x.copy
    new Plot (z(?, 0), z(?, 1)
```

3. Compare the Augmented Lagrangian Method with the Penalty Method by simply removing the Lagrange multiplier from the code.

A.14 Alternating Direction Method of Multipliers

For problems that have non-differentiable points, it may work better to approach such points from two directions. This happens with ℓ_1 regularization, e.g., in Lasso regression. Given a differentiable function $f(\mathbf{x})$ along with a regularization term $\alpha\|\mathbf{x}\|_1$, it is non-differentiable in each dimension as x_j approaches zero, with the slopes for the second term jumping from negative α to positive α.

$$\text{minimize} \;\; f(\mathbf{x}) + \alpha\|\mathbf{x}\|_1 \tag{A.80}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$.

The Alternating Direction Method of Multipliers (ADMM) approach [22] is to separate the objective function into two parts

$$\text{minimize} \;\; f(\mathbf{x}) + g(\mathbf{z}) \tag{A.81}$$

by introducing a new vector $\mathbf{z} \in \mathbb{R}^n$ and constraining it to $\mathbf{x}$ via the equality constraint

$$\text{subject to} \;\; \mathbf{x} - \mathbf{z} \; = \; \mathbf{0} \tag{A.82}$$

and letting

$$g(\mathbf{z}) \; = \; \alpha\|\mathbf{z}\|_1 \tag{A.83}$$

The augmented Lagrangian is then of the form

$$L_{\rho_k}(\mathbf{x}, \mathbf{z}, \boldsymbol{\lambda}) \; = \; f(\mathbf{x}) + g(\mathbf{z}) + \frac{\rho_k}{2}\|\mathbf{x} - \mathbf{z}\|_2^2 - \boldsymbol{\lambda} \cdot (\mathbf{x} - \mathbf{z}) \tag{A.84}$$

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers and ρ_k is the penalty parameter for the k^{th} iteration (see the last section).

The problem can be solved by iterating over k and solving three sub-problems for each iteration (the prime indicates the new value for iteration k). The $\mathbf{x}$ and $\mathbf{z}$ variable vectors are updated in an alternating fashion.

1. Minimize the augmented Lagrangian w.r.t. the $\mathbf{x}$ vector with $\mathbf{z}$ and $\boldsymbol{\lambda}$ fixed with initial values at the start of iteration k.

$$\mathbf{x}' \; = \; \text{argmin}_{\mathbf{x}} \; L_{\rho_k}(\mathbf{x}, \mathbf{z}, \boldsymbol{\lambda}) \tag{A.85}$$

2. Minimize the augmented Lagrangian w.r.t. the $\mathbf{z}$ vector with $\mathbf{x}$ fixed at its new value and $\boldsymbol{\lambda}$ fixed at its start value.

$$\mathbf{z}' \; = \; \text{argmin}_{\mathbf{z}} \; L_{\rho_k}(\mathbf{x}', \mathbf{z}, \boldsymbol{\lambda}) \tag{A.86}$$

3. Update the Lagrange multipliers based on the new values for $\mathbf{x}$ and $\mathbf{z}$.

$$\boldsymbol{\lambda}' \; = \; \boldsymbol{\lambda} - \rho_k(\mathbf{x}' - \mathbf{z}') \tag{A.87}$$

A.14.1 Example Problem

As a reformulation of the half sse loss function for regression, $f(\mathbf{x})$ may be written,

$$f(\mathbf{x}) \;=\; \frac{1}{2}\,\|\mathbf{y} - D\mathbf{x}\|_2^2 \tag{A.88}$$

where $D \in \mathbb{R}^{m\times n}$ is the given data matrix, $\mathbf{x} \in \mathbb{R}^n$ is the parameter vector to be fit, and $\mathbf{y} \in \mathbb{R}^m$ is the given response vector. Letting the shrinkage parameter α be constant, the regularization term becomes,

$$g(\mathbf{z}) \;=\; \alpha\|z\|_1 \tag{A.89}$$

Therefore, the augmented Lagrangian is now,

$$L_{\rho_k}(\mathbf{x},\mathbf{z},\boldsymbol{\lambda}) \;=\; \frac{1}{2}\,\|\mathbf{y} - D\mathbf{x})\|_2^2 \,+\, \alpha\,\|\mathbf{z}\|_1 \,+\, \frac{\rho_k}{2}\|\mathbf{x}-\mathbf{z}\|_2^2 \,-\, \boldsymbol{\lambda}\cdot(\mathbf{x}-\mathbf{z}) \tag{A.90}$$

1. The first sub-problem can be solved by setting the following gradient to zero.

$$\frac{\partial L}{\partial \mathbf{x}} \;=\; -\,D^{\intercal}(\mathbf{y} - D\mathbf{x}) \,+\, \rho_k(\mathbf{x}-\mathbf{z}) \,-\, \boldsymbol{\lambda} \;=\; \mathbf{0} \tag{A.91}$$

   Collecting terms gives an equation that can be solved by matrix factorization (like it was done for ridge regression) to determine $\mathbf{x}'$.

$$(D^{\intercal}D \,+\, \rho_k I)\mathbf{x} \;=\; D^{\intercal}\mathbf{y} \,-\, \rho_k\mathbf{z} \,+\, \boldsymbol{\lambda} \tag{A.92}$$

2. The first term in the second sub-problem is constant w.r.t. $\mathbf{z}$ and may be ignored. Using a soft-thresholding function for the gradient of $\alpha\,\|\mathbf{z}\|_1$ (see the first exercise) and taking the overall gradient w.r.t. $\mathbf{z}$ and dividing by ρ_k gives the following:

$$\frac{\partial L}{\partial \mathbf{z}} \;=\; S_{\alpha/\rho_k}(\mathbf{z}) \,-\, (\mathbf{x}'-\mathbf{z}) \,+\, \frac{\boldsymbol{\lambda}}{\rho_k} \;=\; \mathbf{0} \tag{A.93}$$

3. The update to the Lagrange multipliers is the same as the general equation.

$$\boldsymbol{\lambda}' \;=\; \boldsymbol{\lambda} - \rho_k(\mathbf{x}'-\mathbf{z}') \tag{A.94}$$

A.14.2 `LassoAddm` Object

```
    object LassoAdmm:

    def reset: Unit =
    def solve (a: MatrixD, b: VectorD, λ: Double = 0.01): VectorD =
    def solveCached (ata_ρI_inv: MatrixD, atb: VectorD, λ: Double): VectorD =
```

A.14.3 Exercises

1. The sub-deferential of the scalar absolute value function f,

$$f(x) \;=\; \alpha|x|$$

is $\alpha\,\mathrm{sign}(x)$ where

$$\mathrm{sign}(x) = \begin{cases} 1 & x > 0 \\ [-1, 1] & x = 0 \\ -1 & x < 0 \end{cases}$$

Unfortunately, $\mathrm{sign}(x)$ is a relation, but not a function. To avoid this problem, one can approximate the derivative of $\alpha|x|$ with the following soft-thresholding function.

$$S_\theta(x) = \begin{cases} x - \theta & x > \theta \\ 0 & |x| \leq \theta \\ x + \theta & x < -\theta \end{cases}$$

```
    @param x   the scalar to threshold
    @param th  the threshold (theta)

    def softThresh (x: Double, th: Double): Double =
        if x > th then x - th
        else if x < -th then x + th
        else 0
    end softThresh
```

Plot this function for various values of θ.

2. Rework the above example problem using the scaled version of ADMM [22].

$$\mathbf{x}' = \mathrm{argmin}_{\mathbf{x}}\ (f(\mathbf{x}) + \frac{\rho_k}{2}\|\mathbf{x} - \mathbf{z} + \mathbf{u}\|_2^2) \quad (A.95)$$

$$\mathbf{z}' = \mathrm{argmin}_{\mathbf{z}}\ (g(\mathbf{z}) + \frac{\rho_k}{2}\|\mathbf{x}' - \mathbf{z} + \mathbf{u}\|_2^2) \quad (A.96)$$

$$\mathbf{u}' = \mathbf{u} + (\mathbf{x}' - \mathbf{z}') \quad (A.97)$$

where $\mathbf{u} = -\dfrac{\boldsymbol{\lambda}}{\rho_k}$.

3. Show how the equations from the above problem relate to those given in the Lasso section of the Prediction chapter.

4. The equality constraint can be generalized to give a standard formulation [22]:

$$\text{minimize } f(\mathbf{x}) + g(\mathbf{z}) \quad (A.98)$$

$$\text{subject to } A\mathbf{x} + B\mathbf{z} = \mathbf{c} \quad (A.99)$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$ are variable vectors, and $A \in \mathbb{R}^{p\times n}$ and $B \in \mathbb{R}^{p\times m}$ are coefficient matrices. Rewrite the augmented Lagrangian for this more general case.

A.15 Nelder-Mead Simplex

Efficient nonlinear optimization without the use of gradients is challenging. Of the derivative-free methods, the Nelder-Mead Simplex [131, 177] method is surprisingly robust and effective. A simplex is a triangular shape defined by $n + 1$ vertices in n-dimensional space, e.g., a triangle in a two-dimensional plane. The algorithm begins with an initial simplex that is systematically moved downhill by methods that expand, reflect, contract-out, contract-in and shrink the simplex.

More specifically, the algorithm improves the simplex by replacing the worst vertex ($\mathbf{x}_h$) with a better one found on the line containing $\mathbf{x}_h$ and the centroid ($\mathbf{x}_c$). It tries reflection, expansion, outer contraction and inner contraction points, in that order. If none succeeds, it shrinks the simplex. The algorithm iterate until the distance between the best and worst points in the simplex drops below a given tolerance.

Consider the following objective function.

$$f(\mathbf{x}) \;=\; (x_0 - 2)^2 + (x_1 - 3)^2 + 1 \tag{A.100}$$

The optimal (minimal) solution is the green point at $(2, 3)$ with a functional value of 1 with contours drawn around it The initial simplex is shown as the blue triangle in Figure A.6: the worst point is in red with a functional value of 6, the second worst is in purple with a functional value of 5, and the best point is in blue with a functional value of 3. In this case, the `reflect` method will succeed with new better point $(2, 2)$ with a functional value of 2 and shown in cyan. The new cyan point will replace the red point, forming a new simplex closer to the optimal solution.

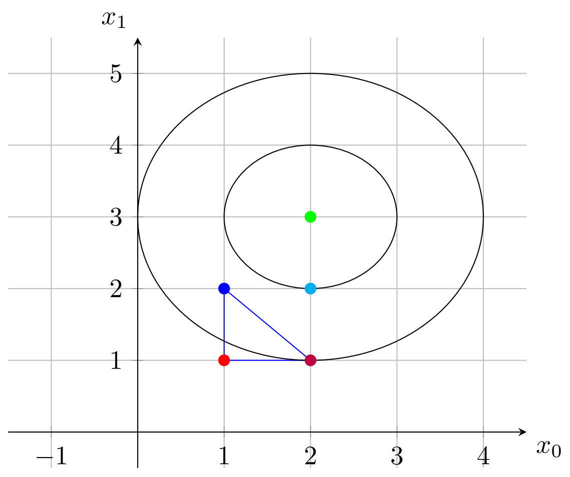


Figure A.6: Contour Curves for Nelder-Mead Simplex (red = 6, purple = 5, blue = 3)

Point/vertex replacement is based on finding points along a line that includes the worst point $\mathbf{x_h}$ and the centroid $\mathbf{x_c}$ formed from the other points (excluding the worst). In this example, the centroid $\mathbf{x_c} = (1.5, 1.5)$.

- Reflection. The reflection point $\mathbf{x_r}$ is the centroid plus α times the distance from the worst point to the centroid and is given as the cyan point in Figure A.6. In 2D and $\alpha = 1$, it flips the triangle to the other side.

$$\mathbf{x_r} \;=\; \mathbf{x_c} + (\mathbf{x_c} - \mathbf{x_h}) * \alpha \tag{A.101}$$

- Expansion. The expansion point $\mathbf{x_e}$ push further away from the centroid than the reflection point.

$$\mathbf{x_e} \;=\; \mathbf{x_c} + (\mathbf{x_r} - \mathbf{x_c}) * \gamma \tag{A.102}$$

- Outer Contraction. The outer contraction point $\mathbf{x_o}$ is between the centroid and the reflection point.

$$\mathbf{x_o} \;=\; \mathbf{x_c} + (\mathbf{x_r} - \mathbf{x_c}) * \beta \tag{A.103}$$

- Inner Contraction. The inner contraction point $\mathbf{x_i}$ is between the worst point and the centroid.

$$\mathbf{x_i} \;=\; \mathbf{x_c} + (\mathbf{x_h} - \mathbf{x_c}) * \beta \tag{A.104}$$

For the above example with $\alpha = 1, \gamma = 2$, and $\beta = 0.5$, these points are $\mathbf{x_r} = (2, 2), \mathbf{x_e} = (2.5, 2.5), \mathbf{x_o} = (1.75, 1.75)$, and $\mathbf{x_i} = (1.25, 1.25)$.

The `transformation` method applies some simple rules for selecting one of these points to replace the worst point (see the code for details). If none of these points are selected to replace the worst point $\mathbf{x_h}$, then all the points are pulled toward the best point $\mathbf{x_l}$ by a factor of δ. by the `shrink` method.

A.15.1 `NelderMeadSimplex` Class

