Introduction to Data Science Using ScalaTion Release 2

John A. Miller
Department of Computer Science
University of Georgia

March 16, 2020

Contents

Chapter 1

Introduction to Data Science

1.1 Data Science

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

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

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

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

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 uncertainity.

1.2 ScalaTion

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

In ScalaTion, 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 distibuted processing. Databases are also scaling up to handle greater amounts of data, while at the same time increasing their analytics capabilities beyond the traditional On-Line Analytic Processing (OLAP). ScalaTion provides many analytics techniques found in tools like MATLAB, R and Weka. The analytics component contains six types of tools: **predictors**, **classifiers**, **forecasters**, **clusterers**, **recommenders** and **reducers**. A trait is defined for each type.

The latest version, ScalaTion 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` 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 A Data Science Project

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

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

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

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

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

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

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

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

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

where the parameters $\mathbf{b} = [b_0, b_1]$ are to be estimated. The line generated by the model should be plotted along with the $\{(x_{ij}, y_i)\}$ data points. Visually, look for patterns such white noise, linear relationship, quadratic relationship, etc. Plotting the residuals $\{(x_{ij}, \epsilon_i)\}$ will also be useful.

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

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

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

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

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

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

1.4 Additional Textbooks

More detailed development of this material can be found in textbooks on statistical learning, such as "An Introduction to Statistical Learning" (ISL) [13] and "The Elements of Statistical Learning" (ESL) [11]. See Table 1.1 for a mapping between the chpaters in three textbooks.

Table 1.1: Source Material Chapter Mappings

Topic	ScalaTion	**ISL**	**ESL**
Mathematical Preliminaries	Ch. 2	-	-
Data Management	Ch. 3	-	-
Prediction	Ch. 4	Ch. 3, 5, 6	Ch. 3
Classification	Ch. 5	Ch. 2, 5, 8	Ch. 4, 12, 13, 15
Classification - Continuous	Ch. 6	Ch. 4, 8, 9	Ch. 4, 12, 13, 15
Generalized Linear Models	Ch. 7	-	-
Generalized Additive Models	Ch. 8	-	-
Non-Linear Models and Neural Networks	Ch. 9	Ch. 7	Ch. 11
Time-Series/Temporal Models	Ch. 10	-	-
Clustering	Ch. 11	Ch. 10	Ch. 14
Dimensionality Reduction	Ch. 12	Ch. 6, 10	Ch. 14
Functional Data Analysis	Ch. 13	Ch. 7	Ch. 5
Simulation Models	Ch. 14	-	-
Optimization Used in Data Science	Appendix	-	-
Parallel/Disributed Computing	Appendix	-	-

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:

$$\begin{aligned} &P(A) \geq 0 \;\text{ for any event } A \subseteq S \\ &P(S) = 1 \\ &P(\cup A_i) = \sum P(A_i) \;\text{ for a countable collection of disjoint events} \end{aligned} \tag{2.1}$$

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

$$P(AB) = P(A \cap B) \;\in\; [0,\ min(P(A),\ P(B))] \tag{2.2}$$

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

$$P(A|B) \;=\; \frac{P(AB)}{P(B)} \tag{2.3}$$

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*.

$$P(A|B) \;=\; \frac{P(B|A)P(A)}{P(B)} \tag{2.4}$$

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.

$$y \;\in\; D_y \tag{2.5}$$

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

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

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.

$$F_y(y) = P(y \le y) \tag{2.6}$$

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 $F_y(6) = P(y \le 6) = 5/12$. The entire CDF for the discrete random variable y (roll of two dice), $F_y(y)$ is

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

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

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

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

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.

$$p_y(y_i) \;=\; F_y(y_i) - F_y(y_{i-1}) \tag{2.7}$$

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 uniformally spread amounst 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.

$$f_y(y) \;=\; \frac{dF_y(y)}{dy} \tag{2.8}$$

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

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

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

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

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.

$$\mathbb{E}\,[y] \;=\; \int_{D_y} y\, dF_y(y) \tag{2.9}$$

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:

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

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

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

Discrete Case

When y is a discrete random variable, we may write

$$\mathbb{E}\left[y\right] \;=\; \sum_{y \in D_y} y\, p_y(y) \tag{2.11}$$

The mean for rolling two dice is $\mathbb{E}\left[y\right] = 7$. 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

$$\mathbb{V}\left[y\right] \;=\; \mathbb{E}\left[(y - \mathbb{E}\left[y\right])^2\right] \tag{2.12}$$

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

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

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

$$\mathbb{SD}\left[y\right] \;=\; \sqrt{\mathbb{V}\left[y\right]} \tag{2.13}$$

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

$$\mathbb{C}\left[x, y\right] \;=\; \mathbb{E}\left[(x - \mathbb{E}\left[x\right])(y - \mathbb{E}\left[y\right])\right] \tag{2.14}$$

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 $\mathbf{z}$ be a vector of k random variables, then a *covariance matrix* is produced.

$$\mathbb{C}[\mathbf{z}] \;=\; \big[\mathbb{C}[z_i, z_j]\big]_{0 \le i,j < k}$$

2.1.9 Quantiles

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

$$\mathbb{Q}[y] \;=\; F_y^{-1}(\frac{1}{2}) \tag{2.15}$$

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

$$_p\mathbb{Q}[y] \;=\; F_y^{-1}(p) \tag{2.16}$$

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

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

Taking the inverse yields the iCDF.

