Introduction to Data Science
Using ScalaTion

John A. Miller
Department of Computer Science
University of Georgia

April 4, 2019
Contents

1 Introduction to Data Science ........................................... 11
  1.1 Data Science ......................................................... 11
  1.2 SCALATION ......................................................... 11
  1.3 Additional Textbooks .............................................. 12

2 Mathematical Preliminaries ............................................ 13
  2.1 Probability .......................................................... 14
    2.1.1 Probability Measure ......................................... 14
    2.1.2 Random Variable ............................................. 14
    2.1.3 Cumulative Distribution Function .......................... 15
    2.1.4 Probability Mass Function .................................. 15
    2.1.5 Probability Density Function ............................... 16
    2.1.6 Expectation .................................................. 16
    2.1.7 Variance ...................................................... 17
    2.1.8 Covariance ................................................... 17
    2.1.9 Quantiles .................................................... 18
    2.1.10 Mode ........................................................ 18
    2.1.11 Conditional Mass and Density .............................. 18
    2.1.12 Conditional Expectation ................................... 19
    2.1.13 Odds ........................................................ 19
    2.1.14 Example Problems .......................................... 20
    2.1.15 Exercises .................................................. 24
    2.1.16 Further Reading ............................................. 25
  2.2 Linear Algebra ...................................................... 26
    2.2.1 Linear System of Equations ................................. 26
    2.2.2 Matrix Inversion ............................................. 26
    2.2.3 Vector ........................................................ 27
    2.2.4 Vector Operations ............................................ 27
    2.2.5 Vector Calculus .............................................. 29
    2.2.6 Matrix ........................................................ 30
    2.2.7 Matrix Operations .......................................... 31
    2.2.8 Matrix Factorization ....................................... 33
    2.2.9 Internal Representation .................................... 33
## 3 Data Management and Preprocessing

### 3.1 Analytics Databases
- Columnar Relational Algebra API
- SQL-Like API
- Map-Reduce API

### 3.2 Preprocessing
- Remove Identifiers
- Convert String Columns to Numeric Columns
- Identify Missing Values
- Detect Outliers
- Imputation Techniques
- Preliminary Feature Selection
- Align Multiple Time Series
- Creating Vectors and Matrices

### 3.3 Exercises

## 4 Prediction

### 4.1 Predictor
- Fit
- PredictorMat
- PredictorVec

### 4.2 Null Model

### 4.3 Simpler Regression
- Derivative

### 4.4 Simple Regression
- Gradient

### 4.5 Regression
- Matrix Inversion Technique
- LU Factorization Technique
- Cholesky Factorization Technique
- QR Factorization Technique
- Singular Value Decomposition Technique
- Use of Factorization in Regression
- Model Assessment
- Model Validation
- Collinearity
- Feature Selection
10 Temporal Models
  10.1 ForecasterVec
      10.1.1 Auto-Correlation Function
  10.2 Auto-Regressive (AR) Models
      10.2.1 AR(1) Model
      10.2.2 AR(p) Model
      10.2.3 Training
      10.2.4 Forecasting
      10.2.5 Exercises
  10.3 Moving-Average (MA) Models
      10.3.1 Selection Based on ACF and PACF
      10.3.2 MA(q) Model
  10.4 ARMA
  10.5 ARIMA
  10.6 ARIMAX
  10.7 SARIMA
  10.8 Exponential Smoothing
  10.9 Dynamic Linear Models
      10.10 Recurrent Neural Networks (RNN)
          10.10.1 Gate Recurrent Unit (GRU) Networks
          10.10.2 Long Short Term Memory (LSTM) Networks
  10.11 Temporal Convolutional Networks (TCN)
  10.12 ODE Parameter Estimation
      10.12.1 Non-Linear Least Squares (NLS)
      10.12.2 Least Squares Approximation (LSA)

11 Spatial Models
  11.1 Convolutional Neural Networks

12 Clustering
  12.1 Clusterer
      12.2 K-Means Clustering
      12.3 Hierarchical Clustering
      12.4 Markov Clustering

13 Dimensionality Reduction
  13.1 Reducer
      13.2 Principal Component Analytics

14 Functional Data Analysis
  14.1 Basis Functions
  14.2 Functional Smoothing
  14.3 Functional Principal Component Analysis
  14.4 Functional Regression
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1</td>
<td>MIMD - Multithreading</td>
<td>280</td>
</tr>
<tr>
<td>17.2</td>
<td>SIMD - Vector Instructions</td>
<td>281</td>
</tr>
<tr>
<td>17.3</td>
<td>Message Passing</td>
<td>282</td>
</tr>
<tr>
<td>17.4</td>
<td>Distributed Shared Memory</td>
<td>283</td>
</tr>
<tr>
<td>17.5</td>
<td>Microservices</td>
<td>284</td>
</tr>
<tr>
<td>17.6</td>
<td>Distributed Functional Programming</td>
<td>285</td>
</tr>
</tbody>
</table>
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.

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 SCALAppATION

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

In SCALAppATION, the analytics 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). SCALAppATION 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.

The latest version, SCALAppATION 1.6, consists of five modules. Each module contains many packages (a key package is given for each module below).
1. scalation_mathematics: e.g., scalation.linalgebra
2. scalation_statistics: e.g., scalation.stat
3. scalation_database: e.g., scalation.columnar_db
4. scalation_modeling: e.g., scalation.analytics
5. scalation_models: e.g., apps.analytics

To use SCALATION, go to the Website [http://www.cs.uga.edu/~jam/scalation.html](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).

### 1.3 Additional Textbooks

More detailed development of this material can be found in textbooks on statistical learning, such as “An Introduction to Statistical Learning” (ISL) [12] and “The Elements of Statistical Learning” (ESL) [10].

<table>
<thead>
<tr>
<th>Topic</th>
<th>SCALATION</th>
<th>ISL</th>
<th>ESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathematical Preliminaries</td>
<td>Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data Management</td>
<td>Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prediction</td>
<td>Ch. 4</td>
<td>Ch. 3, 5, 6</td>
<td>Ch. 3</td>
</tr>
<tr>
<td>Classification</td>
<td>Ch. 5</td>
<td>Ch. 2, 5, 8</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Classification - Continuous</td>
<td>Ch. 6</td>
<td>Ch. 4, 8, 9</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Generalized Linear Models</td>
<td>Ch. 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Generalized Additive Models</td>
<td>Ch. 8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Non-Linear Models</td>
<td>Ch. 9</td>
<td>Ch. 7</td>
<td>Ch. 11</td>
</tr>
<tr>
<td>Temporal Models</td>
<td>Ch. 10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spatial Models</td>
<td>Ch. 11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Clustering</td>
<td>Ch. 12</td>
<td>Ch. 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Dimensionality Reduction</td>
<td>Ch. 13</td>
<td>Ch. 6, 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Functional Data Analysis</td>
<td>Ch. 14</td>
<td>Ch. 7</td>
<td>Ch. 5</td>
</tr>
<tr>
<td>Optimization</td>
<td>Ch. 15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Simulation Models</td>
<td>Ch. 16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Parallel/Distributed Computing</td>
<td>Ch. 17</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1.1: Source Material Chapter Mappings
Chapter 2

Mathematical Preliminaries

This chapter serves as a quick review of the two principal mathematical foundations for data science, probability and linear algebra.
2.1 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 $S$ is the set of all possible outcomes.

2.1.1 Probability Measure

A probability measure $P$ can be defined axiomatically as follows:

$$P(A) \geq 0 \text{ for any event } A \subseteq S$$
$$P(S) = 1$$
$$P(\cup A_i) \leq \sum P(A_i) \text{ for a countable collection of disjoint events}$$

(2.1)

Consequently, given an event $A$, the probability of its occurrence is restricted to the unit interval, $P(A) \in [0, 1]$.

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

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

(2.2)

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

$$P(AB) = P(A)P(B)$$

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

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

(2.3)

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

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

In other words, the occurrence of event $B$ has no affect on the probability of event $A$ occurring. An important theorem involving conditional probability is Bayes Theorem.

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

(2.4)

2.1.2 Random Variable

Rather than just looking at individual events, e.g., $A$ or $B$, one is often more interested in the probability that random variables take on certain values. A random variable $y$ (blue font) takes on values from a given domain $D_y$.

$$y \in D_y$$

(2.5)

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 $P(y \in A)$. For example, the probability of rolling a natural in dice (sum of 7 or 11 with two dice) is given by

$$P(y \in \{7, 11\}) = \frac{6}{36} + \frac{2}{36} = \frac{8}{36} = \frac{2}{9}$$
2.1.3 Cumulative Distribution Function

It is often easier to examine the probability measure for a random variable in terms of a Cumulative Distribution Function (CDF). 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.

$$F_y(y) = P(y \leq y)$$  \hspace{1cm} (2.6)

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 $F_y(6) = P(y \leq 6) = 5/12$. 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

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

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

2.1.4 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, a probability mass function (pmf) may be defined.

$$p_y(y_i) = F_y(y_i) - F_y(y_{i-1})$$  \hspace{1cm} (2.7)

This indicates the amount of mass/probability at point $y_i$, 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. They both follow the Discrete Uniform Distribution. 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.
2.1.5 Probability Density Function

Suppose \( y \) is defined on the continuous domain \([0, 2]\) and that mass/probability is uniformly spread among 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.

\[
f_y(y) = \frac{dF_y(y)}{dy}
\]  
(2.8)

For example, the pdf for the uniformly distributed random variable \( y \) on \([0, 2]\) is

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

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

\[
val rv = \text{Uniform} \ (0, \ 2) \\
val y = rv \text{.gen}
\]

Going the other direction, the CDF \( F_y(y) \) can be computed by summing the pmf \( p_y(y) \) or integrating the pdf \( f_y(y) \).

2.1.6 Expectation

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

\[
E[y] = \int_{D_y} y \, dF_y(y)
\]  
(2.9)

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.

Continuous Case

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

\[
E[y] = \int_{D_y} y \, f_y(y) \, dy
\]  
(2.10)

The mean of \( y \sim \text{Uniform} \ (0, \ 2) \) is

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

When \( y \) is a discrete random variable, we may write

\[
E[y] = \sum_{y \in D_y} y p_y(y)
\]  

(2.11)

The mean for rolling two dice is \( E[y] = 7 \). 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.

2.1.7 Variance

The variance of random variable \( y \) is given by

\[
V[y] = E[(y - E[y])^2]
\]  

(2.12)

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

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

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.

\[
SD[y] = \sqrt{V[y]}
\]  

(2.13)

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.

\[
f_y(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2}
\]

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\%.

2.1.8 Covariance

The covariance of two random variable \( x \) and \( y \) is given by

\[
C[x, y] = E[(x - E[x])(y - E[y])]
\]  

(2.14)

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 \(z\) be a vector of \(k\) random variables, then a covariance matrix is produced.

\[
\mathbb{C}[z] = [\mathbb{C}[z_i, z_j]]_{0 \leq i, j < k}
\]

### 2.1.9 Quantiles

In addition, one may be interested in the median or half quantile

\[
Q[y] = F_y^{-1}\left(\frac{1}{2}\right)
\]

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

\[
pQ[y] = F_y^{-1}(p)
\]

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 \(Q[y] = F_y^{-1}(\frac{1}{2}) = 1\).

### 2.1.10 Mode

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

\[
M[y] = \arg\max_{y \in D_y} p_y(y)
\]

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

\[
M[y] = \arg\max_{y \in D_y} f_y(y)
\]

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

### 2.1.11 Conditional Mass and Density

Conditional probability can be examined locally.
Discrete Case

Given two discrete random variables $x$ and $y$, the *conditional mass function* of $x$ given $y$ is defined as follows:

$$p_{x|y}(x, y) = \frac{P(x = x, y = y)}{p_y(y)}$$  \hspace{1cm} (2.19)

where $p_{x,y}(x, y)$ is the *joint mass function*. The *marginal mass function* for $x$ is

$$p_x(x) = \sum_{y \in D_y} p_{x,y}(x, y)$$  \hspace{1cm} (2.20)

Continuous Case

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

$$f_{x|y}(x, y) = \frac{f_{x,y}(x, y)}{f_y(y)}$$  \hspace{1cm} (2.21)

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

$$f_x(x) = \int_{y \in D_y} f_{x,y}(x, y) dy$$  \hspace{1cm} (2.22)

2.1.12 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:

$$E[x|y = y] = \int_{D_x} x dF_{x|y}(x, y)$$  \hspace{1cm} (2.23)

When $y$ is a continuous random variable, we may write

$$E[x|y = y] = \int_{D_x} x f_{x|y}(x, y) dx$$  \hspace{1cm} (2.24)

When $y$ is a discrete random variable, we may write

$$E[x|y = y] = \sum_{x \in D_x} x p_{x|y}(x, y)$$  \hspace{1cm} (2.25)

2.1.13 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 $S - A$.

$$\text{odds}(y \in A) = \frac{P(y \in A)}{P(y \in S - A)} = \frac{P(y \in A)}{1 - P(y \in A)}$$  \hspace{1cm} (2.26)

For example, the odds of rolling a pair dice and getting natural is 8 to 28.
odds\( (y \in \{7, 11\}) = \frac{8}{28} = \frac{2}{7} = 0.2857 \)

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

\[
\text{odds}(y \in \{7, 11\}) = \frac{P(y \in \{7, 11\})}{1 - P(y \in \{7, 11\})} = \frac{2/9}{7/9} = \frac{2}{7} = 0.2857
\]

Calculating probability from odds may be done as follows:

\[
P(y \in \{7, 11\}) = \frac{\text{odds}(y \in \{7, 11\})}{1 + \text{odds}(y \in \{7, 11\})} = \frac{2/7}{9/7} = \frac{2}{9} = 0.2222
\]

### 2.1.14 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.

\[
P(y \in \{7, 11\}) = \frac{1}{6} + \frac{1}{18} = \frac{2}{9}
\]

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

\[
P(y \in \{5, 7\} \mid y \in \{7, 11\}) = \frac{P(y \in \{5, 7\}, y \in \{7, 11\})}{P(y \in \{7, 11\})} = \frac{1/6}{2/9} = \frac{3}{4}
\]

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 \( x \in B \).

\[
P(y \in A \mid x \in B) = \frac{P(y \in A, x \in B)}{P(x \in B)}
\]

where

\[
P(y \in A, x \in B) = P(x \in B \mid y \in A) P(y \in A)
\]

Therefore, the conditional probability of \( y \) given \( x \) is

\[
P(y \in A \mid x \in B) = \frac{P(x \in B \mid y \in A) P(y \in A)}{P(x \in B)}
\]

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

\[
\frac{P(y \in \{7, 11\} \mid y \in \{5, 7\}) P(y \in \{5, 7\})}{P(y \in \{7, 11\})} = \frac{(3/5) \cdot (5/18)}{2/9} = \frac{3}{4}
\]
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?

$$P(x = 1) = \frac{1}{3}$$

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,

$$P(x = 1 | y = 3) = \frac{P(y = 3 | x = 1) P(x = 1)}{P(y = 3)} = \frac{\frac{1}{3} \times \frac{1}{3}}{\frac{5}{12}} = \frac{4}{5}$$

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

3. 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). Create a sample of size $m = 100$ data points, using a Normal random variate generator. The population values for the mean $\mu$ and standard deviation $\sigma$ are typically unknown and need to estimated from the sample, hence the names sample mean $\tilde{\mu}$ and sample standard deviation $\tilde{\sigma}$. Show the generated sample, by plotting the data points and displaying a histogram.

```scala
val (mu, sig) = (70.0, 8.0) // population mean and standard deviation
val m = 100 // sample size
val t = VectorD.range (0, m) // time/index vector
val rvg = Normal (mu, sig * sig) // Normal random variate generator
val sample = new VectorD (m) // vector to hold sample
for (i <- sample.range) sample(i) = rvg.gen // sample from Normal distribution
val (mu_, sig_) = (sample.mean, sample.stddev) // sample mean and standard deviation
println (s"(mu_, sig_) = ($mu_, $sig_)")
new Plot (t, sample)
new Histogram (sample)
```


4. 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 the sample values into a vector $y$. Then the mean is simply

$$\tilde{\mu} = \frac{1}{m} \cdot y$$
To create a confidence interval, we need to determine the variability or variance in the estimate $\hat{\mu}$.

$$\text{V}[\hat{\mu}] = \frac{\text{V}[y]}{m} = \frac{\sigma^2}{m}$$

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 $\mu$, i.e., $\mathbb{E}[\hat{\mu}] = \mu$).

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

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

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

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 $\mu$ 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 $\frac{2\sigma}{\sqrt{m}} = 1.6$, which is to be subtracted and added to the sample mean.

Use ScalaTion to determine the probability that $\mu$ is within such confidence intervals?

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

22
The probability is one minus twice this value. If 1.96 is used instead of 2, what is the probability, expressed as a percentage.

Also, the population standard deviation is unlikely to be known. It would need to estimated by using the sample standard deviation. This substitution introduces more variability into the estimation of the confidence interval and results in the Standard Normal distribution (z-distribution)

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

being replace by the Student t-distribution

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

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.

5. 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
for (it <- 1 to iter) {
  val sample = new VectorD (m) // vector to hold sample
  for (i <- sample.range) sample(i) = rvg.gen // sample from Normal distribution
  val (mu_, sig_) = (sample.mean, sample.stddev) // sample mean and standard deviation
  val interv = sample.interval () // interval half width: t-distribution
  val ci = sample.ci (mu_, interv) // confidence interval
  val inside = ci._1 <= mu && mu <= ci._2
  val interv2 = sample.interval2 (sig_) // interval half width: z-distribution
  val ci2 = sample.ci (mu_, interv2) // confidence interval
  val inside2 = ci2._1 <= mu && mu <= ci2._2
  if (inside) count += 1
  if (inside2) count2 += 1
}
``` // for

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?

6. Explain why the probability mass function (pmf) for flipping a coin \( n \) times with the experiment resulting in \( k \) heads is given by the Binomial Distribution having parameter \( p \), the probability of getting a head for any particular coin flip.

\[
p_n(k) = P(k = k) = \binom{n}{k} p^k (1 - p)^{n-k}
\]

Now suppose an experiment is run and \( k = k \), 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 \textit{Likelihood function} and it is used in Maximum Likelihood Estimation (MLE) \textsuperscript{[15]}.

2.1.15 Exercises

Several random number and random variate generators can be found in SCALATion’s \texttt{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: \( P(h = 2 \mid h \geq 1) \).

2. Prove Bayes Theorem.

\[ P(A \mid B) = \frac{P(B \mid A) P(A)}{P(B)} \]

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. Show that the variance may be written as follows:

\[ \mathbb{V}[y] = \mathbb{E}[(y - \mathbb{E}[y])^2] = \mathbb{E}[y^2] - (\mathbb{E}[y])^2 \]

5. Use the \texttt{Randi} random variate generator to run experiments to check the pmf and CDF for rolling two dice.

```scala
import scalation.linalgebra.VectorD
import scalation.plot.Plot
import scalation.random.Randi

object DiceTest extends App {
  val dice = Randi(1, 6)
  val x = VectorD.range(0, 13)
  val freq = new VectorD(13)
  for (i <- 0 until 10000) {
    val sum = dice.igen + dice.igen
    freq(sum) += 1
  } // for
  new Plot(x, freq)
} // DiceTest object
```

6. Use the Uniform random variate generator and the Histogram class to run experiments illustrating the Central Limit Theorem (CLT).
import scalation.linalgebra.VectorD
import scalation.stat.Histogram
import scalation.random.Uniform

object CLTTest extends App
{
    val rg = Uniform ()
    val x = VectorD (for (i <- 0 until 100000) yield rg.gen + rg.gen + rg.gen + rg.gen)
    new Histogram (x)
} // CLTTest object

7. 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
your 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.

object MontyHall extends App
{
    val rg = Randi (0, 2) // door selection (0, 1 or 2) random generator
    val coin = Bernoulli () // 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) { // 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) winStay += 1 // stay with initial pick
        else winSwitch += 1 // switch to the other door
    } // for

    println ("winStay = " + winStay)
    println ("winSwitch = " + winSwitch)
}

} // MontyHall object

2.1.16 Further Reading

1. Probability and Mathematical Statistics [21]
2.2 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 \( \mathbf{X} \) and the parameters/coefficients \( \mathbf{b} \) are fit to the data.

2.2.1 Linear System of Equations

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

\[
\begin{align*}
y_0 &= x_{00}b_0 + x_{01}b_1 \\
y_1 &= x_{10}b_0 + x_{11}b_1
\end{align*}
\]

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?

\[
\begin{align*}
100 &= 1b_0 + 1b_1 \\
1600 &= 10b_0 + 20b_1
\end{align*}
\]

The solution is \( b_0 = 40 \) children and \( b_1 = 60 \) adults.

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

\[
\mathbf{y} = \mathbf{X}\mathbf{b}
\]

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

2.2.2 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 \( \mathbf{X} \), \( \mathbf{X}^{-1} \)

\[
\mathbf{b} = \mathbf{X}^{-1}\mathbf{y}
\]

Multiplying matrix \( \mathbf{X} \) and its inverse \( \mathbf{X}^{-1} \), \( \mathbf{X}^{-1}\mathbf{X} \) results in an \( n \)-by-\( n \) identity matrix \( \mathbf{I}_n = [I_{i=j}] \), where the indicator function \( I_{i=j} \) 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 \( \mathbf{X} \) into lower \( \mathbf{L} \) and upper \( \mathbf{U} \) triangular matrices.

\[
\mathbf{X} = \mathbf{LU}
\]

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

\[
\begin{bmatrix}
1 & 3 & 1 \\
0 & -5 & 5
\end{bmatrix}
\]

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

\[
b = \begin{bmatrix} 4 \\ -1 \end{bmatrix}
\]

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

Vector notation is used in this technical report, 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 linalgbra and linalgbra_gen packages. A commonly used operation is the dot (inner) product, $x \cdot y$, or in SCALATION, x dot y.

### 2.2.3 Vector

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

\[
x = [x_0, x_1, x_2] = [0.57735, 0.55735, 0.57735] \\
y = [y_0, y_1, y_2] = [1.0, 1.0, 0.0]
\]

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

### 2.2.4 Vector Operations

Vectors may be added ($x + y$), subtracted ($x - y$), multiplied element-by-element (Hadamard product) ($x * y$), and divided element-by-element ($x / y$). These operations are also supported when one of the arguments is a scalar. A particularly important operation, the dot product of two vectors is simply the sum of the products of their elements.

\[
x \cdot y = \sum_{i=0}^{n-1} x_i y_i = 1.1547 \quad (2.32)
\]

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

\[
\|x\| = \sqrt{\sum_{i=0}^{n-1} x_i^2} = 1 \quad (2.33)
\]

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

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

so the angle \( \theta = 0.616 \) radians. Vectors \( x \) and \( y \) are orthogonal if the angle \( \theta = \pi/2 \) radians (90 degrees).

In general there are \( \ell_p \) norms. The two that are used here are the \( \ell_2 \) norm \( \|x\| = \|x\|_2 \) (Euclidean distance) and the \( \ell_1 \) norm \( \|x\|_1 \) (Manhattan distance).

\[
\|x\|_1 = \sum_{i=0}^{n-1} |x_i|
\]

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.

\[
\begin{align*}
\mu(x) &= \mu_x = \frac{1}{m} \cdot x \\
\sigma^2(x) &= \sigma^2_x = \frac{(x - \mu_x) \cdot (x - \mu_x)}{m} \\
\sigma(x, y) &= \sigma_{x,y} = \frac{(x - \mu_x) \cdot (y - \mu_y)}{m} \\
\rho(x, y) &= \rho_{x,y} = \frac{\sigma_{x,y}}{\sigma_x \sigma_y}
\end{align*}
\]

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.
2.2.5 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.

Gradient

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

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

The gradient of function \( f \) consists of a vector formed from the two partial derivatives and

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

indicates the direction of steepest increase. Its norm indicates the magnitude of the rate of change. 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 \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) of vector \( \mathbf{x} \in \mathbb{R}^n \) is

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

(2.34)

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.

\[
    \nabla (f(\mathbf{x}) \cdot g(\mathbf{x})) = \nabla f(\mathbf{x}) \cdot g(\mathbf{x}) + f(\mathbf{x}) \cdot \nabla g(\mathbf{x})
\]

(2.35)

Jacobian

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 \( \mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m \) of vector \( \mathbf{x} \in \mathbb{R}^n \) is

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

(2.36)
Vector operations are illustrated by the `VectoD` trait, which 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>
<thead>
<tr>
<th>op</th>
<th>vector op vector</th>
<th>vector op scalar</th>
<th>vector element op scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>def + (b: VectoD): VectoD</td>
<td>def + (s: Double): VectoD</td>
<td>def + (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>+=</td>
<td>def += (b: VectoD): VectoD</td>
<td>def += (s: Double): VectoD</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>def - (b: VectoD): VectoD</td>
<td>def - (s: Double): VectoD</td>
<td>def - (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>-=</td>
<td>def -= (b: VectoD): VectoD</td>
<td>def -= (s: Double): VectoD</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>def * (b: VectoD): VectoD</td>
<td>def * (s: Double): VectoD</td>
<td>def * (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>*=</td>
<td>def *= (b: VectoD): VectoD</td>
<td>def *= (s: Double): VectoD</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>def / (b: VectoD): VectoD</td>
<td>def / (s: Double): VectoD</td>
<td>def / (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>/=</td>
<td>def /= (b: VectoD): VectoD</td>
<td>def /= (s: Double): VectoD</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Vector Arithmetic Operations

2.2.6 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

\[ f : \mathbb{R}^n \to \mathbb{R}^m \]  

that map vectors in \( \mathbb{R}^n \) 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:

```scala
tval 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 \( \mathbb{R}^3 \).

\[ \mathbf{A} \mathbf{u} = \mathbf{v} \] (2.38)

30
For example,

\[
A \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \\ 17 \end{bmatrix}
\]

### 2.2.7 Matrix Operations

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

\[
\begin{align*}
A & = \text{matrix} \\
A & = a() = \text{underlying array} \\
a_i & = a(i) = \text{row vector } i \\
a_{:j} & = a.col(j) = \text{column vector } j \\
a_{ij} & = a(i, j) = \text{the element at row } i \text{ and column } j \\
A_{i:k,j:l} & = a(i \text{ to } k, j \text{ to } l) = \text{row and column matrix slice}
\end{align*}
\]

Note that in Scala, \(i \text{ to } k\) is a Range that includes \(k\), while \(i \text{ until } k\) does not include \(k\). Common operations on matrices are supported as well.

**Matrix Addition and Subtraction**

Matrix addition \(\text{val c = a + b}\)

\[
c_{ij} = a_{ij} + b_{ij}
\]

and matrix subtraction \(\text{val c = a - b}\) are supported.

**Matrix Multiplication**

A frequently used operation in data science is matrix multiplication \(\text{val c = a * b}\).

\[
c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}
\]

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 \(A^t\) (\(\text{val t = a.t}\)), simply exchanges the roles of rows and columns.
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.

Matrix Dot Product

SCALaTION provides several types of dot products on both vectors and matrices, three of which are shown below. The first method computes the usual dot product between two vectors, while the second and third methods are between two matrices. The second method simply takes dot products of the corresponding columns of each matrix. The third method provides another (and sometimes more efficient) way to compute \(A^t B = A \cdot B = a.t * b = a.mdot b\).

```scala
def dot (b: VectorD): Double =
{
  var s = 0.0
  for (i <- range) s += v(i) * b.v(i)
  s
} // dot

def dot (b: MatrixD): VectorD =
{
  if (dim1 != b.dim1) flaw ("dot", "matrix dot matrix - incompatible first dimensions")
  val c = new VectorD (dim2)
  for (i <- range1; j <- range2) c(j) += v(i)(j) * b.v(i)(j)
  c
} // dot

def mdot (b: MatrixD): MatrixD =
{
  if (dim1 != b.dim1) flaw ("mdot", "matrix mdot matrix - incompatible first dimensions")
  val c = new MatrixD (dim2, b.dim2)
  val at = this.t // transpose the 'this' matrix
  for (i <- range2) {
    val at_i = at.v(i) // ith row of 'at' (column of 'a')
    for (j <- b.range2) {
      var sum = 0.0
      for (k <- range1) sum += at_i(k) * b.v(k)(j)
      c.v(i)(j) = sum
    } // for
  } // for
} // for
```
2.2.8 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

<table>
<thead>
<tr>
<th>Factorization</th>
<th>Factors</th>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>$A = LU$</td>
<td>lower left triangular</td>
<td>upper right triangular</td>
<td>Fac_LU</td>
</tr>
<tr>
<td>Cholesky</td>
<td>$A = LL^t$</td>
<td>lower left triangular</td>
<td>its transpose</td>
<td>Fac_Cholesky</td>
</tr>
<tr>
<td>QR</td>
<td>$A = QR$</td>
<td>orthogonal</td>
<td>upper right triangular</td>
<td>Fac_QR_H</td>
</tr>
<tr>
<td>SVD</td>
<td>$A = U\Sigma V^t$</td>
<td>orthogonal</td>
<td>diagonal, orthogonal</td>
<td>SVD</td>
</tr>
</tbody>
</table>

Table 2.2: Matrix Factorization Techniques

See Chapter 4 to see how matrix factorization is used in Ordinary Least Squares estimation.

2.2.9 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 submatrices (blocks) may improve performance because of caching efficiency or improved performance for parallel and distributed versions.

The `linalg` package provides several traits and classes implementing multiple types of vectors and matrices. The `VectoD` trait has dense, sparse and compressed class implementations, while the `MatriD` trait has dense, sparse, compressed, symmetric tridiagonal and bidiagonal class implementations.

<table>
<thead>
<tr>
<th>trait</th>
<th>VectoD</th>
<th>MatriD</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense</td>
<td>VectorD</td>
<td>MatrixD</td>
</tr>
<tr>
<td>sparse</td>
<td>SparseVectorD</td>
<td>SparseMatrixD</td>
</tr>
<tr>
<td>compressed</td>
<td>RleVectorD</td>
<td>RleMatrixD</td>
</tr>
<tr>
<td>tridiagonal</td>
<td>-</td>
<td>SymTriMatrixD</td>
</tr>
<tr>
<td>bidiagonal</td>
<td>-</td>
<td>BidMatrixD</td>
</tr>
</tbody>
</table>

Table 2.3: Types of Vectors and Matrices: Implementing Classes

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’, StrNum ‘S’, and TimeNum ‘T’. There are also generic implementations in `linalg.gen`, but they tend to run more slowly.

ScalaTion supports many operations involving matrices and vectors, including the following:
<table>
<thead>
<tr>
<th>Product</th>
<th>Method</th>
<th>Example</th>
<th>in Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector dot</td>
<td>def dot (y: VectoD): Double</td>
<td>x dot y</td>
<td>x \cdot y</td>
</tr>
<tr>
<td>vector elementwise</td>
<td>def * (y: VectoD): VectoD</td>
<td>x * y</td>
<td>xy</td>
</tr>
<tr>
<td>vector outer</td>
<td>def outer (y: VectoD): MatriD</td>
<td>x outer y</td>
<td>x \otimes y</td>
</tr>
<tr>
<td>matrix mult</td>
<td>def * (y: MatriD): MatriD</td>
<td>x * y</td>
<td>XY</td>
</tr>
<tr>
<td>matrix dot</td>
<td>def dot (y: MatriD): VectoD</td>
<td>x dot y</td>
<td>X \cdot Y</td>
</tr>
<tr>
<td>matrix mdot</td>
<td>def mdot (y: MatriD): MatriD</td>
<td>x mdot y</td>
<td>X^t Y</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def * (y: VectoD): VectoD</td>
<td>x * y</td>
<td>X y</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def ** (y: VectoD): MatriD</td>
<td>x ** y</td>
<td>X \text{diag}(y)</td>
</tr>
</tbody>
</table>

Table 2.4: Types of Vector and Matrix Products

2.2.10 Exercises

1. Draw two 2-dimensional non-zero vectors whose dot product is zero.

2. Given the matrix $X$ and the vector $y$, solve for the vector $b$ in the equation $y = Xb$ using matrix inversion and $LU$ factorization.

   ```scala
   import scalation.linalgebra.{MatrixD, VectorD, Fac_LU}
   val x = new 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.

3. If $Q$ is an orthogonal matrix, then $Q^tQ$ becomes what type of matrix? What about $QQ^t$? Illustrate with an example 3-by-3 matrix. What is the inverse of $Q$?

2.2.11 Further Reading

1. Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares

2. Matrix Computations
2.3 Notational Conventions

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

<table>
<thead>
<tr>
<th>variable type</th>
<th>case</th>
<th>font</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>lower</td>
<td>italics</td>
<td>black</td>
</tr>
<tr>
<td>vector</td>
<td>lower</td>
<td>bold</td>
<td>black</td>
</tr>
<tr>
<td>matrix</td>
<td>upper</td>
<td>italics</td>
<td>black</td>
</tr>
<tr>
<td>random scalar</td>
<td>lower</td>
<td>italics</td>
<td>blue</td>
</tr>
<tr>
<td>random vector</td>
<td>lower</td>
<td>bold</td>
<td>blue</td>
</tr>
</tbody>
</table>

Table 2.5: Notational Conventions Followed

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

```scala
type FunctionS2S = Double => Double // function of a scalar - Double
type FunctionV2S = VectorD => Double // function of a vector - VectorD
type FunctionV_2S = VectorD => Double // function of a vector - VectorD
type FunctionV2V = VectorD => VectorD // vector-valued function of a vector - VectorD

type FunctionM2M = MatrixD => MatrixD // matrix-valued function of a matrix - MatrixD

type FunctionM_2M = MatrixD => MatrixD // matrix-valued function of a matrix - MatrixD
```

These function types are defined in `scalation.math` and `svcalation.linalgebra`. 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., \( f \).

\[
\begin{align*}
\text{S2S function } f & : \mathbb{R} \rightarrow \mathbb{R} \\
\text{V2S function } f & : \mathbb{R}^n \rightarrow \mathbb{R} \\
\text{V}_2\text{S function } f & : \mathbb{R}^n \rightarrow \mathbb{R}
\end{align*}
\]

Mathematically, the vector-valued and matrix-valued functions are denoted by a bold symbol, e.g., \( f \).

\[
\begin{align*}
\text{V2V function } f & : \mathbb{R}^n \rightarrow \mathbb{R}^m \\
\text{V}_2\text{V function } f & : \mathbb{R}^n \rightarrow \mathbb{R}^m \\
\text{M2M function } f & : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{p \times q} \\
\text{M}_2\text{M function } f & : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{p \times q}
\end{align*}
\]
Chapter 3

Data Management and Preprocessing

Data Science relies on having large amounts of quality data. Collecting data and handling data quality issues are of upmost 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 and preprocessing. Data management capabilities are provided by SCALATION’s Time Series DataBase (TSDB). 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.
3.1 Analytics Databases

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. 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 three Application Programming Interfaces (APIs) for convenient access to the data [?].

3.1.1 Columnar Relational Algebra API

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.

Relation Creation

A Relation is created by invoking a constructor or factory apply function. For example, the following four Relations may be useful in a traffic forecasting study.

```scala
val sensor = Relation ("sensor", Seq ("sensorID", "model", "latitude", "longitude", "on"), Seq (), 0, "ISDDI")
val road = Relation ("road", Seq ("roadID", "rdName", "lat1", "long1", "lat2", "long2"), Seq (), 0, "ISDDDDDD")
val mroad = Relation ("road", Seq ("roadID", "rdName", "lanes", "lat1", "long1", "lat2", "long2"), Seq (), 0, "ISIDDDDD")
val traffic = Relation ("traffic", Seq ("time", "sensorID", "count", "speed"), Seq (), Seq (0, 1), "TIID")
val wsensor = Relation ("sensor", Seq ("sensorID", "model", "latitude", "longitude"), Seq (), 0, "ISDD")
val weather = Relation ("weather", Seq ("time", "sensorID", "precipitation", "wind"), Seq (), Seq (0, 1), "TIID")
```

The name of the first relation is “sensor”, the first sequence is the attribute names, the second sequence is the data (currently empty), 0 is the column number for the primary key, “ISDDI” indicates the domains for the attributes (Integer, String, Double, Double, Integer). It stores information about traffic sensors. The second relation stores the ID, name, beginning and ending latitude-longitude coordinates. The third relation is for multi-lane roads. The fourth relation 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 stores information about weather sensors. Finally, the sixth relation stores data collected from the weather sensors.
Relation Population

There are several ways to populate the Relations. A row/tuple can be added one at a time using the `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.