```
@param f  the vector-to-scalar objective function
@param n  the dimension of the search space

class NelderMeadSimplex (f: FunctionV2S, n: Int)
      extends Minimize:

def initSimplex (x0: VectorD, step: Double): Unit =
def transform (): Double =
def solve (x0: VectorD, step: Double = 1): FuncVec =
```

A.15.2 Exercises

1. Make a table showing the new candidate points/vertices for each of the first five iterations of the Nelder-Mead algorithm. Include the functional values of these points.

2. Consider ways to normalize the search space and how this can improve the performance of the Nelder-Mead algorithm.

3. The performance of the Nelder-Mead algorithm can be affected by the choice of the initial simplex. Discuss issues related to (a) initial general location, (b) initial size, and (c) initial shape.

   Hint: see "Practical Initialization of the Nelder–Mead Method for Computationally Expensive Optimization Problems," [188].

Appendix B

Graph Databases and Analytics

Graph databases enhance traditional, row-oriented relational databases by making implicit relationships where a foreign key references a primary key, explicit. For example, in the `sensor` relation rather having a foreign key `roadId` that references the primary key `roadId` in the `road` relation, the relationship is made explicit via an edge-type.

Table B.1 shows the correspondence between concepts in the relational database model and the graph database model.

Table B.1: Mapping from Relational Database Concept to Graph Database Concept

Relational	**Graph**	**Description**
Domain	Domain	a set of values or datatype
Value	Value	an individual/atomic data value
Attribute	Property	named components in a vertex/edge
Tuple	Vertex	aggregation of values related to an entity
Relation	Vertex-Type	a collection of vertices with the same type
Primary Key	Primary Key	unique combination of attributes used a the main identifier
Foreign Key	Edge	a directed edge links a source vertex u to a target vertex v
-	Edge-Type	a collection of edges with the same type
-	PGraph	a collection of vertex-types and edge-types
Database	Database	a collection of graphs

The notion that all the vertices in a vertex-type have the same type means that their properties have the same names and domains. This also implies that they will have the same arity. Furthermore, the notion that all edges in an edge-type have the type means that their properties have the same names and domains as well as they connect vertices from the same vertex-types. For example, edges in the `Takes` edge-type connect vertices from the `Student` vertex-type to the `Course` vertex-type. Note, property graphs may have fixed schema, be schema-less or have flexible schema.

The subsections below define several types of graphs that are useful for graph analytics and graph databases. Such graphs need information content that is provided by labels or properties. In some cases, the labels or properties may only be associated with vertices/nodes, but more generally, they are provided for both vertices and edges. Labels associate a single value, while properties associate multiple values. For graph analytics and some types of graph databases, labels may suffice, but properties are usually required for robust database applications.

B.1 Directed Graphs

The starting point for graph databases is the basic definition of a directed graph from *graph theory*. A *directed graph* consists of vertices (nodes) connected via one-way (directed) edges. More formally, it is a two-tuple $G(V, E)$ where

- V = set of vertices/nodes
- $E \subseteq V \times V$ set of directed edges

The edge from vertex/node u to vertex/node v is an ordered pair (u, v), also denoted as $u \to v$, or more concisely uv. The *from* vertex u is referred to as the *source* and the *to* vertex v is referred to as the *target*.

Consider the directed graph shown in Figure B.1

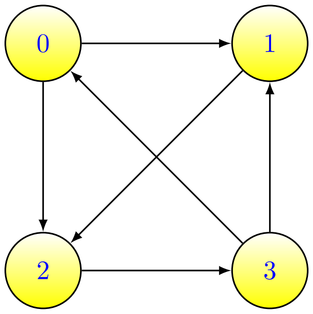


Figure B.1: Example Directed Graph with 4 Vertices and 6 Edges

It can be represented in multiple ways:
As a two-tuple $G(V, E)$.

$$\begin{aligned} V &= \{0, 1, 2, 3\} \\ E &= \{(0, 1), (0, 2), (1, 2), (2, 3), (3, 0), (3, 1)\} \end{aligned}$$

As an array of adjacency sets/lists.

$$\begin{aligned} a(0) &= \{1, 2\} \\ a(1) &= \{2\} \\ a(2) &= \{3\} \\ a(3) &= \{0, 1\} \end{aligned}$$

As an adjacency matrix.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

As a transition matrix giving the probability of moving from vertex u to an adjacent vertex v is d_u^{-1} = one over the out-degree of vertex u.

$$P \;=\; \begin{bmatrix} 0 & .5 & .5 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ .5 & .5 & 0 & 0 \end{bmatrix}$$

This assumes a uniform probability of selecting the next vertex to move to. In general, the transition matrix may be defined using Markov Chain transition probabilities (also see the Chapter on State Space Models).

Movement in a directed graph must be along the edges and is called a *walk* meaning a sequence of edges $(e_0, e_1, ...)$, where the source vertex of e_i must be the target vertex of e_{i-1}. A *trail* is a walk where all the edges are distinct. Finaly, a *path* is a trail where all the vertices are distinct.

A directed graph may be disconnected, weakly connected, or strongly connected. It is *strongly connected* if a path exists between any two vertices $u, v \in V$. It is *weakly connected* if is underlying undirected graph is connected, and is *disconnected* otherwise.

B.1.1 Adding Vertex Labels

The previous definition of a graph captures the topological structure, but needs to be augmented to attach information. The simplest way to do this is to just add labels to vertices. A *vertex-labeled directed graph* is a four-tuple $G(V, E, L, l)$ where

- V = set of vertices/nodes
- $E \subseteq V \times V$ { source-vertex, target-vertex }
- L = set of labels
- $l : V \to L$ is a labelling function

Notice that if a graph has n_v vertices/nodes, the number of edges may range from 0 to n_v^2. There are two general ways to represent the connectivity structure of a graph, an *adjacency matrix* and an *adjacency list*. When the graph is sparse, i.e., $n_e \ll n_v^2$, then an adjacency list will be more efficient.

Under this approach, a vertex/node needs to keep track of its outgoing edges (also known as children). The `Graph0` class below has a `name` and a flag `inverse` indicating whether to also store the incoming edges (parents). The rest of the arguments to the constructor are `Arrays`, where each element of the array corresponds to a vertex in the graph. These arrays record the children `ch` and the vertex `label`. In addition, vertex ids `id` are maintained internally. The children of a node are a set of integers. For example, vertex 1 may have children { 2, 4, 5 }.

Consider the graph shown in Figure B.2. If `Bob` and `Sue` know each other, then the graph consists of 2 vertices and 2 edges.

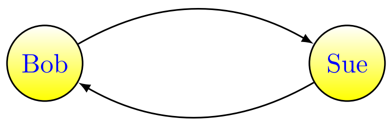


Figure B.2: Example Vertex-Labeled Directed Graph

`Graph0` **Class**

Class Methods:

```
    @param ch         the array of child (adjacency) vertex sets (outgoing edges)
    @param label      the array of vertex labels: v -> vertex label
    @param inverse    whether to store inverse adjacency sets (parents)
    @param name       the name of the digraph

    case class Graph0 (ch: Array [SET [Int]],
                       label: Array [ValueType],
                       inverse: Boolean = false,
                       name: String = "g",
         extends Cloneable:

        val id = Array.range (0, ch.size)                  // array of vertex id's
```

B.1.2 Adding Edge Labels

Providing both edge and vertex labels allows richer information content to be stored. The vertex label describes the vertex. In general, it may be unique or non-unique. Similarly, the edge label describes the edge and is often used to specify a category or type (usually non-unique).

The graph shown in Figure B.3 takes the previous figure and adds labels (`knows` or `employs`) to the edges.

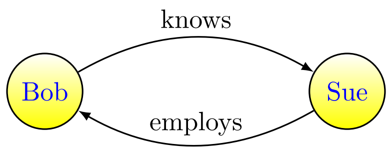


Figure B.3: Example Fully-Labeled Directed Graph

A *fully-labeled directed graph* is a five-tuple $G(V, E, L, l_v, l_e)$ where

- V = set of vertices/nodes
- $E \subseteq V \times V \qquad \{$ source-vertex, target-vertex $\}$

- L = set of labels
- $l_v : V \to L$ is a vertex labeling function
- $l_e : E \to L$ is an edge labeling function

Of course, it may also be useful to split the set of labels L into L_v = set of vertex/node labels and L_e = set of edge labels.

The `Graph` class in `scalation.database.graph_pm` shown below provides more functionality than the `Graph0` class, mainly adding edge labels `elabel`. The edge labels are stored in a `Map` for looking up the label based on the source and target vertices. An optional `schema` specification is also provided to introduce a type structure. (This will not be discussed further here.)

`Graph` Class

Class Methods:

```
    @param ch       the array of child (adjacency) vertex sets (outgoing edges)
    @param label    the array of vertex labels: v -> vertex label
    @param elabel   the map of edge labels: (u, v) -> edge label
    @param inverse  whether to store inverse adjacency sets (parents)
    @param name     the name of the multi-digraph
    @param schema   optional schema: map from label to label type

    case class Graph (ch: Array [SET [Int]],
                      label: Array [ValueType],
                      elabel: Map [(Int, Int), ValueType],
                      inverse: Boolean = false,
                      name: String = "g",
                      schema: Array [String] = Array ())
         extends Cloneable:

    val id = Array.range (0, ch.size)                    // array of vertex id's
```

Example: Create a `Graph`

```
val links = new Graph (Array (SET (1), SET (0), SET (0, 1)),     // children
                       Array ("Bob", "Sue", "Joe"),              // vertex labels
                       Map ((0, 1) -> "knows",                   // edge labels
                            (1, 0) -> "employs",
                            (2, 0) -> "knows",
                            (2, 1) -> "knows"),
                       false, "links")
```

Notice that since there is a single label per edge and only one edge between any two vertices, the basic structure borrowed from graph theory has not changed. It has only been embellished.

B.1.3 Directed Multi-Graphs

In order to better model the real-world, if one thinks of vertices representing entities and edges representing relationships, then it makes sense for two entities to be in different relationships, e.g., `Sue knows Bob` and `Sue employs Bob`. Allowing multiple edges (and edge labels) between two vertices, (u, v), introduces a fundamental change to the graph.

No longer is an edge uniquely identified by its source and target vertices, $u \rightarrow v$. So long as the edge labels are different, multiple edges may connect the same source and target vertices. For example, if `Bob` and `Sue` know each other and `Sue` employs `Bob`, then the graph consists of 2 vertices and 3 edges as shown in Figure B.4.

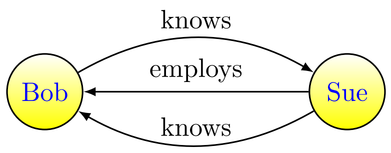


Figure B.4: Example Directed Multi-Graph

Allowing multiple edges between vertices along with both edge and vertex labels allows rich information content to be stored. The easiest way to achieve this to simply make the edge labels multi-valued.

Following this approach, a *fully-labeled directed multi-graph* may be represented as a five-tuple $G(V, E, L, l_v, l_e)$ where

- V = set of vertices/nodes
- $E \subseteq V \times V$ { source-vertex, target-vertex }
- L = set of labels
- $l_v : V \rightarrow L$ is a vertex labeling function
- $l_e : E \rightarrow 2^L$ is an edge labeling function

Note, 2^L is the power-set of the labels, so l_e may map to any element of the power-set (i.e., any subset).

Such graphs may implemented as follows (see the `scalation.database.mugraph_pm` package):

`MuGraph` Class

Class Methods:

```
    @param ch        the array of child (adjacency) vertex sets (outgoing edges)
    @param label     the array of vertex labels: v -> vertex label
    @param elabel    the map of edge labels: (u, v) -> set of edge label
    @param inverse   whether to store inverse adjacency sets (parents)
    @param name      the name of the multi-digraph
    @param schema    optional schema: map from label to label type

    case class MuGraph (ch: Array [SET [Int]],
                        label: Array [ValueType],
                        elabel: Map [(Int, Int), SET [ValueType]],
                        inverse: Boolean = false,
                        name: String = "g",
                        schema: Array [String] = Array ())

        val id = Array.range (0, ch.size)                 // array of vertex id's
```

Making Multiple Edges Explicit

One may say the easiest approach allows an edge to have multiple labels, but does not really support having multiple edges between a pair of vertices. However, a straightforward algorithm can convert it to a representation that does. A natural storage structure that makes the multiple edges explicit is one that treats edges as triples.

Using triples, a *fully-labeled directed multi-graph* may be represented as a four-tuple $G(V, E, L, l)$ where

- V = set of vertices/nodes
- $E \subseteq V \times L \times V$ $\quad$ { source-vertex, edge-label, target-vertex }
- L = set of labels
- $l : V \to L$ $\quad$ is a vertex labeling function

Again, it may also be useful to split the set of labels L into L_v = set of vertex/node labels and L_e = set of edge labels.

As mentioned, an edge is no longer an ordered pair, as it becomes a *triple* where the 3 parts are known by various names as shown in Table B.2

Table B.2: Components of a Triple

(u, l, v)	source-vertex	edge-label	target-vertex
(h, r, t)	head	relation	tail
(s, p, o)	subject	predicate	object

This concept may implemented as follows (see the `scalation.database.triplegraph` package): The `Triple` class holds information about a triple (3 part edge).

```
    @param h   the head vertex
    @param r   the relation/edge-label
    @param t   the tail vertex

    case class Triple (h: Int, r: ValueType, t: Int)
```

`TripleGraph` **Class**

Class Methods:

```
    @param label    the array of vertex labels
    @param triples  the bag of triples in the triple-graph
    @param name     the name of the triple-graph
    @param schema   optional schema: map from label to label type

    case class TripleGraph (label: Array [ValueType],
                            triples: Bag [Triple],
                            name: String = "g",
                            schema: Array [String] = Array ())
         extends Cloneable:
```

Such a structure provides the basis for Resource Description Framework (RDF) graphs, to be discussed later in this chapter.

B.1.4 Exercises

1. Given the above adjacency matrix A, compute the k-hop adjacency A^k for $A^2 = AA$.
2. For the graph shown in Figure B.1 given an example of a path, a trail that is not a path, and a walk that is not a trail.
3. Draw the directed multi-graph called `links` given in Figure B.4.
4. Survey the literature to compare different ways of creating storage structures for fully-labeled directed multi-graphs.
5. Implement an algorithm in Scala 3 that converts a `MuGraph` to a `TripleGraph`.
6. Implement an algorithm in Scala 3 that converts a `TripleGraph` to a `MuGraph`.

B.2 A Graph Database with Relational Roots

Before moving onto Property Graphs (the predominate form of today's Graph Databases), a further extension of the `Table` class is discussed. The main difference is that it uses a tuple construct to store vertex and edge attributes/properties, following more closely to the Relational Model and requiring the specification of a schema. Property Graphs use maps for storing properties of vertices and edge and support having schema and being schema-less.

B.2.1 The `GTable` Class

```
@param name_    the name of the graph-table
@param schema_  the attributes for the graph-table
@param domain_  the domains/data-types for attributes ('D', 'I', 'L', 'S', 'X', 'T')
@param key_     the attributes forming the primary key

case class GTable (name_ : String, schema_ : Schema, domain_ : Domain, key_ : Schema)
     extends Table (name_, schema_, domain_, key_)
        with Serializable:
```

The `GTable` class (for Graph-Table) supports many-to-many relationships with efficient navigation in both directions. Supporting this is much more complicated than what is needed for `LTable`, but provides for *index-free adjacency*, similar to what is provided by Graph Database systems.

The `GTable` model is graph-like in that it (as did `VTable`) elevates tuples into vertices as first-class citizens of the data model. Also, a directed edge has attributes and serves to link a source (from) vertex to a target (to) vertex. Now, if distance is included as one of the edge attributes, shortest path algorithms may be applied.

The `Edge class` includes three parts: The edge attributes in the form of a tuple of values, the source (from) vertex and target (to) vertex.

```
@param tuple  the tuple part of the edge
@param from   the source vertex
@param to     the target vertex

case class Edge (tuple: Tuple, from: Vertex, to: Vertex):

    def reverse: Edge = Edge (tuple, to, from)
    override def toString: String = s"edge: ${stringOf (tuple)}"

end Edge
```

The `Vertex` class extends the notion of `Tuple` into values stored in the tuple part, along with foreign keys links captured as outgoing edges. The edge `Map` has a key that is the edge label (e.g., `employs`) and a value that is a set/bag of outgoing edges (e.g., all of the outgoing employs edges). Each edge in turn references the target vertex (e.g., the person employed).