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

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

2.1.10 Mode

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

$$\mathbb{M}[y] \;=\; \underset{y \in Dy}{\text{argmax}}\; p_y(y) \tag{2.17}$$

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

$$\mathbb{M}[y] \;=\; \underset{y \in Dy}{\text{argmax}}\; f_y(y) \tag{2.18}$$

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:

$$p_{x|y}(x, y) \;=\; P(x = x|y = y) \;=\; \frac{p_{x,y}(x, y)}{p_y(y)} \tag{2.19}$$

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

$$p_x(x) \;=\; \sum_{y \in D_y} p_{x,y}(x, y) \tag{2.20}$$

Continuous Case

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

$$f_{x|y}(x, y) \;=\; \frac{f_{x,y}(x, y)}{f_y(y)} \tag{2.21}$$

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

$$f_x(x) \;=\; \int\limits_{y \in D_y} f_{x,y}(x, y)dy \tag{2.22}$$

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:

$$\mathbb{E}\left[x|y = y\right] \;=\; \int\limits_{D_x} x\, dF_{x|y}(x, y) \tag{2.23}$$

When y is a continuous random variable, we may write

$$\mathbb{E}\left[x|y = y\right] = \int\limits_{D_x} x\, f_{x|y}(x, y)dx \tag{2.24}$$

When y is a discrete random variable, we may write

$$\mathbb{E}\left[x|y = y\right] = \sum_{x \in D_x} x\, p_{x|y}(x, y) \tag{2.25}$$

2.1.13 Conditional Independence

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

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

means that

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

2.1.14 Odds

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

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

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

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

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

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

Calculating probability from odds may be done as follows:

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

2.1.15 Example Problems

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

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

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

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

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

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

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

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

   where

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

Therefore, the conditional probability of y given x is

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

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

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

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

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

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

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

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

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

2.1.16 Estimating Parameters from Samples

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

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

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

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

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

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

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

$$\mathbb{E}\left[||\boldsymbol{\epsilon}||^2\right] \;=\; \mathbb{V}\left[||\boldsymbol{\epsilon}||\right] + \mathbb{E}\left[||\boldsymbol{\epsilon}||\right]^2 \tag{2.27}$$

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

Sample Mean

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

$$y \;=\; \mu + \epsilon$$

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

```
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)
```

See `scalation.stat.StatVectorTest6`.
Imports: `scalation.{linalgebra.VectorD, plot.Plot, random.Normal, stat.{Histogram, vectorD2StatVector}}`.

Confidence Interval

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

$$\hat{\mu} \;=\; \frac{\mathbf{1} \cdot \mathbf{y}}{m}$$

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

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

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

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

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

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

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

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

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

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

In other words,

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

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

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

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

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

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

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

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

$$\left[\hat{\mu} - \frac{z^*\sigma}{\sqrt{m}},\, \hat{\mu} + \frac{z^*\sigma}{\sqrt{m}}\right] \tag{2.28}$$

being replace by the Student t-distribution

$$\left[\hat{\mu} - \frac{t^*\hat{\sigma}}{\sqrt{m}},\, \hat{\mu} + \frac{t^*\hat{\sigma}}{\sqrt{m}}\right] \tag{2.29}$$

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.

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.

```
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?

Estimation for Discrete Outcomes/Responses

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

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

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

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

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

What value of p maximizes the function $L(p)$? The function $L(p)$ is called the *Likelihood function* and it is used in Maximum Likelihood Estimation (MLE) [16].

2.1.17 Exercises

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

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

2. Prove Bayes Theorem.

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

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

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

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

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

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

```
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 you initial choice or switch to the remaining door. What are the probabilities of winning if you (a) stay with your initial choice, or (b) switch to the other door? Finish the code below to validate your results.

```
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
```

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

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

2.1.18 Further Reading

1. Probability and Mathematical Statistics [22]

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 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.,

$$y_0 = x_{00}b_0 + x_{01}b_1$$
$$y_1 = x_{10}b_0 + x_{11}b_1$$

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?

$$100 = 1b_0 + 1b_1$$
$$1600 = 10b_0 + 20b_1$$

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

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

$$\mathbf{y} \;=\; X\mathbf{b} \tag{2.30}$$

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

2.2.2 Matrix Inversion

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

$$\mathbf{b} \;=\; X^{-1}\mathbf{y} \tag{2.31}$$

Multiplying matrix X and its inverse X^{-1}, $X^{-1}X$ results in an n-by-n identity matrix $I_n = [I_{i=j}]$, 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 X into lower L and upper U triangular matrices.

$$X = LU \tag{2.32}$$

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

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

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

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

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

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

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

Vector notation is used in this 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 `linalgebra` and `linalgebra_gen` packages. A commonly used operation is the dot (inner) product, $\mathbf{x} \cdot \mathbf{y}$, 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