```scala
val sensor = Relation ("sensor", "austin_traffic_sensors.csv")
```

Data files are stored in subdirectories of ScalaTion’s `data` directory.

Columnar Relational Algebra Operators

Table 3.1 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 ∪ 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`).

Table 3.1: Columnar Relational Algebra (`r = road, s = sensor, t = traffic, q = mroad, w = weather`)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Unicode</th>
<th>Example</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select</code></td>
<td>σ</td>
<td><code>r.σ (&quot;rdName&quot;, _ == &quot;I285&quot;)</code></td>
<td>rows of <code>r</code> where <code>rdName</code> == &quot;I285&quot;</td>
</tr>
<tr>
<td><code>project</code></td>
<td>π</td>
<td><code>r.π (&quot;rdName&quot;, &quot;lat1&quot;, &quot;long1&quot;)</code></td>
<td>the <code>rdName</code>, <code>lat1</code>, and <code>long1</code> columns of <code>r</code></td>
</tr>
<tr>
<td><code>union</code></td>
<td>∪</td>
<td><code>r ∪ q</code></td>
<td>rows that are in <code>r</code> or <code>q</code></td>
</tr>
<tr>
<td><code>minus</code></td>
<td>−</td>
<td><code>r − q</code></td>
<td>rows that are in <code>r</code> but not <code>q</code></td>
</tr>
<tr>
<td><code>product</code></td>
<td>×</td>
<td><code>r × t</code></td>
<td>concatenation of each row of <code>r</code> with those of <code>t</code></td>
</tr>
<tr>
<td><code>rename</code></td>
<td>ρ</td>
<td><code>r.ρ(&quot;r2&quot;)</code></td>
<td>a copy of <code>r</code> with new name <code>r2</code></td>
</tr>
<tr>
<td><code>join</code></td>
<td>∞</td>
<td><code>r ∞ s</code></td>
<td>rows in natural join of <code>r</code> and <code>s</code></td>
</tr>
<tr>
<td><code>intersect</code></td>
<td>∩</td>
<td><code>r ∩ q</code></td>
<td>rows that are in <code>r</code> and <code>q</code></td>
</tr>
<tr>
<td><code>groupby</code></td>
<td>γ</td>
<td><code>t.γ (&quot;sensorId&quot;)</code></td>
<td>rows of <code>t</code> grouped by <code>sensorId</code></td>
</tr>
<tr>
<td><code>eproject</code></td>
<td>Π</td>
<td><code>t.Π(avg, &quot;acount&quot;, &quot;count&quot;)(&quot;sensorId&quot;)</code></td>
<td>the average of the <code>count</code> column of <code>t</code></td>
</tr>
<tr>
<td><code>orderBy</code></td>
<td>ω</td>
<td><code>t.ω (&quot;sensorId&quot;)</code></td>
<td>rows of <code>t</code> ordered by <code>sensorId</code></td>
</tr>
<tr>
<td><code>compress</code></td>
<td>ζ</td>
<td><code>t.ζ (&quot;count&quot;)</code></td>
<td>compress the <code>count</code> column of <code>t</code></td>
</tr>
<tr>
<td><code>uncompress</code></td>
<td>Z</td>
<td><code>t.Z (&quot;count&quot;)</code></td>
<td>uncompress the <code>count</code> column of <code>t</code></td>
</tr>
</tbody>
</table>

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.γ ("sensorId").Π (avg, "acount", "count")("sensorId")`

In addition to the natural join shown in Table 3.1 the ScalaTion TSDB also supports equi-join, general theta join, left outer join, and right outer join, as shown below.
Example Queries

Several example queries for the traffic study are given below.

1. 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).

   \[
   \text{val austin} = \text{latLong2UTMxy} \ (\text{LatitudeLongitude} \ (30.266667, -97.733333)) \\
   \text{val alat} = (\text{austin}_1 - 100000, \text{austin}_1 + 100000) \\
   \text{val along} = (\text{austin}_2 - 100000, \text{austin}_2 + 100000) \\
   \text{traffic} \bowtie \sigma [\text{Double}] \ (\text{"latitude"}, _. \in \text{alat}).\sigma [\text{Double}] \ (\text{"longitude"}, _. \in \text{along})
   \]

3.1.2 SQL-Like API

3.1.3 Map-Reduce API
3.2 Preprocessing

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.

3.2.1 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.

3.2.2 Convert String Columns to Numeric Columns

In SCALATION, columns with strings (of type StrNum) 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 mapToInt function in the scalation.linalg.Converter object.

3.2.3 Identify Missing Values

Missing Values are common in 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.

3.2.4 Detect Outliers

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.

SCALATION currently provides the following techniques for outlier detection: so many standard deviation units from the mean, DistanceOutlier; the smallest and largest percent values, QuantileOutlier; and an expansion multiplier beyond the middle two quartiles, QuartileXOutlier. For example, 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.

\[ \text{DistanceOutlier.rmOutlier (traffic.column ("speed"))} \]
3.2.5 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.

**ScalaTion** currently supports the following imputation techniques: average of values before and after, **Interpolate**; value of the vector’s mean, **ImputeMean**; random value generated from Normal distribution, **ImputeNormal**; and average seasonally displaced prior values, **ImputeMovingAverage**.

3.2.6 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.

3.2.7 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.

\[
\text{traffic} \Join_{(0.01)(\text{“time”}, \text{“time”}, \text{weather})} \text{approximate left outer join}
\]

The second operator ...

3.2.8 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.

\[
\text{val mat} = \pi \text{“time”, “count”} (\text{traffic}).\text{toMatriD}
\]

This matrix may then be passed into multiple modeling techniques: (1) a Multiple Linear Regression, (2) an AutoRegressive, Integrated, Moving Average (ARIMA) model.

\[
\text{val model1} = \text{Regression (mat)}
\]
\[
\text{val model2} = \text{ARIMA (mat)}
\]
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.

### 3.3 Exercises

1. 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.

2. Find a dataset at the UCI Machine Learning Repository and carry out the same steps [https://archive.ics.uci.edu/ml/index.php].
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 \( x \) 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:

\[
y = f(x, t; b) + \epsilon
\]

where

- \( y \) is a response/output scalar,
- \( x \) is a predictor/input vector,
- \( t \) is a scalar representing time,
- \( b \) is the vector of parameters of the function, and
- \( \epsilon \) represents remaining error or residuals.

The parameters can be adjusted so that the predictive model matches available data. Note, in the definition of a function, the *arguments* appear before the “;”, while the *parameters* appear after. The input/predictor vector could be random as well, but could also be controlled/designed and thus treated as deterministic. 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 \( y \) and the parameters into a matrix \( B \).

When a model is time-independent or time can be treated as just another dimension within the \( x \) vectors, prediction functions can be represented as follows:

\[
y = f(x; b) + \epsilon
\]
Another way to look at such models, is that we are trying to estimate the conditional expectation of $y$ given $x$.

$$y = \mathbb{E}[y|x] + \epsilon$$

$$\epsilon = y - f(x; b)$$

Given a dataset ($m$ instances of data), each instance contributes to an overall residual/error vector $\epsilon$. One of the simpler ways to estimate the parameters $b$ 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 ($sse$)

$$sse = ||\epsilon||^2 = \epsilon \cdot \epsilon$$

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 $b$. 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 $x$, etc. We consider several cases in the subsections below.
4.1 Predictor

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 either the Predictor trait, or one of two abstract classes extending the Predictor trait, namely PredictorMat or PredictorVec. These two abstract classes also extend the Fit class to enable quality of fit (QoF) evaluation.

Predictor Trait

Trait Methods:

```scala
trait Predictor

  def train (yy: VectoD): Predictor
  def eval (xx: MatriD, yy: VectoD)
  def parameter: VectoD = b
  def residual: VectoD = e
  def predict (z: VectoD): Double
  def predict (z: VectoI): Double = predict (z.toDouble)
```

Implementations of the `train` method take the class data/input matrix `x` and a given response/output vector `yy` and optimize the parameter vector `b` to minimize error or maximize likelihood. Implementations of the `eval` method take a test data/input matrix `xx` and the corresponding test response/output vector `yy` to compute errors and evaluate the Quality of Fit (QoF). Implementations of the `parameter` method simply return the optimized parameter vector `b`. Implementations of the `residual` method return the error vector (difference between actual and predicted values) Implementations of the `predict` method take a data vector (e.g., a new data instance) and predict its response.

4.1.1 Fit

The related Fit class 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 `y`. After the parameters `b` (an `n` vector) have been fit/estimated, the error vector `e` may be calculated. The basic QoF measures involve taking either `L_1` or `L_2` norms of the error vector as indicated in Table 4.1.

<table>
<thead>
<tr>
<th>error/residual</th>
<th>absolute</th>
<th><code>L_1</code> norm</th>
<th>squared</th>
<th><code>L_2</code> norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>sum of absolute errors</td>
<td>`sae =</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>mean</td>
<td>mean absolute error</td>
<td><code>mae^0 = sae/m</code></td>
<td>mean squared error</td>
<td><code>mse^0 = sse/m</code></td>
</tr>
<tr>
<td>unbiased mean</td>
<td>mean absolute error</td>
<td><code>mae = sae/df</code></td>
<td>mean squared error</td>
<td><code>mse = sse/df</code></td>
</tr>
</tbody>
</table>

Table 4.1: Quality of Fit
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 \( b \), in which case \( \epsilon = 0 \), 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, \( df = m - n \).

**Fit Class**

---

**Class Methods:**

```scala
@param y the values in the \( m \)-dimensional response vector
@param n the number of parameters (b.dim)
@param df the degrees of freedom (df._1, df._2) for (regression, error)

class Fit (y: VectoD, n: Int, private var df: PairD = (0.0, 0.0))

def resetDF (df_update: PairD)
def mse_ : Double = mse
def diagnose (e: VectoD, w: VectoD = null, yp: VectoD = null, y_: VectoD = y)
def fit: VectoD = VectorD (rSq, sst, sse, mse0, rmse, mae,
                          df._2, rBarSq, fStat, aic, bic, mape)
def fitLabel: Seq[String] = Seq ("rSq", "sst", "sse", "mse0", "rmse", "mae",
                               "df", "rBarSq", "fStat", "aic", "bic", "mape")
def fitMap: Map[String, String] =
def summary (b: VectoD, stdErr: VectoD = null, show: Boolean = false): String =
```

For modeling, a user chooses one 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 `eval` 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. Some of the quality of fit measures are shown below, see the `Fit` class for details.

```scala
val m = y.dim // number of instances
sse  = e dot e // sum of squared errors
sst  = (y dot y) - y.sum^2 / m // sum of squares total
ssr  = sst - sse // sum of squares regression (not returned)
mse0 = sse / m // raw mean square error
rmse = sqrt (mse0) // root mean square error
mae  = e.norm1 / m // mean absolute error
rSq  = ssr / sst // coefficient of determination R^2
```

Note, `^` is the exponentiation operator provided in Scala/Tion, where the first character is `^` to give the operator higher precedence than multiplication (`*`).

48
The sum of squares total ($sst$) measures the variability of the response $y$,

$$
ssst = y \cdot y - m \mu_y^2 = y \cdot y - \frac{1}{m} [\sum y_i]^2
$$

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.

$$
R^2 = \frac{ssr}{sst} \leq 1
$$

Values for $R^2$ would be nonnegative, unless the proposed model is so bad (worse than the Null Model that simply predicts the mean) that the proposed model actually adds variability.

### 4.1.2 PredictorMat

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-Matrix (or PredictorMat) abstract class takes datasets of this form. Also, using an apply method in PredictorMat’s companion object both $x$ and $y$ may be passed together in a combined data matrix $xy$. As the Predictor trait and the Fit class are used to together in most models, the PredictorMat abstract class merges them.

**PredictorMat Abstract 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
- @param hparam the hyper-parameters for the model

abstract class PredictorMat (protected val x: MatrixD, protected val y: VectorD,
protected var fname: Strings = null,
hparam: HyperParameter = null)

extends Fit (y, x.dim2, (x.dim2-1, x.dim1-x.dim2)) with Predictor with Error

def hparameter: HyperParameter = hparam
def train (yy: VectorD = y): PredictorMat =
def train2 (yy: VectorD = y): PredictorMat =
def eval (xx: MatrixD = x, yy: VectorD = y)
def report: String =
def summary: String =
def predict (z: VectorD): Double = b dot z
def predict (z: MatrixD = x): VectorD = VectorD (for (i <- z.range1) yield predict (z(i)))
1. 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).

2. The *train* method takes the dataset passed into the model (either the full dataset or a training dataset) and optimizes the model parameters $b$.

3. 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.

4. The *eval* method evaluates the Quality of Fit (QoF) either on the full dataset or a designated test dataset that is passed into the *eval* method.

5. The *report* method return a basic report on the model’s hyper-parameters, parameters and overall quality of fit.

6. The *summary* method return a statistical summary for each of the model parameters/coefficients including $t$ and $p$ values for the variable/parameter, where low values for $p$ indicate a strong contribution of the variable/parameter to the model.

7. The two *predict* methods use the parameter vector that results from training to predict the response/output for a new instance/input. One version takes a single instance in the form of vector, while the other overloaded method takes multiple instances in the form of a matrix.

8. 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.

9. The *backwardElim* 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.

10. The *crossValidate* protected method is used by implementing classes to facilitate $k$-fold cross-validation, where a dataset is divided into a training dataset and a test dataset. The training dataset is used by the *train* method, while the test dataset is used by the *eval* method. This is repeated $k$ times.

11. Finally, the *crossVal* method is the actual method called to perform $k$-fold cross-validation, as it customizes the cross-validation to the particular modeling technique (e.g., *Regression*, *RidgeRegression*).
4.1.3 PredictorVec

A few modeling techniques use the `PredictorVec` abstract class rather than the `PredictorMat` class. It is used when a single variable is to be automatically expanded into many variables. Polynomial Regression (or `PolyRegression`) is an example where a vector $t$ is expanded into a matrix consisting of four columns when the order $ord$ is three, namely $[1, t, t^2, t^3]$. This abstract class delegates much of its work to the `Regression` class, which is described later in this chapter.

**PredictorVec Abstract Class**

Class Methods:

- @param t the input vector: $t_i$ expands to $x_i = \text{vector}$
- @param y the response vector
- @param ord the order of the expansion

abstract class PredictorVec (t: VectoD, y: VectoD, ord: Int)
  extends Predictor with Error

```scala
def expand (t: Double): VectoD
def expand (t: VectoD): MatriD =
def train (yy: VectoD = y): Regression = rg.train (yy)
def eval (tt: VectoD, yy: VectoD) { rg.eval (expand (tt), yy) }
def eval (xx: MatriD, yy: VectoD = y) { rg.eval (xx, yy) }
override def parameter: VectoD = rg.parameter
override def residual: VectoD = rg.residual
def fit: VectoD = rg.fit
def fitLabel: Seq [String] = rg.fitLabel
def fitMap: Map [String, String] = rg.fitMap
def predict (z: Double): Double
def predict (z: VectoD): Double = rg.predict (z)
def forwardSel (cols: Set [Int], adjusted: Boolean): (Int, VectoD, VectoD) = rg.forwardSel (cols, adjusted)
def backwardElim (cols: Set [Int], adjusted: Boolean, first: Int): (Int, VectoD, VectoD) = rg.backwardElim (cols, adjusted)
def vif: VectoD = rg.vif
protected def crossValidate (algor: (VectoD, VectoD, Int) => PredictorVec, k: Int = 10, rando: Boolean = true): Array [Statistic] =
def crossVal (ord: Int, k: Int = 10, rando: Boolean = true): Array [Statistic]
```
4.2 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.

\[
y = \mathbb{E}[y] + \epsilon
\]

This could happen if the predictors \( x \) are not relevant, not collected in a useful range or the relationship is too complex for the modeling techniques you have applied. Ignoring the predictor variables gives the following simple model.

\[
y = b_0 + \epsilon
\] (4.3)

The optimal solution for the parameter vector \( b \) is simple to compute. It can be shown that (see exercises) the optimal value for the parameter is

\[
b_0 = \mu_y
\] (4.4)

In ScalaTion this requires just one line of code inside the `train` method.

```scala
def train (yy: VectoD = y): NullModel =
{
  b = VectorD (yy.mean)  // parameter vector [b0]
  this
} // train
```

The `eval` method will compute the residual/error vector \( \epsilon \) and then call the `diagnose` method.

```scala
def eval (xx: MatriD = null, yy: VectoD = y)
{
  e = yy - b(0)
  diagnose (e)
} // eval
```

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.

```scala
def predict (z: VectoD): Double = b(0)
```

NullModel Class

Class Methods:
@param y the response vector

class NullModel (y: VectoD)
    extends Fit (y, 1, (1, y.dim)) with Predictor with Error

def hparameter: HyperParameter = null
def train (yy: VectoD = y): NullModel =
def eval (xx: MatriD = null, yy: VectoD = y)
def predict (z: VectoD): Double = b(0)
def predict (z: MatriD): VectoD = VectorD.fill (z.dim1)(b(0))

4.2.1 Exercises

1. Determine the value for the parameter $b_0$ that minimizes the sum of squared errors $sse = \epsilon \cdot \epsilon$.

2. Let the response vector $y$ be

   val y = VectorD (1, 3, 3, 4)

and execute the NullModel.
For context, assume the corresponding predictor vector $y$ is

   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 $y$.

   val y = VectorD (2.0, 3.0, 5.0, 4.0, 6.0) // response vector y
   println (s"y = $y")
   val rg = new NullModel (y) // create a NullModel
   rg.train ().eval () // train on data and evaluate
   println (rg.report) // parameter values and QoF
   val z = VectorD (5.0) // predict y for one point
   val yp = rg.predict (z) // yp (y-predicted or y-hat)
   println (s"predict ($z) = $yp")

4. Execute the NullModel on the Auto MPG dataset. See scalation.analytics.Auto MPG_Regression.
What is the quality of the fit (e.g., $R^2$ or rSq)? Is this value expected? Is it possible for a model to perform worse than this?
4.3 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., $\mathbf{x} = [x_0]$. The goal is to fit the parameter vector $\mathbf{b} = [b_0]$ in the regression equation

$$ y = b_0 x_0 + \epsilon $$

(4.5)

where $\epsilon$ represents the residuals/errors (the part not explained by the model).

A dataset or training set 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.

$$ \mathbf{y} = b_0 \mathbf{x}_0 + \epsilon $$

(4.6)

One way to find a value for $b_0$ is to minimize the norm of residual/error vector $\epsilon$.

$$ \min_{b_0} ||\epsilon|| $$

Since $\epsilon = \mathbf{y} - b_0 \mathbf{x}_0$, we may solve the following optimization problem:

$$ \min_{b_0} ||\mathbf{y} - b_0 \mathbf{x}_0|| $$

This is equivalent to minimizing the dot product ($||\epsilon||^2 = \mathbf{\epsilon} \cdot \mathbf{\epsilon} = sse$)

$$ (\mathbf{y} - b_0 \mathbf{x}_0) \cdot (\mathbf{y} - b_0 \mathbf{x}_0) $$

4.3.1 Derivative

Taking the derivative $\frac{d}{db_0}$ using the derivative product rule (for dot products)

$$ (f \cdot g)' = f' \cdot g + f \cdot g' $$

(4.7)

and setting it equal to zero yields the following equation.

$$ -2 \mathbf{x}_0 \cdot (\mathbf{y} - b_0 \mathbf{x}_0) = 0 $$

Therefore, the optimal value for the parameter $b_0$ is

$$ b_0 = \frac{\mathbf{x}_0 \cdot \mathbf{y}}{\mathbf{x}_0 \cdot \mathbf{x}_0} $$

(4.8)

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 $b_0 = \frac{16}{15}$, we may obtain predicted values for each of the $x$-values.

$$ \hat{y} = \hat{y} = predict(\mathbf{x}_0) = b_0 \mathbf{x}_0 = [1.067, 2.133, 3.200, 4.267] $$

Therefore, the error/residual vector is

\[54\]
\[ \epsilon = y - \hat{y} = [1, 3, 3, 4] - [1.067, 2.133, 3.200, 4.267] = [-0.067, 0.867, -0.2, -0.267]. \]

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.

**SimplerRegression Class**

Class Methods:

- @param x the data/input matrix
- @param y the response/output vector
- @param fname_ the feature/variable names

```scala
class SimplerRegression (x: MatrixD, y: VectorD, fname_ : Strings = null) extends PredictorMat (x, y, fname_)
```

```scala
def train (yy: VectorD = y): SimplerRegression =
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
```

**4.3.2 Exercises**

1. For \( x_0 = [1, 2, 3, 4] \) and \( y = [1, 3, 3, 4] \), try various values for the parameter \( b_0 \). Plot the sum of squared errors (sse) vs. \( b_0 \).

```scala
import scalation.linalgebra.VectorD
import scalation.plot.Plot

object SimplerRegression_exer_1 extends App {
  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.range) {
    val e = ?
    sse(i) = e dot e
  } // for
  new Plot (b0, sse, lines = true)
} // SimplerRegression_exer_1 object
```
Where do you think the minimum occurs?

Note, to run your code you should make a separate project directory. See [https://alvinalexander.com/scala/how-to-create-sbt-project-directory-structure-scala](https://alvinalexander.com/scala/how-to-create-sbt-project-directory-structure-scala) for the directory structure. Copy the `scalation_mathstat` jar file `scalation_mathstat_2.12-1.5.jar` into your lib directory. Create a file called `SimplerRegression_exer_1.scala` in the `src/main/scala` directory. In the project’s base directory, type `sbt`. Within `sbt` type `compile` and then `run`.

2. From the $X$ matrix and $y$ vector, plot the set of data points \( \{(x_i, y_i) | 0 \leq i < m\} \) and draw the line that is nearest to these points. What is the slope of this line? Pass the $X$ matrix and $y$ vector as arguments to the `SimplerRegression` class to obtain the $b = [b_0]$ vector.

```scala
// 4 data points: x0
val x = new MatrixD((4, 1), 1.0, // x 4-by-1 matrix
                        2.0,
                        3.0,
                        4.0)
val y = VectorD(1.0, 3.0, 3.0, 4.0) // y vector

val rg = new SimplerRegression(x, y) // create a SimplerRegression
rg.train().eval() // train on data and evaluate
println("parameter = " + rg.parameter) // parameter/coefficient values
println(" = " + rg.fitMap) // quality of fit
```

3. Redo the last exercise using a spreadsheet by making columns for each vector: $x_0, y, \hat{y}, \epsilon,$ and $\epsilon^2$. Sum the last column to obtain the sum of squared errors $sse$. The sum of squares total $sst$ is the same as the result for the `NullModel` see the Exercise 4.2.1:1. Finally, compute the coefficient of determination $R^2$.

$$R^2 = 1 - \frac{sse}{sst}$$

4. From the $X$ matrix and $y$ vector, plot the set of data points \( \{(x_i, y_i) | 0 \leq i < m\} \) 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 $y$ vector as arguments to the `SimplerRegression` class to obtain the $b = [b_0]$ vector.

```scala
// 5 data points: x0
val x = new MatrixD((5, 1), 0.0, // x 5-by-1 matrix
                        1.0,
                        2.0,
                        3.0,
                        4.0)
val y = VectorD(2.0, 3.0, 5.0, 4.0, 6.0) // y vector

val rg = new SimplerRegression(x, y) // create a SimplerRegression
rg.train().eval() // train on data and evaluate
```
println ("parameter = " + rg.parameter)  // parameter values
println ("fitMap = " + rg.fitMap)    // quality of fit

val z = VectorD (5.0)                 // predict y for one point
val yp = rg.predict (z)              // y-predicted
println (s"predict ($z) = $yp")

5. Execute the SimplerRegression on the Auto MPG dataset. See scalation.analytics.ExampleAuto_MPG.
What is the quality of the fit (e.g., $R^2$ or $r^2$)? Is this value expected? What does it say about this model? Try using different columns for the predictor variable.
4.4 Simple Regression

The SimpleRegression class supports simple linear regression. In this case, the predictor vector \( x \) consists of the constant one and a single variable \( x_1 \), i.e., \([1, x_1]\). The goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = [b_0, b_1] \cdot [1, x_1] + \epsilon = b_0 + b_1 x_1 + \epsilon
\]

(4.9)

where \( \epsilon \) represents the residuals (the part not explained by the model). Given \( m \) data points/vectors, stored row-wise in an \( m \)-by-2 matrix \( X \) and \( m \) response values, stored in an \( m \) dimensional vector \( y \), solve the following optimization problem,

\[
\min_b \| \epsilon \|
\]

Substituting \( \epsilon = y - Xb \) yields

\[
\min_b \| y - Xb \|
\]

\[
\min_{[b_0, b_1]} \| y - [1 x_1] \|_{b_1}^2
\]

\[
\min_{[b_0, b_1]} \| y - (b_0 1 + b_1 x_1) \|
\]

This is equivalent to minimizing the dot product (\( \| \epsilon \|^2 = \epsilon \cdot \epsilon = sse \))

\[
(y - (b_0 1 + b_1 x_1)) \cdot (y - (b_0 1 + b_1 x_1))
\]

Since \( x_0 \) is just \( 1 \), for simplicity we drop the subscript on \( x_1 \).

\[
(y - (b_0 1 + b_1 x)) \cdot (y - (b_0 1 + b_1 x))
\]

4.4.1 Gradient

Taking the gradient \( \nabla = \left[ \frac{\partial}{\partial b_0}, \frac{\partial}{\partial b_1} \right] \) of \( sse \) using the derivative product rule and setting it equal to zero yields two equations.

\[
\nabla sse(b) = \left[ \frac{\partial sse}{\partial b_0}, \frac{\partial sse}{\partial b_1} \right] = 0
\]

Partial Derivative w.r.t. \( b_0 \)

The first equation results from setting \( \frac{\partial}{\partial b_0} \) of \( sse \) to zero.

\[
-21 \cdot (y - (b_0 1 + b_1 x)) = 0
\]

\[
1 \cdot y - 1 \cdot (b_0 1 + b_1 x) = 0
\]

\[
b_0 1 \cdot 1 = 1 \cdot y - b_1 1 \cdot x
\]

Since \( 1 \cdot 1 = m \), \( b_0 \) may be expressed as

\[
b_0 = \frac{1 \cdot y - b_1 1 \cdot x}{m}
\]

(4.10)
Partial Derivative w.r.t. $b_1$

Similarly, the second equation results from setting $\frac{\partial}{\partial b_1}$ of sse to zero.

\[-2x \cdot (y - (b_01 + b_1x)) = 0\]
\[x \cdot y - x \cdot (b_01 + b_1x) = 0\]
\[b_01 \cdot x + b_1x \cdot x = x \cdot y\]

Multiplying by both sides by $m$ produces

\[mb_01 \cdot x + mb_1x \cdot x = m \cdot x \cdot y\]

Substituting for $mb_0 = 1 \cdot y - b_11 \cdot x$ yields

\[[1 \cdot y - b_11 \cdot x][1 \cdot x + mb_1x \cdot x = m \cdot x \cdot y\]
\[b_0[m \cdot x \cdot x - (1 \cdot x)^2] = m \cdot x \cdot y - (1 \cdot x)(1 \cdot y)\]

Solving for $b_1$ gives

\[b_1 = \frac{m \cdot x \cdot y - (1 \cdot x)(1 \cdot y)}{m \cdot x \cdot x - (1 \cdot x)^2}\]

The $b_0$ parameter gives the intercept, while the $b_1$ parameter gives the slope of the line that best fits the data points. 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) $b_1$.

\[
b_1 = \frac{4[1,2,3,4] \cdot [1,3,3,4] - (1 \cdot [1,2,3,4]) (1 \cdot [1,3,3,4])}{4[1,2,3,4] \cdot [1,2,3,4] - (1 \cdot [1,2,3,4])^2} = \frac{128 - 110}{120 - 100} = \frac{18}{20} = 0.9
\]

\[
b_0 = \frac{1 \cdot [1,3,3,4] - 0.9(1 \cdot [1,2,3,4])}{4} = \frac{11 - 0.9 \cdot 10}{4} = 0.5
\]

Concise Formulas for the Parameters

More concise and intuitive formalas for the parameters $b_0$ and $b_1$ may be derived.

- Using the definition for mean from section 2.2.1 for $\mu_x$ and $\mu_y$, it can be shown that the expression for $b_0$ shortens to

\[b_0 = \mu_y - b_1 \mu_x\]  \hfill (4.12)

Draw a line through the following two points $[0, b_0]$ (the intercept) and $[\mu_x, \mu_y]$ (the center of mass). How does this line compare to the regression line.

- Now, using the definitions for covariance $\sigma_{x,y}$ and variance $\sigma_x^2$ from section 2.2.1, it can be shown that the expression for $b_1$ shortens to

\[b_1 = \frac{\sigma_{x,y}}{\sigma_x^2}\]  \hfill (4.13)

If the slope of the regression line is simply the ratio of the covariance to the variance, what would the slope be if $y = x$. 

59
SimpleRegression Class

Class Methods:

@param x the data/input matrix augmented with a first column of ones
@param y the response/output vector
@param fname_ the feature/variable names

class SimpleRegression (x: MatD, y: VecD, fname_ : Strings = null) extends PredictorMat (x, y, fname_)

def train (yy: VecD = y): SimpleRegression =
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]

4.4.2 Exercises

1. From the $X$ matrix and $y$ vector, plot the set of data points \{$(x_i, y_i) | 0 \leq i < m$\} and draw the line that is nearest to these points (i.e., that minimize $||\epsilon||$) Using the formulas from the previous question, what are the intercept and slope $[b_0, b_1]$ of this line.

Also, pass the $X$ matrix and $y$ vector as arguments to the SimpleRegression class to obtain the $b$ vector.

```scala
// 4 data points: constant x1
val x = new MatrixD ((4, 2), 1.0, 1.0, // x 4-by-2 matrix
                    1.0, 2.0,
                    1.0, 3.0,
                    1.0, 4.0)
val y = VectorD (1.0, 3.0, 3.0, 4.0) // y vector
val rg = new SimpleRegression (x, y) // create a SimpleRegression
rg.train ().eval () // train on data and evaluate
println ("parameter = " + rg.parameter) // parameter values
println ("fitMap = " + rg.fitMap) // quality of fit
```

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 $b$, found in the previous problem. Compute values for $sse$ for each point in the grid. Plot $sse$ versus $b_0$ across the optimal point. Do the same for $b_1$. 60
For gradient descent, pick a starting point $b^0$, compute the gradient $\nabla sse$ and move $-\eta \nabla sse$ from $b^0$ where $\eta$ is the learning rate. Repeat for a few iterations. What is happening to the value of $sse$.

$$\nabla sse = [-2 \mathbf{1} \cdot (y - (b_0 \mathbf{1} + b_1 \mathbf{x})), -2 \mathbf{x} \cdot (y - (b_0 \mathbf{1} + b_1 \mathbf{x}))]$$

Substituting $\epsilon = y - (b_0 \mathbf{1} + b_1 \mathbf{x})$, half $\nabla sse$ may be written as

$$[-1 \cdot \epsilon, -\mathbf{x} \cdot \epsilon]$$

3. From the $X$ matrix and $y$ vector, plot the set of data points $\{(x_i, y_i) | 0 \leq i < m\}$ and draw the line that is nearest to these points. What are the intercept and slope of this line. Pass the $X$ matrix and $y$ vector as arguments to the `SimpleRegression` class to obtain the $b$ vector.

```scala
// 5 data points: constant  x1
val x = new MatrixD((5, 2), 1.0, 0.0, // x 5-by-2 matrix
                    1.0, 1.0,
                    1.0, 2.0,
                    1.0, 3.0,
                    1.0, 4.0)
val y = VectorD(2.0, 3.0, 5.0, 4.0, 6.0) // y vector
val rg = new SimpleRegression(x, y) // create a SimpleRegression
rg.train().eval() // train on data and evaluate
println("parameter = "+ rg.parameter) // parameter values
println("fitMap = "+ rg.fitMap) // quality of fit
val z = VectorD(1.0, 5.0) // predict y for one point
val yp = rg.predict(z) // y-predicted
println(s"predict ($z) = $yp")
```

4. Execute the `SimpleRegression` on the Auto MPG dataset. See `scalation.analytics.ExampleAuto_MPG`. 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.
4.5 Regression

The Regression class supports multiple linear regression. In this case, the predictor vector \( x \) is multi-dimensional \([1, x_1, \ldots, x_k]\). The goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + \epsilon \quad (4.14)
\]

where \( \epsilon \) represents the residuals (the part not explained by the model).

Using several data samples as a training set, the Regression class in ScalaTion can be used to estimate the parameter vector \( b \). Each sample pairs an \( x \) input vector with a \( y \) response value. The \( x \) vectors are placed into a data/input matrix \( X \) row-by-row with a column of ones as the first column in \( X \). The individual response values taken together form the response vector \( y \). The matrix-vector product \( Xb \) provides an estimate for the response vector.

\[
y = Xb + \epsilon
\]

The goal is to minimize the distance between \( y \) and its estimate. i.e., minimize the norm of residual/error vector.

\[
\min_b \| \epsilon \|
\]

Substituting \( \epsilon = y - Xb \) yields

\[
\min_b \| y - Xb \|
\]

This is equivalent to minimizing the dot product \((\| \epsilon \|^2 = \epsilon \cdot \epsilon = sse)\)

\[
(y - Xb) \cdot (y - Xb)
\]

\[
(y - Xb)^t(y - Xb)
\]

Taking the gradient \( \nabla sse \) with respect to the parameter vector \( b \) and setting it equal to the zero vector yields

\[
-2X^t(y - Xb) = 0
\]

\[
-2X^t y + 2X^t Xb = 0
\]

A more detailed derivation of this equation is given in section 3.4 of “Matrix Calculus: Derivation and Simple Application” [11]. Dividing the equation by 2 and moving the term involving \( b \) to the left side, results in the Normal Equations.

\[
X^t Xb = X^t y \quad (4.15)
\]

Note: equivalent to minimizing the distance between \( y \) and \( Xb \) is minimizing the sum of the squared residuals/errors (Least Squares method).

ScalaTion provides five techniques for solving for the parameter vector \( b \) based on the Normal Equations: Matrix Inversion, LU Factorization, Cholesky Factorization, QR Factorization and SVD Factorization.
4.5.1 Matrix Inversion Technique

Starting with the Normal Equations

\[ X^t X b = X^t y \]

a simple technique is Matrix Inversion, which involves computing the inverse of \( X^t X \) and using it to multiply both sides of the Normal Equations.

\[ b = (X^t X)^{-1} X^t y \]  \hspace{1cm} (4.16)

where \((X^t X)^{-1}\) is an \( n \)-by-\( n \) matrix, \( X^t \) is an \( n \)-by-\( m \) matrix and \( y \) is an \( m \)-vector. The expression involving the \( X \) matrix is referred to as the pseudo-inverse \( X^{-1} \).

\[ X^{-1} = (X^t X)^{-1} X^t \]

Using the pseudo-inverse, the parameter vector \( b \) may be solved for as follows:

\[ b = X^{-1} y \]

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 \( (x \text{.pinv}) \) looks similar to the math.

```scala
val x_pinv = (x.t * x).inverse * x.t
```

A more robust approach is

```scala
val fac = new Fac_Inv (x.t * x).factor ()
val x_pinv = fac.factors._2 * x.t
```

For efficiency, the code in Regression does not calculate \( x \text{.pinv} \), rather is directly solves for the parameters \( b \).

```scala
val b = fac.solve (x.t * y)
```

4.5.2 LU Factorization Technique

Lower, Upper 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 \( X^t X \), an \( n \)-by-\( n \) matrix, is factored

\[ X^t X = LU \]

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

\[ LU b = X^t y \]

Letting \( w = U b \) allows the problem to solved in two steps. The first is solved by forward substitution to determine the vector \( w \).
Finally, the parameter vector $b$ is determined by backward substitution.

$$Ub = w$$

### 4.5.3 Cholesky Factorization Technique

A faster and slightly more stable technique is to use Cholesky Factorization. Since the product $X^tX$ is a positive definite, symmetric matrix, it may factored using Cholesky Factorization into

$$X^tX = LL^t$$

where $L$ is a lower triangular $n$-by-$n$ matrix. Then the normal equations may be rewritten

$$LL^tb = X^ty$$

Letting $w = L^tb$, we may solve for $w$ using forward substitution

$$Lw = X^ty$$

and then solve for $b$ using backward substitution.

$$L^tb = w$$

### 4.5.4 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,

$$X^tXb = X^ty$$

simply substitute $QR$ for $X$.

$$(QR)^tQRb = (QR)^ty$$

Taking the transpose gives

$$R^tQ^tQRb = R^tQ^ty$$

and using the fact that $Q^tQ = I$, we obtain the following:

$$R^tRb = R^tQ^ty$$

Multiply both sides by $(R^t)^{-1}$ yields
\[ Rb = Q'y \]

Since \( R \) is an upper triangular matrix, the parameter vector \( b \) can be determined by backward substitution. Alternatively, the pseudo-inverse may be computed as follows:

\[ X^{-1} = R^{-1}Q' \]

SCALATION uses Householder Orthogonalization (alternately Modified Gram-Schmidt Orthogonalization) to factor \( X \) into the product of \( Q \) and \( R \).

### 4.5.5 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](http://www.ime.unicamp.br/~marianar/MI602/material%20extra/svd-regression-analysis.pdf), we start with the equation estimating \( y \) as the product of the data matrix \( X \) and the parameter vector \( b \).

\[ y = Xb \]

We then perform a singular value decomposition on the \( m \)-by-\( n \) matrix \( X \)

\[ X = U\Sigma V^t \]

where in the full-rank case, \( U \) is an \( m \)-by-\( n \) orthogonal matrix, \( \Sigma \) is an \( n \)-by-\( n \) diagonal matrix of singular values, and \( V^t \) is an \( n \)-by-\( n \) orthogonal matrix. The \( r = \text{rank}(A) \) equals the number of nonzero singular values in \( \Sigma \), so in general, \( U \) is \( m \)-by-\( r \), \( \Sigma \) is \( r \)-by-\( r \), and \( V^t \) is \( r \)-by-\( n \). The singular values are the square roots of the nonzero eigenvalues of \( X^tX \). Substituting for \( X \) yields

\[ y = U\Sigma V^t b \]

Defining \( d = \Sigma V^t b \), we may write

\[ y = Ud \]

This can be viewed as a estimating equation where \( X \) is replaced with \( U \) and \( b \) is replaced with \( d \). Consequently, a least squares solution for the alternate parameter vector \( d \) is given by

\[ d = (U^tU)^{-1}U^ty \]

Since \( U^tU = I \), this reduces to

\[ d = U^ty \]

If \( \text{rank}(A) = n \) (full-rank), then the conventional parameters \( b \) may be obtained as follows:

\[ b = V\Sigma^{-1}d \]

where \( \Sigma^{-1} \) is a diagonal matrix where elements on the main diagonal are the reciprocals of the singular values.
4.5.6 Use of Factorization in Regression

By default, SCALaTION uses QR Factorization to compute the pseudo-inverse $X^{-1}$. The other techniques may be selected by using the third argument (technique) in the constructor, setting it to Cholesky, SVD, LU or Inverse. For more information see [http://see.stanford.edu/materials/lsoeldsee263/05-ls.pdf](http://see.stanford.edu/materials/lsoeldsee263/05-ls.pdf).

```scala
object RegTechnique extends Enumeration {
  type RegTechnique = Value
  val QR, Cholesky, SVD, LU, Inverse = Value
  val techniques = Array (QR, Cholesky, SVD, LU, Inverse)
}

import RegTechnique._

Based on the selected technique, the appropriate type of matrix factorization is performed. The first part of the code below constructs a factorization object fac, and then calls the factor method on it.

```scala
protected val fac: Factorization = technique match { // select factorization technique
  case QR => new Fac_QR (x, false) // QR Factorization
  case Cholesky => new Fac_Cholesky (x.t * x) // Cholesky Factorization
  case SVD => new SVD (x) // Singular Value Decomposition
  case LU => new Fac_LU (x.t * x) // LU Factorization
  case _ => new Fac_Inv (x.t * x) // Inverse Factorization
}

fac.factor () // factor the matrix, X or X.t * X
```

The `train` method below computes parameter/coefficient vector $b$ by calling the solve method provided by the factorization classes.

```scala
def train (yy: VectoD = y): Regression =
{
  b = technique match { // solve for coefficient vector b
    case QR => fac.solve (yy) // R * b = Q.t * yy
    case Cholesky => fac.solve (x.t * yy) // L * L.t * b = X.t * yy
    case SVD => fac.solve (yy) // b = V * ^-1 * U.t * yy
    case LU => fac.solve (x.t * yy) // b = (X.t * X) \ X.t * yy
    case _ => fac.solve (x.t * yy) // b = (X.t * X)^-1 * X.t * yy
  }
}
```

After training, the `eval` method overridden for efficiency from PredictorMat does two things: First, the residual/error vector $\epsilon$ is computed. Second, several quality of fit measures are computed by calling the `diagnose` method.
override def eval (xx: MatriD = x, yy: VectoD = y)
{
    e = yy - xx * b // compute residual/error vector e
    diagnose (e)    // compute diagnostics
} // eval

See Exercise 2 to see how to train and eval a Regression model.