```
@param tuple  the tuple part of the vertex

case class Vertex (tuple: Tuple):

    val edge = Map [String, Bag [Edge]] ()       // map edge-label -> { edges }

```

```
        def neighbors: Bag [Vertex] =
        def neighbors (elab: String): Bag [Vertex] =
        def neighbors (ref: (String, GTable)): Bag [Vertex] =
        override def toString: String = s"vertex: ${stringOf (tuple)}"

    end Vertex
```

B.2.2 Creating Graph Databases

A graph database may be created by constructing `GTable`s and linking them via edge-types, as shown below:

```
    val student   = GTable ("student",   "sid, sname, street, city, dept, level",
                                         "I, S, S, S, S, I", "sid")
    val professor = GTable ("professor", "pid, pname, street, city, dept",
                                         "I, S, S, S, S", "pid")
    val course    = GTable ("course",    "cid, cname, hours, dept",
                                         "I, X, I, S", "cid")

    student.addEdgeType ("cid", course, false)                 // student has M courses
    course.addEdgeType ("sid", student, false)                 // course has M students
    course.addEdgeType ("pid", professor)                      // course has 1 professor
```

B.2.3 Graph Algebra

Implementations of the main graph algebra operators are analogs of those in relational algebra. Project simply projects on the tuple part of a vertex.

```
    override def project (x: Schema): GTable =
        val newKey = if subset (key, x) then key else x
        val s = new GTable (s"${name}_p_${cntr.inc ()}", x, pull (x), newKey)
        s.vertices ++= (for v <- vertices yield Vertex (pull (v.tuple, x)))
        s
    end project
```

Select applies a given predicate to all vertices in this `GTable`, keeping those where the predicate evaluates to true.

```
    override def select (predicate: Predicate): GTable =
        val s = new GTable (s"${name}_s_${cntr.inc ()}", schema, domain, key)
        s.vertices ++= (for v <- vertices if predicate (v.tuple) yield v)
        s
    end select
```

Union takes the union of all vertices in the two tables. An index may be created to eliminate duplicates.

```
    override def union (r2: Table): GTable =
        if incompatible (r2) then return this
        val s = new GTable (s"${name}_u_${cntr.inc ()}", schema, domain, key)
        s.vertices ++= (
        if r2.isInstanceOf [GTable] then vertices ++ r2.asInstanceOf [GTable].vertices
        else vertices ++ r2.tuples.map (Vertex (_)))
        s
    end union
```

Minus will keep each vertex in this table only if it is not the second table.

```
    override def minus (r2: Table): GTable =
        if incompatible (r2) then return this
        val s = new GTable (s"${name}_m_${cntr.inc ()}", schema, domain, key)
        for v <- vertices do
            if ! (r2 contains v.tuple) then s.vertices += v
        end for
        s
    end minus
```

Having a graph database where the vertices are explicitly linked via edges, reduces the frequency with which join operations are needed. Starting with a subset of vertices in a first `GTable`, relevant attribute values can be efficiently extracted from this and a second table without performing a join.

For example, if one wishes to run a query to retrieve the courses each student is taking, one may write the following:

```
    student extract ("sname, cname", ("cid", course))
```

This query will extract `sname` values from the `Student` table and pair them with `cname` values from the `Course` table. Use of `cid` indicates the edge-type to follow. (In general, there may be multiple types of outgoing edges.)

B.2.4 Exercises

1. Use Scala 3 to complete the implementation of `GTable` in the `scalation.database.table` package.

2. Using your implementation for `GTable`, create a schema where vertices represent cities and edges represent roads connecting them. City attributes include id, name, state, lat, long, and population. Road attributes include distance.

3. Populate the database with sample data for Northeast Georgia, with major roads (US and State Roads) connecting the following cities: Athens, Jefferson, Watkinsville, Monroe, Bethlehem, Winder, Bogart, Statham, Bishop, Good Hope, Hull, Colbert, Crawford, Nicholson, Danielsville.

4. Using `GTable`'s graph algebra, list all major roads leaving Athens, GA.

5. List all paths from Athens, GA to Winder, GA.

6. Find the shortest path (distance-wise) from Athens, GA to Winder, GA.

B.3 Property Graphs

Property graphs [16] enhance plain labeled directed multi-graphs by replacing labels with a list/map of properties. (Note, some database models and systems such as Neo4j keep labels and assign them a special purpose.) As such, information may be flexibly added to a property graph. The structure and type of a property graph is defined by specifying its structural organization (analogously to specifying relational schema). Some Graph Databases/Property Graphs are schema-less to add greater flexibility and agility.

Property Graphs may be divided into two groups: Labeled Property Graph and Typed Property Graphs. A typed property graph requires all vertices to have a unique Vertex-Type that defines the vertex-schema (set of properties) for vertices of a given type. Similarly, it requires all edges to have a unique Edge-Type. In addition to defining the edge-schema, it specifies the source and target Vertex-Types. Some graph databases engines, however, chose to provide users with greater flexibiilty. For instance, Neo4j does not give a vertex a type, but instead may give one or more labels to a vertex (node). For example, the label `Person` may be use to indicate the kind of node (this is done without creating a Vertex-Type). Some vertices may have two (or more) labels, e.g., `Person` and `Golfer`. As vertices are often accessed based on their labels, indices are automatically created to find the vertices with given labels. Different graph database engines make choices regarding whether to use labels or types for vertices, as well as such choices for edges. Furthermore, labels may be optional vs. required, as well as unique vs. multivalued. See the following document for the design choices made by some of the popular graph database engines: `https://medium.com/geekculture/labeled-vs-typed-property-graphs-all-graph-databases-are-not-the-same-efdbc782f099`.

B.3.1 Structure of a Property Graph

The following structural definitions specify datatypes and structural organization for property graphs as defined in the `scalation.database.graph` package. In order to support rapid access to into collections `VEC` is used. Currently, the fastest such Scala 3 structure for immutable collections is `Vector`, while for mutable collections it is `ArrayBuffer`.

```scala
//  import scala.collection.immutable.{Vector => VEC}
    import scala.collection.mutable.{ArrayBuffer => VEC}
```

ScalaTion's graph database supports property graphs and is organized as follows: The primary concepts are vertex and edge. Similar vertices and edges are collected into vertex-types and edge-types. A property graph consists of multiple vertex-types and edge-types. Element and element type are introduced as generalizations that are useful in defining graph algebra operators.

- The `ValueType` type is a Scala 3 union type for atomic database values.

```scala
    type ValueType = ( Double | Int | Long | String | TimeNum )
```

- The `Property` type corresponds to the notion of an attribute in a relational database. It maps property names to property values (e.g., `Map ("name" -> "Bob", "salary" -> 85000.0)`.

```scala
    type Property = Map [String, ValueType]
```

An example of a property is shown below.

```scala
    val knows: Property = Map ("type" -> "knows")
```

`Vertex` Class

The `Vertex` class specifies the form of vertices/nodes in the property graph. A vertex maintains properties for a vertex, e.g., a `person`. It is analogous to a tuple in a relational database.

```
    @param _name  the name of this vertex ('name' from `Identifiable`), vertex label
    @param prop   maps vertex property names into property values
    @param _pos   the position (Euclidean coordinates) of this vertex ('pos' from `Spatial`)

    class Vertex (_name: String, val prop: Property, _pos: VectorD = null)
          extends Identifiable (_name)
             with Spatial (_pos)
             with PartiallyOrdered [Vertex]
             with Serializable:
```

For example, three vertices can easily be created.

```
@main def vertexTest (): Unit =

    val x0 = VectorD (200, 100, 40, 40)
    val x1 = VectorD (100, 400, 40, 40)
    val x2 = VectorD (500, 500, 40, 40)

    // use the class constructor to explicitly assign vertex label

    val vertices = Array (
        new Vertex ("Bob", Map ("name" -> "Bob", "state" -> "GA", "salary" -> 85000.0), x0),
        new Vertex ("Sue", Map ("name" -> "Sue", "state" -> "FL", "salary" -> 95000.0), x1),
        new Vertex ("Joe", Map ("name" -> "Joe", "state" -> "GA", "salary" -> 99000.0), x2))

    banner ("Vertices with explicitly assigned vertex labels")
    Vertex.show (vertices)

    // use the companion object's apply method to use system generated vertex label

    val vertices2 = Array (
        Vertex (Map ("name" -> "Bob", "state" -> "GA", "salary" -> 85000.0), x0),
        Vertex (Map ("name" -> "Sue", "state" -> "FL", "salary" -> 95000.0), x1),
        Vertex (Map ("name" -> "Joe", "state" -> "GA", "salary" -> 99000.0), x2))

    banner ("Vertices with system generated vertex labels")
    Vertex.show (vertices2)

end vertexTest
```

`VertexType` Class

The `VertexType` class corresponds to the notion of an entity type in an Entity-Relationship Model. A vertex-type collects vertices of the same type, e.g., a person vertex-type. Its schema specification determines the types of properties that vertices in this collection are allowed to have. The `color` and `shape` are for display purposes. It is analogous to a relation with no foreign keys in a relational database.

```
    @param _name   the name of this vertex type ('name' form `Identifiable`)
    @param schema  the property names for this vertex type
    @param verts   the set of vertices having this vertex type (extension)
```

```
@param color   the display color for vertices of this type
@param shape   the display shape template for vertices of this type

class VertexType (_name: String, val schema: Schema,
                  val verts: VEC [Vertex] = VEC [Vertex] (),
                  val color: Color = yellow,
                  val shape: Shape = Ellipse ())
      extends Identifiable (_name)
         with Serializable:
```

The three vertices defined above may be collected into a vertex-type.

```
val vt0 = VertexType ("person", "name, state, salary", vertices)
```

The information content in a vertex-type can be shown in a tabular format.

Table B.3: Person VertexType

vertex	name	state	salary
v_0	“Bob”	“GA”	85000.0
v_1	“Sue”	“FL”	95000.0
v_2	“Joe”	“GA”	99000.0

Edge Class

The Edge class allows explicit relationships to be formed between vertices. An edge connects a source vertex to a target vertex and may have its own properties. It may be thought of a triple (from, prop, to), but also allows for an edge label and shift used for diplay purposes. It is roughly analogous to an implicit relationship manifest via foreign key-primary key pairs in a relational database.

```
@param _name  the name of this edge ('name' from `Identifiable`), edge label
@param from   the source/from vertex of this edge
@param prop   maps edge property names into property values
@param to     the target/to vertex of this edge
@param shift  number of units to shift to accomodate a bundle of egdes in a composite
edge

class Edge (_name: String, val from: Vertex, val prop: Property, val to: Vertex, val
shift: Int = 0)
      extends Identifiable (_name)
         with Spatial (if from == null then to.pos else from.pos)
         with Serializable:
```

For example, three edges can easily be created.

```
// use the class constructor to explicitly assign edge label

val edges = VEC (
    new Edge ("knows", v(0), Map ("type" -> "knows", "since" -> 5), v(1)),
    new Edge ("knows", v(1), Map ("type" -> "knows", "since" -> 2), v(0), -1),
    new Edge ("knows", v(2), Map ("type" -> "knows", "since" -> 4), v(0)))

banner ("Edges with explicitly assigned edge labels")
Edge.show (edges)

// use the companion object's apply method to use system generated edge label

val edges2 = VEC (
    Edge (v(0), Map ("type" -> "knows", "since" -> 5), v(1)),
    Edge (v(1), Map ("type" -> "knows", "since" -> 2), v(0), -1),
    Edge (v(2), Map ("type" -> "knows", "since" -> 4), v(0)))

banner ("Edges with system generated edge labels")
Edge.show (edges2)
```

`EdgeType` Class

The `EdgeType` class corresponds to the notion of an relationship type in an Entity-Relationship Model. An edge-type collects edges of the same type and has a source `VertexType` and a target `VertexType`. Its schema specification determines the type of properties that edges in this collection are allowed to have. The `color` and `shape` are for display purposes. An edge-type is analogous to a relation with foreign keys in a relational database.

```
    @param _name   the name of this edge-type ('name' from 'Identifiable')
    @param from    the source vertex
    @param schema  the property names for this edge-type
    @param to      the target vertex
    @param edges   the set of edges having this edge-type (extension)
    @param color   the display color for edges of this type
    @param shape   the display shape template for edges of this type

    class EdgeType (_name: String,
                    val from:  VertexType, val schema: Schema, val to: VertexType,
                    val edges: VEC [Edge] = VEC [Edge] (),
                    val color: Color = blue,
                    val shape: CurvilinearShape = Arrow ())
          extends Identifiable (_name)
             with Serializable:
```

The three edges defined above may be collected into an edge-type.

```
    val et0 = EdgeType ("knows", vt0, "type", vt0, edges)
```

The information content in an edge-type can be shown in a tabular format.

Table B.4: Knows Edge-Type

edge	**from-vertex**	**type**	**since**	**to-vertex**
e_0	v_0	"knows"	5	v_1
e_1	v_1	"knows"	2	v_0
e_2	v_2	"knows"	4	v_0

`PGraph` Class

The `PGraph` class is used to store property graphs. A property graph has a name, zero or more `VertexTypes` and zero or more `EdgeTypes`. Each of the `EdgeTypes` can only reference `VertexTypes` specified in this `PGraph`. The `animating` and `aniRatio` are used for diaplay purposes.

```
    @param name        the name of the property graph
    @param vt          the set of vertex types
    @param et          the set of edges types
    @param animating   whether to animate the model (defaults to false)
    @param aniRatio    the ratio of simulation speed vs. animation speed

    class PGraph (val name: String,
                  val vt: VEC [VertexType] = VEC [VertexType] (null),
                  val et: VEC [EdgeType]   = VEC [EdgeType] (null),
                  animating: Boolean = false, aniRatio: Double = 1.0)
          extends Serializable:
```

A `SocialNetwork` property graph can be created as shown below. See the example below to see the full construction of the property graph.

```
    val g = PGraph ("SocialNetwork", VEC (vt0), VEC (et0, et1))
```

A `PGraph` with no `EdgeTypes` essentially allows a similar organization to a relational database. Simply add a property that acts as a foreign key and use a join operation.

Example: Create a `PGraph`

```
object SocialNetwork:

    // Populate the property-graph

    val employs: Property = Map ("type" -> "employs")

    val x0 = VectorD (200, 100, 40, 40)                            // vertex coordinates,
    size: x, y, w, h
    val x1 = VectorD (100, 400, 40, 40)
    val x2 = VectorD (500, 500, 40, 40)

    // Build vertices and vertex-types

    val v = VEC (
        new Vertex ("Bob", Map ("name" -> "Bob", "state" -> "GA", "salary" -> 85000.0), x0),
        new Vertex ("Sue", Map ("name" -> "Sue", "state" -> "FL", "salary" -> 95000.0), x1),
        new Vertex ("Joe", Map ("name" -> "Joe", "state" -> "GA", "salary" -> 99000.0), x2))
    val vt0 = VertexType ("person", "name, state, salary", v)

    println (s"check schema for vertex-type vt0 = ${vt0.check}")
    vt0.buildIndex ("name")

    // Build edges and edge-types

    val et0 = EdgeType ("knows", vt0, "type", vt0, VEC (
        new Edge ("knows", v(0), Map ("type" -> "knows", "since" -> 5), v(1)),
        new Edge ("knows", v(1), Map ("type" -> "knows", "since" -> 2), v(0), -1),
        new Edge ("knows", v(2), Map ("type" -> "knows", "since" -> 4), v(0))))
    val et1 = EdgeType ("employs", vt0, "type", vt0, VEC (
        new Edge ("employs", v(1), employs, v(0), 1),
        new Edge ("employs", v(2), employs, v(1))))

    println (s"check schema for edge-type et0 = ${et0.check}")
    println (s"check schema for edge-type et1 = ${et1.check}")

end SocialNetwork
```

As an exercise, draw the property graph called `SocialNetwork`.

B.3.2 Native Storage

Achieving the performance potential for graph databases requires appropriate data structures and organizations, whether in main memory or in files. It is common for graph database engines to support *index-free adjacency*, although various hybrids approaches are also possible [128].

There is growing popularity of graph database engines that support native storage, see `https://db-engines.com/en/ranking/graph+dbms` including Neo4j, JanusGraph, TigerGraph, and TinkerGraph.

Index-Free Adjacency

Notice the basic structure for property graphs, makes it easy to find vertices connected to an edge. Some types of navigation will require finding edges that a vertex is connected to. Suppose there are vertex-types `Student` and `Course` connected with edge-type `Takes`. The meta-graph for this is shown in the next subsection. Suppose a student named `"Bob"` wishes to know the courses he is taking. One would like to traverse from the `"Bob"` vertex to his course vertices. In general this will not work, since some queries may depend upon an edge property (e.g., the `grade`). In order to handle such cases, the `"Bob"` vertex should reference its outgoing edges in the `Takes` edge-type. Unfortunately, `"Bob"` may have taken many courses (i.e., the `Takes` edge/relationship type is many-to-many). A storage solution is to have the vertex reference the first edge in the `"Bob"` group of courses and have this edge reference the next edge in the group, etc. The chain of edges may be organized as either a singly or doubly linked list, however, to make edge deletions efficient the list should be doubly linked. This is the approach utilized by Neo4j's native graph storage engine, as described in Chapter 6 of [156]. Note, ScalaTion's `GTable` class provides an alternative of collecting all a vertex's outgoing edges into an `ArrayBuffer`.

To illustrate details about the storage, example vertex-types and edge-types will be shown for a simple course-enrollment database.

Meta-Graph for Course Enrollment Database

Property Graphs are rich enough to depict their own conceptual model. A meta-graph showns the meta-data for a graph database. The meta-graph in Figure B.5 shows three vertex-types, `Student`, `Course` and `Professor`. These are connected via the `Takes` and `TaughtBy` edge-types. The `Takes` edge/relationship type is many-to-many, while the `TaughtBy` edge/relationship type is many-to-one.

The arrow on the `TaughtBy` edge-type indicates a many-to-one relationship from `Course` to `Professor`. That is, a course is taught by one `Professor`, while a professor may teach many `Courses`.

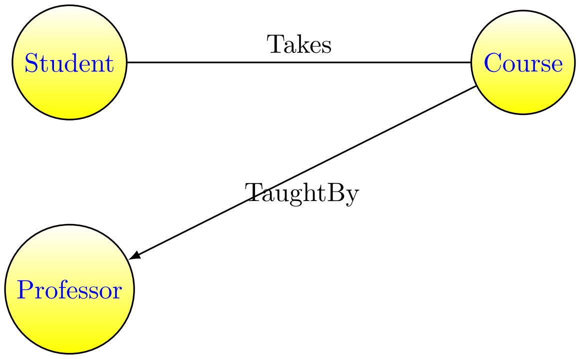


Figure B.5: Meta-Graph for the Course Enrollment Database

Storage for Course Enrollment Database

The `Course` and `Professor` vertex-types as well as the `TaughtBy` edge-type are shown in Figure B.5 in tabular format. Using the index-free adjacency approach, each `Course` vertex maintains a reference to its first outgoing `TaughtBy` edge. Similarly, each `Professor` vertex maintains a reference to its first incoming `TaughtBy` edge.

To handle the case when there are multiple edges (e.g., a `Professor` may teach multiple `Course`s), an edge chain is maintained in the `TaughtBy` Edge-Type. Following the edge chain for "Dr. Bill", gives the first edge as e_0, with the next edge e_1 and then e_2; a null terminates the edge chain.
The edge chains are shown here as singly-linked lists for simplicity, while in practice doubly-linked lists allow for more efficient maintenance of the edge chains.

Table B.5: Course Enrollment Database

(a) `Course` Vertex-Type

vertex	**cname**	**hours**	**→taughtBy**
v_0	“DB”	4	e_0
v_1	“SE”	4	e_1
v_2	“OS”	4	e_2
v_3	“AI”	3	e_3
v_4	“ML”	3	e_4

(b) `Professor` Vertex-Type

vertex	**pname**	**rank**	**taughtBy→**
v_5	“Dr. Bill”	AssocProf	e_0
v_6	“Dr. John”	Professor	e_3

(c) `TaughtBy` Edge-Type

edge	**from-vertex**	**type**	**to-vertex**	**next edge**
e_0	v_0	“taughtBy”	v_5	e_1
e_1	v_1	“taughtBy”	v_5	e_2
e_2	v_2	“taughtBy”	v_5	null
e_3	v_3	“taughtBy”	v_6	e_4
e_4	v_4	“taughtBy”	v_6	null

B.3.3 High-Level Query Language for Graph Databases

High-level query languages for graph database have the potential to be more concise and more intuitive than the SQL language for relational databases. Examples of high-level query languages for graph databases include *Cypher*, *Gremlin* and `GQL`.

- Neo4j’s Cypher Query Language. The `MATCH` statement is structured according the `TaughtBy` edge/relationship, with the `WHERE` constraining the professors and the `RETURN` indicating the returned results.

```
    MATCH (c: Course) - [: TaughtBy] -> (p: Professor)
    WHERE p.pname = 'Dr. John'
    RETURN c.cname
```

- Apache TinkerPop’s Gremlin Query Language. Using syntax from the Groovy programming language (a JVM cousin of Scala), the same query can be written in a functional programming style.

```
    g.V().has ('Professor', 'pname', 'Dr. John')
         .out ('TaughtBy')
         .values ('cname')
```

- Graph Query Language (GQL) is an emerging standard for graph database. It utilizes concepts from multiple existing languages, see `https://www.gqlstandards.org/existing-languages`

The Cypher Query Language

In order to execute queries using the Cypher query language, a graph database needs to be populated. This is done by first creating several nodes. These nodes may be referenced to make several edges.

```
CREATE
(s1: Student   { sid: 101,  sname: 'Peter', city: 'Athens' }),
(s2: Student   { sid: 102,  sname: 'Paul', city: 'Bogart' }),
(s3: Student   { sid: 103,  sname: 'Mary', city: 'Athens' }),
(c1: Course    { cid: 4370, cname: 'DB', hours: 4 }),
(c2: Course    { cid: 4550, cname: 'AI', hours: 3 }),
(p1: Professor { pid: 201,  pname: 'Dr. Bill', rank: 'AssocProf' }),
(p2: Professor { pid: 202,  pname: 'Dr. John', rank: 'Professor' }),
(s1)-[: Takes]->(c1),
(s2)-[: Takes]->(c1),
(s3)-[: Takes]->(c1),
(s2)-[: Takes]->(c2),
(s3)-[: Takes]->(c2),
(p1)<-[: TaughtBy]-(c2),
(p2)<-[: TaughtBy]-(c1)
```

Conventions:

- ( ) indicates a node, e.g., (s) or (s: Student)
- -[ ]-> indicates a forward edge, e.g., (s2)-[: Takes]->(c1)
- <-[ ]- indicates a backward edge, e.g., (p2)<-[: TaughtBy]-(c1)
- { } indicates a list of properties, e.g., { cid: 4370, cname: 'DB', hours: 4 }

Note: In a MATCH statement -[ ]- will match in either direction.

Queries can be concisely specified using MATCH statements. The following query retrieves all nodes with label/type `Student` (like returning a `Student` table in a relational database). If the label `Student` is left out, all nodes will be returned.

```
MATCH (s: Student)
RETURN s
```

A WHERE clause can be added to, for example, restrict the returned nodes to students living in 'Athens'.

```
MATCH (s: Student)
WHERE s.city = 'Athens'
RETURN s
```

This filter can be specified inside the node specification itself, as shown below.

```
MATCH (s: Student { city: 'Athens'} )
RETURN s
```

Many meaningful queries correspond to following paths in the graph database. Cypher provides convenient syntax for specifying path patterns. Those paths matching the pattern serve as the basis for what is returned. In the query below, the `sid` and `sname` of students taking the 'Database' course will be returned.

```
MATCH (s: Student)-[: Takes]->(c: Course { cname: 'Database' } )
RETURN s.sid, s.sname
```

This can be compared to equivalent SQL queries.

```
SELECT s.sid, s.sname
FROM   Student s, Takes t, Course c
WHERE  c.cname = 'Database' and s.sid = t.sid and t.cid = c.cid
```

```
SELECT s.sid, s.sname
FROM   Student s NATURAL JOIN Takes t NATURAL JOIN Course c
WHERE  c.cname = 'Database'
```

It is easy to make connections that are multiple hops away in the graph. To find the professors teaching 'Peter', the following 2-hop path pattern may be used.

```
MATCH (s: Student { sname: 'Peter'})-[: Takes]->(c: Course)-[: TaughtBy]->(p: Professor)
RETURN p
```

Like SQL, Cypher allows results to be combined with UNION (AS is needed for renaming, so the columns agree).

```
MATCH (s: Student)
RETURN s.sname AS name
UNION
MATCH (p: Professor)
RETURN p.sname AS name
```

Note: Like SQL, UNION removes duplicates, while UNION ALL does not.

Other commonly used clauses include ORDER BY to sort the answer to a query and LIMIT to restrict the size of the answer.

For more information on the syntax of Cypher and more example queries, see the Neo4j Cypher Manual, `https://neo4j.com/docs/cypher-manual/current/introduction/`.

B.3.4 Graph Algebra

Often high-level query languages are implemented by translating to an algebraic language that consists of basic operators, such as `select`, `project` and `join`. Although graph algebra is analogous to relational algebra, it is richer and more complex. The operators in ScalaTion's Graph Algebra are implemented in three classes: `VertexType`, `EdgeType`, and `PGraph`.

Operators for Vertex-Types

Several operators apply only to a single vertex-type. For the discussion below, the `person` vertex-type will be used.

```
val person = g.vmap("person")
```

- The **project** operator for vertex-types will project onto the properties given in the specified subschema `x`.

```
def project (x: Schema): VertexType =
    if ! subset (x, schema) then flaw ("project", "subschema x does not follow schema")
    new VertexType (name + "_p", x,
                    for v <- verts yield Vertex (v.prop.filter (x contains _._1)))
end project
```

For example, to return the vertices with the properties trimmed down to just names the following two queries may be used. The corresponding Cypher query is also shown.

```
    val q0 = person.project (Array ("name"))
    val q2 = person.project ("name")
    MATCH (p: Person) RETURN p.name
```

- The **select** operator for vertex-types will return the subset of vertices that satisfy the predicate `pred`.

```
def select (pred: Property => Boolean): VertexType =
    new VertexType (name + "_s", schema,
                    for v <- verts if pred (v.prop) yield v)
end select
```

For example, to return the vertices where the name property is “Sue”, the following two queries may be used (the second one is an abbreviated form of the first one). The corresponding Cypher query is also shown.

```
    val q2 = person.select ((p: Property) => p("name") == "Sue")
    val q3 = person.select (_("name") == "Sue")
    MATCH (p: Person) WHERE p.name = 'Sue' RETURN p
```

- The **unionAll** operator for vertex-types will return all the vertices that from the two vertex-types.

```
def unionAll (vt2: VertexType): VertexType =
    new VertexType (name + "_ua_" + vt2.name, schema, verts ++ vt2.verts)
end unionAll
```

For example, the union of `person` and `android` will return person and android vertices. The corresponding Cypher query is also shown.

```
    val q4 = person unionAll android
    MATCH (p: Person) RETURN p UNION ALL MATCH (a: Andoid) RETURN a
```

An issue to address is defining union-compatibility in a more flexible manner than in relational databases.

- The **union** operator for vertex-types will return the vertices that are in either of two vertex-types, without duplication.

```
def union (vt2: VertexType): VertexType =
    new VertexType (name + "_u_" + vt2.name, schema, (verts ++ vt2.verts).distinct)
end union
```

For example, the union of `person` and `android` will return person and android vertices. If any two vertices have exactly the same properties, the duplicate will be removed. The corresponding Cypher query is also shown.

```
    val q4 = person union android
    MATCH (p: Person) RETURN p UNION MATCH (a: Andoid) RETURN a
```

- The **intersect** operator for vertex-types will return vertices both to both vertex-types (`this` and `vt2`).

```
def intersect (vt2: VertexType): VertexType =
    new VertexType (name + "_i_" + vt2.name, schema, (verts intersect vt2.verts))
end intersect
```

- The **minus** operator for vertex-types will return the vertices that are in either of two vertex-types. The Cypher query language does not currently provide a `MINUS` operation (although a work-around can be used).

```
def minus (vt2: VertexType): VertexType =
    new VertexType (name + "_m_" + vt2.name, schema, verts diff vt2.verts)
end minus
```

  For example, starting with the vertices in `q4` and subtracting the vertices in `android`.

```
    val q5 = q4 minus android
```

- The **groupBy** operator groups the vertices based on sharing a property value `pname`.

```
def groupBy (pname: String, agg_name: String, agg_fn: Double => Double): VertexType =
    debug ("groupBy", s"group $schema by $pname")
    if ! (schema contains pname) then flaw ("groupBy", s"property $pname missing from
    schema")
    if checkMissing (pname) then flaw ("groupBy", s"property $pname missing from a
    vertex")

    val groups   = verts.groupBy [ValueType] (_.prop(pname))      // discriminator
    val vertices = VEC [Vertex] ()
    for g <- groups; v <- g._2 do
        vertices += Vertex (Map (pname -> v.prop(pname),
                                (agg_name -> agg_fn (v.prop(agg_name).toDouble))))
    end for
    new VertexType (name + "_g", Array (pname, agg_name), vertices)
end groupBy
```

- The **orderBy** orders the vertices within this vertex-type by the values of the given property name `pname`.

```
def orderBy (pname: String): VertexType =
    new VertexType (name + "_o", schema, verts.sortWith (_.prop(pname) < _.prop(pname))
    )
end orderBy
```

Operators for Edge-Types

Other operators apply to edge-types. The expand operators expand into vertices that are targets (`to`) or sources (`from`) of the edges in the edge-type. Several other operators are analogs of those provided by vertex-types.

- The **expandTo** operator expand this edge-type with its 'to' vertex-type, appending its properties

```
def expandTo: EdgeType =
    val edgez = for e <- edges yield Edge (e.from, e.prop +++ e.to.prop, null)
    new EdgeType (name + "_et", from, schema ++ to.schema, null, edgez)
end expandTo
```

  Similar operators are **expandFrom** and **expand**.

- Versions of operators in `VertexType` are also provided in `EdgeType`.