4.5.7 Model Assessment

The quality of fit measures includes the coefficient of determination $R^2$ as well as several others. Given $m$ instances (data points) and $n$ parameters in the regression model, the degrees of freedom captured by the regression model is $dfr$ and left for error is $df$.

$$
dfr = n - 1 = k
$$
$$
df = m - n
$$

If the model is without an intercept, $dfr = n$. The ratio of total degrees of freedom to degrees of freedom for error is

$$
r_{df} = \frac{dfr + df}{df}
$$

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 $\hat{R}^2$

$$
\hat{R}^2 = 1 - r_{df}(1 - R^2)
$$

Dividing $sse$ and $ssr$ by their respective degrees of freedom gives the mean square error and regression, respectively

$$
msr = \frac{sse}{df}
$$
$$
msr = \frac{ssr}{df}
$$

The mean square error $mse$ follows a Chi-squared distribution with $df$ degrees of freedom, while the mean square regression $msr$ follows a Chi-squared distribution with $dfr$ degrees of freedom. Consequently, the ratio $\frac{msr}{mse}$ follows an $F$-distribution with $(dfr, df)$ degrees of freedom. If this number exceeds the critical value, one can claim that the parameter vector $b$ 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 SCALAion the following quantities are computed by the diagnose method in the Fit class.

```scala
def diagnose (e: VectoD, yy: VectoD, yp : VectoD = null, w: VectoD = null)
{
    val m = yy.dim // size of the response vector (test/full)
    if (e.dim != m) flaw ("diagnose", s"e.dim = ${e.dim} != yy.dim = $m")

    sse = e dot e // sum of squares for error
    if (w == null) {
```
```scala
sst = (yy dot yy) - yy.sum^2 / m // sum of squares total (ssr + sse)
ssr = sst - sse // sum of squares regression/model
}
else {
  ssr = (w * (yp - (w * yp / w.sum).sum)^2).sum // regression sum of squares
  sst = ssr + sse
}
// if
mse0 = sse / m // raw/ML mean squared error
rmse = sqrt (mse0) // root mean squared error
mae = e.norm1 / m // mean absolute error
rSq = ssr / sst // coefficient of determination (quality of fit)
if (df._1 <= 0 || df._2 <= 0) flaw ("diagnose", s"degrees of freedom df = $df must be positive")
mse = sse / df._2 // mean of squares for error
msr = ssr / df._1 // mean of squares for regression/model
rse = sqrt (mse) // residual standard error
rBarSq = 1 - (1-rSq) * r_df // adjusted R-squared
fStat = msr / mse // F statistic (quality of fit)
p_fS = 1.0 - fisherCDF (fStat, df._1.toInt, df._2.toInt) // p-value for fStat
if (p_fS.isNaN) p_fS = 0.0
aic = m * log (mse0) + 2 * n // Akaike Information Criterion = -2LL + 2n
bic = aic + n * (ln_m - 2) // Bayesian Information Criterion = -2LL + n ln(m)
mape = (e / yy).norm1 / m // mean absolute percentage error / 100
} // diagnose
```

### 4.5.8 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. 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 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 $\bar{R}^2$ 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 dataset and a testing dataset.

**Leave-One-Out**

When fitting the parameters $b$ 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 $y_t - b \cdot x_t$. 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. [12].

**$k$-Fold Cross-Validation**

A more generally applicable strategy is called cross-validation, where a dataset is divided into $k$ test datasets. For each test dataset, the corresponding training dataset is all the instances not chosen for that test dataset. 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.

```plaintext
val tsize = m / k // test dataset size
for (l <- 0 until k) {
    x_te = x.slice (l * tsize, ((l+1) * tsize)) // l-th test dataset
    x_tr = x.sliceEx (l * tsize, ((l+1) * tsize)) // l-th training dataset
} // for
```

The model is trained $k$ times using each of the training datasets. The corresponding test dataset is then used to estimate the test $sse$ (or other quality measure such as $mse$). 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 datasets.

Typically, training QoF measures such as $R^2$ will be better than testing QoF measures such as $R^2_{cv}$. Adjusted measures such as $R^2$ are intending to more closely follow $R^2_{cv}$ than $R^2$.

**4.5.9 Collinearity**

Consider the matrix-vector equation used for estimating the parameters $b$ via the minimization of $||\epsilon||$.

$$ y = Xb + \epsilon $$

The parameter/coefficient vector $b = [b_0, b_1, \ldots, b_k]$ may be viewed as weights on the column vectors in the data/predictor matrix $X$.

$$ y = b_01 + b_1x_{-1} + \ldots + b_kx_{-k} + \epsilon $$

69
A question arises when two of these column vectors are nearly the same. 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 $y$. This is the same example used for `SimpleRegression` in Exercise 2 with new variable $x_2$ added (i.e., $y = b_0 + b_1 x_1 + b_2 x_2 + \epsilon$).

```scala
val x = new MatrixD ((4, 3), 1, 1, 1,
                        1, 2, 2,
                        1, 3, 3,
                        1, 4, -1)

val y = VectorD (1, 3, 3, 4)
```

Try changing the value of element $x_{32}$ from 1 to 4 by .5 and observe what happens to the correlation matrix. What effect do these changes have on the parameter values $b_0$, $b_1$, and $b_2$?

```scala
println (s"corr (x) = ${corr (x)}")
```

The `corr` function is provided by the `scalation.stat.StatVector` object. For this function, if either column vector has zero variance, when the column vectors are the same, it returns 1.0, otherwise -0.0 (indicating undefined).

**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 function $vif$. For a particular parameter $b_j$ for the variable/predictor $x_j$, the function is evaluated as follows:

$$vif(b_j) = \frac{1}{1 - R^2(x_j)}$$

where $R^2(x_j)$ is $R^2$ for the regression of variable $x_j$ onto rest of the predictors. It measures how well the variable $x_j$ (or its vector $x_j$) can be predicted by all $x_l$ for $l \neq j$. Values above 10 are considered problematic. 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` function. Another is to use regularized regression such as `RidgeRegression` or `LassoRegression`.

### 4.5.10 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 $x_j$ ($j^{th}$ column of matrix $X$) with the response vector $y$. A more intuitive way to do this would be to plot the response vector $y$ versus each predictor variable $x_j$. 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 are forward selection and backward elimination.

**Forward Selection**

The `forwardSel` method performs *forward selection* by adding the most predictive variable to the existing model, returning the variable to add, the new parameter vector and the new quality of fit.

```java
def forwardSel (cols: Set[Int], adjusted: Boolean = true): (Int, VectoD, VectoD)
```

Selecting the most predictive variable to add boils down to comparing on the basis of a quality of fit measure. The default is the Adjusted Coefficient of Determination $\bar{R}^2$. Setting the optional parameter `adjusted` to `false` will cause $R^2$ to be used instead. To start with a minimal model, set `cols = Set (0)` for an intercept-only model. The method will consider every variable/column $x$.range2 not already in `cols` and pick the best one for inclusion.

```java
for (j <- x.range2 if ! (cols contains j)) {
    // To find the best model, the forwardSel method should be called repeatedly while the quality of fit measure is sufficiently improving.
}
```

**Backward Elimination**

The `backwardElim` method performs *backward elimination* by removing the least predictive variable from the existing model, returning the variable to eliminate, the new parameter vector and the new quality of fit.

```java
def backwardElim (cols: Set[Int], adjusted: Boolean = true, first: Int = 1):
    (Int, VectoD, VectoD)
```

To start with a maximal model, set `cols = Set (0, 1, ..., k)` for a full model. As with `forwardSel` the `adjusted` optional parameter allows toggling between $\bar{R}^2$ and $R^2$. The second 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.

```java
for (j <- first until x.dim2 if cols contains j) {
    // To find the best model, the backwardElim method should be called repeatedly until the quality of fit measure sufficiently decreases.
}
```

More advanced 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`. 71
Categorical Variables/Features

For **Regression**, the variables/features are treated as continuous/ordinal. Some of the variable may be categorical, where there is no ordering of the values for a categorical variable. In such cases it may be useful to replace a categorical variable with multiple dummy, binary variables. For details on how to do this, see the section on **ANCOVA**.

**Regression 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-vector
@param fname_ the feature/variable names
@param hparam the hyper-parameters (currently none)
@param technique the technique used to solve for b in x.t*x*b = x.t*y

class Regression (x: MatriD, y: VectoD, fname_ : Strings = null,
                  hparam: HyperParameter = null, technique: RegTechnique = QR)
    extends PredictorMat (x, y, fname_, hparam)
```

```scala
def getX = x
def train (yy: VectoD = y): Regression =
override def eval (xx: MatriD = x, yy: VectoD = y): Regression =
def forwardSel (cols: Set [Int], adjusted: Boolean = true):
    (Int, VectoD, VectoD) =
def backwardElim (cols: Set [Int], adjusted: Boolean = true, first: Int = 1):
    (Int, VectoD, VectoD) =
def vif: VectoD =
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
```

### 4.5.11 Exercises

1. For Exercise 2 from the last section, compute $A = X^t X$ and $z = X^t y$. Now solve the following linear systems of equations for $b$.

   $$Ab = z$$

2. Solving a regression problem in **SCALATION** simply involves creating a data/input matrix $X$ and response/output vector $y$ and then creating a **Regression** object upon which `train`, `eval`, `parameter` and `fitMap` methods are called. The Texas Temperature data-set below from [http://www.stat.ufl.edu/~winner/cases/txtemp.ppt](http://www.stat.ufl.edu/~winner/cases/txtemp.ppt) is used to illustrate how to use **SCALATION** for a regression problem.
// 16 data points: Constant x1 x2 x3
// Lat Elev Long County
val x = new 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, 30.367, 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)
val rg = new Regression (x, y)
rg.train ().eval ()
println ("parameter = " + rg.parameter)
println ("full mod fit = " + rg.fitMap)

More details about the parameters/coefficients including standard error, t-values and p-values are shown by the summary method.

println (rg.summary)

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) = ${rg.predict (z)}")

Feature selection may be carried out by using either forwardSel or backwardElim.

println ("reduced mod fit = " + rg.backwardElim (cols))

The source code for this example is at

3. Gradient descent can be used for Multiple Linear Regression as well. For gradient descent, pick a starting point b^0, compute the gradient ∇sse and move −η∇sse from 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 \( sse \).

\[
\nabla sse = -2X^t(y - Xb)
\n\]

Substituting \( \epsilon = y - Xb \), half \( \nabla sse \) may be written as

\[-X^t\epsilon\]

Starting with data matrix \( x \), response vector \( y \) and parameter vector \( b \), in SCALATiON, the calculations become

```scala
val yp = x * b // y predicted
val e = y - yp // error
val g = x.t * e // - gradient
b += g * eta // update parameter b
val sse = e dot e // sum of squared errors
```

Unless the dataset is normalized, finding an appropriate learning rate \( \eta \) may be difficult. See the MatrixTransform object for details.

4. Consider the relationships between the predictor variables and the response variable in the AutoMPG dataset. This is a well known dataset that is available at multiple websites including the UCI Machine Learning Repository [http://archive.ics.uci.edu/ml/datasets/Auto+MPG](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 \( y \)) and the seven predictor variables (each column vector \( x_j \) in matrix \( X \)).

```scala
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_1.6/src/main/scala/scalation/analytics/ExampleAutoMPG.scala](http://www.cs.uga.edu/~jam/scalation_1.6/src/main/scala/scalation/analytics/ExampleAutoMPG.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 Management and Preprocessing Chapter.

```scala
val auto_tab = Relation (BASE_DIR + "auto-mpg.csv", "auto_mpg", null, -1)
val (x, y) = auto_tab.toMatriDD (1 to 6, 0)
println (s"x = $x")
println (s"y = $y")
```
5. 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?

6. Examine the collinearity and multi-collinearity of the column vectors in the AutoMPG dataset.

7. 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 $R^2$, select the best models from the forward and backward approaches. Are they the same?

8. 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.

4.5.12 Further Reading

1. Introduction to Linear Regression Analysis, 5th Edition [14]
4.6 Ridge Regression

The RidgeRegression class supports multiple linear ridge regression. In this case, \( x \) is multi-dimensional \([x_1, \ldots, x_k]\). Ridge regression adds a penalty based on the \( \ell_2 \) norm of the parameters \( b \) to reduce the chance of them taking on large values that may lead to less robust models. The penalty is not to be included on the intercept parameter \( b_0 \). If the \( X \) matrix is centered, then \( b_0 = \mu_y \), so \( \mu_y \) can be added back during prediction.

Therefore, both the data/input matrix \( X \) and the response/output vector \( y \) should be centered (zero mean). The combined centering takes care of the intercept, so it is not included in the model. Thus, the goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_1 x_1 + \ldots b_k x_k + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). The regularization of the model adds an \( \ell_2 \)-penalty on the parameters \( b \). The objective function to minimize is now \( sse \) plus the penalty.

\[
f_{obj} = sse + \lambda \| b \|^2 = \epsilon \cdot \epsilon + \lambda b \cdot b
\]  \hspace{1cm} (4.17)

where \( \lambda \) is the shrinkage parameter. A large value for \( \lambda \) will drive the parameters \( b \) toward zero, while a small value can help stabilize the model (e.g., for nearly singular matrices or high multi-collinearity).

\[
f_{obj} = (y - Xb) \cdot (y - Xb) + \lambda b \cdot b
\]

Taking the gradient of \( f_{obj} \) with respect to \( b \) and setting it equal to zero yields

\[
-2X^t(y - Xb) + 2\lambda b = 0
\]

\[
-X^ty + X^tXb + \lambda b = 0
\]

\[
X^tXb + \lambda b = X^ty
\]

Since \( \lambda b = \lambda I b \) where \( I \) is the the \( n \)-by-\( n \) identity matrix, we may write

\[
(X^tX + \lambda I)b = X^ty
\]  \hspace{1cm} (4.18)

Matrix factorization may now be used to solve for the parameters \( b \) in the modified Normal Equations.

4.6.1 Centering

Before creating a RidgeRegression model, the \( X \) data matrix and the \( y \) response vector should be centered. This is accomplished by subtracting the means (vector of column means for \( X \) and a mean value for \( y \)).

\[
\text{val \: mu_x = x.mean } \quad \text{// columnwise mean of \( x \)}
\]

\[
\text{val \: mu_y = y.mean } \quad \text{// mean of \( y \)}
\]

\[
\text{val \: x_c = x - mu_x} \quad \text{// centered \( x \) (columnwise)}
\]

\[
\text{val \: y_c = y - mu_y} \quad \text{// centered \( y \)}
\]

The centered matrix \( x_c \) and center vector \( y_c \) are then passed into the RidgeRegression constructor.

\[
\text{val \: rrg = new RidgeRegression (x_c, y_c)}
\]

\[
rrg.train ().eval ()
\]

\[
println (rrg.report)
\]
Now, when making predictions, the new data vector \( z \) needs to be centered by subtracting \( \mu_x \). Then the `predict` method is called, after which the mean of \( y \) is added.

```scala
val z_c = z - mu_x // center z first
yp = rrg.predict (z_c) + mu_y // predict z_c and add y's mean
println (s"predict ($z) = $yp")
```

### 4.6.2 The \( \lambda \) Hyper-parameter

The value for \( \lambda \) can be user specified (typically a small value) or chosen by a procedure like Generalized Cross-Validation (GCV).

**RidgeRegression Class**

**Class Methods:**

- `@param x` the centered data \( m \)-by-\( n \) matrix, NOT augmented with a column of 1's
- `@param y` the centered response \( m \)-vector
- `@param fname_` the feature/variable names
- `@param hparam` the shrinkage hyper-parameter, lambda \( (0 \Rightarrow \text{OLS}) \) in penalty
- `@param technique` the technique used to solve for \( b \) in \( (x^t x + \lambda I) b = x^t y \)

```scala
class RidgeRegression (x: Matrix, y: Vector, fname_: Strings = null,
  hparam: HyperParameter = RidgeRegression.hp,
  technique: RegTechnique = Cholesky)
  extends PredictorMat (x, y, fname_, hparam)
```

```scala
def xtx_I (lambda: Double)
def train (yy: Vector = y): RidgeRegression =
def gcv (yy: Vector): Double =
def forwardSel (cols: Set[Int], adjusted: Boolean = true):
  (Int, Vector, Vector) =
def backwardElim (cols: Set[Int], adjusted: Boolean = true, first = 0):
  (Int, Vector, Vector) =
def vif: Vector =
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
```

### 4.6.3 Exercises

1. Compare the results of `RidgeRegression` with those of `Regression`. Examine the parameter vectors, quality of fit and predictions made.
// 5 data points: x_0  x_1
val x = new MatrixD ((5, 2), 36.0, 66.0,  // 5-by-2 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)
val z = VectorD (20.0, 80.0)

// Compute centered (zero mean) versions of x and y, x_c and y_c

// Create a Regression model with an intercept
val ox = VectorD.one (y.dim) +^: x
val rg = new Regression (ox, y)

// Create a RidgeRegression model using the centered data
val rrg = new RidgeRegression (x_c, y_c)

// 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 \( \lambda \) and examine its effect on the parameter vector \( b \), the quality of fit and predictions made.

import RidgeRegression.hp

println (s"hp = $hp")
val hp2 = hp.updateReturn ("lambda", 1.0)
println (s"hp2 = $hp2")

3. Why is it important to center (zero mean) both the data matrix \( X \) and the response vector \( y \)? What is *scale invariance* and how does it relate to centering the data?
4.7 Lasso Regression

The LassoRegression class supports multiple linear regression using the Least absolute shrinkage and selection operator (Lasso) that constrains the values of the \( b \) parameters and effectively sets those with low impact to zero (thereby deselecting such variables/features). Rather than using an \( \ell_2 \)-penalty (Euclidean norm) like RidgeRegression, it uses and an \( \ell_1 \)-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 [12] for a more detailed explanation on how LassoRegression can eliminate a variable/feature by setting its parameter/coefficient to zero.

The regularization of the model adds an \( \ell_1 \)-penalty on the parameters \( b \). The objective function to minimize is now \( sse \) plus the penalty.

\[
\text{f}_{\text{obj}} = \frac{1}{2} \text{sse} + \lambda \|b\|_1 = \frac{1}{2} \|\epsilon\|^2 + \lambda \|b\|_1
\]

(4.19)

where \( \lambda \) is the shrinkage parameter. Substituting \( \epsilon = y - Xb \) yields

\[
\text{f}_{\text{obj}} = \frac{1}{2} \|y - Xb\|^2 + \lambda \|b\|_1
\]

Although similar to the \( \ell_2 \) penalty used in Ridge Regression, it may often be more effective. Still, the \( \ell_1 \) penalty for Lasso has a disadvantage that the absolute values in the \( \ell_1 \) norm make the objective function non-differentiable. 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 work. Instead, the objective function needs to be minimized using a search based optimization algorithm.

4.7.1 Optimization Strategies

Coordinate Descent

Alternative Direction Method of Multipliers

SCALATion also uses the Alternative Direction Method of Multipliers (ADMM) [3] algorithm to optimize the \( b \) parameter vector. The algorithm for using ADMM for Lasso Regression is outlined in section 6.4 of [3]. We follow their development closely, but change to the notation to that used herein. Optimization problems in ADMM form separate the objective function into two parts \( f \) and \( g \).

\[
\min f(b) + g(z) \text{ subject to } b - z = 0
\]

For Lasso Regression, the \( f \) function will capture the loss function \( \frac{1}{2} sse \), while the \( g \) function will capture the \( \ell_1 \) regularization, i.e.,

\[
f(b) = \frac{1}{2} \|y - Xb\|^2, \quad g(z) = \lambda \|z\|_1
\]

Therefore, the iterative step in the ADMM algorithm becomes

\[
b = (X^t X + \rho I)^{-1}(X^t y + \rho (z - u)) \]
\[
z = S_{\lambda/\rho}(b + u) \]
\[
u = u + b - z
\]
where $S$ is the soft thresholding function and $u$ is the Lagrangian vector. See scalation.minima.LassoAdmm for coding details.

### 4.7.2 The $\lambda$ Hyper-parameter

The shrinkage parameter $\lambda$ can be tuned to control feature selection. The larger the value of $\lambda$, the more features (predictor variables) whose parameters/coefficients will be set to zero.

**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
- `@param hparam` the shrinkage hyper-parameter, lambda (0 => OLS) in the penalty term 'lambda * b dot b'

```scala
class LassoRegression (x: MatrixD, y: VectorD, fname_ : Strings = null,
                     hparam: HyperParameter = LassoRegression.hp)
  extends PredictorMat (x, y, fname_, hparam)
```

```scala
def f (yy: VectorD)(b: VectorD): Double =
def train (yy: VectorD = y): LassoRegression =
def forwardSel (cols: Set[Int], adjusted: Boolean = true): (Int, VectorD, VectorD) =
def backwardElim (cols: Set[Int], adjusted: Boolean = true, first: Int = 1): (Int, VectorD, VectorD) =
def crossVal (k: Int = 10, rando: Boolean = true): Array[Statistic]
```

### 4.7.3 Exercises

1. Compare the results of `LassoRegression` with those of `Regression` and `RidgeRegression`. Examine the parameter vectors, quality of fit and predictions made.

```scala
// 5 data points: one x_0 x_1
val x = new 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 \( \lambda \) and examine its effect on the parameter vector \( \mathbf{b} \), the quality of fit and predictions made.

```scala
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 \( (\mu = 0, \sigma = 1) \).

```scala
import MatrixTransforms._
val x_n = normalize (x, (mu_x, sig_x))
val y_n = y.standardize
```

4. 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.

5. Compare LassoRegression the with Regression on the AutoMPG dataset. Specifically, compare the quality of fit measures as well as how well feature selection works.

6. Elastic Nets combine both \( \ell_2 \) and \( \ell_1 \) penalties to try to combine the best features of both RidgeRegression and LassoRegression. Elastic Nets naturally includes two shrinkage parameters, \( \lambda_1 \) and \( \lambda_2 \). Is the additional complexity worth the benefits?

### 4.7.4 Further Reading

1. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers [3]

2. Feature Selection Using LASSO [7]
4.8 Transformed Regression

The `TranRegression` class supports transformed multiple linear regression. In this case, the predictor vector \( \mathbf{x} \) is multi-dimensional \([1, x_1, ... x_k]\). In many 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, but four 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.

The capability can be easily implemented by introducing a transform function into `Regression`. The transform function and its inverse are passed into the `TranRegression` class which extends the `Regression` class. The transform and inverse functions for the four cases are as follows:

\[
(\log, \exp), (\sqrt, ^2), (^2, \sqrt), (\exp, \log)
\]

The goal then is to fit the parameter vector \( \mathbf{b} \) in the transformed regression equation

\[
\text{tran}(y) = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + ... b_k x_k + \epsilon
\]  

(4.20)

where \( \epsilon \) represents the residuals (the part not explained by the model) and `tran` is the function (defaults to log) used to transform the response \( y \). For example, for a log transformation, equation 4.2 becomes the following:

\[
\log(y) = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + ... b_k x_k + \epsilon
\]

The transformation is done in the implementation of the `TranRegression` class by transforming \( y \) and passing it to the `Regression` superclass (multiple linear regression).

\[
\text{Regression}(x, \text{y.map (tran)}, \text{technique})
\]

The inverse transform (`itran`) is then applied in the `predict` method.

\[
\text{override def predict (z: VectoD): Double = itran (b dot z)}
\]

Traditionally, diagnostics are performed using the transformed response vector \( \text{y.map (tran)} \). Sca-lation also provides overall diagnostics based on the original response values using the `eval2` method. Box-Cox transformations (see Exercise 2) are provided in the class’ companion object.
TranRegression Class

Class Methods:

@param x the data/input matrix
@param y the response/output vector
@param fname_ the feature/variable names
@param tran the transformation function (defaults to log)
@param itran the inverse transformation function to rescale predictions to original y scale (defaults to exp)
@param technique the technique used to solve for b in x.t*x*b = x.t*y

class TranRegression (x: MatriD, y: VectoD, fname_ : Strings = null, tran: FunctionS2S = log, itran: FunctionS2S = exp, technique: RegTechnique = QR)
extends Regression (x, y.map (tran), fname_, null, technique)

override def eval (xx: MatriD = x, yy: VectoD = y): TranRegression =
def eval2 (xx: MatriD = x, yy: VectoD = y): TranRegression =
override def predict (z: VectoD): Double = itran (b dot z)
override def predict (z: MatriD = x): VectoD = (z * b).map (itran)
override def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =

4.8.1 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) x(cap * i + j) = VectorD (1, i, j)
    for (k <- y.range) y(k) = sq (10 + 2 * x(k, 1) + err.gen)
    // for (k <- y.range) y(k) = sq (10 + 2 * x(k, 1) + 0.3 * x(k, 2) + err.gen)
    val t = VectorD.range (0, y.dim)

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 rg = new Regression (x, y)
rg.train ().eval ()
println (rg.report)
println (rg.summary)

val yp = rg.predict ()
val e = y - yp

new Plot (t, y, yp, "Original Regression y and yp vs. t")
new Plot (t, 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, yp2 and 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 to match the actual response y, by using the inverse transformation function sq. Now, compute and plot the corresponding y, yp3 and 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 (Seq (y2, yp2))
   val ys3 = MatrixD (Seq (y, yp3, yp))
   new PlotM (t, ys2.t, null, "Transformed")
   new PlotM (t, 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.train ().eval ()
   println (rg.report)
   println (rg.summary)

   banner ("TranRegression")
   val trg = new TranRegression (x, y, sqrt _, sq _)
   trg.train ().eval () // try eval2 ()
   println (rg.report)
   println (rg.summary)
5. Consider the following family of transformation functions.

\[ f_{\text{tran}}(y) = \frac{y^\lambda - 1}{\lambda} \]

where \( \lambda \) determines the power function on \( y \), e.g., 0.5 for \texttt{sqrt} and 2.0 for \texttt{sq}. What is the inverse function? Try various Box-Cox transformations (values for \texttt{lambda}) for the above problem.

\texttt{TranRegression (x, y, lambda)}
4.9 Quadratic Regression

The QuadRegression class extends the Regression class by automatically adding 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.

The QuadRegression class 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 add an intercept column (column of all ones) and the index calculations will be thrown off if the initial data matrix one.

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the quadratic regression equation is the following:

$$ y = \mathbf{b} \cdot \mathbf{x'} + \epsilon = b_0 + b_1 x_1 + b_2 x_1^2 + b_4 x_2 + b_5 x_2^2 + \epsilon $$

(4.21)

where $\mathbf{x'} = [1, x_1, x_1^2, x_2, x_2^2]$ and $\epsilon$ 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:

$$ nt = 2n + 1 \quad e.g., \quad nt = 5 \quad for \quad n = 2 $$

(4.22)

Each column in the initial data matrix is expanded into two in the expanded data matrix and an intercept column is added.

The addition of squared columns is performed by functions in the companion object. For example the function qForms will take an unexpanded vector $v$ and the number of terms for the expanded form and will make a new vector where the zero-th element is 1.0, the odd elements are original values and the rest of the even elements are the squared values.

```scala
def qForms (v: VectoD, nt: Int): VectoD =
{
  VectorD (for (j <- 0 until nt) yield
    if (j == 0) 1.0  // intercept term
    else if (j % 2 == 1) v(j/2)  // linear terms
    else v((j-1)/2)~^2 // quadratic terms
  ) // for
} // qForms
```

QuadRegression Class

Class Methods:

@param x_ the initial data/input matrix (before quadratic term expansion)
@param y the response/output vector
@param fname_ the feature/variable names
4.9.1 Exercises

1. Perform Quadratic Regression on the `ExampleBPressure` dataset using the first two columns of its data matrix `x`.

   ```scala
   import ExampleBPressure.{x01 => x, y}
   ```

2. 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 $\bar{R}^2$?

3. Generate a dataset with data matrix `x` and response vector `y` using the following loop where `noise = new Normal (0, 10 * m * m)`.

   ```scala
   for (i <- x.range1) {
     x(i, 0) = i
     y(i) = i*i + i + noise.gen
   } // for
   ```

   Compare the results of `Regression` vs. `QuadRegression`. 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 `QuadRegression`. Also plot the residuals `e` vs. `x` for both. Note, `t` is the index vector `VectorD.range (0, m)`.

4. 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`.

   ```scala
   var k = 0
   for (i <- grid; j <- grid) {
     x(k) = VectorD (i, j)
     y(k) = x(k, 0)^2 + 2 * x(k, 1) + noise.gen
     k += 1
   } // for
   ```

   Compare the results of `Regression` vs. `QuadRegression`. Try modifying the equation for the response and see how Quality of Fit changes.
4.10 Response Surface Regression

The ResponseSurface class supports the development of Response Surface Regression models. These models fit a multi-dimensional surface with low-order multi-dimensional polynomials. For order one models, regular multiple regression Regression should be used.

4.10.1 Quadratic with Cross Terms

For order two models, the goal is to fit a quadratic surface. In two dimensions (2D) where \( x = [x_1, x_2] \), the quadratic regression equation is the following:

\[
y = b \cdot x' + \epsilon = b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_2 + b_4 x_2^2 + b_5 x_1 x_2 + \epsilon
\]

(4.23)

where \( x' = [1, x_1, x_1^2, x_1 x_2, x_2^2] \) and \( \epsilon \) represents the residuals (the part not explained by the model). 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 \( (n \rightarrow n + 1) \).

\[
nt = \binom{n+2}{2} = \frac{(n+2)(n+1)}{2} \quad \text{e.g., } nt = 6 \text{ for } n = 2
\]

(4.24)

Such models generalize QuadRegression by introducing cross terms, e.g., \( x_1 x_2 \). Adding cross-terms makes the number of terms increase quadratically rather than linearly with the dimensionality. Consequently, multi-collinearity problems may be intensified and the need for feature selection, therefore, increases.

4.10.2 Cubic

For order three models, the goal is to fit a cubic surface. In two dimensions (2D) where \( x = [x_1, x_2, x_3] \), the cubic regression equation is the following:

\[
y = b \cdot x' + \epsilon = b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1^3 + b_4 x_2 + b_5 x_2^2 + b_6 x_2 x_1 + b_7 x_3 + b_8 x_3^2 + b_9 x_2 x_3 + \epsilon
\]

(4.25)

where \( x' = [1, x_1, x_1^2, x_1 x_2, x_1 x_3, x_2, x_2^2, x_2 x_3, x_3^2] \) and \( \epsilon \) represents the residuals (the part not explained by the model). 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 \( (n \rightarrow n + 1) \).

\[
nt = \binom{n+3}{3} = \frac{(n+3)(n+2)(n+1)}{6} \quad \text{e.g., } nt = 10 \text{ for } n = 2
\]

(4.26)

When \( n = 10 \), the number of terms and corresponding parameters \( nt = 286 \), whereas for Regression, QuadRegression and ResponseSurface order 2, it would 11, 21 and 66, respectively. Issues related negative degrees of freedom, overfitting and multi-collinearity will need 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.
ResponseSurface Class

Class Methods:

@param x_ the input vectors/points
@param y the response vector
@param cubic the order of the surface (defaults to quadratic, else cubic)
@param technique the technique used to solve for b in x.t*x*b = x.t*y

class ResponseSurface (x_ : MatriD, y: VectoD, cubic: Boolean = false, technique: RegTechnique = QR)
    extends Regression (ResponseSurface.allForms (x_, cubic), y, technique)

    override def predict (z: VectoD): Double =
    override def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]

4.10.3 Exercises

1. Perform Response Surface Regression on the ExampleBPressure dataset using the first two columns of its data matrix x.

   import ExampleBPressure.{x01 => x, y}

2. 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 \( \bar{R}^2 \)?

3. 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) {
       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
   } // for

Compare the results of Regression QuadRegression vs. ResponseSurface. Try modifying the equation for the response and see how Quality of Fit changes.
4. Prove that the number of terms for a quadratic function \( f(x) \) in \( n \) dimensions is \( \binom{n+2}{2} \), by decomposing the function into its quadratic (both squared and cross), linear and constant terms,

\[
f(x) = x'Ax + b'x + c
\]

where \( A \) in an \( n \)-by-\( n \) matrix, \( b \) 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 \( b \) become scalars, yielding the usual quadratic function \( ax^2 + bx + c \).

5. How does Response Surface Regression related to Response Surface Methodology?
4.11 Weighted Least Squares Regression

The Regression_WLS class supports weighted multiple linear regression. In this case, the predictor vector \( x \) is multi-dimensional \([1, x_1, ..., x_k]\). As before the regression equation is

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + ... b_k x_k + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). Under multiple linear regression, the parameter vector \( b \) is estimated using matrix factorization with the Normal Equations.

\[
X^t X b = X^t y
\]

Let us look at the error vector \( \epsilon = y - X b \) in more detail. A basic assumption is that \( \epsilon_i \sim NID(0, \sigma_i) \), i.e., it is Normally and Independently Distributed (NID). If this is violated substantially, the estimate for the parameters \( b \) may be less accurate than desired. One way this can happen is that the variance changes \( \epsilon_i \sim NID(0, \sigma_i) \). This is called heteroscedasticity and it would imply that certain instances (data points) would have greater influence \( b \) than they should. The problem can be corrected by weighting each instance by the inverse of its residual/error variance.

\[
w_i = \frac{1}{\sigma_i^2}
\]

This begs the question on how to estimate the residual/error variance. This can be done by performing unweighted regression of \( y \) onto \( X \) to obtain the error vector \( \epsilon \). It is used to compute a Root Absolute Deviation (RAD) vector \( r \).

\[
r = \sqrt{\left| \epsilon \right|}
\]

A simple approach would be to make the weight \( w_i \) inversely proportional to \( r_i \).

\[
w_i = \frac{n}{r_i}
\]

More commonly, a second unweighted regression is performed, regressing \( r \) onto \( X \) to obtain the predictions \( \hat{r} \). See Exercise 1 for a comparison or the two methods setWeights0 and setWeights.

\[
w_i = \frac{n}{\hat{r}_i}
\]

See [14] 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

\[
X^t W X b = X^t W y
\]

In SCALATIOn, this is accomplished by computing a weight vector \( w \) and taking its square root \( \omega = \sqrt{W} \). The data matrix \( X \) is then reweighted by premultiplying it by \( \omega \) (rtW in the code), as if it is a diagonal matrix \( rtW ** x \). The response vector \( y \) is reweighted using vector multiplication \( rtW * y \). The reweighted matrix and vector are passed into the Regression class, which solves for the parameter vector \( b \).

In summary, Weighted Least-Squares (WLS) is accomplished by reweighting and then using Ordinary Least Squares (OLS). See [14] and [15] for additional discussion.
Regression_WLS Class

Class Methods:

@param xx the data/input m-by-n matrix
(augment with a first column of ones to include intercept in model)
@param yy the response/output m vector
@param fname_ the feature/variable names
@param technique the technique used to solve for b in x.t*w*x*b = x.t*w*y
@param w the weight vector (if null, computed in companion object)

class Regression_WLS (xx: MatriD, yy: VectoD, technique: RegTechnique = QR, 
private var w: VectoD = null)
extends Regression ({ setWeights (xx, yy, technique, w); reweightX (xx, w) }, 
reweightY (yy, w), technique)

def weights: VectoD = w
override def diagnose (e: VectoD, w_ : VectoD, yp: VectoD, y_ : VectoD = null)
override def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]

4.11.1 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: constant term, x_1 coordinate, x_2 coordinate
val x = new 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. Prove that reweighting the data matrix \(X\) and the response vector \(y\) and solving for the parameter vector \(b\) in the standard Normal Equations \(X'Xb = X'y\) gives the same result as not reweighting and solving for the parameter vector \(b\) in the Weighted Normal Equations \(X'WXb = X'Wy\).

3. Given an error vector \(\epsilon\), what does its covariance matrix \(C[\epsilon]\) represent? How can it be estimated? What are its diagonal elements?

92
4. 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?
4.12 Polynomial Regression

The `PolyRegression` class supports polynomial regression. In this case, \( x \) is formed from powers of a single parameter \( t \), \([1, t, t^2, \ldots t^k]\). The goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1t + b_2t^2 + \ldots b_k t^k + \epsilon
\]

where \( \epsilon \) 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 \( t \) and one for the \( m \)-vector \( y \). An easy way to implement polynomial regression is to expand each \( t \) value into an \( x \) 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 \( t \).

\[
X = \begin{bmatrix} 1, t, t^2, \ldots t^k \end{bmatrix}
\]

In SCALATION the vector \( t \) 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.range) x(i) = expand (t(i))
val rg = new Regression (x, y, technique)
```

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 raw polynomials [22]. This is done in SCALATION by changing the `raw` flag to `false`.

PolyRegression Class

Class Methods:

- `@param t` the input vector: \( t_i \) expands to \( x_i = [1, t_i, t_i^2, \ldots t_i^k] \)
- `@param y` the response vector
- `@param ord` the order \( (k) \) of the polynomial (max degree)
- `@param technique` the technique used to solve for \( b \) in \( x \cdot t \cdot x \cdot b = x \cdot t \cdot y \)
- `@param raw` whether to use raw or orthogonal polynomials

```scala
class PolyRegression (t: VectoD, y: VectoD, ord: Int, technique: RegTechnique = Cholesky, raw: Boolean = true)
extends PredictorVec (t, y, ord)
```

```scala
def expand (t: Double): VectoD =
def orthogonalize (x: MatriD): (MatriD, MatriD) =
def orthoVector (v: VectoD): VectoD =
def predict (z: Double): Double =
```
4.12.1 Exercises

1. Generate two vectors \( t \) and \( y \) as follows.

   ```scala
   val noise = Normal (0.0, 100.0)
   val t = VectorD.range (0, 100)
   val y = new VectorD (t.dim)
   for (i <- 0 until 100) y(i) = 10.0 - 10.0 * i + i^2 + i * noise.gen
   ```

   Test new PolyRegression \((t, y, \text{order}, \text{technique})\) for various orders and factorization techniques. Test for multi-collinearity using the correlation matrix and vif.

2. Test new PolyRegression \((t, y, \text{order}, \text{technique}, \text{false})\) for various orders and factorization techniques. Setting the \text{raw} flag to \text{false} will cause orthogonal polynomials to be used instead or raw polynomials. Again, test for multi-collinearity using the correlation matrix and vif.
4.13 Trigonometric Regression

The TrigRegression class supports trigonometric regression. In this case, \( x \) 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. The goal is to fit the parameter vector \( \mathbf{b} \) in the regression equation

\[
y = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1\sin(\omega t) + b_2\cos(\omega t) + \ldots b_{2k-1}\sin(k\omega t) + b_{2k}\cos(k\omega t) + \epsilon
\]

where \( \omega \) is the base angular displacement in radians (e.g., \( \pi \)) and \( \epsilon \) 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 \( x \) 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 \( t \).

\[
X = [1, \sin(\omega t), \cos(\omega t), \sin(2\omega t), \cos(2\omega t), \ldots \sin(k\omega t), \cos(k\omega t)]
\]

For a model with \( k \) harmonics (maximum multiplier of \( \omega t \)), the data matrix can be formed as follows:

```scala
val x = new MatrixD (t.dim, 1 + 2 * k)
for (i <- t.range) x(i) = expand (t(i))
val rg = new Regression (x, y, technique)
```

TrigRegression Class

Class Methods:

```scala
class TrigRegression (t: VectoD, y: VectoD, ord: Int, technique: RegTechnique = QR)
  extends PredictorVec (t, t, ord)

  def expand (t: Double): VectoD =
  def predict (z: Double): Double = rg.predict (expand (z))
  def crossVal (ord: Int, k: Int = 10, rando: Boolean = true): Array [Statistic]
```

96
4.13.1 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.