```
    def project (x: Schema): EdgeType =
    def select (pred: Property => Boolean): EdgeType =
    def unionAll (et2: EdgeType): EdgeType =
    def union (et2: EdgeType): EdgeType =
    def intersect (et2: EdgeType): EdgeType =
    def minus (et2: EdgeType): EdgeType =
    def orderBy (pname: String): EdgeType =
```

Operators for Property Graphs

The following four operators work on both vertex-types and edge-types to produce new property graphs.

```
    def expandOut (from: VertexType, ets: VEC [EdgeType], tos: VEC [VertexType],
                   newName: String): PGraph =
    def expandIn (froms: VEC [VertexType], ets: VEC [EdgeType], to: VertexType,
                  newName: String): PGraph =
    def expandBoth (froms: VEC [VertexType], ets: VEC [EdgeType], tos: VEC [VertexType],
                    newName: String): PGraph =
    def join (g2: PGraph, vt1: VertexType, vt2: VertexType,
              newName: String): PGraph =
```

As Cypher and Gremlin are the two major query languages for graph databases, graph algebras have been developed for each: Cypher [79, 120, 185], Gremlin [193, 192]. Additional information on graph algebra may be found in the text on *Graph Data Warehousing* [53].

B.3.5 Query Processing in Graph Databases

With a larger and more complex graph algebra as well as excellent potential for high-performance query processing, query processing in graph databases is challenging [150].

- Query Processing and Optimization in Graph Databases [64]
- Demystifying Graph Databases [16]

B.4 Special Types of Graph Databases

Graph databases attempt to strike the right balance between (i) rich and flexible database models, (ii) efficient query processing, and (iii) reduction in complexity. For example, compared to relational databases, graph databases trade off (iii) for gains in (i) and (ii). Different organizational structures for graph databases will be positioned differently among these three competing goals.

- RDF, RDFS, RDF-Graph [51]
- Query Language: SPARQL 1.1 `https://www.w3.org/TR/sparql11-query`, [6]
- Ontologies, Description Logics [100], and OWL 2 EL [99, 126]

B.4.1 Embedding Relationships in Vertex-Types

The implicit relationships (foreign-key, primary-key) pairs in relational databases only support many-to-one relationships (i.e., the foreign-key references a single primary-key). In order to support many-to-many relationships, relational databases require such relationships to be translated in a new association relation that includes two foreign keys. This can be viewed a disadvantage of having a database model that is too simple. On the other hand, is allows tables to be connected using a single join. If all relationships were stored in a separate table, two joins would be required. This is analogous to what is occuring with `EdgeTypes`. It is possible the replace `EdgeTypes` by adding child references to `VertexTypes`.

The `database.graph_relation` package in ScalaTion adds a `Map` that takes edge names and maps them to another vertex. As in ScalaTion's `database.graph` package, a vertex has a property `Map` called `prop` that maps property names to property values, e.g., `"name" -> "Sue"`. In `graph_relation`, the `Vertex` class also has a edge `Map` called `edge` that maps edge names to other vertices, e.g., `"taughtBy" -> professor.verts(1)`.

```
    @param prop   maps vertex property names to property values
    @param edge   maps vertex edge names to other vertices

    case class Vertex (prop: Property, edge: Reference = emptyRef)
         extends Serializable:
```

The `VertexType` class has a name, a schema for its properties, and edge schema for its edge references, as well the vertices contained in it.

```
    @param name      the name of this vertex-type
    @param schema    the property names for this vertex-type
    @param eschema   the edge names for this vertex-type
    @param verts     the set of vertices having this vertex-type (extension)

    case class VertexType (name: String, schema: VEC [String], eschema: VEC [String],
                           verts: VEC [Vertex])
         extends Flaw ("VertexType") with Serializable:
```

A `Course` vertex-type may have a reference `TaughtBy` to a `Professor` vertex-type. At the instance level this would indicates that for example, the Database course is taught by professor 2.

```
    val Professor = VertexType ("professor",
                    VEC ("pid", "name", "phone"),
                    VEC (),
                    VEC (Vertex (Map ("pid" -> 1, "name" -> "Bob", "phone" -> 1234567)),
                         Vertex (Map ("pid" -> 2, "name" -> "Sue", "phone" -> 2345678)),
                         Vertex (Map ("pid" -> 3, "name" -> "Joe", "phone" -> 3456789))))

val Course = VertexType ("course",
                 VEC ("cid", "cname", "dept"),
                 VEC ("taughtBy"),
                 VEC (Vertex (Map ("cid" -> 1, "cname" -> "database", "dept" -> "CSCI"),
                              Map ("taughtBy" -> professor.verts(0))),
                      Vertex (Map ("cid" -> 2, "cname" -> "networks", "dept" -> "CSCI"),
                              Map ("taughtBy" -> professor.verts(1))),
                      Vertex (Map ("cid" -> 3, "cname" -> "ai",       "dept" -> "ARTI"),
                              Map ("taughtBy" -> professor.verts(2)))))
```

The `database.graph_relation` package is similar to a traditional relational model, except that foreign key values are no longer stored in a column of a table whose correspondence with the primary key value facilitates a join. Rather a direct reference to the vertex containing that primary key value is used. The trade-off is a slight increase in the complexity of the database model, for the potential of faster joins. See the exercises for a comparison of property graphs, graph relations and relations.

B.4.2 Resource Description Framework (RDF) Graphs

Various restrictions/simplifications can be added to the above organizational structure to make query processing more efficient. For example, the edge properties may be replaced with an edge label.

Similar to a graph database, an RDF store holds a collection of triples of the form (subject, predicate, object) or (s, p, o). A predicate is also referred to as an RDF property. With certain restrictions on RDF triples, an RDF Graph may be built upon the concept of a Vertex-Edge-Labeled Directed Multi-Graph. RDF Graphs may be viewed as a refinement as follows: The vertices are partitioned into entity nodes and literal nodes. An entity node's label is an International Resource Identifier (IRI), while a literal node's label is a primitive value (e.g., number or string). An edge represent a triple of the form (subject, predicate, object) or (s, p, o). The subject is the from-vertex, the predicate may be viewed as an edge label, and the object is the to-vertex (see `https://www.w3.org/TR/rdf11-primer/`).

The `RDFTriple` class holds information about a triple (3 part edge). It may be viewed as a statement with a subject, predicate and object.

```
@param s  the head vertex (subject = International Resourse Indentifier (IRI))
@param p  the relation/edge-label (predicate)
@param o  the tail vertex (object = IRI or literal value)

case class RDFTriple (s: String, p: ValueType, o: ValueType)
```

Resource Description Framework Schema (RDFS) Graphs

On the flip side, various extensions can be added to the above organizational structure. For example, type hierarchies can be imposed on vertex-types as well as edge-types.

The IRIs are identifiers for (Web) resources and the Resource Description Framework Schema (RDFS) can by used to assign types to resources, dividing them into groups called classes. The `rdf:type` predicate may be used to assign a type to a resource, e.g.,.

```
resourse1 rdf:type Class1
```

states that `resource1` is an instance of `Class1` an `rdfs:Class`. Subclasses may be defined as well (see `https://www.w3.org/TR/rdf-schema/`).

```
    Class2 rdfs:subClassOf Class1
```

Similarly, properties can be defined using `rdf:Property` and subtypes defined using `rdfs:subPropertyOf`

A convenient way to define an RDF/RDFS dataset is via the *Turtle* language (see `https://www.w3.org/TR/2014/REC-turtle-20140225/`). In addition, JSON-LD provides an alternative convenient syntax (see `https://www.w3.org/TR/json-ld/`).

To handle certain use cases in RDF and RDFS, more complicated graph forms are needed, including bipartite graphs, hypergraphs and Labeled Directed Multigraph with Triple Nodes (LDM-3N) [134]. There are also efforts underway to provide interoperability between RDF and Property Graphs [8].

The Shapes Constraint Language (SHACL) provides constraints for RDF Graph.

SPARQL Query Language

The SPARQL query language makes it easy to express queries in that the `WHERE` clause consists of constraints in the form of triple patterns that are similar to triple statements (see `https://www.w3.org/TR/sparql11-query/`).

```
    PREFIX university: <http://.../univerity>
    SELECT ?p
    WHERE { ?c rdf:type university:Course .
            ?c university:taughtBy ?p . }
```

The first triple pattern constrains the variable `?c` to be a university course, while the second one constrains the `?p` to be someone who teaches a course at the university. The `SELECT` clause indicates what to to return as the answer to query, while the `PREFIX` indicates the data source and its namespace.

B.4.3 From Relational to Graph Databases

ScalaTion supports data models spanning the space from traditional relational data models to graph data models, including `Table`, `LTable`, `VTable`, `GTable`, and finally `PGraph`. Table B.6 shows ten distinct types of data models.

Table B.6: Types of Data Models (Relational (R), Graph-Relational (GR), Graph (G)

Type	**Features**	**Example**
R	Nested Loop Join	Oracle
R	Referential Integrity, Join Using Primary Index	Oracle 7
R	NU Index on Foreign Key, Efficient Joins in Both Directions	`Table`
R	Joins via Index-Free Adjacency (Many-to-One)	`LTable`
GR	Information-Bearing Adjacency (May Drop Foreign Keys)	CODASYL
GR	Index-Free Adjacency (Many-to-Many), Tuple → Vertex	`VTable`
G	Adjacency via Edges, Edges as First Class Citizens	-
G	Attributes for Edges	`GTable`
G	Replace Attributes with Property Map	`PGraph`
G	Replace Property Map with Edges	RDF Graph

B.5 Knowledge Graphs

Some view knowledge graphs as RDF/RDFS graphs and their extensions such as using the Web Ontology Language (OWL) to specify type hierarchies and constraints. Another viewpoint is that they are graph databases, primarily property graphs, used to store knowledge in addition to data.

A more common view is that knowledge graphs go beyond the structural issues related to basing them on RDF graphs versus property graphs, by including the following deeper issues [77, 78]:

1. the type of logic used to express constraints and make inferences,
2. methodologies for adding vertices/nodes and edges/relationships, and
3. their synergy with machine learning.

For example, various sub-languages for OWL support particular forms of *description logic*. Description logics are especially helpful in deducing if one class/concept *subsumes* another class/concept, purely from the logical specification (e.g., `Animal` subsumes `Mammal`).

Also see https://web.stanford.edu/ vinayc/kg/notes/KG_Notes_v1.pdf

One could with a graph database based on Property Graph and add some following features/capabilities discussed in the subsections below.

B.5.1 Type Hierarchies

From an object-oriented point of view, each table or vertex-type defines a class. Indeed, object-relational database systems allow the creation of *table data types* using a `CREATE TYPE` statement. In ScalaTion these are defined dynamically and these is no type hierarchy specified.

On the other hand, RDFS defines a type hierarchy as follows:

Again, object-relational database systems support this via the `UNDER` clause. For example, the student-course-professor database may be defined as follows (for example using PostgreSQL):

```
    CREATE TYPE personType AS (
        id     INTEGER,
        name   VARCHAR(30),
        street VARCHAR(30),
        city   VARCHAR(30))

    CREATE TYPE studenType AS (
        dept   VARCHAR(30),
        level  INTEGER)
        UNDER personType

    CREATE TYPE professorType AS (
        dept   VARCHAR(30))
        UNDER personType

    CREATE TYPE courseType AS (
        cid    INTEGER,
        cname  VARCHAR(30),
        hours  INTEGER,
        dept   VARCHAR(30),
        pid    INTEGER)

    CREATE TYPE takesType AS (
        id      INTEGER,
        cid     INTEGER)
```

RDFS also allows type hierarchies to be created among relationships (edge-types). For example, a relationship (edge-type) `mother-of` could could be defined to be a subtype of `parent-of`. Then any query looking for parents of a `Person` would follow the `parent-of` relationship and any subtype relationships.

B.5.2 Constraints and Rules

In addition to the structural requirements for membership in a table/vertex-type, a vertex may need to satisfy constraints, e.g., a `Voter` type may extend `Person` and add a constraint that the voter must be at least 18 years of age. The challenging issue with constraints is that they make it hard to know is one type subsumes another. For example, if type `Employee` had a constraint the age must be 18 to 70, then `Voter` subsumes `Employee` in the sense that every employee can be a voter, but not vice versa.

For efficiency/decidability constraint languages are designed to have restricted expressive power. Constraints can be written in a description logic. For example, the Web Ontology Language, Version 2 (OWL 2) includes OWL 2 DL (Description Logic), OWL 2 EL (Existential Language), OWL 2 QL (Query Language), and OWL 2 RL (Rule Language) profiles that trade-off expressiveness for efficiency in various ways to meet particular needs [65, 100].

In the OWL 2 EL description logic, the schema defined above for PostgreSQL may be defined as follows:

$$\begin{aligned}
Person &\sqsubseteq \exists hasId.\top \sqcap \exists hasName.\top \sqcap hasStreet.\top \sqcap \exists hasCity.\top \\
Student &\sqsubseteq Person \sqcap \exists hasDept.\top \sqcap hasLevel.\top \\
Professor &\sqsubseteq Person \sqcap \exists hasDept.\top \\
Course &\sqsubseteq \exists hasCid.\top \sqcap \exists hasName.\top \sqcap \exists hasHours.\top \sqcap \exists hasDept.\top \\
Student &\sqsubseteq \exists takes.Course \\
Professor &\sqsubseteq \exists teaches.Course
\end{aligned}$$

It consists of five concepts/classes (*Person, Student, Professor, Course,* $\top$), where $\top$ is the top concept (anything) and nine roles/binary relationships (*hasId, hasName, hasStreet, hasCity, hasDept, hasLevel, hasCid, takes, teaches*). The $\sqcap$ symbol indicates *concept intersection/conjunction*, thus specifying the *Person* concept has all four roles/properties listed in its definition. The $\exists$ symbol is an *existential restriction.*

To permit inferencing in polynomial time, OWL EL is limited to (1) negation ($\neg C$), (2) conjunction ($C \sqcap D$), (3) disjunction ($C \sqcup D$), (4) existential restriction ($\exists R.C$), (5) concept inclusion ($C \sqsubseteq D$), (6) role inclusion ($R \sqsubseteq S$), and (7) role chain ($R_1 \circ R_2 \sqsubseteq R$).

Concept inclusion $C \sqsubseteq D$ is the flip side of concept subsumption $D \sqsupseteq C$. When $C \sqsubseteq D$, if $c \in C^I$, then $c \in D^I$. Note, C is a concept, while C^I is the application of the concept to a knowledge base (or in general any interpretation) and indicates the set of individuals classified under C, asserted or inferred.

Instances/individuals may be associated with concepts and roles as follows: (1) concept assertion $C(a)$, e.g., $Student(peter)$, $Course(database)$ and (2) role assertion $R(a, b)$, e.g., $takes(peter, database)$.

In addition to the top concept ($\top$) containing all individuals, there is also a bottom concept ($\bot$) that can never have individuals.

See [99, 126] for the complete definitions for OWL 2 EL.

A *reasoner* can be applied to make *inferences*, such as *concept inclusion* ($\sqsubseteq$), concept $C \sqsubseteq D$, meaning that concept D is more general than concept C, e.g., $Student \sqsubseteq Person$.

Related to constraints are *rules*. While inferencing with constraints is used to determine whether something (e.g., *Person* subsumes *Student*) is true, inferencing with rules produces new facts. Consider the following rule written in Datalog [62]

$$grandparent(x, z) :- parent(x, y) \ \& \ parent(y, z).$$

It produces new information that is not stored in the database or knowledge base. Although this could be easily accomplished using relational algebra on a table $parent(x, y)$,

$$\pi_{xz}(parent \bowtie_{y=p2.x} parent)$$

the following rules (with the second one being recursive) are beyond the capabilities of standard relational algebra.

$$ancester(x, y) :- parent(x, y).$$
$$ancester(x, z) :- parent(x, y) \ \& \ ancester(y, z).$$

Other implementations of rules used for knowledge graphs or more knowledge bases include the Semantic Web Rule Language (SWRL) and the Rule Interchange Format (RIF) [152].

For a discussion of future direction of research on knowledge graphs, see [76].

B.5.3 KGTable

The `KGTable` class in ScalaTion allows a knowledge-graph-table (or simply kg-table) to specify a parent or supertype.

```
@param name_     the name of the graph-table
@param schema_   the attributes for the graph-table
@param domain_   the domains/data-types for attributes ('D', 'I', 'L', 'S', 'X', 'T')
@param key_      the attributes forming the primary key
@param parent    the parent (super-type) table

class KGTable (name_ : String, schema_ : Schema, domain_ : Domain, key_ : Schema,
               val parent: KGTable = null)
      extends GTable (name_,
                      if parent == null then schema_ else parent.schema ++ schema_,
                      if parent == null then domain_ else parent.domain ++ domain_,
                      if parent == null then key_    else parent.key)
         with Serializable:
```

The schema defined above for PostgreSQL may be specified in ScalaTion as follows:

```
val person    = KGTable ("person",    "id, name, street, city",
                                      "I, S, S, S", "id")
val student   = KGTable ("student",   "dept, level",
                                      "S, I", null, person)
val professor = KGTable ("professor", "dept",
                                      "S", null, person)
val course    = KGTable ("course",    "cid, cname, hours, dept",
                                      "I, X, I, S", "cid")

student.addEdgeType ("cid", course, false)                    // student has M courses
course.addEdgeType ("id", student, false)                     // course has M students
course.addEdgeType ("pid", professor)                         // course has 1 professor
```

In a manner similar to the GTable graph database, a KGTable knowledge graph may be populated as follows:

```
    val v_Joe    = person.addV (91, "Joe", "Birch St", "Athens")
    val v_Sue    = person.addV (92, "Sue", "Ceder St", "Athens")

    val v_Peter  = student.addV (101, "Peter", "Oak St",   "Bogart",       "CS", 3)
    val v_Paul   = student.addV (102, "Paul",  "Elm St",   "Watkinsville", "CE", 4)
    val v_Mary   = student.addV (103, "Mary",  "Maple St", "Athens",       "CS", 4)

    val v_DrBill = professor.addV (104, "DrBill", "Plum St", "Athens",       "CS")
    val v_DrJohn = professor.addV (105, "DrJohn", "Pine St", "Watkinsville", "CE")

    val v_Database     = course.addV (4370, "Database Management", 4, "CS")
    val v_Architecture = course.addV (4720, "Comp. Architecture",  4, "CE")
    val v_Networks     = course.addV (4760, "Computer Networks",   4, "CS")

    student.add2E ("cid", Edge (v_Peter, v_Database),     "id", course)
           .add2E ("cid", Edge (v_Peter, v_Architecture), "id", course)
           .add2E ("cid", Edge (v_Paul, v_Database),      "id", course)
           .add2E ("cid", Edge (v_Paul, v_Networks),      "id", course)
           .add2E ("cid", Edge (v_Mary, v_Networks),      "id", course)

    course.addE ("pid", Edge (v_Database,     v_DrBill))
          .addE ("pid", Edge (v_Architecture, v_DrBill))
          .addE ("pid", Edge (v_Networks,     v_DrJohn))
```

The all operator will retrieve vertices from the complete hierrachy that starts with the given class. The first show will display Joe and Sue, while the second will display all people.

```
    person.show ()
    person.all().show ()
```

B.6 Exercises - Part I

1. Present a Group Lecture Series on **High Level Query Languages for Graph Databases**.
2. Present a Group Lecture Series on **Graph Algebra**.
3. Present a Group Lecture Series on **Query Processing in Graph Database**.
4. Present a Group Lecture Series on **Special Types of Graph Databases**.
5. Use Scala 3 to complete the implementation of `VertexType` in the `scalation.database.graph` package. It should provide operators like those in the Neo4j graph algebra.
6. Use Scala 3 to complete the implementation of `EdgeType` and `PGraph` in the `scalation.database.graph` package. Again, it should provide operators like those the Neo4j graph algebra. The exiting operators in these classes may need to renamed and/or modified.
7. Using your implementation for `PGraph` translate the road sensor schema.
8. Create the same schema using Neo4j.
9. Populate the database with sample data.
10. Using Neo4j's Cypher Query language, retrieve the sensors that are on I35.
11. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).
12. **Project PG1 and PG2**.

    PG1: Finish and test the implementations of the ScalaTion data models: `Table`, `LTable`, `VTable`, `GTable`, and `PGraph`. See Exercise 1 in Chapter 4, section 5 for details. The first four are in the `scalation.database.table` package, while the last is in the `scalation.database.graph` package. Each group picks one to work on.

    PG2: Compare the complexity and performance of the following database systems:

    - ScalaTion data models: `Table`, `LTable`, `VTable`, `GTable`, and `PGraph`. Each group picks one to work on.
    - Relations - using PostgreSQL/MySQL
    - Properties Graphs - using Neo4j

    Plot the performance of various types of queries executed with each of the four database systems. For each plot, the x-axis will be the number of tuples/vertices, while the y-axis will be the time in milliseconds (using `nanoTime`) to execute the query. ScalaTion allows its conveinent use as follows:

```
    def time [R] (block: => R): R =
        val t0 = nanoTime ()
        val result = block                                  // call-by-name
        val t1 = nanoTime ()
        println ("Elapsed time: " + (t1 - t0) * NS_PER_MS + " ms")
        result
    end time
```

Be sure to skip the timing results for the first iteration and average the next five to get reliable timing results. Note, the JIT compiler is used during the first iteration, so it tends to be slow. The performance evaluation will require a large number of tuples/vertices, so a tuple/vertex generator will be needed. Make sure to test path queries common in social networking applications.

B.7 Graph Data Science

Graph Analytics or Graph Data Science is emerging as an important area. It has two sides: first, how can data science/machine learning be used to build graph databases or knowledge graphs, second how can these be used to enhance data science/machine learning. Enhancement includes making existing predictive models better as well as supporting new types of analysis.

The first place to starts is to see how graph algorithms are used in graph data science. Graph database providers such as Neo4j and TigerGraph supply libraries to support graph data science [130, 94, 153, 149].

B.7.1 Path Finding

There are many types of path finding problems, including single-source shortest path, all-pairs shortest path, short-path from vertex u to w, through vertex v, vertex search, including Depth-First Search (DFS), Breath-First Search (BFS) and A^*.

B.7.2 Centrality and Importance

The importance of vertices varies depending on their connections to other vertices. For example, a vertex that is connected to only one other vertex would not viewed as central, a vertex with long paths in all directions would be considered as central. Also, a vertex with high in-degree would likely be considered to be important. A *page rank* algorithms may be used to score the importance of vertices.

B.7.3 Community Detection

The basic idea of community in a graph is that vertices in a group will have more connections within the group than outside.

B.8 Graph Pattern Matching

Graph Pattern Matching [19] captures the idea that one can specify a query as a (typically small) graph or more generally a graph pattern and then find all subgraphs in the graph database that match it. If one is interested in subgraphs that match in terms of labels/properties and topology, then the graph matching problem is called *subgraph isomorphism*. Various forms of *graph simulation* may also be used, including simple, dual, strong, strict and tight simulation. The graph simulation algorithms return all the subgraph isomorphism matches plus some that while they fail the subgraph isomorphism test, they may be of interest to the user; in addition, these algorithms are much faster.

Another direction that also relaxes subgraph isomorphism is *graph homomorphism* [95, 184].

B.8.1 Graph Simulation

Graph simulation establishes a mapping of each vertex in the query graph Q to a set of vertices in the data graph G. Initially the mapping is based on matching vertex labels.

$$\phi(u) = \{v \in G.V : l_v(v) = l_v(u)\} \tag{B.1}$$

The **GraphSim** class implements the pruning method defined for graph simulation.

```
    @param g  the data graph  G(V, E, l)
    @param q  the query graph Q(U, D, k)

    class GraphSim (g: Graph, q: Graph)
          extends GraphMatcher (g, q):
```

Pruning is used to remove cases where vertex $v \in \phi(u)$ does not have children with vertex labels matching those of vertex u.

$$\phi(u) -= \{v : l_v(ch(u)) \not\subseteq l_v(ch(v))\} \tag{B.2}$$

```
def prune0 (ϕ: Array [SET [Int]]): Array [SET [Int]] =
    var (rem, alter) = (SET [Int] (), true)                 // vertices to be removed
    breakable {
        while alter do                                      // check for matching children
            alter = false                                   // no vertices removed yet

            for u <- qRange; u_c <- q.ch(u) do              // for u in q; its children u_c
                debug ("prune0", s"for u = $u, u_c = $u_c")
                val ϕ_u_c = ϕ(u_c)                          // u_c mapped to vertices in g