```scala
val noise = Normal (0.0, 10000.0)
val t = VectorD.range (0, 100)
val y = new VectorD (t.dim)
for (i <- 0 until 100) {
  val x = (i - 40)/2.0
  y(i) = 1000.0 + x + x*x + x*x*x + noise.gen
}
```

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.

```scala
val noise = Normal (0.0, 10.0)
val t = VectorD.range (0, 200)
val y = new VectorD (t.dim)
for (i <- 0 until 5) {
  for (j <- 0 until 20) {
    val x = j - 4
    y(40*i+j) = 100.0 + x + x*x + x*x*x + noise.gen
  } // for
  for (j <- 0 until 20) {
    val x = 16 - j
    y(40*i+20+j) = 100.0 + x + x*x + x*x*x + noise.gen
  } // for
} // for
```

3. Is the problem of multi-collinearity an issue for Trigonometric Regression?

4. How does Trigonometric Regression relate to Fourier Series?
4.14 ANCOVA

An ANalysis of COVAriance (ANCOVA) model may be developed using the `ANCOVA` 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.

4.14.1 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 $x_j \in \{\text{Red, Black}\}$, it would be replaced by one encoded/dummy variable, $x_{j0}$ as shown in Table 4.2.

<table>
<thead>
<tr>
<th>$x_j$</th>
<th>encoded $x_j$</th>
<th>dummy $x_{j0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Black</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 $n_{dv}$. 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.

$$\text{for } (k \leftarrow 0 \text{ until } n_{dv} - 1) \text{ } x_{jk} = I\{x_j = k+1\}$$

$I\{\}$ is an indicator function that returns 1 when the condition evaluates to true and 0 otherwise. In this way, $x_j \in \{\text{English, French, German, Spanish}\}$ would be replaced by three dummy variables, $x_{j0}$, $x_{j1}$ and $x_{j2}$, as shown in Table 4.3.

Conversion from strings to an integer encoding can be accomplished using the `map2Int` function in the `Converter` object within the `scala.linalgebra` package. The vector of encoded integers $x_e$ can be
Table 4.3: Encoding a Categorical Variable

<table>
<thead>
<tr>
<th></th>
<th>encoded</th>
<th>dummy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_j$</td>
<td>$x_j0$, $x_j1$, $x_j2$</td>
</tr>
<tr>
<td>English</td>
<td>0</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>French</td>
<td>1</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>German</td>
<td>2</td>
<td>0, 1, 0</td>
</tr>
<tr>
<td>Spanish</td>
<td>3</td>
<td>0, 0, 1</td>
</tr>
</tbody>
</table>

made into a matrix using `MatrixI (Seq (xe))`. To produce the dummy variable columns the `dummyVars` function within the ANCOVA companion object may be called. See the first exercise for an example.

Multi-column expansion may be done by the caller in cases where there are fewer categorical variables, by expanding the input data matrix before passing it to the Regression class. The expansion occurs automatically when the ANCOVA class is called. This class performs the expansion and then delegates to the work to the Regression class.

Before continuing the discussion of the ANCOVA class, a restricted form is briefly discussed.

### 4.14.2 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 (GLM) 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, $x$ consists of $n_{dv} - 1$ dummy variables $d_k$ for $k \in \{1, n_{dv} - 1\}$

$$y = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1d_1 + \ldots + b_l d_l + \epsilon$$

where $l = n_{dv} - 1$ and $\epsilon$ 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](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 $t$. As with multiple linear regression, the $y$ vector holds the response values. Multi-way Analysis of Variance may be performed using the more general ANCOVA class. Also, a more traditional implementation called Anova, not following the GLM approach, is also provided in the stat package.

**ANOVA1 Class**

**Class Methods:**

- `@param t` the binary/categorical treatment variable vector
double with integer values
- `@param y` the response vector
4.14.3 ANCOVA Implementation

When there is only one categorical/binary variable, \( x \) 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.

\[
t \quad \text{expands to} \quad d = [d_0, \ldots, d_l] \quad \text{where} \quad l = n_{dv} - 2
\]

Therefore, the regression equation becomes the following:

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \cdots + b_k x_k + b_{k+1} d_0 + \cdots + b_{k+l} d_l + \epsilon
\]

The dummy variables are binary and are used to determine the level of a categorical variable. See [http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx](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, ANCOVA 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 \( y \) vector holds the response values.

**ANCOVA 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
@param technique the technique used to solve for b in \( x^t * x * b = x^t * y \)

class ANCOVA (x_: MatriD, t: MatriI, y: VectoD, fname_: Strings = null,
technique: RegTechnique = QR)
    extends Regression (x_ ++^ ANCOVA.dummyVars (t), y, fname_, null, technique)

override def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]

4.14.4 Exercises

1. Use the \texttt{map2Int} function in the \texttt{Converter} object within the \texttt{scalation.linalgebra} package to convert the given strings into encoded integers. Turn this vector into a matrix and pass it into the \texttt{dummyVars} function to produce the dummy variable columns. Print out the values \( x_e \), \( x_m \) and \( x_d \).

   ```scala
   val x1 = VectorS ("English", "French", "German", "Spanish")
   val (xe, map) = Converter.map2Int (x1) // map strings to integers
   val xm = MatrixI (Seq (xe)) // form a matrix from vector
   val xd = ANCOVA.dummyVars (xm) // make dummy variable columns
   ``

   Add code to recover the string values from the encoded integers using the returned \texttt{map}.

2. Use the \texttt{ANOVA1} class to predict responses based on treatment levels trained using the following treatment \( t \) and response \( y \) vectors. Plot the given versus predicted responses.

   ```scala
   val t = VectorI (1, 1, 1, 2, 2, 2, 3, 3, 3) // treatment level data
   val y = VectorD (755.0, 865.0, 815.0,
                    442.0, 420.0, 401.0,
                    282.0, 250.0, 227.0)
   val levels = 3
   val arg = new ANOVA1 (t, y, levels)
   arg.train ().eval ()
   println ("parameter = " + arg.parameter)
   println ("fitMap = " + arg.fitMap)
   banner ("test predictions")
   val yp = new VectorD (y.dim)
   for (i <- yp.range) yp(i) = arg.predict (t(i))
   println (s" y = $y \n yp = $yp")
   new Plot (t.toDouble, y, yp, "ANOVA1")
   ```
3. Compare the results of using the ANCOVA class versus the Regression class for the following dataset.

// 5 data points: constant term, x_1 coordinate, x_2 coordinate
val x = new 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 = new MatrixI ((6, 1), 0, 0, 1, 1, 2, 2)  // 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, 1.0, 0.0)
4.15 General Linear Models

A General Linear Model (GLM) may be developed using the GLM object. Currently, SCALATION only supports Univariate forms, i.e., models having only one response variable $y$.

$$y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \ldots b_k x_k + b_{k+1} d_1 + \ldots b_{k+l} d_l + \epsilon$$

The GLM object is simply a convenient wrapper for more specific types of general linear models: (1) simple linear regression, (2) multiple linear regression, (3) weighted least squares regression, (4) ridge regression, (5) transformed multiple linear regression, (6) polynomial regression, (7) trigonometric regression, (8) response surface regression, (9) ANOVA and (10) ANCOVA. It provides factory apply functions for creating specific types of general linear models, based on the signatures of the parameters.

Example Problem:

Class Methods:

```scala
trait GLM {
  def apply (x: VectoD, y: VectoD): SimpleRegression =
  def apply (x: MatriD, y: VectoD): Regression [MatriD, VectoD] =
  def apply (xy: MatriD): Regression [MatriD, VectoD] =
  def apply (x: MatriD, y: VectoD, w: VectoD): Regression_WLS [MatriD, VectoD] =
  def apply (x: MatriD, y: VectoD, lambda: Double): RidgeRegression [MatriD, VectoD] =
  def apply (x: MatriD, y: VectoD, transform: FunctionS2S, tranInv: FunctionS2S): TranRegression [MatriD, VectoD] =
  def apply (t: VectoD, y: VectoD, k: Int): PolyRegression =
  def apply (ty: MatriD, k: Int): PolyRegression =
  def apply (t: VectoD, y: VectoD, k: Int, p: Int): TrigRegression =
  def apply (ty: MatriD, k: Int, p: Int): TrigRegression =
  def apply (x_: MatriD, y: VectoD, cubic: Boolean): ResponseSurface [MatriD, VectoD] =
  def apply (t: VectoI, y: VectoD, levels: Int): ANOVA =
  def apply (x_ : MatriD, t: VectoI, y: VectoD, levels: Int): ANCOVA =
}
```

object GLM extends GLM

103
Chapter 5

Classification

When the output/response $y$ is defined on small domains (categorical response), e.g., $\mathbb{B}$ or $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$, then the problem shifts from prediction to classification. This facilitates giving the response meaningful class names, e.g., low-risk, medium-risk and high-risk. When the response is discrete, but unbounded (e.g., Poisson Regression), the problem is considered to be a prediction problem.

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

As with Regression in continuous domains, some of the modeling techniques in this chapter will focus on estimating the conditional expectation of $y$ given $\mathbf{x}$.

$$y = \mathbb{E}[y|\mathbf{x}] + \epsilon$$ (5.1)

Others will focus on maximizing the conditional probability of $y$ given $\mathbf{x}$, i.e., finding the conditional mode.

$$y^* = \arg\max P(y|\mathbf{x}) = \mathbb{M}[y|\mathbf{x}]$$ (5.2)

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 function is still used, but the classify method replaces the predict method.

Let us briefly contrast the two approaches based on the two equations (5.1 and 5.2). 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).
5.1 Classifier

The Classifier trait provides a common framework for several classifiers such as NaiveBayes.

Trait Methods:

```
trait Classifier

def size: Int // typically = m
def train (itest: IndexedSeq[Int]): Classifier
def train (testStart: Int, testEnd: Int): Classifier = train (testStart until testEnd)
def train (): Classifier = train (0, 0)
def classify (z: VectorI): (Int, String, Double)
def test (itest: IndexedSeq[Int]): Double
def test (testStart: Int, testEnd: Int): Double = test (testStart until testEnd)
def crossValidate (nx: Int = 10, show: Boolean = false): Double =
```

For modeling, a user chooses one of the classes extending the trait Classifier (e.g., DecisionTreeID3) to instantiate an object. Next the train method would be typically called. 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 way to utilize 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. The first cross-validation method takes the number of folds $k$ (nx in the software) and a show flag. It performs $k$ iterations of training (train method) and testing (test method).

```
def crossValidate (nx: Int = 10, show: Boolean = false): Double =
{
  val testSize = size / nx // number of instances in test set
  var sum = 0.0
  for (it <- 0 until nx) {
    val testStart = it * testSize // test set start index (inclusive)
    val testEnd   = testStart + testSize // test set end index (exclusive)
    train (testStart, testEnd) // train on opposite instances
    val acc = test (testStart, testEnd) // test on test set
    if (show) println (s"crossValidate: for it = $it, acc = $acc")
    sum += acc // accumulate accuracy
  }
}
```
The second cross-validation method is more complicated, but usually preferred, since it randomizes the instances selected for the test dataset, so that patterns coincidental to the index are broken up.

```scala
def crossValidateRand (nx: Int = 10, show: Boolean = false): Double =
```

The `crossValidateRand` method calls the following methods:

```scala
  train (itest: IndexedSeq[Int])
  test (itest: IndexedSeq[Int])
```

while the `crossValidate` method calls the following methods:

```scala
  train (testStart: Int, testEnd: Int)
  test (testStart: Int, testEnd: Int)
```

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.
5.2 ClassifierInt

The ClassifierInt abstract class provides a common foundation for several classifiers that operate on integer-valued data.

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the integer-valued classification vector, where y_i = class for row i of matrix x
@param fn the names of the features/variables
@param k the number of classes
@param cn the names for all classes

abstract class ClassifierInt (x: MatrixI, y: VectorI, fn: Strings, k: Int, cn: Strings)
    extends Classifier with Error

    def size: Int = m
    def vc_default: Array[Int] = Array.fill (n)(2)
    def vc_fromData: Array[Int] = (for (j <- x.range2) yield x.col(j).max() + 1).toArray
    def vc_fromData2 (rg: Range): Array[Int] = (for (j <- rg) yield x.col(j).max() + 1).toArray
    def shiftToZero () { x -= VectorI (for (j <- x.range2) yield x.col(j).min()) }
    def classify (z: VectorD): (Int, String, Double) = classify (roundVec (z))
    def classify (xx: MatrixI): VectorI =
    def test (itest: IndexedSeq[Int]): Double =
    def test (xx: MatrixI, yy: VectorI): Double =
    def calcCorrelation: MatrixD =
    def calcCorrelation2 (zrg: Range, xrg: Range): MatrixD =
    def featureSelection (TOL: Double = 0.01)

ClassifierInt 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 and methods for calculating correlations. Finally, the featureSelection method will eliminate features that have little positive impact on the quality of the model. Rather than considering all n features/variables, a proper subset fset C {0, 1, ... n − 1} of the features is selected. Various algorithms can be used to search for an optimal feature set fset. SCALATION currently uses a simple backward elimination algorithm that removes the least significant feature, in terms of cross-validation accuracy, in each round.
5.3 Confusion Matrix

The ConfusionMat class provides methods to produce a confusion matrix and associated quality metrics. In Scala, when $k = 2$, the confusion matrix $C$ is configured as follows:

$$
\begin{bmatrix}
    c_{00} = tn & c_{01} = fp \\
    c_{10} = fn & c_{11} = tp
\end{bmatrix}
$$

The first column indicates the classification is negative (no or 0), while the second column indicates it is positive (yes or 1). The first letter indicates whether the classification is correct (true) or not (false). The row (0, 1) indicates the actual class label, while the column (0, 1) indicates the response of the classifier.

Class Methods:

- @param y the actual class labels
- @param yp the predicted class labels
- @param k the number class values

```scala
class ConfusionMat (y: Vector[Int], yp: Vector[Int], k: Int = 2)

  def confusion: Matrix = conf
  def pos_neg (con: Matrix = conf): (Double, Double, Double, Double) =
  def accuracy: Double = conf.trace / conf.sum.toDouble
  def prec_recl: (Vector[Double], Vector[Double], Double, Double) =
  def f1_measure (prec: Double, recl: Double): Double = 2.0 * prec * recl / (prec + recl)
  def kappa: Double =
```

109
5.4 Bayes Classifier

The `BayesClassifier` abstract class provides common methods for several Bayes classifiers.

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where y(l) = class for row l of the matrix x, x(l)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes

abstract class BayesClassifier (x: MatrixI, y: VectorI, fn_: Strings = null, k: Int = 2, cn_: Strings = null) extends ClassifierInt (x, y, fn_, k, cn_) with BayesMetrics

def toggleSmooth () { smooth = ! smooth }
def calcCMI (idx: IndexedSeq[Int], vca: Array[Int]): MatrixD =
def cmiJoint (p_y: VectorD, p_Xy: HMatrix3[Double], p_XZy: HMatrix5[Double]): MatrixD =
def getParent: Any = null
protected def updateFreq (i: Int) {}
def printClassProb () { println (s"ClassProb = $p_y") }
5.5 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 $y$ within the training dataset. These probabilities are estimated based on the frequency $\nu$ (nu in the code) with which each class value occurs.

$$\nu(y = c) = |\{i \mid y_i = c\}| = m_c$$

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 $y$ where $y_i$ equals $c$ divided by the total number of elements.

$$P(y = c) = \frac{\nu(y = c)}{m} = \frac{m_c}{m} \quad (5.3)$$

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.,

$$P(y = 0) = \frac{5}{14} \quad \text{and} \quad P(y = 1) = \frac{9}{14}$$

This information, class frequencies and class probabilities, can be placed into a Class Frequency Vector (CFV) as shown in Table 5.1 and

Table 5.1: Class Frequency Vector

<table>
<thead>
<tr>
<th>$y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

a Class Probability Vector (CPV) as shown in Table 5.8

Table 5.2: Class Probability Vector

<table>
<thead>
<tr>
<th>$y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5/14</td>
<td>9/14</td>
</tr>
</tbody>
</table>

Picking the maximum probability case, one should always predict that tennis will be played, i.e., $y^* = 1$.

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.
NullModel Class

Class Methods:

```scala
@param y the class vector, where y(i) = class for instance i
@param k the number of classes
@param cn_ the names for all classes

class NullModel (y: VectorI, k: Int = 2, cn_: Strings = null)
  extends ClassifierInt (null, y, null, k, cn_)

def train (itest: IndexedSeq [Int]): NullModel =
  def classify (z: VectorI): (Int, String, Double) =
  override def classify (xx: MatrixI): VectorI = VectorI.fill (xx.dim1)(p_y.argmax ())
  def classify (xx: MatrixD): VectorI = classify (xx.toInt)
  override def test (itest: IndexedSeq [Int]): Double =
  def reset () { /* NA */ }
```

The \texttt{train} method for this modeling technique is very simple. It takes the parameter \texttt{itest} as input that indicates which instance/row indices make up the test dataset. The training dataset is made up of the rest on the instances.

```scala
def train (itest: IndexedSeq [Int]): NullModel =
  {
    val idx = 0 until m diff itest // training dataset - opposite of testing
    nu_y = frequency (y, k, idx) // frequency vector for y
    p_y = toProbability (nu_y, idx.size) // probability vector for y
    if (DEBUG) println (s" nu_y = $nu_y 
      p_y = $p_y")
    this
  } // train
```

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.

The \texttt{frequency} and \texttt{toProbability} are functions from the \texttt{Probability} object in the \texttt{scalation.analytics} package.

```scala
def frequency (x: VectorI, k: Int, idx_: IndexedSeq [Int] = null): VectorI =
  {
    val idx = if (idx_ == null) IndexedSeq.range (0, x.dim) else idx_
    val nu = new VectorI (k)
    for (i <- idx) nu(x(i)) += 1
    nu
```
} // frequency

def toProbability (nu: VectoI, n: Int): VectoD =
{
    val nd = n.toDouble
    VectorD (for (i <- nu.range) yield nu(i) / nd)
} // toProbability

5.5.1 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.analytics.classifier.ExampleTennis. Use the NullModel for classification and evaluate its effectiveness using cross-validation.

/** The 'ExampleTennis' object is used to test all integer based classifiers.
   * This 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.sliceCol (0, 4) // columns 0, 1, 2, 3
   * val y = xy.col (4) // column 4
   */

object ExampleTennis
{

    // 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 data matrix
    val xy = new 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
} // ExampleTennis object

5.6 Naive Bayes

The NaiveBayes class implements a Naive 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 ExampleTennis exercise. For Naive Bayes, weather conditions (or other data relevant to decision making) are captured in an \( n \)-dimensional vector of random variables.

\[
x = [x_0, \ldots, x_{n-1}],
\]

For the exercise, \( n = 4 \) 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 \( P(y|x) \).

Bayesian classifiers are designed to find the class (value for random variable \( y \)) that maximizes the conditional probability of \( y \) given \( x \).

It may be complex and less robust to estimate \( P(y|x) \) directly. Often it is easier to examine the conditional probability of \( x \) 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.

\[
P(y|x) = \frac{P(x|y)P(y)}{P(x)}
\]

Since the denominator is the same for all \( y \), it is sufficient to maximize the right hand side of the following proportionality statement.

\[
P(y|x) \propto P(x|y)P(y)
\]

Notice that the right hand side is the joint probability of all the random variables.

\[
P(x, y) = P(x|y)P(y) \quad (5.4)
\]

One could in principle represent the joint probability \( P(x, y) \) or the conditional probability \( P(x|y) \) 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.

5.6.1 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 \( P(x|y) \) into the product of their conditional probabilities (independence rule).

\[
P(x|y) = \prod_{j=0}^{n-1} P(x_j|y)
\]

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, Naive Bayes still tends to perform well \[25\]. Substituting this factorization in equation 5.4 yields

\[
P(x, y) = P(y) \prod_{j=0}^{n-1} P(x_j|y) \quad (5.5)
\]
The classification problem then is to find the class value for $y$ that maximizes this probability, i.e., let $y^*$ 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.

$$y^* = \text{argmax}_{y \in \{0, \ldots, k - 1\}} P(y) \prod_{j=0}^{n-1} P(x_j | y)$$ (5.6)

### 5.6.2 Estimating Conditional Probabilities

For Integer-based classifiers $x_j \in \{0, 1, \ldots, v_{c_j} - 1\}$ where $v_{c_j}$ 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 $y$. 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 $x_{.j} = h$ and vector $y = c$ is as follows:

$$\nu(x_{.j} = h, y = c) = |\{i \mid x_{ij} = h, y_i = c\}|$$

The conditional probability for random variable $x_j$ given random variable $y$ can be estimated as the ratio of two frequencies.

$$P(x_j = h \mid y = c) = \frac{\nu(x_{.j} = h, y = c)}{\nu(y = c)}$$ (5.7)

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)*.

#### ExampleTennis Problem

For the ExampleTennis problem, the *Joint Frequency Matrix/Table (JFT)* for the $Outlook$ random variable $x_0$ is shown in Table 5.3

\[
\begin{array}{c|cc}
\text{x}_0 & 0 & 1 \\
\hline
0 & 2 & 3 \\
1 & 0 & 4 \\
2 & 3 & 2 \\
\end{array}
\]

Table 5.3: JFT for $x_0$

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., $P(x_0 = h \mid y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum.
Table 5.4: CPT for \( x_0 \)

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>3/9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4/9</td>
</tr>
<tr>
<td>2</td>
<td>3/5</td>
<td>2/9</td>
</tr>
</tbody>
</table>

Continuing with the ExampleTennis problem, the Joint Frequency Matrix/Table for Wind random variable \( x_3 \) is shown in Table 5.5.

\[ \nu(x_3 = h, y = c) \] for \( h \in \{0,1\}, c \in \{0,1\} \)

Table 5.5: JFT for \( x_3 \)

<table>
<thead>
<tr>
<th>( x_3 )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

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., \( P(x_0 = h \mid y = c) \), is computed by dividing each entry in the joint frequency matrix by its column sum as shown in Table 5.6.

Table 5.6: CPT for \( x_3 \)

<table>
<thead>
<tr>
<th>( x_3 )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>6/9</td>
</tr>
<tr>
<td>1</td>
<td>3/5</td>
<td>3/9</td>
</tr>
</tbody>
</table>

Similar matrices/tables can be created for the other random variables: Temperature \( x_1 \) and Humidity \( x_2 \).

5.6.3 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:

\[
P(x_j = h \mid y = c) = \frac{\nu(x_j = h, y = c) + m_e/vc_j}{\nu(y = c) + m_e}
\]

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 5.7 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>
<thead>
<tr>
<th>$x_0 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7/18</td>
<td>10/30</td>
</tr>
<tr>
<td>1</td>
<td>1/18</td>
<td>13/30</td>
</tr>
<tr>
<td>2</td>
<td>10/18</td>
<td>7/30</td>
</tr>
</tbody>
</table>

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 5.3). The question now is whether this is due to the combination of $x_0 = 1$ 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 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.

### 5.6.4 Hypermatrices

The values within the class probability table and the conditional probability tables are assigned by the `train` method. In SCALATION, vectors and third level hypermatrices are used for storing frequencies ($\nu$) and probabilities ($\mathbf{p}$).

```scala
val nu_y = new VectorI (k) // frequency of y with classes 0, ..., k-1
val nu_Xy = new HMatrix3[Int] (k, n, vc) // joint frequency of features x_j's and class y
val p_y = new VectorD (k) // probability of y with classes 0, ..., k-1
val p_Xy = new HMatrix3[Double] (k, n, vc) // conditional probability of features x_j's // given class y
```

where $k$ is the number of class values, $n$ is the number of $x$-random variables (features) and $vc$ is the value count per feature. Note, one third level hypermatrix is able to store multiple matrices.

For the `ExampleTennis` problem where $k = 2$ and $n = 4$, the frequency counters ($\nu$) would be defined as follows:

```scala
nu_y = new VectorI (2) // Class Frequency Vector (CFV)
nu_Xy = new HMatrix3[Int] (2, 4, Array (3, 3, 2, 2)) // all Joint Frequency Tables (JFTs)
```

The dimensionality of the hypermatrix $\nu_Xy$ could have been 2-by-4-by-3, but this would in general be wasteful of space. Each variable only needs space for the values it allows, as indicated by `Array (3, 3, 2, 2)` for the value counts $vc$. 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.

118
5.6.5 The classify Method

A new instance can now be classified by simply matching its values 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.

```scala
def classify (z: Vector[Int]): (Int, String, Double) =
{
    val prob = new VectorD (p_y)
    for (c <- 0 until k; j <- 0 until n) prob(c) *= p_Xy(c, j, z(j)) // P(x_j = z_j | y = c)
    val best = prob.argmax () // class with the highest relative probability
    (best, cn(best), prob(best)) // return the best class and its name
}  // classify
```

In situations where there are many variables/features the product calculation may underflow. An alternative calculation would be to take the log of the probability.

\[
\log P(x, y) = P(y) + \sum_{j=0}^{n-1} P(x_j | y)
\]

5.6.6 Feature Selection

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>
<thead>
<tr>
<th></th>
<th>( 0 )</th>
<th>( 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>5/14</td>
<td>9/14</td>
</tr>
<tr>
<td>( z_0 )</td>
<td>3/5</td>
<td>2/9</td>
</tr>
<tr>
<td>( z_3 )</td>
<td>3/5</td>
<td>3/9</td>
</tr>
<tr>
<td>( z, y )</td>
<td>9/70</td>
<td>1/21</td>
</tr>
</tbody>
</table>

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 an `fset` array that indicates the features/variables to be kept in the model. This array is assigned by calling the `featureSelection` method in the `ClassifierInt` abstract class.

5.6.7 Efficient Cross-Validation

There are actually two classes `NaiveBayes0` and `NaiveBayes`. The former uses conventional “additive” cross-validation where frequency counters are reset to zero and are incremented for each fold. The latter uses a more efficient “subtractive” cross-validation where frequency counters are reset to the counts for the entire dataset and are decremented for each fold.
NaiveBayes0 Class

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where y(l) = class for row l of the matrix x, x(l)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param vc the value count (number of distinct values) for each feature
@param me use m-estimates (me == 0 => regular MLE estimates)

class NaiveBayes0 (x: Matrix, y: Vector, fn_: Strings = null, k: Int = 2, cn_: Strings = null, 
                  protected var vc: Array[Int] = null, me: Double = me_default)
  extends BayesClassifier (x, y, fn, k, cn)

  def train (itest: IndexedSeq[Int]): NaiveBayes0 =
  protected def frequencies (idx: IndexedSeq[Int])
  protected def updateFreq (i: Int)
  def classify (z: Vector): (Int, String, Double) =
  def lclassify (z: Vector): (Int, String, Double) =
  protected def vlog (p: Vector): Vector = p.map (log (_))
  def reset ()
  def printConditionalProb ()

NaiveBayes Class

This class is the same as the one above, but uses an optimized cross-validation technique.

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where y(l) = class for row l of the matrix x, x(l)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param vc_ the value count (number of distinct values) for each feature
@param me use m-estimates (me == 0 => regular MLE estimates)

class NaiveBayes (x: Matrix, y: Vector, fn_: Strings = null, k: Int = 2, cn_: Strings = null, 
                  vc_: Array[Int] = null, me: Float = me_default)
  extends NaiveBayes0 (x, y, fn, k, cn, vc_, me)
def frequenciesAll ()
protected override def updateFreq (i: Int)
override def reset ()

5.6.8 Exercises

1. Complete the ExampleTennis problem given in this section by creating CPTs for random variables $x_1$ and $x_2$ and then computing the relative probabilities for $z = [2, 2, 1, 1]$.

2. Use SCALATION’s Integer-based NaiveBayes class to build a classifier for the ExampleTennis problem.

```scala
import scalation.analytics.classifier.ExampleTennis._
println ("Tennis Example")
println ("xy = " + xy) // combined matrix [x | y]
val nb = NaiveBayes (xy, fn, k, cn, null, 0) // create a classifier
nb.train () // train the classifier
val z = VectorI (2, 2, 1, 1) // new data vector
println (s"classify ($z) = ${nb.classify (z)}") // classify z
```

3. Compare the confusion matrix, accuracy, precision and recall of NaiveBayes on the full dataset to that of NullModel.

```scala
val x = xy.sliceCol (0, xy.dim2 - 1) // data matrix
val y = xy.col (xy.dim2 - 1) // response/class label vector
val yp = new VectorI (xy.dim1) // predicted class label vector
for (i <- x.range1) {
  yp(i) = nb.classify (x(i))._1
  println (s"Use nb to classify (${x(i)}) = ${yp(i)}")
}
val cm = new ConfusionMat (y, yp, k) // confusion matrix
println ("Confusion Matrix = " + cm.confusion)
println ("accuracy = " + cm.accuracy)
println ("prec-recall = " + cm.prec_recl)
```

4. Compare the accuracy of NaiveBayes using 10-fold cross-validation (cv) to that of NullModel.

```scala
println ("nb cv accu = " + nb.crossValidateRand (10, true)) // 10-fold cross-validation
```

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 ExampleTennis 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.
5.7 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 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 given in equation 5.5,

\[
P(x, y) = P(x|y)P(y)
\]

we can obtain a better factored approximation (better than Naïve Bayes) by keeping the most important dependencies amongst the random variables. Each \( x_j \), except a selected \( x \)-root, \( x_r \), will have one \( x \)-parent \( (x_{p(j)}) \) in addition to its \( y \)-parent. 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 parent.

\[
P(x, y) = P(y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)
\]

Since the root \( x_r \), has no \( x \)-parent, it can be factored out as special case.

\[
P(x, y) = P(y)P(x_r|y) \prod_{j \neq r} P(x_j|x_{p(j)}, y) \quad (5.8)
\]

As with Naïve Bayes, the goal is to find an optimal value for the random variable \( y \) that maximizes the probability.

\[
y^{*} = \arg\max_{y \in D_y} P(y)P(x_r|y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)
\]

5.7.1 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 \( x_j \) and \( x_l \) is

\[
I(x_j; x_l) = \sum_{x_j} \sum_{x_l} p(x_j, x_l) \log \frac{p(x_j, x_l)}{p(x_j)p(x_l)} \quad (5.9)
\]

The Conditional Mutual Information (CMI) between two random variables \( x_j \) and \( x_l \) given a third random variable \( y \) is

\[
I(x_j; x_l|y) = \quad (5.10)
\]

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.

### 5.7.2 Conditional Probability Tables

For the *Example Tennis* problem limited to two variables, \( x_0 \) and \( x_3 \), suppose that structure learning algorithm found the \( x \)-parents as shown in Table 5.9.

<table>
<thead>
<tr>
<th>( x_j )</th>
<th>( x_{P(j)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>null</td>
</tr>
</tbody>
</table>

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 5.10.

<table>
<thead>
<tr>
<th>( x_0 ) ( x_3, y )</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 5.11.

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:

\[
P(x_j = h \mid x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c)}{\nu(x_p = l, y = c)}
\] (5.11)
Table 5.11: Extended CPT for $x_{0,0}$

<table>
<thead>
<tr>
<th>$x_0 \backslash x_3, y$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>2/3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/3</td>
<td>0</td>
<td>2/3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1/6</td>
<td>1/3</td>
<td>1/3</td>
</tr>
</tbody>
</table>

5.7.3 Smoothing

The analog of Laplace smoothing used in Naïve Bayes is the following.

$$P(x_j = h | x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c) + m_e/\nu_j}{\nu(x_p = l, y = c) + m_e}$$

In Friedman’s paper [8], he suggests using the marginal distribution rather than uniform (as shown above), which results in the following formula.

$$P(x_j = h | x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c) + m_e * m p_j}{\nu(x_p = l, y = c) + m_e}$$

where

$$m p_j = \frac{\nu(x_j)}{m}$$

5.7.4 The classify Method

As with Naïve Bayes, the classify simply multiplies entries in the CPV and CPTs (all except the root are extended). Again the class with the highest product is chosen.

```scala
def classify (z: VectoI): (Int, String, Double) =
{
  val prob = new VectorD (p_y)
  for (i <- 0 until k; j <- 0 until n if fset(j)) {
    prob(i) *= (if (parent(j) > -1) p_XyP(i, j, z(j), z(parent(j)))
      else p_XyP(i, j, z(j), 0))
  } // for
  val best = prob.argmax ()
  (best, cn(best), prob(best))
} // classify
```

5.7.5 Cross-Validation

Again there are two classes: TANBayes0 that uses conventional “additive” cross-validation and TANBayes that uses more efficient “subtractive” cross-validation.
The \texttt{TANBayes} class is similar, but uses a more efficient cross-validation method.

\textbf{5.7.6 \textit{Exercises}}

1. Use the Integer-based \texttt{TANBayes} to build classifiers for (a) the ExampleTennis problem and (b) the Breast Cancer problem (data in \texttt{breast-cancer.arff} file). Compare its accuracy to that of \texttt{NullModel} and \texttt{NaiveBayes}.

2. Re-engineer \texttt{TANBayes} to use correlation instead of conditional mutual information. Compare the results with the current \texttt{TANBayes} implementation.
5.8 Forest Augmented Naïve Bayes

The FANBayes class implements a Forest Augmented Naïve (FAN) Bayes Classifier suitable for discrete input data.
5.9 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 known as a Bayesian Network Classifier.

5.9.1 Bayesian Network Classifier

A Bayesian Network Classifier [1] is used to classify a discrete input data vector \( x \) by determining which of \( k \) classes has the highest Joint Probability of \( x \) and the response/outcome \( y \) (i.e., one of the \( k \) classes) of occurring.

\[
P(y, x_0, x_1, \ldots x_{n-1})
\]

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:

\[
P(x_0, x_1, x_2) = P(x_0)P(x_1|x_0)P(x_2|x_0, x_1)
\]

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,

\[
y^* = \arg\max_{y \in D_y} P(y) \prod_{j=0}^{n-1} P(x_j|x_p(j), y)
\]

where \( x_p(j) \) 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.

5.9.2 Structure Learning

For TwoNANBayes the parents of variable \( x_j \) are recoded in a vector \( x_{p(j)} \) 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 [1].

5.9.3 Conditional Probability Tables

Example Problem:

Class Methods:
@param dag the directed acyclic graph specifying conditional dependencies
@param table the array of tables recording conditional probabilities
@param k the number of classes

class BayesNetwork (dag: DAG, table: Array [Map [Int, Double]], k: Int)
   extends Classifier with Error

def jp (x: VectoI): Double =
def cp (i: Int, key: VectoI): Double =
def train ()
override def classify (z: VectoI): Int =
def classify (z: VectoD): Int =
5.10 Decision Tree ID3

A Decision Tree (or Classification Tree) classifier\[20,19\] will take an input vector $x$ 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 $f$.

$$y = f(x) = f(x_0, x_1, \ldots x_{n-1}) \tag{5.12}$$

The DecisionTreeID3\[16\] 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 $y$. Each data vector in the matrix is classified into one of $k$ classes numbered $0, 1, \ldots k - 1$. Each column in the matrix represents a $x$-variable/feature (e.g., Humidity). The value count $vc$ vector gives the number of distinct values per feature (e.g., 2 for Humidity).

### 5.10.1 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: $y = (1, 1, \ldots 1, 1)$ or $y' = (1, 0, \ldots 1, 0)$. In the first case all the decisions are yes, while in the second, there are an equal number of yes and no decisions. One way to measure the level of disorder is Shannon entropy. To compute the Shannon entropy, first convert the $m$-dimensional decision/classification vector $y$ into a $k$-dimensional probability vector $p$. 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, $p = (1, 0)$ and $p' = (.5, .5)$, so computing the Shannon entropy $H(p)$,

$$H(p) = -\sum_{i=0}^{k-1} p_i \log_2(p_i) \tag{5.13}$$

we obtain $H(p) = 0$ and $H(p') = 1$, which indicate that there is no disorder in the first case and maximum disorder in the second case.

```scala
def entropy (p: VectoD): Double =
{
  var sum = 0.0
  for (pi <- p if pi > 0.0) sum -= pi * log2 (pi)
  sum // return entropy, a number in the interval [0, max]
} // entropy
```

Letting the dimensionality of the probability vector be $k$, the maximum entropy is given by $\log_2(1/k)$, which is 1 for $k = 2$. The maximum base-$k$ entropy is always 1.

$$H(p) = -\sum_{i=0}^{k-1} p_i \log_k(p_i)$$

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 \[19\]. For a deeper dive into entropy, relative entropy and mutual information see \[5\].
5.10.2 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 5.12 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

\[
H(p) = H\left(\frac{5}{14}, \frac{9}{14}\right) = -\frac{5}{14} \log_2\left(\frac{5}{14}\right) - \frac{9}{14} \log_2\left(\frac{9}{14}\right) = 0.9403
\]

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:

\[
\sum_{v=0}^{v_{c_j}-1} \frac{\nu(x_j = v)}{m} H(p_{x_j=v})
\]

where \(\nu(x_j = v)\) is the frequency count of value \(v\) for column vector \(x_j\) in matrix \(X\). The sum is the weighted average of the entropy over all possible \(v_{c_j}\) values for variable \(j\).

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.

\[
\sum_{v=0}^{2} \frac{\nu(x_0 = v)}{m} H(p_{x_0=v})
\]

Table 5.12: Tennis Example

<table>
<thead>
<tr>
<th>Day</th>
<th>(x_{-0})</th>
<th>(x_{-1})</th>
<th>(x_{-2})</th>
<th>(x_{-3})</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
For \( v = 0 \), we have 2 no (0) cases and 3 yes (1) cases (2−, 3+), for \( v = 1 \), we have (0−, 4+) and for \( v = 2 \), we have (3−, 2+).

\[
\frac{\nu(x_{-0} = 0)}{14} H(p_{x_{-0}=0}) + \frac{\nu(x_{-0} = 1)}{14} H(p_{x_{-0}=1}) + \frac{\nu(x_{-0} = 2)}{14} H(p_{x_{-0}=2})
\]

\[
\frac{5}{14} H(p_{x_{-0}=0}) + \frac{4}{14} H(p_{x_{-0}=1}) + \frac{5}{14} H(p_{x_{-0}=2})
\]

We are left with computing three entropy values:

\[
H(p_{x_{-0}=0}) = H\left(\frac{2}{5}, \frac{3}{5}\right) = -\frac{2}{5} \log_2\left(\frac{2}{5}\right) - \frac{3}{5} \log_2\left(\frac{3}{5}\right) = 0.9710
\]

\[
H(p_{x_{-0}=1}) = H\left(\frac{0}{4}, \frac{4}{4}\right) = -\frac{0}{4} \log_2\left(\frac{0}{4}\right) - \frac{4}{4} \log_2\left(\frac{4}{4}\right) = 0.0000
\]

\[
H(p_{x_{-0}=2}) = H\left(\frac{3}{5}, \frac{2}{5}\right) = -\frac{3}{5} \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \log_2\left(\frac{2}{5}\right) = 0.9710
\]

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 5.13, the other entropy drops are 0.0292 for Temperature (1), 0.1518 for Humidity (2) and 0.0481 for Wind (3).