                for v <- ϕ(u) do                            // for each v in g image of u
                    if (g.ch(v) & ϕ_u_c).isEmpty then       // v must have a child in ϕ(u_c)
                        rem += v; alter = true              // match failed => add v to rem
                    end if
                end for
                if remove (ϕ, u, rem) then break ()         // remove vertices from ϕ(u)
            end for
        end while
    } // breakable
    ϕ
end prune0
```

The `prune` method also checks whether the edge labels match. (see `sclation.database.graph_pm.GraphSim`.

Dual Simulation

Strict Simulation

Tight Simulation

B.8.2 Subgraph Isomorphism

B.8.3 Graph Homomorphism

B.8.4 Application to Query Processing in Graph Databases

B.9 Graph Representation Learning

Large graphs are difficult to analyze as a whole. One approach is to convert a graph into a matrix and then use spectral theory (eigenvalues/singular values) to make a lower dimension approximation of the matrix (e.g., for an Adjacency Matrix, Laplacian Matrix, or Normalized Laplacian Matrix). Graph Representation Learning [207] ...

B.9.1 Matrix Representations

Weighted Adjacency Matrix

Given a Directed Multi-Graph $G(V, E, L_e, l, L_v)$, suppose an edge-label (or more generally an edge property) can be interpreted as a weight (e.g., inverse of distance). The weight can be interpreted as vertex proximity or similarity. Between any two vertices (u, v) there can multiple edges and edge labels.

$$\{(u, l_e, v) \in E\} \tag{B.3}$$

These can be aggregated, (e.g., using count, min, max, mean, median) to create a single weight. This weight can be use to create a weighted adjacency non-negative matrix, $A = [a_{ij}]$.

$$\begin{aligned} a_{ij} &= agg(\{l_e\} : (u, l_e, v) \in E \text{ and } u.id = i \text{ and } v.id = j) && \text{(B.4)} \\ &= 0 \qquad\qquad \text{if there no edge from } u \text{ to } v && \text{(B.5)} \end{aligned}$$

Count could be the number of edges from vertex u to v. For example, suppose the vertices represent cities and the edges are connecting one way streets with an edge label/property giving the distance, then *agg* function can be min for the minimum distance between the cities.

Graph Laplacian

Although the Graph Laplacian is usually applied in the context of undirected graph, it can be defined for directed graphs and directed multi-graphs.

$$L = D - A \tag{B.6}$$

where A is the weighted adjacency matrix and D is the out-degree (alternatively the in-degree) diagonal matrix. For directed graphs, L may not be a symmetric matrix. Consequently, a spectral decomposition giving the eigenvalues of the matrix may include complex numbers. In such cases, one could work with singular values [26]. A simpler alternative would be to work with the undirected graph underlying the directed graph, where L is a symmetric matrix and the eigenvalues are real numbers. Having the spectrum of a matrix allows overall characteristics of graphs to be study and graphs compared. For example, if the spectra of two graphs are different, then they cannot be isomorphic [26]. In addition, small eigenvalues may be ignored to obtain a lower dimensional reprentation of the matrix (and consequently the graph).

Normalized Graph Laplacian

The Normalized Graph Laplacian is

$$L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}} \tag{B.7}$$

B.9.2 Graph Embeddings

As many data science/machine learning modeling techniques take vectors and matrices as input, they need to be created from the information content of knowledge graphs. For example, a state in the United States may have demographic and policy information, that can be useful information to be fed into a model. In addition, similiar information about neighboring states or for example mobility data between the states can useful as well. Capturing all this information may lead to a high dimensionality vector that is not ideal for modeling. The goal of this area of research is to embed the information in a lower dimensionality vector, while not losing important structural information [210]. Also see `https://neo4j.com/developer/graph-data-science/graph-embeddings/`.

A vertex embedding function $\phi : V \to \mathbb{R}^d$ maps each vertex to d-dimensional vector. This can be done in numerous ways, but the idea is make it so the dot product of two vectors approximates a similarity function *sim* applied to the vertices, i.e.,

$$sim(u, v) \;=\; \phi(u) \cdot \phi(v) \tag{B.8}$$

FastRP

DeepWalk

A *random walk* in a directed graph will involve moving from vertex to vertex along edges. The selection of the next vertex is governed by the transition matrix P. This may also be viewed as moving from state-to-state in a Discrete-Time Markov Chain $\{x_t | t = 0, 1, ...\}$. The walk will start at a particular vertex (or state) and randomly progress from there.

Given a weighted adjacency matrix A, the transition probability for a standard random walk is given by the following conditional probabilty [145, 83].

$$p(x_{t+1} = u | x_t = v) \;=\; \frac{a_{uv}}{d_u} \tag{B.9}$$

The transition matrix is then $P = D^{-1}A$. As indicated in the section on Markov Chains, the long-term, steady-state solution may be found by solving to $\boldsymbol{\pi}$ in

$$\boldsymbol{\pi} \;=\; \boldsymbol{\pi} P \tag{B.10}$$

Node2Vec

GraphSAGE

B.10 Graph Neural Networks

The convolutional filters used in CNNs allow one to extract hidden features by looking at a collection of nearby points. The notion of nearby may be thought of in terms of Euclidean distance.

Graphs can be used to provide a more versatile notion of what points are nearby. This may be done by counting hops, the number of edges traversed to get form one vertex to another. If distance is an edge property (given or derived) then nearby vertices may be defined as those within a certain distance, as shown in Figure B.6.

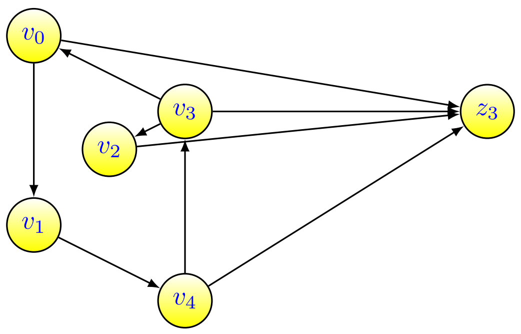


Figure B.6: Neighborhood of vertex v_3 aggregated and processed at hidden node z_3

There are many types of Graph Neural Networks (GNNs) [216, 206] ... A GNN may be defined for several types of graphs.

Consider the following slightly modified definition of a vertex-labeled directed graph as a four-tuple $G(V, E, d, x)$ where

- V = set of vertices
- $E \subseteq V \times V$ set of directed edges
- d = dimensionality of the property vectors
- $\mathbf{x} : V \to \mathbb{R}^d$

Hence, $\mathbf{x}_v$ denotes a property/attribute vector associated with vertex v. In general, a GNN could also use values on edges. The following definitions are needed for understanding how GNNs work [209].

Neighborhood

The neighborhood of vertex $v \in V$ is given as follows for a directed graph.

$$\mathcal{N}(v) \ = \ \{u | (v, u) \in E\} \cup \{u | (u, v) \in E\} \tag{B.11}$$

The neighborhood of vertex v_4 is $\{v_1, v_3\}$, where v_1 is an upstream vertex and v_3 is a downstream vertex.

B.10.1 AGGREGATE and COMBINE Operations

As with CNNs, the initial processing of data occurs with non-neural layers. For each layer l, data from a vertex is mixed with its neighbors' data using the AGGREGATE and COMBINE operations [68, 129]

AGGREGATE

The values for the neighborhood of vertex v are aggregated to form a vector of values representing the neighborhood.

$$\mathbf{x}_{\mathcal{N}(v)} \ = \ \text{AGGREGATE}\,\{\mathbf{x}_u^{(l-1)} | u \in \mathcal{N}(v)\} \tag{B.12}$$

The AGGREGATE operation may be as simple as element-wise mean or max-pooling.

COMBINE

A new value for vertex v, $\mathbf{x}_v^{(l)}$ is calculated by combining the previous vector value for vertex v, $\mathbf{x}_v^{(l-1)}$, and the vector value produced from v's neighborhood.

$$\mathbf{x}_v^{(l)} \ = \ \text{COMBINE}(\mathbf{x}_v^{(l-1)},\ \mathbf{x}_{\mathcal{N}(v)}) \tag{B.13}$$

The COMBINE operation may be a simple as element-wise mean or vector addition (+).

Example from Traffic Domain

As an example, consider applying a Graph Neural Network to vehicle traffic flow on a highway system. Let the vertices in the graph represent sensors. At a particular time t, the sensor records the flow (vehicles in last 5 minutes) and speed (in mph). The neighborhood may capture the notion of upstream flow past one sensor and downstream flow past another sensor. Suppose the initial property/attribute vectors are the following:

Table B.7: Example from Traffic Domain

v	flow	speed
v_4	99	66
v_1	90	68
v_3	80	72

Applying an element-wise mean AGGREGATE operation gives

$$\mathbf{x}_{\mathcal{N}(v_4)} \ = \ \text{AGGREGATE}\,\{\mathbf{x}_u^{(0)} | u \in \{1, 3\}\} \ = \ [85, 70]$$

Applying an element-wise mean COMBINE operation gives

$$\mathbf{x}_v^{(1)} \ = \ \text{COMBINE}(\mathbf{x}_v^{(0)}, \mathbf{x}_{\mathcal{N}(v)}) \ = \ [93, 67]$$

Typically, the COMBINE operation includes a learnable weight/parameter matrix B and an activation function f (e.g., reLU).

$$\mathbf{x}_v^{(l)} \ = \ \mathbf{f}(B[\mathbf{x}_v^{(0)}, \mathbf{x}_{\mathcal{N}(v)}]) \tag{B.14}$$

$\mathbf{f}$ is the vectorization of activation function f. The weights/parameters in the GNN indicate the importance of data from upstream and downstream sensors in, for example, classifying, predicting or forecasting results at a particular sensor. Pooling can also be used to reduce the size of the vectors.

ScalaTion WILL supports `GraphNeuralNet`s of the spatial variety, so those of the spectral variety are not considered (see [206] for a comparison).

B.11 Exercises - Part II

1. Pressent a Group Lecture Series on **Graph Data Science**.
2. Pressent a Group Lecture Series on **Graph Pattern Matching**.
3. Pressent a Group Lecture Series on **Graph Representation Learning**.
4. Pressent a Group Lecture Series on **Graph Neural Networks**.

Bibliography

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[2] Charu C Aggarwa et al. *Data Classification: Algorithms and Applications.* Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2015.

[3] Shahrokh Akhlaghi, Ning Zhou, and Zhenyu Huang. Adaptive adjustment of noise covariance in kalman filter for dynamic state estimation. In *2017 IEEE power & energy society general meeting*, pages 1–5. IEEE, 2017.

[4] Awad H Al-Mohy and Nicholas J Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM Journal on Matrix Analysis and Applications*, 31(3):970–989, 2010.

[5] Christos Alexopoulos, Andrew F Seila, and J Banks. Output data analysis. In *Handbook of Simulation*, number 7. John Wiley & Sons, 1998.

[6] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of rdf stores & sparql engines for querying knowledge graphs. *The VLDB Journal*, pages 1–26, 2021.

[7] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75:245–248, 2009.

[8] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Rdf and property graphs interoperability: Status and issues. In *AMW*, 2019.

[9] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Symposium on Computational Geometry*, volume 6, pages 1–10, 2006.

[10] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[11] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017.

[12] Jerry Banks, John Carson, Barry Nelson, and David Nicol. *Discrete event system simulation, 5th Edition.* Pearson, 2010.

[13] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[14] Anil K Bera and Yannis Bilias. The MM, ME, ML, EL, EF and GMM approaches to estimation: a synthesis. *Journal of Econometrics*, 107(1-2):51–86, 2002.

[15] Joseph Berkson. Minimum chi-square, not maximum likelihood! *The Annals of Statistics*, 8(3):457–487, 1980.

[16] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[17] Concha Bielza and Pedro Larrañaga. Discrete Bayesian Network Classifiers: A Survey. *ACM Computing Surveys (CSUR)*, 47(1):5, 2014.

[18] Nicholas H Bingham and John M Fry. *Regression: Linear models in Statistics*. Springer Science & Business Media, 2010.

[19] Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys (CSUR)*, 54(2):1–35, 2021.

[20] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[21] Stephan Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018.

[22] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[23] Charles G Broyden. The convergence of a class of double-rank minimization algorithms: 2. the new algorithm. *IMA journal of applied mathematics*, 6(3):222–231, 1970.

[24] Yalcin Bulut, D Vines-Cavanaugh, and Dionisio Bernal. Process and measurement noise estimation for kalman filtering. In *Structural Dynamics, Volume 3*, pages 375–386. Springer, 2011.

[25] Brett Burglund and Ryan Street. Golf ball flight dynamics. *Flathead Valley*, 2011.

[26] Steven Kay Butler. *Eigenvalues and structures of graphs*. University of California, San Diego, 2008.

[27] José M Carcione, Juan E Santos, Claudio Bagaini, and Jing Ba. A simulation of a covid-19 epidemic based on a deterministic seir model. *Frontiers in public health*, 8:230, 2020.

[28] David Maxwell Chickering. Learning bayesian networks is np-complete. In *Learning from data*, pages 121–130. Springer, 1996.

[29] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[30] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[31] Zafer Cömert and Adnan Fatih Kocamaz. A study of artificial neural network training algorithms for classification of cardiotocography signals. *J Sci Technol*, 7(2):93–103, 2017.

[32] Pierre Comon. Tensors: A brief introduction. *IEEE Signal Processing Magazine*, 31(3):44–53, 2014.

[33] Jerome Connor, Les E Atlas, and Douglas R Martin. Recurrent networks and narma modeling. In *Advances in neural information processing systems*, pages 301–308, 1992.

[34] Denis Cousineau and Sylvain Chartier. Outliers detection and treatment: A review. *International Journal of Psychological Research*, 3(1):58–67, 2010.

[35] Thomas M. Cover and Joy A. Thomas. Entropy, Relative Entropy and Mutual Information. Technical report, Columbia University, 1991.

[36] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[37] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[38] Holger Dette and Weichi Wu. Prediction in locally stationary time series. *Journal of Business & Economic Statistics*, 40(1):370–381, 2022.

[39] LEE DO Q. Numerically efficient methods for solving least squares problems, 2012.

[40] Justin Domke. Statistical Machine Learning Notes: Trees. Technical report, Uinversity of Massachusetts, 2018.

[41] Rick Durrett. *Probability: theory and examples*, volume 49. Cambridge university press, 2019.

[42] A Ehiagwina. *Application of Mixed Simulation Method to Modelling Port Traffic*. PhD thesis, Liverpool John Moores University, 2021.

[43] Joseph G Eisenhauer. Regression through the origin. *Teaching statistics*, 25(3):76–80, 2003.

[44] Steven Elsworth and Stefan Güttel. Time series forecasting using LSTM networks: A symbolic approach. *arXiv preprint arXiv:2003.05672*, 2020.

[45] Leonhard Euler. *Mechanica Sive Motus Scientia Analytice Exposita: Instar Supplementi Ad Commentar. Acad. Scient. Imper*, volume 2. Ex typographia academiae scientiarum, 1736.

[46] Geir Evensen. The ensemble kalman filter: Theoretical formulation and practical implementation. *Ocean dynamics*, 53(4):343–367, 2003.

[47] Yuval Filmus. Two proofs of the central limit theorem. *Recuperado de http://www. cs. toronto. edu/yuvalf/CLT. pdf*, 2010.

[48] Roger Fletcher. A new approach to variable metric algorithms. *The computer journal*, 13(3):317–322, 1970.

[49] Valeria Fonti and Eduard Belitser. Feature Selection using LASSO. Technical report, VU Amsterdam, 2017.

[50] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[51] Fabien Gandon, Reto Krummenacher, Sung-Kook Han, and Ioan Toma. The resource description framework and its schema, 2011.

[52] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. In *Ninth International Conference on Artificial Neural Networks*. IET, 1999.

[53] Amine Ghrab et al. Graph data warehousing. 2020.

[54] Michael B Giles, Mihai C Duta, Jens-Dominik Muller, and Niles A Pierce. Algorithm developments for discrete adjoint methods. *AIAA journal*, 41(2):198–205, 2003.

[55] Igor Gitman, Hunter Lang, Pengchuan Zhang, and Lin Xiao. Understanding the role of momentum in stochastic gradient methods. *Advances in Neural Information Processing Systems*, 32, 2019.

[56] D Goldfarb. A family of variable metric updates derived by variational means. *Mathematics of Computing*, 24:317–322, 1970.

[57] John Goldsmith. Probability for linguists. *Mathématiques et sciences humaines. Mathematics and social sciences*, (180):73–98, 2007.

[58] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, 4th Edition*, volume 3. JHU Press, 2013.

[59] Maximilian Götzinger, Dávid Juhász, Nima Taherinejad, Edwin Willegger, Benedikt Tutzer, Pasi Liljeberg, Axel Jantsch, and Amir M Rahmani. Rosa: A framework for modeling self-awareness in cyber-physical systems. *IEEE Access*, 8:141373–141394, 2020.

[60] Clive WJ Granger and Paul Newbold. Spurious regressions in econometrics. *Journal of econometrics*, 2(2):111–120, 1974.

[61] Erin Grant and Yan Wu. Predicting generalization with degrees of freedom in neural networks. In *ICML 2022 2nd AI for Science Workshop*, 2022.

[62] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013.

[63] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[64] Andrey Gubichev. *Query processing and optimization in graph databases.* PhD thesis, Technische Universität München, 2015.

[65] Ricardo Guimarães and Ana Ozaki. Reasoning in knowledge graphs. In *International Research School in Artificial Intelligence in Bergen (AIB 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[66] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4805–4814, 2019.

[67] Ernst Hairer, Christian Lubich, Gerhard Wanner, et al. Geometric numerical integration illustrated by the stormer-verlet method. *Acta numerica*, 12(12):399–450, 2003.

[68] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[69] John A Hartigan, Manchek A Wong, et al. A k-means clustering algorithm. *Applied statistics*, 28(1):100–108, 1979.

[70] Hussein Abdulahman Hashem. *Regularized and Robust Regression Methods for High Dimensional Data*. PhD thesis, Brunel University, 2014.

[71] Trevor Hastie and Robert Tibshirani. Efficient quadratic regularization for expression arrays. *Biostatistics*, 5(3):329–340, 2004.

[72] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer-Verlag New York, 2009.

[73] Yanchen He. The application of alternating direction method of multipliers on 1l-norms problems. In *Journal of Physics: Conference Series*, volume 1187, page 042070. IOP Publishing, 2019.