**Table 5.13: Choices for Principal Feature**

<table>
<thead>
<tr>
<th>( j )</th>
<th>Variable/Feature</th>
<th>Entropy</th>
<th>Entropy Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Outlook</td>
<td>0.6936</td>
<td>0.2467</td>
</tr>
<tr>
<td>1</td>
<td>Temperature</td>
<td>0.9111</td>
<td>0.0292</td>
</tr>
<tr>
<td>2</td>
<td>Humidity</td>
<td>0.7885</td>
<td>0.1518</td>
</tr>
<tr>
<td>3</td>
<td>Wind</td>
<td>0.8922</td>
<td>0.0481</td>
</tr>
</tbody>
</table>

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.

**Sub-problem \( x_0 = 0 \)**

The sub-problem for Outlook: Rain (0) see Table 5.14 is defined as follows: Take all five cases/rows in the data matrix \( X \) for which \( x_{-0} = 0 \).

**Table 5.14: Sub-problem for node \( x_0 \) and branch 0**

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

If we select Wind (\( j = 3 \)) as the next variable, we obtain the following cases: For \( v = 0 \), we have (0−, 3+), so the probability vector and entropy are
\( p_{x=3} = (\frac{3}{5}, \frac{2}{5}) \quad H(p_{x=3}) = 0 \)

For \( v = 1 \), we have \((2-, 0+)\), so the probability vector and entropy are

\[
\begin{align*}
\mathbf{p}_{x=3} &= (\frac{2}{5}, \frac{3}{5}) \\
H(\mathbf{p}_{x=3}) &= 0
\end{align*}
\]

If we stop expanding the tree at this point, we have the following rules.

if \( x_0 = 0 \) then
  if \( x_3 = 0 \) then yes
  if \( x_3 = 1 \) then no
if \( x_0 = 1 \) then yes
if \( x_0 = 2 \) then no

The overall entropy can be calculated as the weighted average of all the leaf nodes.

\[
\frac{3}{14} \cdot 0 + \frac{2}{14} \cdot 0 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.9710 = 0.3468
\]

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 6.1 is defined as follows: Take all five cases/rows in the data matrix \( X \) for which \( x_0 = 2 \).

Table 5.15: Sub-problem for node \( x_0 \) and branch 2

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It should be obvious that \( y = 1 - x_{-2} \). For \( v = 0 \), we have \((0-, 2+)\), so the probability vector and entropy are

\[
\begin{align*}
\mathbf{p}_{x=3} &= (\frac{0}{5}, \frac{2}{5}) \\
H(\mathbf{p}_{x=3}) &= 0
\end{align*}
\]

For \( v = 1 \), we have \((3-, 0+)\), so the probability vector and entropy are

\[
\begin{align*}
\mathbf{p}_{x=3} &= (\frac{3}{5}, \frac{0}{5}) \\
H(\mathbf{p}_{x=3}) &= 0
\end{align*}
\]

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:
[ Node[0] b-1 : f = x0 ( 5-, 9+ ) ]
    [ Node[1] b0 : f = x3 ( 2-, 3+ ) ]
        [ Leaf[2] b0 : y = 1 ( 0-, 3+ ) ]
        [ Leaf[3] b1 : y = 0 ( 2-, 0+ ) ]
    [ Leaf[4] b1 : y = 1 ( 0-, 4+ ) ]
[ Node[5] b2 : f = x2 ( 3-, 2+ ) ]
    [ Leaf[6] b0 : y = 1 ( 0-, 2+ ) ]
    [ Leaf[7] b1 : y = 0 ( 3-, 0+ ) ]

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.

5.10.3 Early Termination

Producing a complex decision tree with zero entropy may suggest overfitting, 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 in Chapter 6.

5.10.4 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 Chapter 6). The ScalaTion implementation of ID3 does support pruning.

DecisionTreeID3 Class

Class Methods:
- @param x the data vectors stored as rows of a matrix
- @param y the class array, where y_i = class for row i of the matrix x
- @param fn_ the names for all features/variables
- @param k the number of classes
- @param cn_ the names for all classes
- @param vc the value count array indicating number of distinct values per feature
- @param td the maximum tree depth to allow
5.10.5 Exercises

1. 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 \(H(\mathbf{pp})\) versus \(p\).

   ```scala
   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)
   ```

2. The Tennis example (see NaiveBayes) can also be analyzed using decision trees.

   ```scala
   val id3 = new DecisionTreeID3(x, y, fn, k, cn, vc) // create the classifier
   id3.train()
   val z = VectorI(2, 2, 1, 1) // new vector to classify
   println(s"classify ($z) = \${id3.classify(z)}")
   ```

   Use DecisionTreeID3 to build classifiers for the ExampleTennis problem. Compare its accuracy to that of NullModel, NaiveBayes and TANBayes.

3. Do the same for the Breast Cancer problem (data in breast-cancer.arff file).

4. For the Breast Cancer problem, evaluate the effectiveness of the prune method.

5. Again for the Breast Cancer problem, explore the results for various limitations to the maximum tree depth via the \(td\) parameter.
Chapter 6

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.
6.1 ClassifierReal

The ClassifierReal abstract class provides a common foundation for several classifiers that operate on continuous (or real-valued) data.

Class Methods:

@param x the real-valued training data vectors stored as rows of a matrix
@param y the training classification vector, where y_i = class for row i of the matrix x
@param fn the names of the features/variables
@param k the number of classes
@param cn the names for all classes

abstract class ClassifierReal (x: MatriD, y: VectoI, fn: Strings, k: Int, cn: Strings) extends Classifier with Error

def vc_default: Array[Int] = Array.fill(n)(2)
def size: Int = m
def classify (z: VectoI): (Int, String, Double) = classify (z.toDouble)
def classify (xx: MatriD): VectoI =
def test (itest: IndexedSeq[Int]): Double =
def test (xx: MatriD, yy: VectoI): Double =
def calcCorrelation: MatriD =
def calcCorrelation2 (zrg: Range, xrg: Range): MatriD =
def featureSelection (TOL: Double = 0.01)
6.2 Gaussian Naive Bayes

The NaiveBayesR class implements a Gaussian Naive 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 $y$. Each data vector in the matrix is classified into one of $k$ classes numbered $0, 1, \ldots, k-1$.

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 naive, because it assumes feature independence and therefore simply multiplies the conditional densities.

Starting with main results from the section on Naive Bayes (equation 4.5),

$$y^\star = \underset{y \in \{0, \ldots, k-1\}}{\operatorname{argmax}} P(y) \prod_{j=0}^{n-1} P(x_j | y)$$

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 $y^\star$ that maximizes the product of the conditional densities multiplied by the class probability.

$$y^\star = \underset{y \in \{0, \ldots, k-1\}}{\operatorname{argmax}} P(y) \prod_{j=0}^{n-1} f(x_j | y) \quad (6.1)$$

Although the formula assumes the conditional independence of $x_j$s, 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,

$$x_j | y \sim \text{Normal}(\mu_c, \sigma_c^2)$$

where class $c \in \{0, 1, \ldots, k-1\}$, $\mu_c = \mathbb{E}[x | y = c]$ and $\sigma_c^2 = \mathbb{V}[x | y = c]$. Thus, the conditional density function is

$$f(x_j | y = c) = \frac{1}{\sqrt{2\pi} \sigma_c} e^{-\frac{(x - \mu_c)^2}{2\sigma_c^2}}$$

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 $y$. 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:

$$\hat{\mu}_{cj} = \frac{1}{m_c} \sum_{i=0}^{m-1} (x_{ij} | y_i = c)$$

$$\hat{\sigma}_{cj}^2 = \frac{1}{m_c - 1} \sum_{i=0}^{m-1} ((x_{ij} - \hat{\mu}_{cj})^2 | y_i = c)$$

Using conditional density (cd) functions estimated in the train function (see code for details), an input vector $z$ can be classified using the classify function.
def classify (z: VectoD): Int =
{
    for (c <- 0 until k; j <- 0 until n) prob(c) *= cd(c)(j)(z(j))
    prob.argmax () // class c with highest probability
} // classify

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 fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes

class NaiveBayesR (x: MatriD, y: VectoI, fn_ : Strings = null, k: Int = 2,
    cn_ : Strings = null)
    extends ClassifierReal (x, y, fn_, k, cn_)

def calcStats ()
def calcHistogram (x_j: VectoD, intervals: Int): VectoD =
def train (itest: IndexedSeq [Int]): NaiveBayesR =
override def classify (z: VectoD): (Int, String, Double) =
def reset ()

6.2.1 Exercises

1. Use NaiveBayesR to classify manufactured parts according to 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)
    //
    val xy = new MatrixD ((7, 3), 2.95, 6.63, 0,
                         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,
                         140
val fn = Array ("curvature", "diameter") // feature names
val cn = Array ("pass", "fail") // class names
val cl = NaiveBayesR (xy, fn, 2, cn) // create NaiveBayesR classifier
6.3 Simple Logistic Regression

The SimpleLogisticRegression class supports simple logistic regression. In this case, the predictor vector \( x \) is two-dimensional \([1, x_1]\). Again, the goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \epsilon
\]

where \( \epsilon \) 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 \( p_y(x) \in [0, 1] \), i.e.,

\[
p_y(x) = P(y = 1 | x)
\]

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.

6.3.1 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/a412dfbd88b3db70b7b74b). Try using \( mpg \) to predict/classify the car’s engine as either V-shaped(0) or Straight(1), as in V-6 or S-4. First, use SimpleRegression to predict \( p_y(x) \) where \( y \) is V/S and \( x_1 \) is \( mpg \), \((x = [1, x_1])\). 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 \( p_y(x) \) would correspond to the following equation.

\[
p_y(x) = b_0 + b_1 x_1
\]

6.3.2 Logistic Function

The linear relationship between \( y \) and \( x_1 \) is 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

\[
\text{logistic}(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z}
\]

Letting \( z = b_0 + b_1 x_1 \), we obtain

\[
p_y(x) = \text{logistic}(b_0 + b_1 x_1) = \frac{e^{b_0 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}}
\]

6.3.3 Logit Function

The goal now is to transform the right hand side into the usual linear form (i.e., \( b \cdot x \)).

\[
p_y(x) = \frac{e^{b \cdot x}}{1 + e^{b \cdot x}}
\]
Multiplying through by $1 + e^{b \cdot x}$ gives

$$p_y(x) + e^{b \cdot x} p_y(x) = e^{b \cdot x}$$

Solving for $e^{b \cdot x}$ yields

$$e^{b \cdot x} = \frac{p_y(x)}{1 - p_y(x)}$$

Taking the natural logarithm of both sides gives

$$\ln \frac{p_y(x)}{1 - p_y(x)} = b \cdot x = b_0 + b_1 x_1$$

where the function on the left hand side is called the logit function.

$$\text{logit}(p_y(x)) = b \cdot x = b_0 + b_1 x_1$$

Putting the model in this form shows it is a special case of a Generalized Linear Model (see Chapter 7) and will be useful in the estimation procedure.

### 6.3.4 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 $b$ (treated as a random vector) will impact the quality of the model. Define a function of $b$ that will be maximized when the parameters are ideally set to generate the observed data.

#### 6.3.5 Likelihood Function

We can think of this function as the likelihood of $b$ given the predictor vector $x$ and the response variable $y$.

$$L(b|x, y)$$

In this case, $y \in \{0, 1\}$, so if we estimate the likelihood for a single data instance (or row), we have

$$L(b|x, y) = p_y(x)^y (1 - p_y(x))^{1-y}$$

If $y = 1$, then $L = p_y(x)$ and otherwise $L = 1 - p_y(x)$. 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 $b$ given the predictor matrix $X$ and and the response vector $y$ is then

$$L(b|x, y) = \prod_{i=0}^{m-1} p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}$$

143
6.3.6 Log-likelihood Function

To reduce round-off errors, a log (e.g., natural log, \( \ln \)) is taken

\[
l(b|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i \ln(p_y(x_i)) + (1 - y_i)\ln(1 - p_y(x_i))
\]

This is referred as the log-likelihood function. Collecting \( y_i \) terms give

\[
l(b|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i \ln p_y(x_i) - \ln(1 + e^{b \cdot x_i}) \quad (6.8)
\]

Multiplying the log-likelihood by -2 makes the distribution approximately Chi-square [?].

\[
-2l = -2 \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\]

Or since \( b = [b_0, b_1] \),

\[
-2l = -2 \sum_{i=0}^{m-1} y_i (b_0 + b_1 x_{i1}) - \ln(1 + e^{b_0 + x_{i1}})
\]

Letting \( \beta_i = b_0 + b_1 x_{i1} \) gives

\[
-2l = -2 \sum_{i=0}^{m-1} y_i \beta_i - \ln(1 + e^{\beta_i})
\]

It is more numerically stable to perform a negative rather than positive \( e^z \) function.

\[
-2l = -2 \sum_{i=0}^{m-1} y_i \beta_i - \beta_i - \ln(e^{-\beta_i} + 1) \quad (6.9)
\]

6.3.7 Computation in SCALATION

The computation of \( -2l \) is carried out in SCALATION via the \( \text{ll} \) method. It loops through all instances computing \( \beta_i \) (bx in the code) and summing all the terms given in equation 6.9.
def ll (b: VectoD): Double =
{
  var sum = 0.0
  var bx = 0.0 // beta
  for (i <- y.range) {
    bx = b(0) + b(1) * x(i, 1)
    sum += y(i) * bx - bx - log (exp (-bx) + 1.0)
  } // for
  -2.0 * sum
} // ll

SimpleLogisticRegression Class

Class Methods:

@param x the input/input matrix augmented with a first column of ones
@param y the binary response vector, y_i in {0, 1}
@param fn_ the names for all features/variable
@param cn_ the names for both classes

class SimpleLogisticRegression (x: MatriD, y: VectoI, fn_: Strings = Array ("one", "x1"),
                                 cn_: Strings = null)
  extends ClassifierReal (x, y, fn_, 2, cn_)

def ll (b: VectoD): Double =

def ll_null (b: VectoD): Double =

def train (itest: IndexedSeq[Int]): SimpleLogisticRegression =

def train_null ()

def parameter: VectoD = b

override def fit (y: VectoI, yp: VectoI, k: Int = 2): VectoD =

override def fitLabel: Seq[String] = super.fitLabel ++
  Seq ("n_dev", "r_dev", "aic", "pseudo_rSq")

override def classify (z: VectoD): (Int, String, Double) =

def reset () { /* Not Applicable */ }

6.3.8 Exercises

1. Plot the standard logistic function (sigmoid).

   import scalation.analytics.ActivationFun.sigmoidV
   val z = VectorD.range (0, 160) / 10.0 - 8.0
val fz = sigmoidV (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 \( \ell_l \) function. Use the \( x \) matrix (\texttt{one, mpg}) and \( y \) vector (V/S) from SimpleLogisticRegressionTest.

```scala
// 32 data points: One Mpg
val x = new 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, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1)
```

6.4 Logistic Regression

The LogisticRegression class supports logistic regression. In this case, \( x \) may be multi-dimensional \([1, x_1, \ldots, x_k]\). Again, the goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1x_1 + \ldots + b_kx_k + \epsilon
\]

where \( \epsilon \) 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 \( p_y(x) \in [0, 1] \), i.e.,

\[
p_y(x) = P(y = 1 | x)
\]

Still, \( p_y(x) \) is bounded, while \( b \cdot x \) is not. We therefore, need a transformation, such as the logit transformation, and fit \( b \cdot x \) to this function. Treating this as a GZLM problem,

\[
y = \mu(x) + \epsilon
\]

\[
g(\mu(x)) = b \cdot x
\]

we let the link function \( g = \text{logit} \).

\[
\text{logit}(\mu(x)) = \ln \frac{p_y(x)}{1 - p_y(x)} = b \cdot x
\]

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 \( i \)th training case \( x_i \) with outcome \( y_i \), the likelihood function is based on the Bernoulli distribution.

\[
p_y(x_i)^{y_i}(1 - p_y(x_i))^{1-y_i}
\]

The overall likelihood function is the product over all \( m \) cases. The equation is the same as 6.6 from the last section.

\[
L(b|x, y) = \prod_{i=0}^{m-1} p_y(x_i)^{y_i}(1 - p_y(x_i))^{1-y_i}
\]  (6.10)

Following the same derivation steps, will give the same log-likelihood that is in equation 6.7.

\[
l(b|x, y) = \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\]  (6.11)

Again, multiplying the log-likelihood function by \(-2\) makes the distribution approximately Chi-square.

\[
-2l = -2 \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\]

The likelihood can be maximized by minimizing \(-2l\), which is a non-linear function of the parameter vector \( b \). Various optimization techniques may be used to search for optimal values for \( b \). 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](http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf).
LogisticRegression Class

Class Methods:

@param x the input/data matrix augmented with a first column of ones
@param y the binary response vector, y_i in {0, 1}
@param fn_ the names for all features/variable
@param cn_ the names for all classes

class LogisticRegression (x: Matrix, y: VectorI, fn_: Strings = null,
                                 cn_: Strings = null)
          extends ClassifierReal (x, y, fn_, 2, cn_)

    def ll (b: VectorD): Double =
    def ll_null (b: VectorD): Double =
    def train (itest: IndexedSeq[Int]): LogisticRegression =
    def train_null () =
    def parameter: VectorD = b
    override def fit (y: VectorI, yp: VectorI, k: Int = 2): VectorD =
    override def fitLabel: Seq[String] = super.fitLabel ++
                                  Seq ("n_dev", "r_dev", "aic", "pseudo_rSq")
    override def classify (z: VectorD): (Int, String, Double) =
    def forwardSel (cols: Set[Int], adjusted: Boolean = true):
            (Int, VectorD, VectorD) =
    def backwardElim (cols: Set[Int], adjusted: Boolean = true, first: Int = 1):
            (Int, VectorD, VectorD) =
    def vif: VectorD =
    def reset () { /* Not Applicable */ }

6.4.1 Exercises

1. Use Logistic Regression to classify whether stock market will be increasing or not. The Smarket dataset is in the ISLR library, see [12] section 4.6.2.

2. Use Logistic Regression to classify whether a customer will purchase caravan insurance. The Caravan dataset is in the ISLR library, see [12] section 4.6.6.
6.5 Simple Linear Discriminant Analysis

The SimpleLDA class supports Linear Discriminant Analysis which is useful for multiway classification of continuously valued data. The response/classification variable can take on \( k \) possible values, \( y \in \{0, 1, \ldots, k-1\} \). 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 \( x \).

\[
P(y|x) = \frac{P(x|y)P(y)}{P(x)}
\]

Since \( x \) is now continuous, we need to work with conditional densities as done Gaussian Naïve Bayes classifiers,

\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)} \tag{6.12}
\]

where

\[
f(x) = \sum_{c=0}^{k-1} f(x|y = c)P(y = c)
\]

Now let us assume the conditional probabilities are normally distributed with a common variance.

\[
x|y \sim \text{Normal}(\mu_c, \sigma^2)
\]

where class \( c \in \{0, 1, \ldots, k-1\} \), \( \mu_c = \mathbb{E}[x|y = c] \) and \( \sigma^2 \) is the pooled variance (weighted average of \( \mathbb{V}[x|y = c] \)). Thus, the conditional density function is

\[
f(x|y = c) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}
\]

Substituting into equation 6.10 gives

\[
P(y|x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_y)^2}{2\sigma^2}} P(y) \tag{6.13}
\]

where

\[
f(x) = \sum_{c=0}^{k-1} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y = c)
\]

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.

\[
P(y|x) \propto \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_y)^2}{2\sigma^2}} P(y)
\]

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
\[
\ln(P(y|x)) \propto \frac{-(x - \mu_c)^2}{2\sigma^2} + \ln(P(y))
\]

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).

\[
\ln(P(y|x)) \propto \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y)) \quad (6.14)
\]

The right hand side functions in 4.12 are linear in \(x\) and are called discriminant functions \(\delta_c(x)\).

Given training data vectors \(x\) and \(y\), define \(x_c\) (or \(x_c\) 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:

\[
\hat{\mu}_c = \frac{1}{m_c} \cdot x_c
\]

The common variance may be estimated using a pooled variance estimator.

\[
\hat{\sigma}^2 = \frac{1}{m - k} \sum_{c=0}^{k-1} ||x_c - \mu_c||^2
\]

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 \texttt{train} method. It estimates the class probability vector \(p_y\), the group means vector \(\mu\) and the pooled variance. The vectors \texttt{term1} and \texttt{term2} capture the \(x\)-term \((\mu_c/\sigma^2)\) and the constant term \((\mu_c^2/2\sigma^2 - \ln(P(y)))\) in equation 6.12.

\[
\texttt{def train (itest: IndexedSeq [Int]): SimpleLDA =}
\]

\{
    py = VectorD (xc.map (_.dim / md)) // probability \(y = c\)
    mu = VectorD (xc.map (_.mean)) // group means
    var sum = 0.0
    for (c <- 0 until k) sum += (xc(c) - mu(c)).normSq
    sig2 = sum / (m - k).toDouble // pooled variance
    term1 = mu / sig2
    term2 = mu~^2 / (2.0 * sig2) - py.map (log (_))
    this
\} // train

Given the two precomputed terms, the \texttt{classify} method simply multiplies the first by \(z(0)\) and subtracts the second. Then it finds the \texttt{argmax} of the \texttt{delta} vector to return the class with the maximum \texttt{delta}, which corresponds the most probable classification.

\[
y^* = \argmax_c \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y)) \quad (6.15)
\]

\[
\texttt{override def classify (z: VectoD): (Int, String, Double) =}
\]

\{
    val delta = term1 * z(0) - term2
    val best = delta.argmax ()
    (best, cn(best), delta(best))
\} // classify
Class Methods:

@param x the real-valued training/test data values stored in a vector
@param y the training/test classification vector, where y_i = class for x_i
@param fn_ the name of the feature/variable
@param k the number of possible values for y (0, 1, ... k-1)
@param cn_ the names for all classes

class SimpleLDA (x: VectoD, y: VectoI, fn_ : Strings = Array ("x1"), k: Int = 2,
    cn_ : Strings = null)
    extends ClassifierReal (MatrixD (Seq (x)), y, fn_, k, cn_)

def train (itest: IndexedSeq [Int]): SimpleLDA =
override def classify (z: VectoD): (Int, String, Double) =
def reset () { /* Not Applicable */ }

6.5.1 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.analytics.classifier.SimpleLDAUtil.
6.6 Linear Discriminant Analysis

Like SimpleLDA, the LDA class supports Linear Discriminant Analysis that is used for multiway classification of continuously valued data. Similarly, the response/classification variable can take on \( k \) possible values, \( y \in \{0, 1, \ldots, k-1\} \). Unlike SimpleLDA, this class is intended for cases where the feature vector \( x \) is multi-dimensional. The classification \( y = c \) is chosen to maximize the conditional probability of class \( y \) given the \( n \)-dimensional data/feature vector \( x \).

\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)}
\]

(6.16)

where

\[
f(x) = \sum_{c=0}^{k-1} f(x|y=c)P(y=c)
\]

In the multi-dimensional case, \( x|y \) has a multivariate Gaussian distribution, \( \text{Normal}(\mu_c, \Sigma) \), where \( \mu_c \) are the mean vectors \( \mathbb{E}[x|y=c] \) and \( \Sigma \) is the common covariance matrix (weighted average of \( \mathbb{C}[x|y=c] \). The conditional density function is given by

\[
f(x|y=c) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c)}
\]

Dropping factors independent of \( c \) and multiplying by \( P(y=c) \) gives

\[
f(x|y=c)P(y=c) \propto e^{-\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c)} P(y=c)
\]

Taking the natural logarithm

\[
\ln(P(y|x)) \propto -\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c) + \ln(P(y=c))
\]

The discriminant functions are obtained by multiplying out and again dropping terms independent of \( c \).

\[
\delta_c(x) = x^t \Sigma^{-1} \mu_c - \frac{\mu_c^t \Sigma^{-1} \mu_c}{2} + \ln(P(y=c))
\]

(6.17)

As in the last section, the means for each class \( c (\mu_c) \), the common covariance matrix (\( \Sigma \)), and the class probabilities (\( P(y) \)) must be estimated.

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 \( y_i = \text{class for row } i \) of the matrix \( x \)
@param fn_ the names for all features/variables
@param k the number of classes (\( k \) in \( \{0, 1, \ldots, k-1\} \)
@param cn_ the names for all classes

class LDA (x: MatrixD, y: VectorI, fn_: Strings = null, k: Int = 2, cn_: Strings = null)
extends ClassifierReal (x, y, fn_, k, cn_)

    def corrected_cov (xc: MatrixD): MatrixD = (xc.t * xc) / xc.dim1
def train (itest: IndexedSeq [Int]): LDA =
def reset () { /* Not Applicable */ }
override def classify (z: VectoD): (Int, String, Double) =

6.6.1 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.
6.7 K-Nearest Neighbors Classifier

The KNNClassifier class is used to classify a new vector $z$ into one of $k$ classes $y \in \{0, 1 \ldots k - 1\}$. It works by finding its $\kappa$-nearest neighbors to the point $z$. These neighbors essentially vote according to their classification. The class with the most votes is selected as the classification of vector $z$. Using a distance metric, the $\kappa$ vectors nearest to $z$ are found in the training data, which are stored row-wise in data matrix $X$. The corresponding classifications are given in vector $y$, such that the classification for vector $x_i$ 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 $x$ to vector $z$ is then

$$d(x) = d(x, z) = ||x - z||^2$$

The distance metric is used to collect the $\kappa$ nearest vectors into set $top_\kappa(z)$, such that there does not exists any vector $x_j \notin top_\kappa(z)$ that is closer to $z$.

$$top_\kappa(z) = \{x_i | i \in \{0, \ldots \kappa - 1\} \text{ and } \exists(x_j \notin top_\kappa(z) \text{ and } d(x_j) < d(x_i))\}$$

In case of ties for the most distant point to include in $top_\kappa(z)$ one could pick the first point encountered or the last point. A less biased approach would be to randomly break the tie.

Now $y(top_\kappa(z))$ can be defined to be the vector of votes from the members of the set, e.g., $y(top_3(z)) = [1, 0, 1]$. The ultimate classification is then simply the mode (most frequent value) of this vector (e.g., 1 in this case).

$$y^* = \text{mode } y(top_\kappa(z))$$

6.7.1 Lazy Learning

Training in the KNNClassifier class is lazy, i.e., the work is done in the classify method, rather than the train method.

```scala
override def classify (z: VectorD): (Int, String, Double) =
{
  kNearest (z) // set topK to kappa nearest
  for (i <- 0 until kappa) count(y(topK(i)._1)) += 1 // tally votes per class
  val best = count.argmax () // class with maximal count
  reset () // reset topK and counters
  (best, cn(best), count(best)) // return best class, its name and votes
} // classify
```

The kNearest method finds the $\kappa \times x$ vectors closest to the given vector $z$. This method updates $topK$ by replacing the most distant $x$ 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 $z$. Each of these selected vectors will have their vote taken, voting for the class for which it is labelled. These votes are tallied in the count vector. The class with the highest count will be selected as the best class.
KNN Classifier Class

Class Methods:

@param x the vectors/points of classified data stored as rows of a matrix
@param y the classification of each vector in x
@param fn_ the names of the features/variables
@param k the number of classes
@param cn_ the names for all classes
@param kappa the number of nearest neighbors to consider

class KNN_Classifier (x: MatrixD, y: VectorI, fn_: Strings = null, k: Int = 2, cn_: Strings = null, kappa: Int = 3)
    extends ClassifierReal (x, y, fn_, k, cn_)

def distance (x: VectorD, z: VectorD): Double = (x - z).normSq

def kNearest (z: VectorD)

def train (itest: IndexedSeq[Int]): KNN_Classifier =

override def classify (z: VectorD): (Int, String, Double) =

def reset ()

6.7.2 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 = new MatrixD ((10, 3), 1, 5, 1, 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 adapted to work for prediction problems?
6.8 Decision Tree C45

The DecisionTreeC45 class implements a Decision Tree classifier that uses the C4.5 algorithm. The classifier is trained using an \( m \times n \) data matrix \( X \) and an \( n \)-dimensional classification vector \( y \). 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 \( \text{vc} \) 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 \( \text{isCont} \) if it is continuous or relatively large ordinal. For a column that \( \text{isCont} \), 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 \textit{gain}) is the one that is chosen.

6.8.1 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.

```scala
object ExampleTennisCont {
  val xy = new 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
                        )
}
```
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 \( b_0 \) will lead to a node where as before \( x_3 \) is chosen. Branch \( b_1 \) will lead to a leaf node. Finally, branch \( b_2 \) 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 6.1 is defined as follows: Take all five cases/rows in the data matrix \( X \) for which \( x_0 = 2 \).

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>85</td>
<td>85</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>90</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>95</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>69</td>
<td>70</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>75</td>
<td>70</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

**DecisionTreeC45 Class**

**Class Methods:**

- `@param x` the data vectors stored as rows of a matrix
- `@param y` the class array, where \( y_i \) = class for row \( i \) of the matrix \( x \)
- `@param fn_` the names for all features/variables
- `@param isCont` ‘Boolean’ value to indicate whether according feature is continuous
@param k the number of classes
@param cn_ the names for all classes
@param vc the value count array indicating number of distinct values per feature
@param td the maximum tree depth allowed (defaults to 0 => n, -1 => no depth constraint)

class DecisionTreeC45 (val x: MatriD, val y: VectoI, fn_: Strings = null, isCont: Array[Boolean],
   k: Int = 2, cn_: Strings = null, private var vc: Array[Int] = null,
   private var td: Int = 0)
   extends ClassifierReal (x, y, fn_, k, cn_)

def frequency (dset: (MatriD, VectoI), f: Int, value: Double, cont: Boolean = false, thres: Double = 0):

def gain (dset: (MatriD, VectoI), f: Int): (Double, VectoI) =

def calThreshold (f: Int, dset: (MatriD, VectoI))

def train (itest: IndexedSeq[Int]) = // FIX the logic

def buildTree (dset: (MatriD, VectoI), path: List[(Int, Int)], depth: Int): Node =

def printTree ()

override def classify (z: VectoD): (Int, String, Double) =

def reset ()

6.8.2 Exercises

1. Run DecisionTreeC45 on the ExampleTennis dataset and verify that it produces the same answer as DecisionTreeID3.

2. Complete the C45 Decision Tree for the ExampleTennisComp problem.

3. Run DecisionTreeC45 on the winequality-white dataset. Plot the accuracy versus the maximum tree depth (td).
6.9 Random Forest

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’ and a subset of the columns/features. The fraction of rows used is given by ‘bR’ the bagging ratio, while the number of columns used is given by ‘fS’ the number of features used in building trees. 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.

Class Methods:

@param x the data matrix (instances by features)
@param y the response class labels of the instances
@param nF the number of trees
@param bR bagging ratio (the portion of samples used in building trees)
@param fS the number of features used in building trees
@param k the number of classes
@param s seed for randomness
@param fn_ feature names (array of string)
@param cn_ class names (array of string)

class RandomForest (x: MatrixD, y: VectorI, nF: Int, bR: Double, fS: Int, k: Int, s: Int, 
val fn_: Strings = null, val cn_: Strings = null) 
extends ClassifierReal (x, y, fn_, k, cn_) with Error

def createSubsample (): MatrixD =
def selectSubFeatures (subSample: MatrixD): (MatrixD, VectorI) =
def train (testStart:Int, testEnd:Int) =
def classify (z: VectorD): (Int, String, Double) =
def reset() {}
6.10 Support Vector Machine

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 \( x_i \) is stored as a row in the \( x \) matrix.

**Example Problem:**

**Class Methods:**

- `@param x` the matrix consisting of vectors
- `@param y` the vector of outcomes (e.g., positive(1), negative(-1))
- `@param fn_` the names of the features/variables
- `@param cn_` the class names

```scala
class SupportVectorMachine (x: MatrixD, y: VectorI, fn_ : Strings = null, cn_ : Strings = Array ("-", "+")) extends ClassifierReal (x, y, fn_, 2, cn_)

def l_D (a: VectorD): Double =
def g (a: VectorD): Double = a dot y
def find_w ()
def find_b ()
def train ()
def fit: (VectorD, Double) = (w, b)
def classify (z: VectorD): Int = (signum (w dot z + b)).toInt
```

160
Generalized Linear Models

A Generalized Linear Model (GZLM) can be developed using the GZLM class. One way to think about such models is to separate the GLM regression equation into two steps. In the first step, $y$ is determined by summing a mean function $\mu(x) = E[y|x]$ and an error term (or multiplying in the case of multiplicative errors).

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

In the second step, the mean function is related to a linear combination of the predictor variables, i.e., $b \cdot x$

$$g(\mu(x)) = b \cdot x$$

where $g$ is a function that links $y$’s mean to a linear combination of the predictor variables. When $g$ is the identify function and residuals/errors are Normally distributed, we have a General Linear Model (GLM).

Several additional combinations of link functions and residual distributions are commonly used as shown in the table below.

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Response Type (y)</th>
<th>Link Function</th>
<th>Residual Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>binary {0, 1}</td>
<td>logit</td>
<td>Bernoulli Distribution</td>
</tr>
<tr>
<td>Poisson Regression</td>
<td>integer {0, \ldots \infty}</td>
<td>ln</td>
<td>Poisson Distribution</td>
</tr>
<tr>
<td>Exponential Regression</td>
<td>continuous [0, \infty)</td>
<td>ln or reciprocal</td>
<td>Exponential Distribution</td>
</tr>
<tr>
<td>General Linear Model (GLM)</td>
<td>continuous (−\infty, \infty)</td>
<td>identity</td>
<td>Normal Distribution</td>
</tr>
</tbody>
</table>

Table 7.1: Types of Generalized Linear Models


Since the response variable for Logistic Regression is defined on finite domains, it has been placed under Classification (see the next chapter).

Example Problem:

Class Methods:
object GZLM extends GLM

def apply (x: MatriD, y: VectoI, cn: Array [String]): LogisticRegression =
def apply (x: MatriD, y: VectoI, fn: Array [String], poisson: Boolean): PoissonRegression =
def apply (x: MatriD, nonneg: Boolean, y: VectoD): ExpRegression =

7.0.1 Further Reading

1. Generalized Linear Models (GLM) [24]
7.1 Exponential Regression

The `ExpRegression` class can be used for developing Exponential Regression models. The response variable \( y \) is estimated by the product of a mean function and exponentially distributed residuals/errors \( \epsilon \).

\[
y = \mu(x) \epsilon
\]

The probability density function (pdf) for the Exponential distribution may be defined as follows:

\[
f(t; \lambda) = \lambda e^{-\lambda t}
\]

The link function \( g \) for Exponential Regression is the \( \ln \) function (alternatively the reciprocal function).

\[
g(\mu(x)) = \ln(\mu(x)) = b \cdot x
\]

Expanding the dot product and using the inverse link function yields the following:

\[
\mu(x) = e^{b \cdot x} = e^{b_0 + b_1 x_1 + \ldots + b_k x_k}
\]

The residuals \( \epsilon_i = y_i / \mu(x_i) \) are distributed Exponential(1), so

\[
f(y_i / \mu(x_i)) = \frac{1}{\mu(x_i)} e^{-y_i / \mu(x_i)}
\]

Therefore, the likelihood function for Exponential Regression is as follows:

\[
L = \prod_{i=0}^{m-1} \frac{1}{\mu(x_i)} e^{-y_i / \mu(x_i)}
\]

Substituting for \( \mu(x_i) \) gives

\[
L = \prod_{i=0}^{m-1} e^{-b \cdot x_i} e^{-y_i / e^{b \cdot x_i}}
\]

Taking the natural logarithm gives the log-likelihood function.

\[
LL = \sum_{i=0}^{m-1} -b \cdot x_i - \frac{y_i}{e^{b \cdot x_i}}
\]

See [http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf](http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf) for more details.

ExpRegression Class

Class Methods:
@param x the data/input matrix
@param y the response vector
@param nonneg whether to check that responses are nonnegative

class ExpRegression (x: MatriD, y: VectoD, nonneg: Boolean)
    extends PredictorMat (x, y)

def ll (b: VectoD): Double =
def ll_null (b: VectoD): Double =
def train (yy: VectoD = y): ExpRegression =
def train_null ()
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
7.2 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 nonnegative integer value. The probability density function (pdf) for the Poisson distribution with mean \(\lambda\) may be defined as follows:

\[
f(y; \lambda) = \frac{\lambda^y}{y!} e^{-\lambda}
\]

Again, treating this as a GZLM problem,

\[
y = \mu(x) + \epsilon \quad \text{and} \quad g(\mu(x)) = b \cdot x
\]

The link function \(g\) for Poisson Regression is the \(\ln\) (natural logarithm) function.

\[
\ln(\mu(x)) = b \cdot x
\]

The residuals \(\epsilon_i\) are distributed according to the Poisson distribution.

\[
\frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)}
\]

Therefore, the likelihood function for Poisson Regression is as follows:

\[
L = \prod_{i=0}^{m-1} \frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)}
\]

Taking the natural logarithm gives the log-likelihood function.

\[
LL = \sum_{i=0}^{m-1} y_i \ln(\mu(x_i)) - \mu(x_i) - \ln(y_i!)
\]

Substituting \(\mu(x_i) = e^{b \cdot x_i}\) yields the following:

\[
LL = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i} - \ln(y_i!)
\]

Since the last term is independent of the parameters, removing it will not affect the optimization.

\[
LL_2 = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i}
\]

See [http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf](http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf) for more details.

---

**Example Problem:**

---

**Class Methods:**

---

165
@param x the input/data matrix augmented with a first column of ones
@param y the integer response vector, \( y_i \) in \( \{0, 1, \ldots\} \)
@param fn the names of the features/variable

class PoissonRegression (x: MatriD, y: VectoI, fn: Array[String] = null)
    extends Classifier with Error

def ll (b: VectoD): Double =
def ll_null (b: VectoD): Double =
def train (yy: VectoD) { throw new UnsupportedOperationException("train (yy) not implemented yet") }
def train ()
def train_null ()
override def fit: VectoD =
override def fitLabels: Seq[String] = Seq("n_dev", "r_dev", "aic", "pseudo_rSq")
def predict (z: VectoD): Double = (round (exp (b dot z))).toDouble
Chapter 8

Generalized Additive Models

A Generalized Additive Model (GAM) can be developed using the \texttt{GZLM} class.
8.1 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 \( \theta_k \) be a threshold for splitting variable \( x_j \), we may split the rows in the \( X \) matrix into left and right groups.

\[
\text{left}_k(X) = \{x_i | x_{ij} \leq \theta_k\} \\
\text{right}_k(X) = \{x_i | x_{ij} > \theta_k\}
\]

For splitting variable \( x_j \), the threshold \( \theta_k \) should be chosen to minimize the 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 \( \mu(y) \).

8.1.1 Example Problem

Consider the following small dataset with just one predictor variable \( x_0 \).

\[
\begin{align*}
\text{val } x &= \text{new MatrixD ((10, 1), 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)} \\
\text{val } y &= \text{VectorD (5.23, 5.7, 5.91, 6.4, 6.8, 7.05, 8.9, 8.7, 9.0, 9.05)}
\end{align*}
\]

In this case, \( \theta_0 = 6.5 \) divides the dataset into

\[
\begin{align*}
\text{left}_0(X) = \{1, 2, 3, 4, 5, 6\} \\
\text{right}_0(X) = \{7, 8, 9, 10\}
\end{align*}
\]

with means \( \mu_0(y) = 6.18 \) (left) and \( \mu_1(y) = 8.91 \) (right). 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 \( x_0 \) in (-Inf, 6.5]
   Leaf \( x_0 \) in (-Inf, 3.5]
   Leaf \( x_0 \) in (3.5, 6.5]
Node \( x_0 \) in (6.5, Inf]
   Leaf \( x_0 \) in (6.5, 8.5]
   Leaf \( x_0 \) 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 \( \theta_2 = 8.5 \).
8.1.2 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,

$$ I_l(x) = x_j \in (a_l, b_l] $$

which simply indicates (false/true) whether variable $x_j$ is in the interval $(a_l, b_l]$. Now define $I^*_l(x)$ as the product of the indicator functions from leaf $l$ until (not including) the root of the tree,

$$ I^*_l(x) = \prod_{h \in \text{anc}(l)} I_h(x) $$

where $\text{anc}(l)$ is the set of ancestors of leaf node $l$ (inclusive of $l$, exclusive of root). Since only one of these $I^*$ indicator functions can be true for any given $x$ vector, we may concisely express the regression tree model as follows:

$$ y = \sum_{l \in \text{leaves}} I^*_l(x) \mu(y) + \epsilon $$

Thus, given a predictor vector $x$, predicting a value for the response variable $y$ corresponds to taking the mean $y$-value of the vectors in $x$’s composite region (the intersection of regions from the leaf until the root). As locality determines the prediction for Regression Trees, they are similar to K-NN Predictors.

8.1.3 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 $\theta_k$ that minimizes the sum of the MSEs.

$$ \min_{\theta_k} \text{mse}(\text{left}_k(X)) + \text{mse}(\text{right}_k(X)) $$

Possible values for $\theta_k$ are the values between any two consecutive values in vector $x_j$ sorted. This will allow any possible split of $x_j$ 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 straightforward way to implement determining the next variable $x_j$ and its threshold $\theta_k$ would be to iterate over all features/variables and split points. Calculating the sum of left and right mse (or sse) from scratch for each candidate split point is inefficient. These values may be computed incrementally using the fast thresholding algorithm [6].

RegressionTree Class

Class Methods:

- @param x the data vectors stored as rows of a matrix
- @param y the dependent value
- @param fn the names for all features/variables
class RegressionTree (x: MatrID, y: VectoD, fn: Array[String], maxDepth: Int,
curDepth: Int, branchValue: Int, thres: Double, feature: Int)
extends PredictorMat (x, y)

def split (f: Int, thresh: Double): (Array[Int], Array[Int]) =
def fastThreshold (f: Int, subSamle: VectoI = null)
def nextXY (f: Int, side: Int): (MatrID, VectoD) =
def train (yy: VectoD): RegressionTree =
def train (interval: VectoI)
def buildTree (opt: (Int, Double))
override def eval (xx: MatrID, yy: VectoD) =
def printTree ()
override def predict (z: VectoD): Double =
override def predict (z: MatrID): VectorD =
def crossVal (k: Int = 10, rando: Boolean = true): Array[Statistic]
def reset ()

8.1.4 Exercises

1.

2. Consider the following two-dimensional Regression Tree problem. FIX.

3. Contrast K-NN Predictors with Regression Trees in terms of the shape of and how regions are formed.
Chapter 9

Non-Linear Models
9.1 Non-Linear Regression

The `NonLinRegression` class supports non-linear regression. In this case, \( x \) can be multi-dimensional \([1, x_1, ... x_k]\) and the function \( f \) is non-linear in the parameters \( b \). As before, the goal is to fit the parameter vector \( b \) in the regression equation

\[
y = f(x; b) + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). 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.

\[
y = \frac{(b_0 + b_1 x_1)}{(b_2 + x_1)} + \epsilon
\]

A training set consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector \( b \). Given an input matrix \( X \) consisting of \( m \) input vectors and an output vector \( y \) consisting of \( m \) output values, minimize the distance between the target output vector \( y \) and the predicted output vector \( f(X; b) \).

\[
\min_b \|y - f(X; b)\|
\]

Again, it is convenient to minimize the dot product of the error with itself,

\[
p(b) = (y - f(X; b)) \cdot (y - f(X; b))
\]

For non-linear regression, a Least-Squares (minimizing the residuals) method can be used to fit the parameter vector \( b \). Unlike the linear case (where one simply sets the gradient to zero), since the formula is non-linear in \( b \), Non-Linear Programming (NLP) is used to minimize the Sum of Squares Error (\( SS_E \)). A user defined function \( f: (\text{VectorD}, \text{VectorD}) \Rightarrow \text{Double} \) is passed as a class parameter. This function is used to create a predicted output value \( z_i \) for each input vector \( x_i \). The `sseF` method applies this function to all \( m \) input vectors to compute predicted output values. These are then subtracted from the target output to create an error vector \( e \), which when dot producted with itself yield \( SS_E \).

```scala
def sseF (b: VectorD): Double =
{
    val z = new VectorD (m) // create vector z to hold predicted outputs
    for (i <- 0 until m) z(i) = f (x(i), b) // compute values for z
    val e = y - z // residual/error vector
    e dot e // residual/error sum of squares
} // sseF
```

ScalaTion’s `minima` and `maxima` packages provide several solvers for linear, quadratic, integer and non-linear programming. Currently, the `QuasiNewton` class is used for finding an optimal \( b \) by minimizing `sseF`. The `QuasiNewton` optimizer requires an initial guess for the parameter vector \( b \).

```scala
val bfgs = new QuasiNewton (sseF) // minimize sse using NLP
b = bfgs.solve (b_init) // estimate for b from optimizer
```

NonLinRegression Class

Class Methods:

@param x the input/data matrix augmented with a first column of ones
@param y the response vector
@param f the non-linear function \( f(x, b) \) to fit
@param b_init the initial guess for the parameter vector \( b \)

class NonLinRegression (x: MatriD, y: VectoD,
                             f: (VectoD, VectoD) => Double,
                             b_init: VectoD)
              extends PredictorMat (x, y)

    def sseF (b: VectoD): Double =
    def train (yy: VectoD = y): NonLinRegression =
    override def eval ()
    override def predict (z: VectoD): Double = f(z, b)
    def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
9.2 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 \( x \), while the outputs are given by the output value \( y \). 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. 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 (sometimes referred to as bias) one of the inputs (say \( x_0 \)) must always be set to 1. Alternatively, a bias value can be associated with the output node and added to the weighted sum (see below).

The weights on the edges are analogous to the parameter vector \( b \) in regression. The output \( y \) has an associated parameter vector \( b \), where parameter value \( b_j \) is the edge weight connecting input node \( x_j \) with output node \( y \).

Recall the basic multiple regression model (equation 4.1).

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + ...b_{n-1} x_{n-1} + \epsilon
\]

We now take the linear combination of the inputs, \( b \cdot x \), and apply an activation function \( f \).

\[
y = f(b \cdot x) + \epsilon = f(\sum_{j=0}^{n-1} b_j x_j) + \epsilon \tag{9.1}
\]

9.2.1 Training

Given several input vectors and output values (e.g., in a training dataset), optimize/fit the weights \( b \) connecting the layers. After training, given an input vector \( x \), the net can be used to predict the corresponding output value \( y \).

A training dataset consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight vector \( b \). Given an input matrix \( X \) consisting of \( m \) input vectors and an output vector \( y \) consisting of \( m \) output values, minimize the distance between the actual/target output vector \( y \) and the predicted output vector \( \hat{y} \),

\[
\hat{y} = f(Xb) \tag{9.2}
\]

where \( f \) 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.

\[
\epsilon = y - \hat{y} = y - f(Xb) \tag{9.3}
\]

Using the Euclidean (\( \ell_2 \)) norm, we have

\[
\min_b \| y - f(Xb) \|
\]

As was the case with regression, it is convenient to minimize the dot product of the error with itself (\( ||\epsilon||^2 = \epsilon \cdot \epsilon \)). In particular, we aim to minimize half of this value, half sse (\( hse \)).
\[ hse(b) = \frac{1}{2} (y - f(Xb)) \cdot (y - f(Xb)) \] (9.4)

### 9.2.2 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 Propogation (RMSProp) and Adaptive Moment Estimation (Adam) (see Chapter on Optimization for details).

The gradient of the objective/cost function \( hse \) is calculated by computing all of the partial derivatives with respect to the parameters/weights.

\[ \nabla hse(b) = \frac{\partial hse}{\partial b} = \left[ \frac{\partial hse}{\partial b_0}, \ldots, \frac{\partial hse}{\partial b_{n-1}} \right] \]

#### Partial Derivative for \( b_j \)

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 \( u = Xb \) (the pre-activation response) allows equation 9.4 to be simplified to

\[ hse = \frac{1}{2} (y - f(u)) \cdot (y - f(u)) \] (9.5)

The chain rule from vector calculus to be applied is

\[ \frac{\partial hse}{\partial b_j} = \frac{\partial hse}{\partial u} \cdot \frac{\partial u}{\partial b_j} \] (9.6)

The first partial derivative is

\[ \frac{\partial hse}{\partial u} = -f'(u)(y - f(u)) \] (9.7)

where the first part of the r.h.s. is \( f'(u) \) which is the derivative of \( f \) with respect to vector \( u \) and the second part is the difference between the actual and predicted output/response vectors. The two vectors are multiplied together, elementwise.

The second partial derivative is

\[ \frac{\partial u}{\partial b_j} = x_{j} \] (9.8)

where \( x_{j} \) 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

\[ \frac{\partial hse}{\partial b_j} = -x_{j} \cdot f'(Xb)(y - f(Xb)) \]

Since the error vector \( \epsilon = y - f(Xb) \), we may simplify the expression.

\[ \frac{\partial hse}{\partial b_j} = -x_{j} \cdot f'(Xb) \epsilon \] (9.9)
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 \( hsee \).

\[ \delta \]

\[ \Delta = \frac{\partial hse}{\partial u} = -f'(Xb) \epsilon \]  

(9.10)

It multiplies the gradient of \( f \) by the error vector, elementwise. If the error is small or the gradient is small, the adjustment to the parameter should be small. The partial derivative of \( hse \) with respect to \( b_j \) now simplifies to

\[ \frac{\partial hse}{\partial b_j} = x_j \cdot \delta \]  

(9.11)

Note, if we consider a single instance \( (x_i, y_i) \), equation 9.11 becomes

\[ \frac{\partial hse}{\partial b_j} = -x_{ij}f'(x_i \cdot b) \epsilon_i = x_{ij} \delta_i \]

Example

Assume that the activation function is the sigmoid function. Starting with the parameter/weight vector \( b = [1.1, 2.1] \), compute the \( m \)-dimensional vectors, \( \epsilon \) and \( \delta \), for Exercise 6. With these parameters, the predicted output/response vector \( \hat{y} \) may be computed in two steps: The first step computes the response, pre-activation \( Xb \). The second step takes this vector and applies the activation function to each of its elements. This requires looking ahead to the section on activation functions. The sigmoid function is \( [1 + e^{-t}]^{-1} \).

\[ \hat{y} = \text{sigmoid}(Xb) = \text{sigmoid}([1, 1.15, 2, 2.25, 3, 3.3, 3.5, 4]) \]

\[ = \{0.5249, 0.5374, 0.5498, 0.5498, 0.5621, 0.5744, 0.5744, 0.5866, 0.5986\} \]

The error vector \( \epsilon \) is simply the difference between the actual and predicted output/response vectors.

\[ \epsilon = y - \hat{y} \]

\[ = \{0.5000, 0.3000, 0.2000, 0.8000, 0.5000, 3.0000, 1.0000, 0.8000, 0.5000\} - \]

\[ = \{0.5249, 0.5374, 0.5498, 0.5498, 0.5621, 0.5744, 0.5744, 0.5866, 0.5986\} = \]

\[ = \{-0.0249, -0.2374, -0.3498, 0.2501, 0.0621, -0.2744, 0.4255, 0.2133, -0.0986\} \]

To compute the delta vector \( \delta \), we must look ahead to get the derivative of the activation function.

\[ \text{sigmoid}'(t) = \text{sigmoid}(t) [1 - \text{sigmoid}(t)] \]

Therefore, since \( \text{sigmoid}(Xb) = \hat{y} \)
\[ \delta = -\hat{y}(1 - \hat{y}) \epsilon \]

[0.0062, 0.0590, 0.0865, -0.0619, 0.0153, 0.0670, -0.1040, -0.0517, 0.0237]

Forming the Gradient

Combining the partial derivatives in equation 9.11 into an \(n\)-dimensional vector (i.e., the gradient) yields

\[
\nabla hse(b) = \frac{\partial hse}{\partial b} = -X^t[f'(Xb)\epsilon] = X^t\delta
\]

(9.12)

Since many optimizers such as gradient-descent, move in the direction opposite to the gradient by a distance governed by the learning rate \(\eta\) (alternatively step size), the following term should be added to the weight/parameter vector \(b\).

\[
X^t[f'(Xb)\epsilon]\eta = -X^t\delta \eta
\]

(9.13)

The right hand side is an \(n\)-by-\(m\) matrix, \(m\) vector product yielding an \(n\) vector result. The factor in brackets, \([f'(Xb)\epsilon]\), is the elementwise vector product. Since gradient-based optimizers move in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate \(\eta\), the parameter/weight vector \(b\) is updated as follows:

\[
b = b - X^t \delta \eta
\]

(9.14)

9.2.3 Initializing Weights/Parameters

The weight/parameter vector \(b\) should be randomly set to start the optimization.

Set the initial weight/parameter vector \(b\) with values in \((0, \text{limit})\) before training.

@param stream the random number stream to use
@param limit the maximum value for any weight

```scala
def setWeights (stream: Int = 0, limit: Double = 1.0 / sqrt (x.dim2)) {
  val rvg = new RandomVecD (n, limit, 0.0, stream = stream) // may change stream
  b = rvg.gen
} // setWeights
```

For testing or learning purposes, the weights may also be set manually.

```scala
def setWeights (w0: VectoD) { b = w0 }
```

9.2.4 Activation Functions

An activation function \(f\) takes an aggregated signal and transforms it. In general, to reduce the chance of signals being amplified to infinity, the range of an activation may be limited. 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 7). 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, sigmoid, tanh and gaussian. Several activation functions are compared in [13]. For these activation functions the outputs in the $y$ vector need to be transformed into the range specified for the activation function, see Table 9.1. It may be also useful to transform/standardize the inputs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function $u = f(t)$</th>
<th>Domain</th>
<th>Range</th>
<th>Derivative $f'(t)$</th>
<th>Inverse $t = f^{-1}(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$t$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>1</td>
<td>$u$ for $u &gt; 0$</td>
</tr>
<tr>
<td>reLU</td>
<td>max$(0, t)$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}^+$</td>
<td>$I_{t&gt;0}$</td>
<td>$u$ for $u &gt; 0$</td>
</tr>
<tr>
<td>lReLU</td>
<td>max$(\alpha t, t)$, $\alpha &lt; 1$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>$I_{t&lt;0}(\alpha, 1)$</td>
<td>$\min\left(\frac{\alpha}{\alpha}, u\right)$</td>
</tr>
<tr>
<td>eLU</td>
<td>$\text{if}_{t&lt;0}(\alpha (e^t - 1), t)$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>$I_{t&lt;0}(f(t) + \alpha, 1)$</td>
<td>$I_{t&lt;0}(\ln\left(\frac{\alpha}{\alpha} + 1\right), u) $</td>
</tr>
<tr>
<td>sigmoid</td>
<td>$\frac{1 + e^{-t}}{1}$</td>
<td>$\mathbb{R}$</td>
<td>$(-1, 1)$</td>
<td>$f(t)[1 - f(t)]$</td>
<td>$-\ln\left(\frac{\alpha}{\alpha} - 1\right)$</td>
</tr>
<tr>
<td>tanh</td>
<td>$\tanh(t)$</td>
<td>$\mathbb{R}$</td>
<td>$(-1, 1)$</td>
<td>$1 - f(t)^2$</td>
<td>$0.5 \ln \left(\frac{1 + u}{1 - u}\right)$</td>
</tr>
<tr>
<td>gaussian</td>
<td>$e^{-t^2}$</td>
<td>$\mathbb{R}$</td>
<td>$(0, 1)$</td>
<td>$-2te^{-t^2}$</td>
<td>$\sqrt{-\ln(u)}$</td>
</tr>
</tbody>
</table>

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, $f'(t) = f(t)[1 - f(t)]$, so equation 9.12 becomes

$$\frac{\partial hse}{\partial b} = -X^T[f(Xb)[1 - f(Xb)]e] = X^T \delta$$

A simple form of gradient-descent iteratively moves in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate $\eta$. Therefore, the parameter/weight vector $b$ is adjusted as follows:

$$b = b - X^T \delta \eta$$

Assuming the learning rate $\eta = 1$ and taking the $\delta$ vector from the example, the update to parameter/weight vector $b$ is

$$X^T \delta \eta = [.0402, -.1218, .1886]$$

Consequently, the updated value for the parameter/weight vector $b$ is

$$b = [.1, 2, .1] - [.0402, -.1218, .1886] = [.0597, 3.218, -.0886]$$

Check to see if the new values for $b$ have improved the objective function $hse$.

The iterative process is typically terminated when the drop in $hse$ is small or a maximum number of iterations is exceeded. The parameters $\eta$ and max_epochs need careful adjustment to obtain nearly (locally) optimal values for $hse$. Gradient-descent works by iteratively moving in the opposite direction as the gradient.
until the error changes fall below a threshold. The rate of convergence can be adjusted using the learning rate $\eta$ which multiplies the gradient. Setting it too low, slows convergence, while setting it too high can cause oscillation. In SCALATION, the learning rate $\eta$ defaults to 0.1.

The `train0` method contains the main training loop that is shown below. Inside the loop, new values $y_p$ 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 $b$.

```scala
def train0 (yy: VectoD = y): Perceptron =
{
  if (b == null) b = weightVec (n) // initialize parameters/weights
  var sse0 = Double.MaxValue // hold prior value of sse
  for (epoch <- 1 until maxEpochs) {
    // iterate over each epoch
    val yp = f1.fV (x * b) // predicted output vector yp = f(Xb)
    e = yy - yp // error vector for y
    val d = -f1.dV (yp) * e // delta vector for y
    b -= x.t * d * eta // update the parameters/weights
    val sse = sseF (y, f1.fV (x * b)) // recompute sum of squared errors
    if (DEBUG) println (s"train0: parameters for $epoch th phase: b = $b, sse = $sse")
    if (sse >= sse0) return this // return when sse increases
    sse0 = sse // save prior sse
  }
  // for
  this
} // train
```

The vector function $f1.fV$ is the vectorization of the activation function $f1.f$, and is created in SCALATION using the `vectorize` high-order function, e.g., given a scalar function $f$, it can produce the corresponding vector function $f$.

```scala
def vectorize (f: FunctionS2S): FunctionV_2V = (x: VectoD) => x.map (f(_))
```

The function $f1.dV$ is the derivative of the vector activation function. The core of the algorithm is the first four lines in the loop. Table 9.2 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Equation</th>
<th>Equation Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$yp = f1.fV (x * b)$</td>
<td>$\hat{y} = f(Xb)$</td>
<td>9.1</td>
</tr>
<tr>
<td>$e = yy - yp$</td>
<td>$\epsilon = y - \hat{y}$</td>
<td>9.2</td>
</tr>
<tr>
<td>$d = -f1.dV (yp) * e$</td>
<td>$\delta = -f'(Xb)e$</td>
<td>9.9</td>
</tr>
<tr>
<td>$b -= x.t * d * eta$</td>
<td>$b = b - X^t \delta \eta$</td>
<td>9.13</td>
</tr>
</tbody>
</table>

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
(except Gaussian) are either formulas involving constants or simple functions of the activation function itself, so for efficiency, the yp vector is passed in.

A perceptron can be considered to be a special type of non-linear or transformed regression, see Exercise 8. The Perceptron class defaults to the f_sigmoid Activation Function Family (AFF), which is defined in the ActivationFun object.

```scala
val f_sigmoid = AFF (sigmoid, sigmoidV, sigmoidM, sigmoidDV, sigmoidDM, (0, 1))
```

Other activation functions should be experimented with, as one may produce better results. All the activation functions shown in Table 9.1 are available in the ActivationFun object.

**Perceptron Class**

**Class Methods**:

```scala
class Perceptron (x: MatriD, y: VectoD,
    fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp,
    f1: AFF = f_sigmoid, val itran: FunctionV_2V = null)
    extends PredictorMat (x, y, fname_, hparam)
```

```scala
def setWeights (w0: VectoD)
def reset (eta_: Double) { eta = eta_ }
def train0 (yy: VectoD = y): Perceptron =
def train (yy: VectoD = y): Perceptron =
override def train2 (yy: VectoD = y): Perceptron =
def trainSwitch (which: Int, yy: VectoD = y): Perceptron =
override def predict (z: VectoD): Double = f1.f (b dot z)
override def predict (z: MatriD = x): VectoD = f1.fV (z * b)
def forwardSel (cols: Set[Int], adjusted: Boolean = true): (Int, VectoD, VectoD) =
def backwardElim (cols: Set[Int], adjusted: Boolean = true,
def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =
```

The `train0` method uses Gradient Descent with a simple stopping rule and a non-adaptive learning rate. A better optimizer is used by the `train` method which 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. The `train2` method simply calls the `train` method with various learning rates over a given interval to find the best one. The `trainSwitch` method makes it easy to switch between these three methods by setting the `which` parameter to 0, 1 or 2.

9.2.5 Exercises

1. Plot the sigmoid and tanh activation functions in the same plot and compare them.

2. The Texas Temperature regression problem can also be analyzed using a perceptron.

```scala
// 16 data points: Constant x1 x2 x3 County
// Lat Elev Long
val x = new 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, 3900.0, 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, 2380.0, 100.217, // 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)
val nn = new Perceptron (x, y)
nn.train ().eval ()
println (nn.report)
val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println (s"predict ($z) = ${nn.predict (z)}")
```

3. Analyze the `ExampleConcrete` dataset, which has three output variables `y0`, `y1` and `y2`. Create a perceptron for each output variable.

4. Use the following formula for matrix-vector multiplication

181
\[ u = Xb = \sum_j b_j x_j \]

to derive the formula for the following partial derivative

\[ \frac{\partial u}{\partial b_j} = x_j \]

5. Given equation 9.5, the formula for the objective/cost function \( hse : \mathbb{R}^m \rightarrow \mathbb{R} \) expressed in terms of the pre-activation vector \( u = Xb \) and the vectorized activation function \( f : \mathbb{R}^m \rightarrow \mathbb{R}^m \),

\[ hse(u) = \frac{1}{2} (y - f(u)) \cdot (y - f(u)) \]

derive equation 9.7, the formula for the gradient of \( hse \) with respect to \( u \).

\[ \nabla hse = \frac{\partial hse}{\partial u} = -f'(u)(y - f(u)) \]

**Hint:** Take the gradient, \( \frac{\partial hse}{\partial u} \), using the product rule \( (d_1 \cdot f_2 + f_1 \cdot d_2) \).

\[ \frac{\partial hse}{\partial u} = - \frac{\partial f(u)}{\partial u} \cdot (y - f(u)) \]

where \( f_1 = f_2 = y - f(u) \) and \( d_1 = d_2 = - \frac{\partial f(u)}{\partial u} \). Next, assuming \( \frac{\partial f(u)}{\partial u} \) is a diagonal matrix, show that the above equation can be rewritten as

\[ \frac{\partial hse}{\partial u} = -f'(u)(y - f(u)) \]

where \( f'(u) = [f'(u_0), \ldots, f'(u_{m-1})] \) and the two vectors, \( f'(u) \) and \( y - f(u) \), are multiplied, elementwise.

6. Show that the \( m \)-by-\( m \) Jacobian matrix, \( \mathbb{J} f(u) = \frac{\partial f(u)}{\partial u} \), is a diagonal matrix, i.e.,

\[ \mathbb{J} f(u) = \begin{bmatrix} \frac{\partial f_i(u)}{\partial u_j} \end{bmatrix} = 0 \text{ if } i \neq j \]

where \( f_i = f \) the scalar activation function. Each diagonal element is the derivative of the activation function applied to the \( i^{th} \) input, \( f'(u_i) \).

7. Show the first 10 iterations that update the parameter/weight matrix \( b \) that is initialized to \([.1, .2, .1]\).

Use the following combined input-output matrix. Let the perceptron use the default sigmoid function.

```plaintext
// 9 data points: Constant x1 x2 y
val xy = new 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, 0.5, 0.5,
                      1.0, 0.5, 0.5, 0.5,
                      1.0, 0.5, 0.5, 0.5,
                      1.0, 0.5, 0.5, 0.5,
```
val hp = Optimizer.hp.updateReturn("eta", 2.0)  // try several values for eta
val nn = Perceptron(xy, null, hp)  // create a perceptron

For each iteration, do the following: Print the weight/parameter update vector \( \mathbf{b}_{\text{up}} \) and the new value for weight/parameter vector \( \mathbf{b} \). Make a table with \( m \) rows showing values for

\[
x_1, x_2, y, \hat{y}, \epsilon, \epsilon^2, \hat{y}(1 - \hat{y}) \text{ and } \hat{y}(1 - \hat{y}) \epsilon \eta
\]

Try letting \( \eta = 1 \) then 2. Also, compute \( sse \) and \( R^2 \).

8. Show that when the activation function \( f \) is the id function, that \( f'(u) \) is the one vector, \( \mathbf{1} \). Plug this into equation 9.12 to obtain the following result.

\[
\frac{\partial \text{hse}}{\partial \mathbf{b}} = -X^{\text{t}}[1 \epsilon] = -X^{\text{t}}(y - X\mathbf{b})
\]

Setting the gradient equal to zero, now yields \( X^{\text{t}}X\mathbf{b} = X^{\text{t}}y \), the Normal Equations.

9. Show that a Perceptron with an invertible activation function \( f \) is similar to TranRegression with transform \( f^{-1} \). 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): \( y, \text{ypr, ypt} \) vs. \( t \) and \( y, \text{ypr, ypp} \) vs. \( t \), where \( y \) is the actual response/output, \( \text{ypr} \) is the prediction from Regression, \( \text{ypt} \) is the prediction from TranRegression and \( \text{ypp} \) is the prediction from Perceptron.
9.3 Multi-Output Prediction

The PredictorMat2 abstract class provides the basic structure and API for a variety of modeling techniques that produce multiple responses/outputs, e.g., Neural Networks. It serves the same role that PredictorMat and PredictorVec have for the regression modeling techniques.

9.3.1 Model Equation

For modeling techniques extending this abstract class, the model equation takes an \( n_x \)-dimensional vector \( \mathbf{x} \) as input into the function \( f \) that is parameterized by matrix \( B \),

\[
\mathbf{y} = f(B \cdot \mathbf{x}) + \epsilon = f(B^t \mathbf{x}) + \epsilon
\]  

(9.15)

where \( \mathbf{y} \) is an \( n_y \)-dimensional response random vector and \( \epsilon \) is an \( n_y \)-dimensional residual/error random vector.

9.3.2 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.

\[
\mathbf{Y} = f(\mathbf{X}B) + \mathbf{E}
\]  

(9.16)

where \( \mathbf{X} \) is an \( m \)-by-\( n_x \) data/input matrix, \( \mathbf{Y} \) is an \( m \)-by-\( n_y \) response/output matrix, \( B \) is an \( n_x \)-by-\( n_y \) parameter matrix, \( f \) is a function mapping one \( m \)-by-\( n_y \) matrix to another, and \( \mathbf{E} \) is an \( m \)-by-\( n_y \) residual/error matrix.

If one is interested in referring to the \( k \text{th} \) component (or column) of the output, equation 9.16 becomes.

\[
\mathbf{y}_{:,k} = f(\mathbf{Xb}_{:,k}) + \mathbf{e}_{:,k}
\]  

(9.17)

Recall that \( \mathbf{y}_{:,k} \) indicates the \( k \text{th} \) row of matrix \( \mathbf{Y} \), while \( \mathbf{y}_{:,k} \) indicates the \( k \text{th} \) column.

Analogous to other predictive modeling techniques, PredictorMat2 constructor takes four arguments: the data/input matrix \( \mathbf{x} \), the response/output matrix \( \mathbf{y} \), the feature/variable names \( \text{fname} \), and the hyper-parameters for the model/network \( \text{hparam} \).

PredictorMat2 Class

Class Methods:

- @param \( \mathbf{x} \) the \( m \)-by-\( nx \) data/input matrix (data consisting of \( m \) input vectors)
- @param \( \mathbf{y} \) the \( m \)-by-\( ny \) response/output matrix (data consisting of \( m \) output vectors)
- @param \( \text{fname} \) the feature/variable names (if null, use \( x_j \)’s)
- @param \( \text{hparam} \) the hyper-parameters for the model/network
abstract class PredictorMat2 (x: MatriD, y: MatriD, 
protected var fname: Strings, hparam: HyperParameter) 
extends Predictor with Error

def reset (eta_: Double) { eta = eta_ }
def train0 (yy: MatriD = y): PredictorMat2
def train (yy: MatriD = y): PredictorMat2
def train (yy: VectoD): PredictorMat2 = train (MatrixD (Seq (yy)))
def train2 (yy: MatriD = y): PredictorMat2
def trainSwitch (which: Int, yy: MatriD = y): PredictorMat2 =
def eval (xx: MatriD = x, yy: VectoD = y.col(0)): PredictorMat2 =
def eval (xx: MatriD, yy: MatriD): PredictorMat2 =
def residual: VectoD = ee.col(0)
def residuals: MatriD = ee
def fitLabel: Seq[String] = fitA(0).fitLabel
def fitMap: IndexedSeq[Map[String, String]] =
def hparameter: HyperParameter = hparam
def parameter: VectoD = parameters (0).w(0)
def parameters: NetParams
def report: String =
def forwardSel (cols: Set[Int], adjusted: Boolean): (Int, NetParams, VectoD)
def predict (z: VectoD): Double = predictV (z)(0)
def predictV (z: VectoD): VectoD
def predict (z: MatriD): MatriD
protected def crossValidate (algor: (MatriD, MatriD) => PredictorMat2, k: Int = 10, 
rando: Boolean = true): Array[Statistic] =
def crossVal (k: Int = 10, rando: Boolean = true): Array[Statistic]

The default hyper-parameters are defined in the Optimizer object.

val hp = new HyperParameter
hp += ("eta", 0.1, 0.1) // learning/convergence rate
hp += ("bSize", 10, 10) // mini-batch size
hp += ("maxEpochs", 10000, 10000) // maximum number of epochs/iterations
hp += ("lambda", 0.0, 0.0) // regularization hyper-parameter
9.4 Two-Layer Neural Networks

The \texttt{NeuralNet.2L} class supports multi-valued 2-layer (input and output) Neural Networks. The inputs into a Neural Net are given by the input vector $x$, while the outputs are given by the output vector $y$. 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. The input layer consists of $n_x$ input nodes, while the output layer consists of $n_y$ output nodes. An edge connects each input node with each output node, i.e., there are $n_x n_y$ edges in the network. To include an intercept in the model (sometimes referred to as bias) one of the inputs (say $x_0$) must always be set to 1.

The weights on the edges are analogous to the parameter vector $b$ in regression. Each output variable $y_k$, has its own parameter vector $b_{k}$. 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 $x$, the network can be used to predict the corresponding output vector $y$. 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 $f$.

$$ y_k = f(b_{k} \cdot x) + \epsilon_k = f\left( \sum_{j=0}^{n_x-1} b_{jk} x_j \right) + \epsilon_k $$

The model for \texttt{NeuralNet.2L} can written in vector form as follows:

$$ y = f(B \cdot x) + \epsilon = f(B^T x) + \epsilon $$ (9.18)

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 $X$ consisting of $m$ input vectors and an output matrix $Y$ consisting of $m$ output vectors, minimize the distance between the actual/target output matrix $Y$ and the predicted output matrix $f(XB)$

$$ \min_B \| Y - f(XB) \|_F $$

where $\| \cdot \|_F$ is the Frobenius norm, $X$ is $m$-by-$n_x$, $Y$ is $m$-by-$n_y$, and $B$ is $n_x$-by-$n_y$. Other norms may be used as well, but the square of the Frobenius norm will give the overall sum of squared errors $sse$.

9.4.1 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 objective function $sse(B)$. As in the last section, we work with half of the sum of squared errors $sse$ (or $hse$). Summing the error over each column vector $y_k$ in matrix $Y$ gives

$$ hse(B) = \frac{1}{2} \sum_{k=0}^{n_y-1} (y_k - f(Xb_{k})) \cdot (y_k - f(Xb_{k})) $$ (9.19)

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). For the sigmoid activation function,
\[ \text{sigmoid}(t) = \frac{1}{1 + e^{-t}} \]

the derivative is

\[ \text{sigmoid}(t)[1 - \text{sigmoid}(t)] \]

To minimize the objective function given in equation 9.15, we decompose it into \( n_y \) functions.

\[ hse(b_{:k}) = \frac{1}{2} (y_{:k} - f(Xb_{:k})) \cdot (y_{:k} - f(Xb_{:k})) \]

Notice that this is the same equation as 9.4, just with subscripts on \( y \) and \( b \).

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, just with subscripts added. The boxed equations from the Perceptron section become the following: The prediction vector for the \( k^{th} \) response/output is

\[ \hat{y}_{:k} = f(Xb_{:k}) \] (9.20)

The error vector for the \( k^{th} \) response/output is

\[ e_{:k} = y_{:k} - \hat{y}_{:k} = y_{:k} - f(Xb_{:k}) \] (9.21)

The delta vector for the \( k^{th} \) response/output is

\[ \delta_{:k} = \frac{\partial hse}{\partial u_{:k}} = -f'(Xb_{:k}) e_{:k} \] (9.22)

where \( u_{:k} = Xb_{:k} \). The gradient with respect to the \( k^{th} \) parameter vector is

\[ \frac{\partial hse}{\partial b_{:k}} = - X^t [f'(Xb_{:k}) e_{:k}] = X^t \delta_{:k} \]

Finally, the update for the \( k^{th} \) parameter vector is

\[ b_{:k} = b_{:k} - X^t \delta_{:k} \eta \] (9.23)

**Sigmoid Case**

For the sigmoid function, \( f'(Xb_{:k}) = f(Xb_{:k})[1 - f(Xb_{:k})] \), so

\[ \frac{\partial hse}{\partial b_{:k}} = - X^t [f(Xb_{:k})[1 - f(Xb_{:k})] e_{:k}] \]

Again, moving in the direction opposite to the gradient by a distance governed by the learning rate \( \eta \) the following term should be added to the weight/parameter vector \( b_{:k} \).

\[ X^t [f(Xb_{:k})(1 - f(Xb_{:k})) e_{:k}] \eta = - X^t \delta_{:k} \eta \] (9.24)
9.4.2 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.

$$\hat{Y} = f(XB)$$ (9.25)

The $m$-by-$n_y$ negative of the error matrix $E$ is the difference between the predicted and actual/target output/response.

$$E = \hat{Y} - Y$$ (9.26)

The $m$-by-$n_y$ delta matrix $\Delta$ adjusts the error according to the slopes within $f'(XB)$ and is the elementwise matrix (Hadamard) product of $f'(XB)$ and $E$.

$$\Delta = f'(XB) \odot E$$ (9.27)

In math, the Hadamard product is denoted by the $\odot$ operator, while in SCALATION it is denoted by the $\ast$ operator. Finally, the $n_x$-by-$n_y$ parameter matrix $B$ is updated by $-X'\Delta \eta$.

$$B = B - X'\Delta \eta$$ (9.28)

The corresponding code in the train0 method is shown below:

```scala
def train0 (yy: MatrixD = y): NeuralNet_2L = {
  var sse0 = Double.MaxValue // hold prior value of sse
  for (epoch <- 1 to maxEpochs) { // iterate over each epoch
    val yp = f1.fM (b * x) // Yp = f(XB)
    val ee = yp - yy // negative of error matrix
    val d = f1.dM (yp) ** ee // delta matrix for y
    b -= x.t * d * eta // update 'b' parameters
    val sse = sseF (y, f1.fM (b * x)) // recompute sum of squared errors
    if (DEBUG) println (s"train0: parameters for $epoch th epoch: b = $b, sse = $sse")
    if (sse > sse0) return this // return when sse increases
    sse0 = sse // save prior sse
  } // for
} // train0
```

Note: $f1.fM$ 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.

```scala
def matrixize (f: FunctionV_2V): FunctionM_2M = (x: MatrixD) => x.map (f(_))
val fM = matrixize (fV)
```

Similarly, $f1.dM$ is the matrix version of the derivative of the activation function. The NeuralNet_2L class also provides train (typically better than train0) and train2 (with built in $\eta$ search) methods.
NeuralNet_2L Class

Class Methods:

@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 fname_ the feature/variable names (if null, use x_j’s)
@param hparam the hyper-parameters for the model/network
@param f1 the activation function family for layers 1->2 (input to output)
@param itran the inverse transformation function returns responses to original scale

class NeuralNet_2L (x: MatriD, y: MatriD, fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp, f1: AFF = f_sigmoid, val itran: FunctionV_2V = null)
extends PredictorMat2 (x, y, fname_, hparam)

def parameters: NetParams = Array (b)
def train0 (yy: MatriD = y): NeuralNet_2L =
def train (yy: MatriD = y): NeuralNet_2L =
def train2 (yy: MatriD = y): NeuralNet_2L =
def forwardSel (cols: Set [Int], adjusted: Boolean = true): (Int, NetParams, VectoD) =
def predictV (z: VectoD): VectoD = f1.fV (b dot z)
def predict (z: MatriD = x): MatriD = f1.fM (b * z)
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic] =

9.4.3 Exercises

1. The dataset in ExampleConcrete 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 Perceptrons.

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.
9.5 Three-Layer Neural Networks

The \texttt{NeuralNet.3L} class supports 3-layer (input, hidden and output) Neural Networks. The inputs into a Neural Net are given by the input vector \( x \), while the outputs are given by the output vector \( y \). Between these two layers is a single hidden layer, whose intermediate values will be denoted by the vector \( z \). 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. The input layer consists of \( n_x \) input nodes, the hidden layer consists of \( n_z \) hidden nodes, and the output layer consists of \( n_y \) output nodes.