[74] Korth HF and A Silberschatz. *Database Systems Concepts*. McGraw-Hill, 1986.

[75] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[76] Aidan Hogan. Knowledge graphs: Research directions. *Reasoning Web International Summer School*, pages 223–253, 2020.

[77] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, et al. Knowledge graphs. *arXiv preprint arXiv:2003.02320*, 2020.

[78] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *ACM Computing Surveys (CSUR)*, 54(4):1–37, 2021.

[79] Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2016.

[80] Douglas Holtz-Eakin, Whitney Newey, and Harvey S Rosen. Estimating vector autoregressions with panel data. *Econometrica: Journal of the econometric society*, pages 1371–1395, 1988.

[81] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[82] Pili Hu. Matrix Calculus: Derivation and Simple Application. Technical report, City University of Hong Kong, 2012.

[83] Zexi Huang, Arlei Silva, and Ambuj Singh. A broader picture of random-walk based graph embedding. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 685–695, 2021.

[84] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice.* OTexts, 2018.

[85] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer-Verlag New York, 2013.

[86] Lucas Janson, William Fithian, and Trevor Hastie. Effective degrees of freedom: a flawed metaphor. *Biometrika*, 99(1):1–8, 2012.

[87] Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[88] Jefkine Kafunah. Backpropagation in convolutional neural networks. *DeepGrid—Organic Deep Learning, Nov*, 29:1–10, 2016.

[89] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[90] Anuj Karpatne, Gowtham Atluri, James H Faghmous, Michael Steinbach, Arindam Banerjee, Auroop Ganguly, Shashi Shekhar, Nagiza Samatova, and Vipin Kumar. Theory-guided data science: A new paradigm for scientific discovery from data. *IEEE Transactions on knowledge and data engineering*, 29(10):2318–2331, 2017.

[91] Matthias Katzfuss, Jonathan R Stroud, and Christopher K Wikle. Understanding the ensemble kalman filter. *The American Statistician*, 70(4):350–357, 2016.

[92] Luke Keele and Nathan J Kelly. Dynamic models for dynamic theories: The ins and outs of lagged dependent variables. *Political analysis*, pages 186–205, 2006.

[93] S. Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Karuturi Radha Krishna Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural computation*, 13(3):637–649, 2001.

[94] Samir Khuller and Balaji Raghavachari. Basic graph algorithms. In *Algorithms and theory of computation handbook: general concepts and techniques*, pages 7–7. 2010.

[95] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming subgraph isomorphism for rdf query processing. *arXiv preprint arXiv:1506.01973*, 2015.

[96] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[97] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J Inman. 1d convolutional neural networks and applications: A survey. *arXiv preprint arXiv:1905.03554*, 2019.

[98] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[99] Markus Krötzsch. Efficient inferencing for the description logic underlying owl el. *Institut AIFB, KIT, Karlsruhe*, 2010.

[100] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *arXiv preprint arXiv:1201.4089*, 2012.

[101] Viktor Kunčak and Jad Hamza. Stainless verification system tutorial. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, pages 2–7, 2021.

[102] M. Kutner, Nachtsheim, and Neter. Introduction to nonlinear regression and neural networks. Technical report, University of Minnesota, 2016.

[103] Giovanni Lavezzi, Kidus Guye, and Marco Ciarcià. Nonlinear programming solvers for unconstrained and constrained optimization problems: a benchmark analysis. *arXiv preprint arXiv:2204.05297*, 2022.

[104] Averill M Law. How to build valid and credible simulation models. In *2008 Winter Simulation Conference*, pages 39–47. IEEE, 2008.

[105] Averill M Law. *Simulation Modeling and Analysis. Fifth.* McGraw-Hill Education, 2015.

[106] Robert J Leach, Stephanie E Forrester, AC Mears, and Jonathan R Roberts. How valid and accurate are measurements of golf impact parameters obtained using commercially available radar and stereoscopic optical launch monitors? *Measurement*, 112:125–136, 2017.

[107] Pierre L'ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[108] Hagyeong Lee and Jongwoo Song. Introduction to convolutional neural network using keras; an understanding from a statistician. *Communications for Statistical Applications and Methods*, 26(6):591–610, 2019.

[109] Jun Li. A robust stochastic method of estimating the transmission potential of 2019-ncov. *arXiv preprint arXiv:2002.03828*, 2020.

[110] Michael Y Li and James S Muldowney. Global stability for the seir model in epidemiology. *Mathematical biosciences*, 125(2):155–164, 1995.

[111] Lek-Heng Lim. Tensors and hypermatrices. *Handbook of Linear Algebra*, pages 231–260, 2013.

[112] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[113] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[114] Charles Macal and Michael North. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference 2014*, pages 6–20. IEEE, 2014.

[115] Charles M Macal and Michael J North. Agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 86–98. IEEE, 2009.

[116] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Journal of Simulation*, pages 151–162. IEEE, 2010.

[117] Penelope Maddy. How applied mathematics became pure. *The Review of Symbolic Logic*, 1(1):16–41, 2008.

[118] Franco Manessi and Alessandro Rozza. Learning combinations of activation functions. *arXiv preprint arXiv:1801.09403*, 2018.

[119] Maja Marasović, Tea Marasović, and Mladen Miloš. Robust nonlinear regression in enzyme kinetic parameters estimation. *Journal of Chemistry*, 2017, 2017.

[120] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising opencypher graph queries in relational algebra. In *European Conference on Advances in Databases and Information Systems*, pages 182–196. Springer, 2017.

[121] P McCullagh and JA Nelder. Generalized linear models., 2nd edn.(chapman and hall: London). *Standard book on generalized linear models*, 1989.

[122] John A Miller, Mohammed Aldosari, Farah Saeed, Nasid Habib Barna, Subas Rana, I Budak Arpinar, and Ninghao Liu. A survey of deep learning and foundation models for time series forecasting. *arXiv preprint arXiv:2401.13912*, 2024.

[123] John A Miller, Hao Peng, and Michael E Cotterell. Adding support for theory in open science big data. In *Big Data (BigData Congress), 2017 IEEE International Congress on*, pages 251–255. IEEE, 2017.

[124] Piotr Mirowski. Time series modeling with hidden variables and gradient-based algorithms. *Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, Ph. D. Dissertation*, 2011.

[125] R Mohammadi Farsani and Ehsan Pazouki. A transformer self-attention model for time series forecasting. *Journal of Electrical and Computer Engineering Innovations (JECEI)*, 9(1):1–10, 2020.

[126] Sutapa Mondal, Vijaya Raghava Mutharaju, and Sumit Bhatia. *Embeddings for the EL++ description logic.* PhD thesis, IIIT-Delhi, 2020.

[127] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*, volume 821. John Wiley & Sons, 2012.

[128] Steve Ataky Tsham Mpinda, Lucas Cesar Ferreira, Marcela Xavier Ribeiro, and Marilde Terezinha Prado Santos. Evaluation of graph databases performance through indexing techniques. *International Journal of Artificial Intelligence & Applications (IJAIA)*, 6(5):87–98, 2015.

[129] Nihal V Nayak. Graph neural networks - notes. Technical report, Brown Univerity, 2020.

[130] Mark Needham and Amy E Hodler. *Graph algorithms: practical examples in Apache Spark and Neo4j*. O'Reilly Media, 2019.

[131] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[132] Isaac Newton. *Philosophiae naturalis principia mathematica*, volume 2. typis A. et JM Duncan, 1833.

[133] Anh Nguyen, Khoa Pham, Dat Ngo, Thanh Ngo, and Lam Pham. An analysis of state-of-the-art activation functions for supervised deep neural network. In *2021 International Conference on System Science and Engineering (ICSSE)*, pages 215–220. IEEE, 2021.

[134] Vinh Nguyen, Jyoti Leeka, Olivier Bodenreider, and Amit Sheth. A formal graph model for rdf and its implementation. *arXiv preprint arXiv:1606.00480*, 2016.

[135] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[136] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[137] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[138] Jeremy Orloff and Jonathan Bloom. 18.05 introduction to probability and statistics. *Massachusetts Institute of Technology: MIT OpenCourseWare*, 2014.

[139] Jeremy Orloff and Jonathan Bloom. Maximum Likelihood Estimates. Technical report, MIT, 2014.

[140] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[141] Joanna Maria Papakonstantinou. *Historical development of the BFGS secant method and its characterization properties*. Rice University, 2009.

[142] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[143] Julio L Peixoto. A property of well-formulated polynomial regression models. *The American Statistician*, 44(1):26–30, 1990.

[144] Jolynn Pek, Octavia Wong, and AC Wong. Data transformations for inference with linear regression: Clarifications and recommendations. *Practical Assessment, Research, and Evaluation*, 22(1):9, 2017.

[145] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[146] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[147] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[148] John R Quinlan et al. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. World Scientific, 1992.

[149] Md Saidur Rahman et al. *Basic graph theory*, volume 9. Springer, 2017.

[150] Supriya Ramireddy. *Query processing in graph databases.* PhD thesis, University of Georgia, 2017.

[151] Suhasini Rao. A course in Time Series Analysis. Technical report, Texas A&M University, 2018.

[152] Thanyalak Rattanasawad, Kanda Runapongsa Saikaew, Marut Buranarach, and Thepchai Supnithi. A review and comparison of rule languages and rule-based inference engines for the semantic web. In *2013 International Computer Science and Engineering Conference (ICSEC)*, pages 1–6. IEEE, 2013.

[153] Santanu Saha Ray. *Graph theory with algorithms and its applications: in applied science and technology.* Springer, 2013.

[154] Matthew B Rhudy, Roger A Salguero, and Keaton Holappa. A kalman filtering tutorial for undergraduate students. *International Journal of Computer Science & Engineering Survey*, 8(1):1–9, 2017.

[155] Stephen D Roberts and Dennis Pegden. The history of simulation modeling. In *2017 Winter Simulation Conference (WSC)*, pages 308–323. IEEE, 2017.

[156] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* "O'Reilly Media, Inc.", 2013.

[157] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

[158] Lior Rokack. Decision Trees. Technical report, Tel-Aviv University, 2015.

[159] Michael J Rosenfeld. Ols in matrix form. *NYU Lecture Notes*, 2013.

[160] Sheldon M Ross. *Introduction to probability models, 3rd Edition.* Academic Press, San Diego, 1985.

[161] Sheldon M Ross. *Introduction to probability models, 11th Edition.* Academic press, 2014.

[162] Dan Roth. Decision Trees. Technical report, University of Illinois, 2016.

[163] Prasanna Sahoo. *Probability and Mathematical Statistics.* University of Louisville, Louisville, KY 40292, USA, 2013.

[164] Remi M Sakia. The box-cox transformation technique: A review. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 41(2):169–178, 1992.

[165] Anders W Sandvik. Numerical solutions of classical equations of motion. *PY Comput. Phys*, 502, 2015.

[166] Robert G Sargent. An introductory tutorial on verification and validation of simulation models. In *2015 winter simulation conference (WSC)*, pages 1729–1740. IEEE, 2015.

[167] T Shalab. Regression analysis. chapter 12: Polynomial regression models. *University lectures, Indian Institute of Technology Kanpur*, 2010.

[168] Cosma Shalizi. Advanced data analysis from an elementary point of view, 2013.

[169] Cosma Shalizi. Modern Regression. Technical report, Carneige-Mellon University, 2015.

[170] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[171] Karl Sigman. Acceptance-rejection method. Technical report, Columbia University, 2007.

[172] Karl Sigman. Ieor 6711: Continuous-time markov chains. Technical report, Columbia University, 2009.

[173] Karl Sigman. Limiting distribution for a Markov chain recurrence and transience. Technical report, Columbia University, 2009.

[174] Karl Sigman. Notes on little's law. Technical report, Columbia University, 2009.

[175] Gregory A Silver, John A Miller, Maria Hybinette, Gregory Baramidze, and William S York. An ontology for discrete-event modeling and simulation. *Simulation*, 87(9):747–773, 2011.

[176] Christopher A Sims. Macroeconomics and reality. *Econometrica: journal of the Econometric Society*, pages 1–48, 1980.

[177] Saša Singer and Sanja Singer. Efficient implementation of the nelder–mead search algorithm. *Applied Numerical Analysis & Computational Mathematics*, 1(2):524–534, 2004.

[178] Anders Skajaa. Limited memory bfgs for nonsmooth optimization. *Master's thesis, Courant Institute of Mathematical Science, New York University*, 2010.

[179] A Solonen, J Hakkarainen, A Ilin, M Abbas, and A Bibov. Estimating model error covariance matrix parameters in extended kalman filtering. *Nonlinear Processes in Geophysics*, 21(5):919–927, 2014.

[180] Saul Stahl. The evolution of the normal distribution. *Mathematics magazine*, 79(2):96–113, 2006.

[181] Mark Stamp. A Revealing Introduction to Hidden Markov Models. Technical report, San Jose State University, 2018.

[182] Natalie M Steiger, Emily K Lada, James R Wilson, Jeffrey A Joines, Christos Alexopoulos, and David Goldsman. Asap3: A batch means procedure for steady-state simulation analysis. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(1):39–73, 2005.

[183] Suhartono Suhartono. Time series forecasting by using seasonal autoregressive integrated moving average: Subset, multiplicative or additive model. *J. Math. Stat*, 7:20–27, 2011.

[184] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[185] Gábor Szárnyas, József Marton, János Maginecz, and Dániel Varró. Reducing property graph queries to relational algebra for incremental view maintenance. *arXiv preprint arXiv:1806.07344*, 2018.

[186] Souhaib Ben Taieb. Machine learning strategies for multi-step-ahead time series forecasting. *Universit Libre de Bruxelles, Belgium*, pages 75–86, 2014.

[187] Souhaib Ben Taieb, Rob J Hyndman, et al. *Recursive and direct multi-step forecasting: the best of both worlds*, volume 19. Citeseer, 2012.

[188] Shintaro Takenaga, Yoshihiko Ozaki, and Masaki Onishi. Practical initialization of the nelder–mead method for computationally expensive optimization problems. *Optimization Letters*, 17(2):283–297, 2023.

[189] Srikanth Tammina. Transfer learning using VGG-16 with deep convolutional neural network for classifying images. *International Journal of Scientific and Research Publications*, 9(10):143–150, 2019.

[190] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[191] Thaddeus Tarpey. Generalized Linear Models (GLM). Technical report, Wright State University, 2012.

[192] Harsh Thakkar, Sören Auer, and Maria-Esther Vidal. Formalizing gremlin pattern matching traversals in an integrated graph algebra. In *BlockSW/CKG@ ISWC*, 2019.

[193] Harsh Thakkar, Dharmen Punjani, Soeren Auer, and Maria-Esther Vidal. Towards an integrated graph algebra for graph pattern matching with Gremlin (extended version). *arXiv preprint arXiv:1908.06265*, 2019.

[194] You Tingyan. Multistep Yule-Walker estimation of autoregressive models. Technical report, National University of singapore, 2010.

[195] Luís Fernando Raínho Alves Torgo. Inductive learning of tree-based regression models. 1999.

[196] Wessel N van Wieringen. Lecture notes on ridge regression. *arXiv preprint arXiv:1509.09169*, 2015.

[197] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[198] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[199] Eric A Wan, Rudolph Van Der Merwe, and Simon Haykin. The unscented kalman filter. *Kalman filtering and neural networks*, 5(2007):221–280, 2001.

[200] Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. *Advances in Neural Information Processing Systems*, 31, 2018.

[201] William WS Wei. *Multivariate Time Series Analysis and Applications.* John Wiley & Sons, 2018.

[202] Greg Welch and Gary Bishop. An introduction to the kalman filter: Siggraph 2001 course 8. In *Computer Graphics, Annual Conference on Computer Graphics & Interactive Techniques*, pages 12–17, 2001.

[203] Samuel S Wilks. The large-sample distribution of the likelihood ratio for testing composite hypotheses. *The annals of mathematical statistics*, 9(1):60–62, 1938.

[204] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[205] Ian Witten, Eibe Eibe Frank, Mark Hall, and Christopher Pal. *Data Mining Practical Machine Learning Tools and Techniques, Fourth Edition.* Elsevier Inc., 2017.

[206] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[207] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence*, 2(2):109–127, 2021.

[208] Shufang Xie, Tao Zhang, and Oliver Rose. Agent-based simulation with process-interaction worldview. *Simul. Notes Eur.*, 29(4):169–177, 2019.

[209] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[210] Mengjia Xu. Understanding graph embedding methods and their applications. *SIAM Review*, 63(4):825–853, 2021.

[211] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[212] Harry Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.

[213] Jianshu Zhang, Jun Du, and Lirong Dai. A GRU-based encoder-decoder approach with attention for online handwritten mathematical expression recognition. In *2017 14th IAPR international conference on document analysis and recognition (ICDAR)*, volume 1, pages 902–907. IEEE, 2017.

[214] Zhifei Zhang. Derivation of backpropagation in convolutional neural network (cnn). *University of Tennessee, Knoxville, TN*, 2016.

[215] Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234, 2016.

[216] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[217] Eric Zivot and Jiahui Wang. Unit root tests. *Modeling Financial Time Series with S-Plus*, pages 111–139, 2006.

[218] Eric Zivot and Jiahui Wang. Vector autoregressive models for multivariate time series. *Modeling financial time series with S-PLUS®*, pages 385–429, 2006.