There are two sets of edges. Edges in the first set connect each input node with each hidden node, i.e., there are \( n_x n_z \) such edges in the network. The parameters (or edge weights) for the first set of edges are maintained in matrix \( A = [a_{jh}]_{n_x \times n_z} \). 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 \( B = [b_{hk}]_{n_z \times n_y} \).

The model for \texttt{NeuralNet.3L} can written in vector form as follows:

\[
\mathbf{y} = f_1(B \cdot f_0(A \cdot \mathbf{x})) + \epsilon = f_1(B^t f_0(A^t \mathbf{x})) + \epsilon
\]  
(9.29)

The innermost matrix-vector product multiplies the transpose of the \( n_x \)-by-\( n_z \) matrix \( A \) by the \( n_x \)-by-1 vector \( \mathbf{x} \), producing an \( n_z \)-by-1 vector, which is passed into the \( f_0 \) 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 \( f_1 \) vectorized activation function.

Intercept/Bias

As before, one may include an intercept in the model (sometimes 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. In such case, the computation performed at node \( z_0 \) would be thrown away and replaced with 1. The alternative is to replace the uniform notion of parameters with two types of parameters, weights and biases. \textsc{scalaTion} supports this with the \texttt{NetParam} case class.

```scala
@param w the weight matrix
@param b the optional bias/intercept vector (null => not used)

case class NetParam (w: Matrix, var b: Vector = null)

def dot (x: Vector): Vector = (w dot x) + b
```

Following this approach, there is no need for the special nodes and the \texttt{dot} product is re-defined to add the bias \( b \) to the regular matrix-vector \texttt{dot} product (\( w \cdot x \)). Note, the \texttt{NetParam} class defines several other methods as well. The vector version of the predict method in \texttt{NeuralNet.3L} uses this \texttt{dot} product to make predictions.

```scala
def predictV (v: Vector): Vector = f1.fV (b dot f0.fV (a dot v))
```

191
Training

Training involves an iterative procedure (e.g., stochastic gradient descent) that adjusts parameter values (for weights and biases) to minimize an objective function such as \( sse \) or in this section half \( sse \) (or \( hse \)). Before the main loop, random parameter values (for weights and biases) need to be assigned to \( \text{NetParam} \ A \) and \( \text{NetParam} \ B \). Roughly as outlined in section 3 of [18], the training can be broken into four steps:

1. Compute predicted values for output \( \hat{y} \) and compare with actual values \( y \) to determine the error \( y - \hat{y} \).
2. Back propagate the adjusted error to determine the amount of correction needed at the output layer. Record this as vector \( \delta^1 \).
3. Back propagate the correction to the hidden layer and determine the amount of correction needed at the hidden layer. Record this as vector \( \delta^0 \).
4. Use the delta vectors, \( \delta^1 \) and \( \delta^0 \), to make updates to \( \text{NetParam} \ A \) and \( \text{NetParam} \ B \), i.e., the weights and biases.

9.5.1 Optimization

In this subsection, the basic elements of the backpropagation 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 \( B \) \( \text{NetParams} \) are treated as weight matrices. In the code, the same logic includes the biases (so nothing is lost, see exercises).

1. Compute predicted values: Based on the randomly assigned weights to the \( A \) and \( B \) matrices, predicted outputs \( \hat{y} \) are calculated. First values for the hidden layer \( z \) are calculated, where the values for hidden node \( h \), \( z_h \), is given by

\[
z_h = f_0(a_h \cdot x) \quad \text{for } h = 0, \ldots, n_z - 1
\]

where \( f_0 \) is the first activation function (e.g., sigmoid), \( a_h \) is column-\( h \) of the \( A \) weight matrix, and \( x \) is an input vector for a training sample/instance (row in the data matrix). Typically, several samples (referred to as a batch) are used in each step. Next, the values computed at the hidden layer are used to produce predicted outputs \( \hat{y} \), where the value for output node \( k \), \( \hat{y}_k \), is given by

\[
\hat{y}_k = f_1(b_k \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1
\]

where the second activation function \( f_1 \) may be the same as (or different from) the one used in the hidden layer and \( b_k \) 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 elementwise, the error for the \( k^{th} \) output, \( \epsilon_k \), is given by

\[
\epsilon_k = y_k - \hat{y}_k \quad \text{for } k = 0, \ldots, n_y - 1
\]

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 $\epsilon$, the delta/correction vector $\delta^1$ for the output layer may be calculated, where for output node $k$, $\delta^1_k$ is given by
\[
\delta^1_k = -f'_1(b_k \cdot z) \epsilon_k \quad \text{for } k = 0, \ldots, n_y - 1
\] (9.30)
where $f'_1$ is the derivative of the activation function (e.g., for sigmoid, $f'(t) = f(t)[1 - f(t)]$). The partial derivative of $hse$ with respect to the weight connecting hidden node $h$ with output node $k$, $b_{hk}$, is given by
\[
\frac{\partial hse}{\partial b_{hk}} = z_h \delta^1_k
\] (9.31)

3. Back propagate from hidden layer: Given the delta/correction vector $\delta^1$ from the output layer, the delta vector for the hidden layer $\delta^0$ may be calculated, where for hidden node $h$, $\delta^0_h$ is given by
\[
\delta^0_h = f'_0(a_h \cdot x)[b_h \cdot \delta^1] \quad \text{for } h = 0, \ldots, n_z - 1
\] (9.32)
This equation is parallel to the one given for $\delta^1_k$ in that an error-like factor multiplies the derivative of the activation function. In this case, the error-like factor is the weighted average of the $\delta^1_k$ for output nodes connected to hidden node $h$ times row-$h$ of weight matrix $B$. The weighted average is computed using the dot product.
\[
b_h \cdot \delta^1 = \sum_{k=0}^{n_y-1} b_{hk} \delta^1_k
\]
The partial derivative of $hse$ with respect to the weight connecting input node $j$ with hidden node $h$, $a_{jh}$, is given by
\[
\frac{\partial hse}{\partial a_{jh}} = x_j \delta^0_h
\] (9.33)

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 $\eta$ which moderates the adjustments to the weights.
\[
b_{hk} = b_{hk} - z_h \delta^1_k \eta \quad \text{for } h = 0, \ldots, n_y - 1
\] (9.34)
\[
a_{jh} = a_{jh} - x_j \delta^0_h \eta
\] (9.35)

To improve the stability of the algorithm, weights are adjusted based on accumulated corrections over a batch of instances, where a batch is a subsample 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 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 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 decrease in $hse$ has been sufficiently small for $q$ iterations.

### 9.5.2 Matrix Version

Given a training dataset consisting of an $m$-by-$n_x$ input data matrix $X$ and an $m$-by-$n_y$ output data matrix
$Y$, the optimization equations may be re-written in matrix form as shown below.

The gradient descent optimizer used by the `train0` method has one main loop, while the stochastic
gradient descent optimizer used by the `train` 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 taking differences between the actual/target values and these predicted values to
compute an error matrix. Computed matrices are then used to compute delta matrices that form the basis
for updating the weight matrices.

1. The hidden values for all $m$ instances and all $n_z$ hidden nodes are computed by applying the first
matrixized activation function $f_0$ to the matrix product $XA$. The predicted output values $\hat{Y}$ are
similarly computed by applying the second matrixized activation function $f_1$ to the matrix product $ZB$.

\[
Z = f_0(XA) \tag{9.36}
\]

\[
\hat{Y} = f_1(ZB) \tag{9.37}
\]

2. The negative of the error matrix $E$ is just the difference between the predicted and actual/target
values.

\[
E = \hat{Y} - Y \tag{9.38}
\]

3. This information is sufficient to calculate delta matrices: $\Delta^1$ for adjusting $B$ and $\Delta^0$ for adjusting $A$. The $\Delta^1$ matrix is the elementwise matrix (Hadamard) product of $f_1'(ZB)$ and $E$.

\[
\Delta^1 = f_1'(ZB) \circ E \tag{9.39}
\]

The $\Delta^0$ matrix is the elementwise matrix (Hadamard) product of $f_0'(XA)$ and $\Delta^1B^t$.

\[
\Delta^0 = f_0'(XA) \circ (\Delta^1B^t) \tag{9.40}
\]

4. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate $\eta$) for updating the parameter/weight matrices, $A$ and $B$.

\[
B = B - Z^t\Delta^1 \eta \tag{9.41}
\]
The corresponding \texttt{SCALE} code for the \texttt{train0} method is shown below.

```scala
def train0 (yy: MatrixD = y): NeuralNet_3L =
{
    var sse0 = Double.MaxValue // hold prior value of sse
    for (epoch <- 1 to maxEpochs) {
        // iterate over each epoch
        var z = f0.fM (a * x) // \( Z = f(XA) \)
        var yp = f1.fM (b * z) // \( Yp = f(ZB) \)
        ee = yp - yy // negative of error matrix
        val d1 = f1.dM (yp) ** ee // delta matrix for \( Y \)
        val d0 = f0.dM (z) ** (d1 * b.w.t) // delta matrix for \( Z \)
        a -= (x.t * d0 * eta, d0.mean * eta) // update \( A \) weights & biases
        b -= (z.t * d1 * eta, d1.mean * eta) // update \( B \) weights & biases
        val sse = sseF (y, b * f1.fM (f0.fM (a * x)))
        if (DEBUG) println (s"train0: parameters for $epoch th epoch: sse = $sse")
        if (sse > sse0) return this // return early if moving up
        sse0 = sse // save prior sse
        this
    } // for
} // // train0
```

For stochastic gradient descent in the \texttt{Optimizer} object, the inner loop divides the training dataset into \( nB \) \textit{batches}. A batch is a randomly selected group/batch of rows. Each batch (\( ib \)) is passed to the \texttt{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 correspond to the number of responses. For example, in the \texttt{ExampleConcrete} example there are three response columns, requiring three instances of \texttt{Regression} or one instance of \texttt{NeuralNet_3L}. Three separate \texttt{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 \texttt{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.

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 to the last layer for classification problems.

\[
f_i(t) = \frac{e^{t_i}}{\sum e^t} \quad \text{for } i = 0, \ldots, n - 1
\]

\texttt{NeuralNet_3L} Class

\textbf{Class Methods:}

195
@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 nz the number of nodes in hidden layer
@param fname_ the feature/variable names (if null, use x_j’s)
@param hparam the hyper-parameters for the model/network
@param f0 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 NeuralNet_3L (x: MatriD, y: MatriD,
    private var nz: Int = -1,
    fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp,
    f0: AFF = f_sigmoid, f1: AFF = f_lreLU,
    val itran: FunctionV_2V = null)
extends PredictorMat2 (x, y, fname_, hparam) // sets eta in parent class

def parameters: NetParams = Array (a, b)
def train0 (yy: MatriD = y): NeuralNet_3L =
def train (yy: MatriD = y): NeuralNet_3L =
def train2 (yy: MatriD = y): NeuralNet_3L =
def forwardSel (cols: Set[Int], adjusted: Boolean = true): (Int, NetParams, VectoD) =
def predictV (v: VectoD): VectoD = f1.fV (b dot f0.fV (a dot v))
def predict (v: MatriD = x): MatriD = f1.fM (b * f0.fM (a * v))
def crossVal (k: Int = 10, rando: Boolean = true): Array[Statistic] =

9.5.3 Exercises

1. Draw a 3-layer (input, hidden and output) Neural Network (with sigmoid activation), where \( n_x = 2, n_z = 2 \) and \( n_y = 1 \). Initialize bias vector \( \mathbf{a}^b \) to \([1,1] \) and weight matrix \( A \) (\( n_x \)-by-\( n_z \)) to

\[
\begin{bmatrix}
  .1 & .2 \\
  .3 & .4 \\
\end{bmatrix}
\]

Initialize bias vector \( \mathbf{b}^b \) to \([1] \) and weight matrix \( B \) (\( n_z \)-by-\( n_y \)) to

\[
\begin{bmatrix}
  .5 \\
  .6 \\
\end{bmatrix}
\]

Let \( x_0 = 2, x_1 = 1 \) 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 \( \mathbf{z} \).

\[
\mathbf{z}_h = f_0 (\mathbf{a}_h \cdot \mathbf{x} + a_h^b)
\]
and then compute predicted values at the output layer for $\hat{y}$.

$$y_k = f_1(b_{k} \cdot z + b^b_k)$$

2. For the above problem, calculate the $\delta^1 = [\delta^1_0]_k$ vector. Explain equation 9.30 (with explicit weights and biases included) for computing $\delta^1_k$ and use it for the calculation.

$$\delta^1_k = -f'_1(b_{k} \cdot z + b^b_k) \epsilon_k$$

3. For the above problem, calculate the $\delta^0 = [\delta^0_0, \delta^0_1]$ vector. Explain equation 9.32 (with explicit weights and biases included) for computing $\delta^0_h$ and use it for the calculation.

$$\delta^0_h = f'_0(a_h \cdot x + a^b_h) [b_h \cdot \delta^1]$$

4. Use the $\delta^1$ vector to update weight matrix $B$, i.e., for each row $h$,

$$b_h = b_h - z_h \delta^1$$
and update the bias vector $b^b$ as follows:

$$b^b = b^b - \delta^1 \eta$$

5. Use the $\delta^0$ vector to update weight matrix $A$, i.e., for each row $j$,

$$a_j = a_j - x_j \delta^0 \eta$$
and update the bias vector $a^b$ as follows:

$$a^b = a^b - \delta^0 \eta$$

6. Derive equation 9.31,

$$\frac{\partial hse}{\partial b_{hk}} = z_h \delta^1_k$$

by defining pre-activation value $v_k = b_{k} \cdot z$ and applying the following chain rule:

$$\frac{\partial hse}{\partial b_{hk}} = \frac{\partial hse}{\partial v_k} \frac{\partial v_k}{\partial b_{hk}}$$

7. The dataset in ExampleConcrete 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.

8. 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.
9.6 Multi-Hidden Layer Neural Networks

The `NeuralNet_XL` class supports basic x-layer (input, \{hidden\} and output) Neural Networks. Since arrays of matrices are used in the code, multiple layers of hidden nodes are supported. In particular, parameter $b$ which holds the weights and biases for all layers is of type `NetParams` where

```plaintext
type NetParams = IndexedSeq[NetParam]
```

The equations are the same as those used for `NeuralNet_3L`, except that the calculations are repeated layer by layer in a forward direction for prediction. During training, the predicted values are compared to actual/target values to compute an error matrix. Corrections based on these errors are propagated backward through the network to improve the parameter estimates (weights and biases) layer by layer.

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 ignored 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 fed 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 $f_l$ being applied to the product of the previous layer’s $Z_l$ matrix times its parameter matrix $B_l$.

   $$Z_0 = X$$

   (9.43)

   For each layer $l$ in the forward direction:

   $$Z_{l+1} = f_l(Z_l B_l)$$

   (9.44)

2. The negative of the error matrix $E$ is just the difference between the predicted and actual/target values, where $\hat{Y} = Z_{nl}$.

   $$E = \hat{Y} - Y$$

   (9.45)

3. This information is sufficient to calculate delta matrices $\Delta^l$. For the last layer:

   $$\Delta^{nl-1} = f'_{nl-1}(Z_{nl-1} B_{nl-1}) \circ E$$

   (9.46)

   For the rest of layers in the backward direction with $l$ being decremented:

   $$\Delta^l = f'_l(Z_l B_l) \circ (\Delta^{l+1} B_{l+1}^t)$$

   (9.47)
4. As mentioned, the delta matrices form the basis (a matrix transpose \( \times \) delta \( \times \) the learning rate \( \eta \)) for updating the parameter/weight matrices, \( B_l \) for each layer \( l \).

\[
B_l = B_l - Z_l^t \Delta^t \eta
\]  
(9.48)

The implementation of the \texttt{train0} 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.

```scala
def train0 (yy: Matrix = y): NeuralNet_XL =
{
  var sse0 = Double.MaxValue // hold prior value of sse
  val z = Array.ofDim [Matrix] (nl+1); z(0) = x // storage: activations \( f(b(l), z(l)) \)
  val d = Array.ofDim [Matrix] (nl) // storage: deltas

  for (epoch <- 1 to maxEpochs) { // iterate over each epoch
    for (l <- layers) z(l+1) = f(l).fM (b(l) * z(l)) // feedforward and store activations
    ee = z.last - yy // negative of error matrix
    d(nl-1) = f.last.dM (z.last) ** ee // delta for last layer before output
    for (l <- nl-2 to 0 by -1)
      d(l) = f(l).dM (z(l+1)) ** (d(l+1) * b(l+1).w.t) // deltas for previous hidden layers
    for (l <- layers) b(l) -= (z(l).t * d(l) * eta, d(l).mean * eta) // update (weights, biases)
    val sse = ee.normF ** 2
    if (DEBUG) println (s"train0: parameters for $epoch th epoch: b = $b, sse = $sse")
    if (sse > sse0) return this // return early if moving up
    sse0 = sse // save prior sse
  }
} // train0
```

The \texttt{train0} method is kept simple to facilitate understanding. In practice, the \texttt{train} and \texttt{train2} methods should be used.

### 9.6.1 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 \( nx \) and number of output nodes \( ny \), defaults are utilized according to one of two rules. The comments should be switched to change rules.

```scala
// if (nz == null) nz = compute_nz (nx) // Rule [1] default # nodes for hidden layers
if (nz == null) nz = compute_nz (nx, ny) // Rule [2] default # nodes for hidden layers
val df_m = compute_df_m (nz) // degrees of freedom for model
resetDF (df_m, x.dim1 - df_m) // degrees of freedom for (model, error)
```

If the array is \texttt{null}, then default numbers for the hidden layers are utilized. Rule [1] (drop one), simply decreases the number of nodes in each hidden layer by one, starting with \( nx - 1 \), \( nx - 2 \), etc. Rule [2] (average), takes the average between \( nx \) and \( ny \) for the first hidden layer, averages that with \( ny \) for the second layer, etc. The number of nodes in each layer is used make a rough estimate of the degrees of freedom for the model [?]. Currently, the degrees of freedom is only considered for the first output variable.
9.6.2 Avoidance of Overfitting

If efforts are not made to avoid overfitting, 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 overfitting 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. Regularization of the parameters, as was done for Ridge and Lasso Regression, may help as well.

NeuralNet_XL Class

Class Methods:

@param x the m-by-nx data/input matrix (training data having m input vectors)
@param y the m-by-ny response/output matrix (training data having m output vectors)
@param nz the number of nodes in each hidden layer, e.g., Array (5, 10)
@param fname_ the feature/variable names (if null, use x_j’s)
@param hparam the hyper-parameters for the model/network
@param f the array of activation function families between every pair of layers
@param itran the inverse transformation function returns responses to original scale

extends PredictorMat2 (x, y, fname_, hparam) // sets eta in parent class

def parameters: NetParams = b

def train0 (yy: MatriD = y): NeuralNet_XL =
def train (yy: MatriD = y): NeuralNet_XL =
def train2 (yy: MatriD = y): NeuralNet_XL =
def forwardSel (cols: Set [Int], adjusted: Boolean = true): (Int, NetParams, VectoD) =
def predictV (v: VectoD): VectoD =
def predict (v: MatriD = x): MatriD =
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic] =
9.6.3 Exercises

1. Examine the implementation of the `train0` 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 `train0` 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 `train0` method and the `NetParam` case class and show how the biases `b.b` are updated in the back propagation process.

4. The dataset in `ExampleConcrete` 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`.
Chapter 10

Temporal Models

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. So far, predictive models have been of the form

\[ y = f(x; b) + \epsilon \]

where \( x \) is the vector of predictive variables/features, \( y \) is the response variable and \( \epsilon \) is the residual/error.

In order to fit the parameters \( b \), \( m \) samples are collected into a data/input matrix \( X \) and a response/output vector \( y \). 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 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.

To begin with, one could focus on forecasting the value of response \( y \) at time \( t \) as a function of prior values of \( y \), e.g.,

\[ y_t = f(y_{t-1}, \ldots, y_{t-p}; \phi) + \epsilon_t \quad (10.1) \]

where \( \phi \) is a vector of parameters analogous to \( b \).

Of course, other predictor variables \( x \) may be included in the forecasting model as well, see the section on ARIMAX. Initially in this chapter, time will be treated as discrete time.
10.1 ForecasterVec

The ForecasterVec abstract class within the scalation.analytics.forecaster package provides a common framework for several forecasters. Note, the train method must be called first followed by eval. The forecast method is used to make forecasts h steps (time units) into the future.

ForecasterVec Abstract Class

Class Methods:

- @param y the response vector (time series data)
- @param n the number of parameters
- @param hparam the hyper-parameters

abstract class ForecasterVec (y: VectoD, n: Int, hparam: HyperParameter = null)
extends Fit (y, n, (n - 1, y.dim - n)) with Predictor

def train (yy: VectoD = y): ForecasterVec
def eval (xx: MatriD, yy: VectoD): ForecasterVec =
def eval (yy: VectoD = y): ForecasterVec =
def hparameter: HyperParameter = hparam
def report: String =
def residual: VectoD = e
def predict (yy: VectoD = null): Double = forecast ()(0)
def predictAll: VectoD
def forecast (h: Int = 1): VectoD
def plotFunc (fVec: VectoD, name: String)

ForecasterVec Companion Object

The ForecasterVec companion object defines the Auto-Correlation Function (ACF). It returns the variance, the auto-covariance vector and the auto-correlation vector.

def acf (y_: VectoD, lags: Int = MAX_LAGS): (Double, VectoD, VectoD) =
{
  val y = if (y_.isInstanceOf [VectorD]) y_.asInstanceOf [VectorD]
  else y_.toDense
  val sig2 = y.variance // the sample variance
  val acv = new VectorD (lags + 1) // auto-covariance vector
  for (k <- acv.range) acv(k) = y acov k // k-th lag auto-covariance
  val acr = acv / sig2 // auto-correlation function
  (sig2, acv, acr)
} // acf

204
10.1.1 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 (see cran.r-project.org/web/packages/fpp/fpp.pdf):

\[
\begin{align*}
\text{val } m &= 98 \\
\text{val } t &= \text{VectorD}\text{.range}(0, m) \\
\text{val } y &= \text{VectorD}(580.38, 581.86, 580.97, 580.39, 580.42, 580.82, 581.40, 581.32, 581.44, 581.68, 581.17, 580.53, 580.01, 579.91, 579.14, 579.16, 579.55, 579.67, 578.44, 578.24, 579.10, 579.09, 579.35, 578.82, 579.32, 579.01, 579.00, 579.80, 579.87, 579.72, 579.89, 580.01, 579.37, 578.69, 578.19, 578.67, 579.55, 578.92, 578.09, 579.37, 580.13, 580.14, 579.51, 579.24, 578.66, 578.86, 578.05, 577.79, 576.75, 576.75, 577.82, 578.64, 580.58, 579.48, 577.38, 576.90, 576.94, 576.24, 576.84, 576.85, 576.90, 577.79, 578.18, 577.51, 577.23, 578.42, 579.61, 579.05, 579.26, 579.22, 579.38, 579.10, 577.95, 578.12, 579.75, 580.85, 580.41, 579.96, 579.61, 578.76, 578.18, 577.21, 577.13, 579.10, 578.25, 577.91, 576.89, 575.96, 576.80, 577.68, 578.38, 578.52, 579.74, 579.31, 579.89, 579.96)
\end{align*}
\]

First plot this dataset and then look at its Auto-Correlation Function (ACF).

\[
\text{new Plot}(t, y, \text{null}, \text{"Plot of } y \text{ vs. } t\text{", true})
\]

The Auto-Correlation Function (ACF) measures how much the past can influence a forecast. If the forecast is for time \( t \), then the past time points are \( t - 1, \ldots, t - p \). The \( k^{th} \) lag auto-covariance (auto-correlation), \( \gamma_k (\rho_k) \) is the covariance (correlation) of \( y_t \) and \( y_{t-k} \).

\[
\gamma_k = \mathbb{C}[y_t, y_{t-k}] \quad (10.2)
\]

\[
\rho_k = \text{corr}(y_t, y_{t-k}) \quad (10.3)
\]

Note that \( \gamma_0 = \mathbb{V}[y] \) and \( \rho_k = \gamma_k / \gamma_0 \). These equations assume the stochastic process \( \{y_t | t \in [0, K]\} \) 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 \( y\text{.slice}(2, m) \) and \( y\text{.slice}(0, m-2) \). In SCALATION, the ACF is computed using the \text{acf} function from the \text{ForecasterVec} companion object.

\[
\text{import ForecasterVec._}
\]

\[
\text{val (sig2, acv, acr) = acf(y)}
\]

\[
\text{val zero = new VectorD(acr.dim)}
\]

\[
\text{new Plot(t(0 until acr.dim), acr, zero, \"ACF vs. k\", true)}
\]

The first point in plot is the auto-correlation of \( y_t \) with itself, while the rest of the points are ACF(k), the \( k^{th} \) lag auto-correlation.
10.2 Auto-Regressive (AR) Models

One of the simplest types of forecasting models of the form given in equation 10.1, is to make the future value \( y_t \) be linearly dependent on the last \( p \) of \( y \) values. In particular, a \( p^{th} \)-order Auto-Regressive AR\((p)\) model predicts the next value \( y_t \) from the sum of the last \( p \) values each weighted by its own coefficient/parameter \( \phi_j \)

\[
y_t = \delta + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t
\]

where \( \delta = \mu (1 - 1 \cdot \phi) \) and the error/noise is represented by \( \epsilon_t \). \( \epsilon_t \) represents noise that shocks the system at each time step. We require that \( E[\epsilon_t] = 0, V[\epsilon_t] = \sigma^2 \), and that all the noise shocks be independent.

To better capture the dependency, the data need to be zero-centered, which can be accomplished by subtracting the mean \( \mu \), 

\[
z_t = y_t - \mu
\]

Notice that since \( z_t \) is zero-centered, the formulas for the mean, variance and covariance are simplified.

\[
E[z_t] = 0 \tag{10.5}
\]
\[
V[z_t] = E[z_t^2] \tag{10.6}
\]
\[
C[z_t, z_{t-k}] = E[z_t z_{t-k}] = \gamma_k \tag{10.7}
\]

10.2.1 AR(1) Model

When the future is mainly dependent only on the most recent value, e.g., \( \rho_1 \) is high and rest \( \rho_2, \rho_3, \) etc. are substantially lower, then an AR(1) model may be sufficient.

\[
z_t = \phi_1 z_{t-1} + \ldots + \phi_p z_{t-p} + \epsilon_t
\]

An estimate for the parameter \( \phi_1 \) may be determined from the ACF. Take equation 10.8 and multiply it by \( z_t \) \( (k = 1, 2, \ldots p) \).

\[
z_t z_{t-k} = \phi_1 z_{t-1} z_{t-k} + \ldots + \phi_p z_{t-p} \epsilon_t
\]

Taking the expected value of the above equation gives,

\[
E[z_t z_{t-k}] = \phi_1 E[z_{t-1} z_{t-k}] + E[\epsilon_t z_{t-k}]
\]

Using the definition in Equation 10.7 for \( \gamma_k \), this can be rewritten as

\[
\gamma_k = \phi_1 \gamma_{k-1} + 0
\]

where past value \( z_{t-k} \) is independent of future noise shock \( \epsilon_t \). Now dividing by \( \gamma_0 \) yields

\[
\rho_k = \phi_1 \rho_{k-1} \tag{10.9}
\]

An estimate for parameter \( \phi_1 \) may be easily determined by simply setting \( k = 1 \) in equation 10.9.

\[
\phi_1 = \rho_1 \tag{10.10}
\]
10.2.2 AR($p$) Model

The model equation for an AR($p$) model includes the past $p$ values of $z_t$.

$$y_t = \delta + \phi_1 z_{t-1} + \phi_2 z_{t-2} + \ldots + \phi_p z_{t-p} + \epsilon_t$$

Subtracting the mean $\mu$ from the response $y_t$ gives

$$z_t = \phi_1 z_{t-1} + \ldots + \phi_p z_{t-p} + \epsilon_t \quad (10.11)$$

Multiplying by $z_{t-k}$ and then taking the expectation produces

$$\gamma_k = \phi_1 \gamma_{k-1} + \phi_2 \gamma_{k-2} + \ldots + \phi_p \gamma_{k-p}$$

Dividing by $\gamma_0$ yields

$$\rho_k = \phi_1 \rho_{k-1} + \phi_2 \rho_{k-2} + \ldots + \phi_p \rho_{k-p} \quad (10.12)$$

Equation 10.12 contains $p$ unknowns and by letting $k = 1, 2, \ldots, p$, it can be used to generate $p$ equations, or one matrix equation.

$$\begin{align*}
\rho_1 &= \phi_1 \rho_0 + \phi_2 \rho_1 + \ldots + \phi_p \rho_{p-1} \\
\rho_2 &= \phi_1 \rho_1 + \phi_2 \rho_0 + \ldots + \phi_p \rho_{p-2} \\
\vdots \\
\rho_p &= \phi_1 \rho_{p-1} + \phi_2 \rho_{p-2} + \ldots + \phi_p \rho_0
\end{align*}$$

These are the Yule-Walker equations (often $\rho_0$ is removed since it equals 1). Letting $\rho$ be the $p$-dimensional vector of lag auto-correlations and $\phi$ be the $p$-dimensional vector of parameters/coefficients, we may concisely write

$$\rho = [\rho_{|i-j|}]_{i,j} \phi = P \phi \quad (10.13)$$

where $P$ is a $p$-by-$p$ symmetric Toeplitz matrix with ones on the main diagonal. One way to solve for the parameter vector $\phi$ is to take the inverse (or use related matrix factorization techniques).

$$\phi = P^{-1} \rho \quad (10.14)$$

Due to the special structure of the $P$ matrix, more efficient techniques may be used, see the next subsection.

10.2.3 Training

The first step in training is to zero-center the response and compute the Auto-Correlation Function.

val z = y - mu // work with mean zero time series
val (sig2, acv, acr) = acf (z, ml) // variance, auto-covariance, auto-correlation

207
The parameter/coefficient vector $\phi$ can be estimated from values in the ACF. In SCALATion, the coefficients $\phi$ are estimated using the Durbin-Levinson algorithm and extracted from the $p^{th}$ row of the $\psi$ (psi) matrix.

Define $\psi_{kj}$ to be $\phi_j$ for an AR$(k)$ model. Letting $k$ range up to $p$ allows the $\phi_j$ 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.

private var phi: VecotD = null // AR(p) parameters/coefficients
private var psi = new MatrixD (ml+1, ml+1) // psi matrix (ml = max lags)

Invoke the durbinLevinson method \[17\] passing in the auto-covariance vector $\gamma$ (g) and the maximum number of lags (ml). From 1 up to the the maximum number of lags, iteratively compute the following:

\[
\psi_{kk} = \frac{\gamma_k - \sum_{j=1}^{k-1} \psi_{k-1,j} \gamma_{k-j}}{r_{k-1}}
\]
\[
\psi_{kj} = \psi_{k-1,j} - \psi_{kk} \psi_{k-1,k-j}
\]
\[
r_k = r_{k-1}(1 - \psi_{kk}^2)
\]

private def durbinLevinson (g: VecotD, ml: Int): MatrixD =
{
  val r = new VectorD (m+1); r(0) = g(0)
  for (k <- 1 to ml) { // range up to max lags
    var sum = 0.0
    for (j <- 1 until k) sum += psi(k-1, j) * g(k-j)
    val a = (g(k) - sum) / r(k-1)
    psi(k, k) = a
    for (j <- 1 until k) psi(k, j) = psi(k-1, j) - a * psi(k-1, k-j)
    r(k) = r(k-1) * (1.0 - a * a)
  }
}

The train method simply calls the durbinLevinson method to create the $\Psi$ matrix. The parameter/coefficient vector corresponds to the $p^{th}$ row of the $\Psi$ matrix, while the PACF is found in the main diagonal of the $\Psi$ matrix.

def train (yy: VecotD = y): AR =
{
  durbinLevinson (acv, ml) // pass in auto-covariance and max lags
  pacf = psi.getDiag () // PACF is the diagonal of the psi matrix
  phi = psi(p).slice (1, p+1) // AR(p) coefficients: phi_0, ..., phi_p-1
  this
} // train
Partial Auto-Correlation Function

The Partial Auto-Correlation Function (PACF) that is extracted from the main diagonal of the \( \Psi \) matrix can be used along with ACF to select the appropriate type of model. As \( k \) increases, the \( \rho_k \) will decrease and adding more parameters/coefficients will become of little help. Deciding where to cut off \( \rho_k \) from the ACF is somewhat arbitrary, but \( \psi_{kk} \) drops toward zero more abruptly giving a stronger signal as to what model to select. See the exercises for details.

10.2.4 Forecasting

After the parameters/coefficients have been estimated as part of the \texttt{train} method, the AR(\( p \)) model can be used for forecasting.

\begin{verbatim}
AR.hp("p") = p  // reassign hyper-parameter p
val ar = new AR (y)  // time series data
ar.train ().eval ()  // train for AR(p) model
val ar_f = ar.forecast (2)  // make forecasts for 1 and 2 steps
\end{verbatim}

The \texttt{forecast} method takes the parameter \texttt{steps} to indicate how many steps into the future to forecast. 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 \texttt{steps} gets larger.

\begin{verbatim}
def forecast (steps: Int = 1): VectoD =
{
    val zf = z.slice (z.dim - p) ++ new VectorD (steps)
    for (t <- p until zf.dim) {
        var sum = 0.0
        for (j <- 0 until p) sum += (j) * zf(t-1-j)
        zf(t) = sum
    }  // for
    zf.slice (p) + mu
}  // forecast
\end{verbatim}

AR Class

\begin{verbatim}
Class Methods:

@param y the response/output vector (time series data)
@param hparam the hyper-parameters

class AR (y: VectoD, hparam: HyperParameter = AR.hp)
    extends ForecasterVec (y, hparam("p").toInt, hparam)

def acF: VectoD = acr
\end{verbatim}
def pacF: VectoD = pacf
def train (yy: VectoD = y): AR =
def retrain (pp: Int): AR =
def parameter: VectoD = phi
def predictAll: VectoD =
def forecast (steps: Int = 1): VectoD =

10.2.5 Exercises

1. Compute ACF for the Lake Level Dataset. Set parameter $\phi_1$ to $\rho_1$, the first lag auto-correlation.
   Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$ and for $k <- 1$ until $y$.dim

   $$z_t = y_t - \mu$$
   $$\hat{y}_t = \rho_1 z_{t-1} + \mu$$

   Plot $\hat{y}_t$ and $y_t$ versus $t$.

2. Consider the following AR(2) Model.

   $$z_t = \phi_1 z_{t-1} + \phi_2 z_{t-2} + \epsilon_t$$

   Derive the following equation:

   $$\rho_k = \phi_1 \rho_{k-1} + \phi_2 \rho_{k-2}$$

   Setting $k = 1$ and then $k = 2$ produces two equations which have two unknowns $\phi_1$ and $\phi_2$. Solve for $\phi_1$ and $\phi_2$ in terms of the first and second lag auto-correlations, $\rho_1$ and $\rho_2$. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$, $\hat{y}_1 = y_1$ and for $k <- 2$ until $y$.dim

   $$z_t = y_t - \mu$$
   $$\hat{y}_t = \phi_1 z_{t-1} + \phi_2 z_{t-2} + \mu$$

   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 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 sig2 = 10000.0
val noise = Normal (0.0, sig2)
val n = 50
val t = VectorD.range (0, n)
val y = VectorD (for (i <- 0 until n) yield 40 * (i-1) - (i-2) * (i-2) + noise.gen)

Now create an AR(1) model, train it and show its report.

```scala
val ar = new AR (y) // time series data
ar.train ().eval () // train for AR(1) model
println (ar.report)
```

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.

```scala
ar.plotFunc (ar.acF, "ACF")
ar.plotFunc (ar.pacF, "PACF")
```

Choose a value for \( p \) and retrain the model.

```scala
ar.retrain (p).eval () // retrain for AR(p) model
println (ar.report)
```

Also, plot \( \hat{y}_t \) and \( y_t \) versus \( t \).

5. The \( k^{th} \) lag auto-covariance is useful when the stochastic process \( \{y_t | t \in [0,K]\} \) is covariance stationary.

\[
\gamma_k = C[y_t, y_{t-k}]
\]

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., \( C[y_4, y_3] = C[y_2, y_1] = \gamma_1 \). Covariance stationary also requires that \( E[y_t] = \mu \), 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?
10.3 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 day's errors or shocks. In the simplest case,

\[ y_t = \delta + \theta_1 \epsilon_{t-1} + \epsilon_t \]

To make the data zero-centered, again let \( z_t = y_t - \mu \)

\[ z_t = \theta_1 \epsilon_{t-1} + \epsilon_t \]  \hspace{1cm} (10.15)

Errors are computed in the usual way.

\[ \epsilon_t = y_t - \hat{y}_t \]

The forecast \( \hat{y}_t \) will be \( \mu + \theta_1 \epsilon_{t-1} \), that is, the mean of the process plus some fraction of yesterday's shock. Let us assume that \( \mu = 100 \) and the parameter/coefficient \( \theta_1 \) 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 10.3.

<table>
<thead>
<tr>
<th>t</th>
<th>( y_t )</th>
<th>( z_t )</th>
<th>( \epsilon_{t-1} )</th>
<th>( \hat{z}<em>t = \theta_1 \epsilon</em>{t-1} )</th>
<th>( \hat{y}_t )</th>
<th>( \epsilon_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>110.0</td>
<td>10.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>10.0</td>
</tr>
<tr>
<td>2</td>
<td>120.0</td>
<td>20.0</td>
<td>10.0</td>
<td>8.0</td>
<td>108.0</td>
<td>12.0</td>
</tr>
<tr>
<td>3</td>
<td>110.0</td>
<td>10.0</td>
<td>12.0</td>
<td>9.6</td>
<td>109.6</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>90.0</td>
<td>-10.0</td>
<td>0.4</td>
<td>0.3</td>
<td>100.3</td>
<td>-10.3</td>
</tr>
</tbody>
</table>

The question of AR versus MA may be viewed as which column \( z_t \) or \( \epsilon_t \) leads to better forecasts. It depends on the data, but considering the GDP example again, \( z_t \) indicates how far above or below the mean the current GDP is. One could imagine shocks to GDP, such as new tarriffs, tax cuts or bad weather being influential shocks to the economy. In such cases, an MA model may provide better forecasts than an AR model.

10.3.1 Selection Based on ACF and PACF

10.3.2 MA(\( q \)) Model

The model equation for an MA(\( q \)) model includes the past \( q \) values of \( \epsilon_t \).

\[ z_t = \theta_1 \epsilon_{t-1} + \ldots \theta_q \epsilon_{t-q} + \epsilon_t \]  \hspace{1cm} (10.16)
10.4 ARMA

The ARMA class provide 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.

A $p^{th}$-order Auto-Regressive AR($p$) model predicts the next value $y_t$ from a weighted combination of prior values.

$$y_t = \delta + \phi_1 y_{t-1} + ... + \phi_p y_{t-p} + \epsilon_t$$

A $q^{th}$-order Moving Average MA($q$) model predicts the next value $y_t$ from the combined effects of prior noise/disturbances.

$$y_t = \delta + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} + \epsilon_t$$

There are multiple ways to combine multiple regression with time series analysis. One common technique called Time Series Regression is to use multiple linear regression and model its residuals using ARMA models.

---

**Example Problem:**

**Class Methods:**

@param y the input vector (time series data)
@param t the time vector

class ARMA (y: VectoD, t: VectoD)
    extends Predictor with Error

    def est_ar (p: Int = 1): VectoD =
def durbinLevinson: MatriD =
def ar (phi: VectoD): VectoD =
def est_ma (q: Int = 1): VectoD =
def ma (theta: VectoD): VectoD =
def train ()
def predict (y: VectoD): Double =
def predict (z: MatriD): VectoD =
def plotFunc (fVec: VectoD, name: String)
def smooth (l: Int): VectoD =

---
10.5 ARIMA
10.6 ARIMAX
10.7 SARIMA
10.8 Exponential Smoothing
10.9 Dynamic Linear Models

The next state vector $x_{t+1}$ is dependent on the current state $x_t$ and a noise vector $v_t$.

$$x_{t+1} = Fx_t + Gv_t$$

At time $t$, the observation vector $y_t$ is dependent on the current state $x_t$ and a second noise vector $w_t$.

$$y_t = Hx_t + w_t$$

The transition/coefficient matrices, F, G and H indicate
10.10 Recurrent Neural Networks (RNN)

10.10.1 Gate Recurrent Unit (GRU) Networks

10.10.2 Long Short Term Memory (LSTM) Networks
10.11 Temporal Convolutional Networks (TCN)
10.12 ODE Parameter Estimation

\[ y = x(t) + \epsilon \]

\[ \frac{dx(t)}{dt} = f(x(t); b) \]

10.12.1 Non-Linear Least Squares (NLS)

10.12.2 Least Squares Approximation (LSA)
Chapter 11

Spatial Models
11.1 Convolutional Neural Networks
Chapter 12

Clustering

Clustering is related to classification, except that specific classes are not prescribed. Instead data points (vectors) are placed into clusters based on some distance metric (e.g., Euclidean or Manhattan distance). Points in a cluster are according to some metric closer to each other than to points not in their cluster.
12.1 Clusterer

The Clusterer trait provides a common framework for several clustering algorithms.

Trait Methods:

trait Clusterer

  def cluster (): Array [Int]
  def classify (y: VectoD): Int
  def name_ (n: Array [String])
  def getName (i: Int): String =
12.2 K-Means Clustering

The `KMeansClustering` class clusters several vectors/points using k-means clustering. Either (1) randomly assign points to \( k \) clusters or (2) randomly pick \( k \) points as initial centroids (technique (1) to work better and is the primary technique). Iteratively, reassign each point to the cluster containing the closest centroid. Stop when there are no changes to the clusters.

---

**Example Problem:**

**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 \( s \) the random number stream (to vary the clusters made)
- @param primary true indicates use the primary technique for initiating the clustering

```scala
class KMeansClustering (x: Matrix, k: Int, s: Int = 0, primary: Boolean = true)
  extends Clusterer with Error

  def distance (u: Vector, v: Vector): Double =
  def assign ()
  def reassign (): Boolean =
  def pickCentroids ()
  def calcCentroids ()
  def cluster (): Array[Int] =
  def classify (y: Vector): Int =
```

227
12.3 Hierarchical Clustering

Cluster several vectors/points using hierarchical clustering. Start with each point forming its own cluster and merge clusters until there are only $k$.

Example Problem:

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 HierClustering (x: Matrid, k: Int = 2)
    extends Clusterer with Error

    def distance (u: VectoD, v: VectoD): Double =
    def clustDist (setA: Set [VectoD], setB: Set [VectoD]): Double =
    def initClusters ()
    def cluster (): Array [Int] =
    def finalClusters ()
    def calcCentroids ()
    def classify (y: VectoD): Int =

228
12.4 Markov Clustering

The `MarkovClustering` 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.

---

Example Problem:

---

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

```scala
class MarkovClustering (t: MatriD, k: Int = 2, r: Double = 2.0)
  extends Clusterer with Error
```

```scala
def addSelfLoops (weight: Double = 1.0)
def normalize ()
def expand ()
def inflate (): Boolean =
def processMatrix (): MatriD =
def cluster (): Array [Int] =
def classify (y: VectoD): Int =
```
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 (): MatriD
def recover (): MatriD
13.2 Principal Component Analytics

The PrincipalComponents class computes the Principal Components (PCs) for data matrix $x$. It can be used to reduce the dimensionality of the data. 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).

Example Problem:

Class Methods:

@param x the data matrix to reduce, stored column-wise

class PrincipalComponents (x: MatrID)

def meanCenter (): VectoD =
def computeCov (): MatrID =
def computeEigenVectors (eVal: VectoD): MatrID =
def findPCs (k: Int): MatrID =
def reduceData (): MatrID =
def recover (): MatrID = reducedMat * featureMat.t + mu
def solve (i: Int): (VectoD, VectoD) =
Chapter 14

Functional Data Analysis
14.1 Basis Functions
14.2 Functional Smoothing
14.3 Functional Principal Component Analysis
14.4 Functional Regression
Chapter 15

Simulation Models

15.1 Introduction to Simulation

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics. The focus of this document is simulation modeling. Viewed as black-box, a simple model maps an input vector $\mathbf{x}$ and a scalar time $t$ to an output/response vector $y$.

$$y = f(\mathbf{x}, t)$$

A simulation model adds to these the notion of state, represented by a vector-valued function $\mathbf{s}(t)$. 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.

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 [23].

- **State-Oriented Models.** State-oriented models, including Generalized Semi-Markov Processes (GSMPs), can be defined using three functions,

  - an activation function $\{e\} = a(\mathbf{s}(t))$,
  - a clock function $t' = c(\mathbf{s}(t), e)$ and
  - a state-transition function $\mathbf{s}(t') = \mathbf{d}(\mathbf{s}(t), e)$.

In simulation, advancing to the current state $\mathbf{s}(t)$ 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 $\mathbf{s}(t')$. In this paper we tie in the input and output vectors. The input vector $\mathbf{x}$ 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 state. Essentially, the activation and state transition functions are divided into several simpler functions, one for each event $e$:

$$
\begin{align*}
\{e\} &= a_e(s(t)) \text{ and } \\
{s(t')} &= d_e(s(t)).
\end{align*}
$$

Time advance is simplified to just setting the time $t'$ to the time of the most imminent event on a future event list.

• **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 co-routines (or threads/actors as a more heavyweight alternative). One co-routine for each active entity. The overall state of a simulation is then a combination of the states of each active entity and the global shared state, which may include 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 were recently added to DeMO, since hybrid models that combine continuous and discrete aspects are becoming more popular. In this section, modeling the flight of a golf ball is considered. Let the response vector $y = [y_0, y_1]$ where $y_0$ indicates the horizontal distance traveled, while $y_1$ indicates the vertical height of the ball. Future positions $y$ depends on the current position and time $t$. Using Newton’s Second Law of Motion, $y$ can be estimated by solving a system of Ordinary Differential Equations (ODEs) such as

$$
\dot{y} = f(y, t), \quad y(0) = y_0.
$$

The **Newtons2nd** 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 [?].

In addition to these main modeling paradigms, ScalaTion support a simpler approach called Tableau Oriented Models.
15.2 Tableau Oriented

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 10 customers are simulated, the matrix will have 10 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 [Bank].

```scala
object Bank extends App {
  val stream = 1 // random number stream (0 to 99)
  val lambda = 6.0 // customer arrival rate (per hour)
  val mu = 7.5 // customer service rate (per hour)
  val maxCusts = 10 // stopping rule: simulate maxCusts
  val iArrivalRV = Exponential (HOUR/lambda, stream) // inter-arrival time random variate
  val serviceRV = Exponential (HOUR/mu, stream) // service time random variate
  val label = Array ("ID-0", "IArrival-1", "Arrival-2", "Start-3", "Service-4", "End-5", "Wait-6", "Total-7")
  val mm1 = new Model ("M/M/1 Queue", maxCusts, Array (iArrivalRV, serviceRV), label)
  mm1.simulate ()
  mm1.report
}

15.2.1 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

```scala
class Model (name: String, m: Int, rv: Array [Variate], label: Array [String])
{
  def simulate ()
  def report
}
```
15.3 Event Oriented

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. 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 (FEL). Event Graphs 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.

15.3.1 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.

For example, to create a simple bank simulation model, one could use the three classes defined in the event-scheduling engine to create subclasses of Event, called Arrival and Departure, and one subclass of Model, called BankModel. The complete code for this example may be found in Bank.

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. For the Arrival class, 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, otherwise it schedule 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).

@param customer the entity that arrives, in this case a bank customer

```scala
case class Arrival (customer: Entity) extends Event (customer, this) // entity, model
{
  def occur ()
  {
    if (nArr < nArrivals-1) {
      val iArrivalT = iArrivalRV.gen
      val next2Arrive = Entity (clock + iArrivalT, serviceRV.gen) // next customer
      schedule (iArrivalT, Arrival (next2Arrive))
    } // if
    if (nIn > 0) { // teller is busy
      waitQueue.enqueue (customer)
    } else {
      t_q_stat.tally (0.0)
      t_s_stat.tally (schedule (customer.serviceT, Departure (customer)))
    } // if
    nArr += 1 // update the current state
    nIn  += 1
  }
```

244
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

case class Departure (customer: Entity) extends Event (customer, this) // entity, model
{
  def occur ()
  {
    t_y_stat.tally (clock - customer.arrivalT)
    if (nIn > 1) {
      val next4Service = waitQueue.dequeue () // first customer in queue
      t_q_stat.tally (clock - next4Service.arrivalT)
      t_s_stat.tally (schedule (next4Service.serviceT, Departure (next4Service)))
    } // if
    nIn -= 1 // update the current state
    nOut += 1
  } // occur
}

In order to collect statistical information, the occur methods of both event classes call the tally method from the Statistics class to obtain statistics on the time in queue t_q_stat, the time in service t_s_stat and the time in system t_y_stat.

The three classes used for creating simulation models following the Event Scheduling paradigm are discussed in the next three subsections.

Event.scala

The Event class provides facilities for defining simulation events. 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 act times for events and converting an event to its string representation. Note: unique identification and the event/activation time (actTime) are mixed in via the PQItem trait.

Class Methods:

@param entity the entity involved in this event
@param director the controller/scheduler that this event is a part of
@param proto the prototype (serves as node in animation) for this event

abstract class Event (val entity: Entity, director: Model, val proto: Event = null)
  extends PQItem with Ordered [Event]
def compare (ev: Event): Int = ev.actTime compare actTime

def occur ()

override def toString = entity.toString + "\t" + me

Entity.scala

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 arrivalT the time at which the entity arrived
@param serviceT the amount of time required for the entity’s next service

case class Entity (val arrivalT: Double, var serviceT: Double)

override def toString = "Entity-" + eid

Model.scala

The Model class schedules events and implements the time advance mechanism for event oriented simulation models. It provides methods to schedule and cancel events. Scheduled events are placed in the Future Event List (FEL) in time order. The simulate method will cause the main simulation loop to execute, which will remove the most imminent event from the FEL and invoke its occur method. The simulation will continue until a stopping rule evaluates to true. 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, animation: Boolean = false)
  extends ModelT with Identity

def schedule (timeDelay: Double, event: Event): Double =
def cancel (event: Event)

def simulate (startTime: Double = 0.0): ListBuffer [Statistic] =

def report (eventType: String, links: Array [CausalLink] = Array ())

def report (vars: Array [Tuple2 [String, Double]])

def reports (stats: Array [Tuple2 [String, Statistic]])

def getStatistics: ListBuffer [Statistic] =

def animate (who: Identity, what: Value, color: Color, shape: Shape, at: Array [Double])


The animate methods are used with Event Graphs (see the next section).
15.3.2 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 four classes: `Entity`, `Model`, `EventNode` and `CausalLink`. The first two are shared with Event Scheduling. An Entity, such as a customer in a bank, flows through the simulation. The `Model` serves as a container/controller for the whole simulation. The last two are specific to Event Graphs. 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. 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.

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]. In more complex situations, one would typically define a subclass of `Entity` to represent the customers in the bank.

```scala
class BankModel (name: String, nArrivals: Int, arrivalRV: Variate, serviceRV: Variate)
  extends Model (name)
```

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,

```scala
() => nArr < nArrivals
```

is a 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, one first defines the state variables: `nArr`, `nIn` and `nOut`. For animation of the event graph, a prototype for each type of event is created and displayed as a node. The edges connecting these prototypes represent the causal links. The `aLinks` 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 `dLinks` 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).

```scala
//:: define the state variables for 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 have finished and left the bank
```

```scala
//:: define the nodes in the event graph (event prototypes)
val protoArrival  = Arrival (null) // prototype for all Arrival events
val protoDeparture = Departure (null) // prototype for all Departure events
```
//: define the edges in the event graph (causal links between events)

val aLinks = Array (CausalLink ("link2A", this, () => nArr < nArrivals, protoArrival,
  () => Arrival (null), arrivalRV),
  CausalLink ("link2D", this, () => nIn == 0, protoDeparture,
  () => Departure (null), serviceRV))

val dLinks = Array (CausalLink ("link2D", this, () => nIn > 1, protoDeparture,
  () => Departure (null), serviceRV))

protoArrival.displayLinks (aLinks)
protoDeparture.displayLinks (dLinks)

An animation of the Event Graph consisting of two EventNodes Arrival and Departure and three CausalLinks is depicted in Figure 15.1.

Figure 15.1: Event Graph Animation of a Bank.

The main thing to write within each subclass of EventNode is the occur method. To handle arrival events, the occur method of the Arrival class first calls the super.occur method from the superclass to trigger other events using the causal links and then updates the state by incrementing both the number of arrivals (nArr) and the number in the system (nIn).

@param customer the entity that arrives, in this case a customer

case class Arrival (customer: Entity)
  extends EventNode (customer, this, protoArrival, Array (150.0, 200.0, 50.0, 50.0), aLinks)
{
  override def occur ()
  {
    super.occur () // handle casual links
    nArr += 1 // update the current state
    nIn += 1
  } // occur

} // Arrival class
To handle departure events, the `occur` method of the `Departure` class first calls the `occur` method of the superclass to trigger other events using the causal links and then updates the state by decrementing the number in the system (`nIn`) and incrementing the number of departures (`nOut`).

@param customer the entity that departs, in this case a customer

```scala
case class Departure (customer: Entity)
  extends EventNode (customer, this, protoDeparture, Array(450.0, 200.0, 50.0, 50.0), dLinks)
{
  override def occur ()
  {
    super.occur () // handle casual links
    nIn -= 1 // update the current state
    nOut += 1
  } // occur
}
``` // Departure class

Two of the three classes used for creating simulation models following the Event Scheduling paradigm can be used for Event Graphs, namely `Entity` and `Model`. `Event` must be replaced with its subclass called `EventNode`. These 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.

**EventNode.scala**

The `Event` class provides facilities for defining simulation events. Subclasses of `Event` provide event-logic in their implementation of the `occur` method. Note: unique identification and the event/activation time (`actTime`) are mixed in via the PQItem trait.

**Class Methods:**

- `@param proto` the prototype (serves as node in animation) for this event
- `@param entity` the entity involved in this event
- `@param links` the causal links used to trigger other immediate/future events
- `@param director` the controller/scheduler that this event is a part of
- `@param at` the location of this event

```scala
abstract class EventNode (val proto: Event, entity: Entity, links: Array[CausalLink],
  director: Model, at: Array[Double] = Array ()
  extends PQItem with Ordered [Event]

  def compare (ev: Event): Int = ev.actTime.compare (actTime)
  def occur ()
  def display ()
  def displayLinks (outLinks: Array[CausalLink])
```
The 'CausalLink' class provides casual links between events. After an event has updated the state, it checks its causal links to schedule/cancel other events.

Class Methods:

- @param _name the name of the causal link
- @param condition the condition under which it is followed
- @param makeEvent function to create an event
- @param delay the time delay in scheduling the event
- @param cancel whether to schedule (default) or cancel the event

```scala
case class CausalLink (_name: String, director: Model, condition: () => Boolean, causedEvent: Event,
                      makeEvent: () => Event, delay: Variate, cancel: Boolean = false)
```

```scala
def display (from: Event, to: Event)
def tally (duration: Double) { _durationStat.tally (duration) }
def accumulate (value: Double, time: Double) { _persistentStat.accumulate (value, time) }
def durationStat = _durationStat
def persistentStat = _persistentStat
```
15.4 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 below. Following this world view, one simply constructs the simulation components and then provides a script for entities (SimActors) 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, actors 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 Scala Actors for concurrency. Future implementations will include use of continuations and Akka Actors.

ScalaTion includes several types of model components: Gate, Junction, Resource, Route, Sink, Source, Transport and WaitQueue. 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 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.

The BankModel may be developed as follows: The BankModel first defines 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. Finally, a 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 (Scala Actors) and is analogous to the run method for Java/Scala Threads.

class BankModel (name: String, nArrivals: Int, iArrivalRV: Variate, nUnits: Int, serviceRV: Variate, moveRV: Variate) extends Model (name) {
  val entry = Source ("entry", this, Customer, 0, nArrivals, iArrivalRV, (100, 290))
  val tellerQ = WaitQueue ("tellerQ", (330, 290))
  val teller = Resource ("teller", tellerQ, nUnits, serviceRV, (350, 285))
  val door = Sink ("door", (600, 290))
  val toTellerQ = new Transport ("toTellerQ", entry, tellerQ, moveRV)
  val toDoor = new Transport ("toDoor", teller, door, moveRV)

  addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)

  case class Customer () extends SimActor ("c", this) {
    def act () {
      toTellerQ.move ()
      if (teller.busy) tellerQ.waitIn () else tellerQ.noWait ()
      teller.utilize ()
      teller.release ()
      toDoor.move ()
      door.leave ()
    } // act
  } // Customer
} // BankModel class

Note, that the bank model for event-scheduling did not include time delays and events for moving token along transports. In BankModel2, the impact of transports is reduced 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. Both BankModel and BankModel2 are in the apps.process package as well as CallCenterModel, ERoomModel, IntersectionModel, LoopModel MachineModel and RoadModel.

15.4.1 Component.scala

The Component trait provides basic common feature 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).

Class Methods:
trait Component extends Identity

def initComponent (label: String, loc: Array[Double])
def initStats (label: String)
def director = _director
def setDirector (dir: Model)
def display ()
def tally (duration: Double) { _durationStat.tally (duration) }
def accumulate (value: Double, time: Double) { _persistentStat.accumulate (value, time) }
def durationStat = _durationStat
def persistentStat = _persistentStat

15.4.2 Signifiable.scala

The Signifiable trait defines standard messages sent between actors implementing process interaction simulations.

Class Methods:

trait Signifiable

15.4.3 SimActor.scala

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 Scala’s Actor 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.

Class Methods:

@param name the name of the entity/SimActor
@param director the director controlling the model

abstract class SimActor (name: String, director: Model)
    extends Actor with Signifiable with PQItem with Ordered[SimActor] with Locatable

def subtype: Int = _subtype
def setSubtype (subtype: Int) { _subtype = subtype }
def trajectory: Double = traj
def setTrajectory (t: Double) { traj = t }
def compare (actor2: SimActor): Int = actor2.actTime compare actTime
def act ()
def yetToAct = _yetToAct
def nowActing () { _yetToAct = false }
def time = director.clock
def schedule (delay: Double)
def yieldToDirector (quit: Boolean = false)
15.4.4 Source.scala

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.

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 subtype indicator of the subtype of the entities to make
@param units the number of entities to make
@param iArrivalTime the inter-arrival time distribution
@param at the location of the source (x, y, w, h)

class Source (name: String, director: Model, makeEntity: () => SimActor, subtype: Int, units: Int, iArrivalTime: Variate, at: Array[Double])
  extends SimActor (name, director) with Component

def this (name: String, director: Model, makeEntity: () => SimActor, units: Int, iArrivalTime: Variate, at: Array[Double])
  extends SimActor (name, director) with Component

def display ()
def act ()

15.4.5 Sink.scala

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.

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, director: Model, makeEntity: () => SimActor, subtype: Int, units: Int, iArrivalTime: Variate, xy: Tuple2[Double, Double])
  extends Component

def display ()
def leave ()

15.4.6 Transport.scala

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.

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 = R2 (0.0, 0.0), shift2: R2 = R2 (0.0, 0.0))
    extends Component
    def display ()
    override def at: Array [Double] =
    def jump ()
    def move ()

15.4.7 Resource.scala

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.

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: Tuple2 [Double, Double])
    def changeUnits (dUnits: Int)
    def display ()
    def busy = inUse == units
    def utilize ()
    def utilize (duration: Double)
    def release ()

255
15.4.8 WaitQueue.scala

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).

Class Methods:

```scala
@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
```

```scala
def this (name: String, xy: Tuple2[Double, Double], cap: Int)
def isFull: Boolean = length >= cap
def barred: Int = _barred
def display ()
def waitIn ()
def noWait ()
```

15.4.9 Junction.scala

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.

Class Methods:

```scala
@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
```

```scala
def this (name: String, director: Model, jTime: Variate, xy: Tuple2[Double, Double])
def display ()
def move ()
```

15.4.10 Gate.scala

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 $\Rightarrow$ open, red $\Rightarrow$ shut).

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 at the location of the Gate (x, y, w, h)
@param shut0 Boolean indicating if the gate is 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, at: Array[Double], shut0: Boolean, cap: Int = 10)
  extends SimActor (name, director) with Component
  
  def this (name: String, director: Model, line: WaitQueue, units: Int, onTime: Variate, offTime: Variate, xy: Tuple2[Double, Double], shut0: Boolean, cap: Int)
  
  def shut: Boolean = _shut
  def display ()
  def release ()
  def act ()
  def gateColor: Color = if (shut) red else green
  def flip () { _shut = ! shut }
  def duration: Double = if (shut) offTime.gen else onTime.gen

15.4.11 Route.scala

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.

Class Methods:

@param name the name of the route
@param k the number of lanes/transport 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,
15.4.12 Model.scala

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. It maintains a time-ordered priority queue to activate/re-activate each of the entities. Each entity (SimActor) is implemented as a Scala Actor and may be roughly thought of as running in its own thread.

Class Methods:

@param name the name of the model
@param animating whether to animate the model

class Model (name: String, animating: Boolean = true)
    extends Actor with Signifiable with Modelable with Component

def addComponent (_parts: Component*) { for (p <- _parts) parts += p }
def addComponents (_parts: List [Component]*) { for (p <- _parts; q <- p) parts += q }
def theActor = _theActor
def simulate (startTime: Double = 0.0)
def reschedule (actor: SimActor) { agenda += actor }
def act ()
def report

def reportf { new StatTable (name + " statistics", getStatistics) }
def getStatistics: ListBuffer [Statistic] =
def display ()
def animate (who: Identifiable, what: Value, color: Color, shape: Shape, at: Array [Double])
Chapter 16

Optimization Used 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,

$$\begin{align*}
\text{minimize} & \quad f(\mathbf{x}) \\
\text{subject to} & \quad g(\mathbf{x}) \leq 0 \\
& \quad h(\mathbf{x}) = 0
\end{align*}$$

where $f(\mathbf{x})$ is the objective function, $g(\mathbf{x}) \leq 0$ are the inequality constraints, and $h(\mathbf{x}) = 0$ are the equality constraints. Consider the example below.

$$\begin{align*}
\text{minimize} & \quad f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2 \\
\text{subject to} & \quad g(\mathbf{x}) = [x_1 - 3, x_2 - 1] \leq 0 \\
& \quad h(\mathbf{x}) = x_1 - x_2 = 0
\end{align*}$$

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 $x_1 = x_2$. 

259
16.1 Gradient Descent

One the simplest algorithms for unconstrained optimiztion is Gradient Descent. Imagine you are in a
mountain range at some point \( x \) with elevation \( f(x) \). 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,

\[
\text{minimize } f(x) = (x_1 - 4)^2 + (x_2 - 2)^2
\]

the gradient of the objective function \( \nabla f(x) \) is the vector formed by the partial derivatives \( \left\{ \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2} \right\} \)

\[
\nabla f(x) = [2(x_1 - 4), 2(x_2 - 2)]
\]

In its most elemental form the algorithm simply moves in the direction that is opposite to the gradient
\( -\nabla f(x) \) 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 \( \eta \) (tunable hyper-parameter typically smaller than one). Using a tuned
learning rate, update your current location \( x \) as follows:

\[
x = x - \eta \nabla f(x)
\] (16.1)

Repeat this process until a stopping rule signals sufficient convergence. Examples of stopping rules include
stop when the change to \( x \) or \( f(x) \) becomes small or after the objective function has increased for too many
consecutive iterations/steps.

16.1.1 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. 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 alterative 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.
16.1.2

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 similary Quality
of Fit (QoF) measure, it may be used for the objective function. For a Perceptron, the equation has been
developed for hse(b)
hse(b) = (y − f (Xb) · (y − f (Xb)
in which case the gradient is
∇hse(b) = − X t [f 0 (Xb) ]

(16.2)

where  = y − f (Xb).
For each epoch, the parameter vector b is update according to equation 16.1.
b = b − η∇hse(b)

16.1.3

(16.3)

Exercises

1. Write a ScalaTion program to solve the example problem given above.
// function to optimize
def f(x: VectoD): Double = (x(0) - 4)~^2 + (x(1) - 2)~^2
// gradient of objective function
def grad (x: VectoD): VectoD = VectorD (?, ?)
val x
= new VectorD (2)
val eta = 0.1

// vector to optimize
// learning rate

for (k <- 1 to MAX_ITER) {
x -= grad (x) * eta
println (s"$k: x = $x, f(x) = ${f(x)}, lg(x) = ${lg(x)}, p = $p, l = $l")
} // 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_ITER, 2)
z(k-1) = x.copy
new Plot (z.col(0), z.col(1)

// store x’s trajectory

261


16.2 Stochastic Gradient Descent

Continuing for a Perceptron, starting with $hse(b)$

$$hse(b) = (y - f(Xb)) \cdot (y - f(Xb))$$

an estimate of the gradient is computed for a limited number of instances (a batch). Several non-overlapping batches are created simultaneous by taking a random permutation of the row indices of data/input matrix $X$. The permutation is split into $n_B$ batches. Letting $i_B$ be the indices for the $i^{th}$ batch and $X[i_B]$ be the projection of matrix $X$ onto the rows in $i_B$, the estimate for the gradient is simply

$$\nabla hse(b) = -X[i_B]^t[f'(X[i_B]b) \epsilon]$$

(16.4)

where $\epsilon = y[i_B] - f(X[i_B]b)$. Using the definition of the delta vector

$$\delta = -f'(X[i_B]b) \epsilon$$

the gradient becomes

$$\nabla hse(b) = X[i_B]^t\delta$$

For each epoch, $n_B$ batches are created. For each batch, the parameter vector $b$ is update according to equation 16.1, using that batch’s estimate for the gradient.

$$b = b - \eta \nabla hse(b) = b - X[i_B]^t\delta \eta$$

(16.5)

The corresponding ScalaTion code is in the Optimizer_SGD object.

```scala
def optimize (x: Matrix, y: Vector, b: Vector, 
etta : Double = hp.default("eta"),
bSize: Int = hp.default("bSize").toInt,
maxEpochs: Int = hp.default("maxEpochs").toInt,
f1: AFF = f_sigmoid): (Double, Int) =
{
  val idx = VectorI.range (0, x.dim1) // instance index range
  val permGen = PermutedVecI (idx, ranStream) // permutation vector generator
  val nB = x.dim1 / bSize // the number of batches
  val stop = new StoppingRule () // rule for stopping early
  var eta = etta // set initial learning rate
  for (epoch <- 1 to maxEpochs) { // iterate over each epoch
    val batches = permGen.igen.split (nB) // permute index, split into batches
    for (ib <- batches) b -= updateWeight (x(ib), y(ib)) // iteratively update vector b
    val sse = sseF (y, f1.fV (x * b)) // recompute sum of squared errors
    val (b_best, sse_best) = stop.stopWhen (b, sse)
    if (b_best != null) {
      b.set (b_best)
    }
  }
  (sse_f, epoch)
}
The above code 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 simply looks for early termination based on a stopping rule and records the best solution for b found so far. The final part of the outer loop, increases the learning rate eta at the end of each adjustment period (as the algorithm get closer to an optimal solution, gradients shrink and may slow down the algorithm).

```scala
def updateWeight (x: MatriD, y: VectoD): VectoD =
{
    val yp = f1.fV (x * b) // yp = f(Xb)
    val e = yp - y // negative error vector
    val d = f1.dV (yp) * e // delta vector

    val eta_o_sz = eta / x.dim1 // eta over current batch size
    x.t * d * eta_o_sz // return change in vector b
} // updateWeight

(sseF (y, f1.fV (x * b)), maxEpochs) // return sse and # epochs
} // optimize
```

The above code shows the `updateWeight` nested method and the final line of the outer `optimize` method. 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 batch size normalized learning rate, and finally, returning the parameter vector b update. The final line of `optimize` simply returns the value of the objective function and number epochs used by the algorithm.
16.3 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. These contributions may be collected via the gradient-based parameter updates to the parameter vector $b$.

$$ b_{\text{up}} = \eta \nabla hse(b) = X[i_B]\delta \eta $$(16.6)

Again for a Perceptron, one can include a momentum vector $m_\text{o}$ to hold the weighted average of recent gradients. The momentum vector $m_\text{o}$ is then simply the sum of the old momentum decayed by parameter $\beta$ and $b_{\text{up}}$.

$$ m_\text{o} = \beta m_\text{o} + b_{\text{up}} $$ (16.7)

Then the parameters are updated as follows:

$$ b = b - m_\text{o} $$ (16.8)

If $\beta$ is zero, that algorithm behaves the same as Stochastic Gradient Descent. At the other extreme, if $\beta = 1$, 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. In ScalaTion, BETA is set to 0.9, but can easily be changed.

To add momentum into the code, the `updateWeight` method from the last section needs be slightly modified.

```scala
def updateWeight (x: MatriD, y: VectoD): VectoD =
{
  val yp = f1.fV (x * b) // yp = f(Xb)
  val e = yp - y // negative of the error vector
  val d = f1.dV (yp) * e // delta vector for y
  val eta_o_sz = eta / x.dim1 // eta over the current batch size
  val bup = x.t * d * eta_o_sz // gradient-based change in input-output weights
  mo = mo * BETA + bup // update momentum
  mo // return momentum
}
```

See the Optimzer_SGDM for more coding details.
16.4 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 $\lambda$. At optimality, the gradient of $f$ should be orthogonal to the surface defined by the constraint $h(x) = 0$, otherwise, moving along the surface in the opposite direction to the gradient ($-\nabla f(x)$ for minimization) would improve the solution. Since the gradient of $h$, $\nabla h(x)$, is orthogonal to the surface as well, this implies that the two gradients should only differ by a constant multiplier $\lambda$.

$$-\nabla f(x) = \lambda \nabla h(x)$$ \hfill (16.9)

In general, such problems can solved by defining the Lagrangian

$$L(x, \lambda) = f(x) - \lambda \cdot h(x)$$ \hfill (16.10)

where $\lambda$ is a vector of Lagrange multipliers. When there is a single equality constraint, this becomes

$$L(x, \lambda) = f(x) - \lambda h(x)$$

Taking the gradient of the Lagrangian w.r.t. $x$ and $\lambda$ yields a vector of dimension $n + 1$.

$$\nabla L(x, \lambda) = [\nabla f(x) - \lambda \nabla h(x), h(x)]$$

Now we may try setting the gradient to zero and solving a system of equations.

16.4.1 Example Problem

The Lagrangian for the problem given at the beginning of the chapter is

$$L(x, \lambda) = (x_1 - 4)^2 + (x_2 - 2)^2 - \lambda (x_1 - x_2)$$

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,

$$-2(x_1 - 4) = \lambda$$

$$-2(x_2 - 2) = -\lambda$$

$$x_1 - x_2 = 0$$

The first two equations are from the gradient w.r.t. $x$, while the third equation is simply the constraint itself $h(x) = 0$. The equations may be rewritten in the following form.

$$2x_1 + \lambda = 8$$

$$2x_2 - \lambda = 4$$

$$x_1 - x_2 = 0$$

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 $x = [3, 3]$ with $\lambda = 2$ where $f(x) = 2$. 

265
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. $x$ 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.
16.5  Karush-Kuhn-Tucker Conditions

Introducing inequality constraints makes the situation 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:

\[-\nabla f(x) = \alpha \cdot \nabla g(x) + \lambda \cdot \nabla h(x)\]  \hspace{1cm} (16.11)

The original constraints must also hold.

\[g(x) \leq 0 \quad \text{and} \quad h(x) = 0\]

Furthermore, the Lagrange multipliers for the inequality constraints \(\alpha\) are themselves constrained to be nonnegative.

\[\alpha \geq 0\]

When the objective function is at most quadratic and the constraints are linear, the problem of finding an optimal value for \(x\) 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.

16.5.1  Active and Inactive Constraints
16.6 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.

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad h(x) = 0
\end{align*}
\]

where \( f(x) \) is the objective function and \( h(x) = 0 \) are the equality constraints.

In penalty form, the constrained optimization problem becomes.

\[
\begin{align*}
\text{minimize} & \quad f(x) + \frac{\rho_k}{2} \left\| h(x) \right\|_2^2 \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{minimize} & \quad f(x) + \frac{\rho_k}{2} h(x) \cdot h(x)
\end{align*}
\]

The value of \( \rho_k \) increases (e.g., linearly) with \( k \) and thereby enforces the equality constraints more strongly with each iteration.

An alternative to minimizing \( f(x) \) with a quadratic penalty is to minimize using the Augmented Lagrangian \( L(x, \rho_k, \lambda) \).

\[
L(x, \rho_k, \lambda) = f(x) + \frac{\rho_k}{2} h(x) \cdot h(x) - \lambda \cdot h(x)
\]

where \( \lambda \) is the vector of Lagrange multipliers. After each iteration, the Lagrange multipliers are updated.

\[
\lambda = \lambda - \rho_k h(x)
\]

This method allows for quicker convergence without the need for the penalty \( \rho_k \) 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.

16.6.1 Example Problem

Consider the problem given at the beginning of the chapter with the inequality constraint left out.

\[
\begin{align*}
\text{minimize} & \quad f(x) = (x_1 - 4)^2 + (x_2 - 2)^2 \\
\text{subject to} & \quad h(x) = x_1 - x_2 = 0
\end{align*}
\]

where \( x \in \mathbb{R}^2 \), \( f \) is the objective function and \( h \) is the single equality constraint. The Augmented Lagrangian for this problem is
\[ L(x, \rho_k, \lambda) = (x_1 - 4)^2 + (x_2 - 2)^2 + \frac{\rho_k}{2} (x_1 - x_2)^2 - \lambda (x_1 - x_2) \]  

(16.13)

The gradient of the Augmented Lagrangian \( \nabla L(x, \rho_k, \lambda) \) is made up of the following two partial derivatives.

\[
\begin{align*}
\frac{\partial}{\partial x_1} & = 2(x_1 - 4) + \frac{\rho_k}{2} 2(x_1 - x_2) - \lambda \\
\frac{\partial}{\partial x_2} & = 2(x_2 - 2) - \frac{\rho_k}{2} 2(x_1 - x_2) + \lambda
\end{align*}
\]

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.

### 16.6.2 Exercises

1. Write a ScalaTion program to solve the example problem given above.

   ```scala
   // function to optimize
   def f(x: VectoD): Double = (x(0) - 4)^2 + (x(1) - 2)^2

   // equality constraint to maintain
   def h(x: VectoD): Double = x(0) - x(1)

   // augmented Lagrangian
   def lg (x: VectoD): Double = f(x) + (p/2) * h(x)^2 - l * h(x)

   // gradient of Augmented Lagrangian
   def grad (x: VectoD): VectoD = 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_ITER) {
     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
   } // 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_ITER, 2)  // store x’s trajectory
z(k-1) = x.copy
new Plot (z.col(0), z.col(1))

3. Compare the Augmented Langragian Method with the Penalty Method by simply removing the Lagrange multiplier from the code.
16.7 Quadratic Programming

The **QuadraticSimplex** class solves Quadratic Programming (QP) problems using the Quadratic Simplex Algorithm. Given a constraint matrix \( A \), constant vector \( b \), cost matrix \( Q \) and cost vector \( c \), find values for the solution/decision vector \( x \) that minimize the objective function \( f(x) \), while satisfying all of the constraints, i.e.,

\[
\text{minimize } f(x) = \frac{1}{2}x \cdot Qx + c \cdot x
\]

subject to \( g(x) = Ax - b \leq 0 \)

Before considering the type of optimization algorithm to use, we may simplify the problem by applying the KKT conditions.

\[-\nabla f(x) = Qx + c = \alpha \cdot \nabla g(x) = \alpha \cdot A\]

Adding the constraints gives the following \( n \) equations and \( 2m \) constraints:

\[
Qx + c = \alpha \cdot A \\
Ax - b \leq 0 \\
\alpha \geq 0
\]

These equations have two unknown vectors, \( x \) of dimension \( n \) and \( \alpha \) of dimension \( m \).

The algorithm creates an simplex tableau. This implementation is restricted to linear constraints \( Ax \leq b \) and \( Q \) being a positive semi-definite matrix. Pivoting must now also handle non-linear 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)

```java
class QuadraticSimplex (a: MatrixD, b: VectorD, q: MatrixD, c: VectorD, 
var x_B: Array[Int] = null)
extends Error

def setBasis (j: Int = N, l: Int = M): Array[Int] = 
def entering (): Int =
def comple (l: Int): Int = 
def leaving (l: Int): Int =
```

271
def pivot (k: Int, l: Int)
def solve (): (VectorD, Double) =
def tableau: MatrixD = t
def primal: VectorD =
def dual: VectorD = null // FIX
def objValue (x: VectorD): Double = (x dot (q * x)) * .5 + (c dot x)
def showTableau ()
16.8 Coordinate Descent
16.9 Conjugate Gradient
16.10 Quasi-Newton Method
16.11 Alternating Direction Method of Multipliers
16.12 Nelder-Mead Simplex
Chapter 17

Parallel and Distributed Computing
17.1 MIMD - Multithreading
17.2 SIMD - Vector Instructions
17.3 Message Passing
17.4 Distributed Shared Memory
17.5 Microservices
17.6 Distributed Functional Programming
Bibliography