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

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

2.2.4 Vector Operations

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

$$\mathbf{x} \cdot \mathbf{y} \;=\; \sum_{i=0}^{n-1} x_i y_i \;=\; 1.1547 \tag{2.33}$$

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

$$\|\mathbf{x}\| \;=\; \sqrt{\sum_{i=0}^{n-1} x_i^2} \;=\; 1 \tag{2.34}$$

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

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

Thus, the cosine of θ is,

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

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

In general there are ℓ_p norms. The two that are used here are the ℓ_2 norm $\|\mathbf{x}\| \;=\; \|\mathbf{x}\|_2$ (Euclidean distance) and the ℓ_1 norm $\|\mathbf{x}\|_1$ (Manhattan distance).

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

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

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

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

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

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

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

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

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

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 Vector

Consider the following function $f : \mathbb{R}^2 \to \mathbb{R}$ 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 $f : \mathbb{R}^n \to \mathbb{R}$ of vector $\mathbf{x} \in \mathbb{R}^n$ is

$$\nabla f(\mathbf{x}) \;=\; \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \;=\; \left[\frac{\partial f}{\partial x_0}, \dots, \frac{\partial f}{\partial x_{n-1}}\right] \tag{2.35}$$

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

$$\nabla(f(\mathbf{x}) \cdot g(\mathbf{x})) \;=\; \nabla f(\mathbf{x}) \cdot g(\mathbf{x}) + f(\mathbf{x}) \cdot \nabla g(\mathbf{x}) \tag{2.36}$$

Jacobian Matrix

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

$$\mathbb{J}\,\mathbf{f}(\mathbf{x}) \;=\; \left[\frac{\partial f_i}{\partial x_j}\right]_{0 \le i < m, 0 \le j < n} \;= \tag{2.37}$$

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

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

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

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

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

Taking the partial derivatives gives the following Jacobian matrix.

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

Hessian Matrix

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

$$\mathbb{H}\,f(\mathbf{x}) \;=\; \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]_{0 \le i < n, 0 \le j < n} \tag{2.38}$$

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

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

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

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

Taking the second partial derivatives gives the following Hessian matrix.

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

Vector Operations in ScalaTion

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 2.1: Vector Arithmetic Operations

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

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*

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m \tag{2.39}$$

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

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

to produce matrix A.

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

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

$$A\mathbf{u} \;=\; \mathbf{v} \tag{2.40}$$

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.
Note that in Scala, `i to k` is a `Range` that includes `k`, while `i until k` does not include `k`. Common operations on matrices are supported as well.

A	=	`a`	=	matrix
A	=	`a()`	=	underlying array
$\mathbf{a_i}$	=	`a(i)`	=	row vector i
$\mathbf{a_{:j}}$	=	`a.col(j)`	=	column vector j
a_{ij}	=	`a(i, j)`	=	the element at row i and column j
$A_{i:k,j:l}$	=	`a(i to k, j to l)`	=	row and column matrix slice

Matrix Addition and Subtraction

Matrix addition `val c = a + b`

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

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

Matrix Multiplication

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

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

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 (`val t = a.t`), simply exchanges the roles of rows and columns.

```
def t: MatrixD =
{
    val c = new MatrixD (dim2, dim1)
    for (j <- range1) {
        val v_j = v(j)
        for (i <- range2) c.v(i)(j) = v_j(i)
    } // for
    c
} // t
```

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 $A^tB = A \cdot B =$ `a.t * b = a mdot b`.

```
    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
        c
    } // mdot
```

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 the techniques shown in table 2.2
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

Table 2.2: Matrix Factorization Techniques

Fractorization	Factors	Factor 1	Factor 2	Class
LU	$A = LU$	lower left triangular	upper right triangular	`Fac_LU`
Cholesky	$A = LL^t$	lower left triangular	its transpose	`Fac_Cholesky`
QR	$A = QR$	orthogonal	upper right triangular	`Fac_QR_H`
SVD	$A = U\Sigma V^t$	orthogonal	diagonal, orthogonal	`SVD`

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 `linalgebra` package provides several traits and classes implementing multiple types of vectors and matrices as shown in Table 2.3 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 2.3: Types of Vectors and Matrices: Implementing Classes

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

The suffix 'D' indicates the base element type is `Double`. There are also implementations for `Complex 'C'`, `Int 'I'`, `Long 'L'`, `Rational 'Q'`, `Real 'R'`, `StrNum 'S'`, and `TimeNum 'T'`. There are also generic implementations in `linalgebra_gen`, but they tend to run more slowly.

ScalaTion supports many operations involving matrices and vectors, including the following show in Table 2.4.

Table 2.4: Types of Vector and Matrix Products

Product	Method	Example	in Math
vector dot	def dot (y: VectoD): Double	x dot y	$\mathbf{x} \cdot \mathbf{y}$
vector elementwise	def * (y: VectoD): VectoD	x * y	$\mathbf{x}\,\mathbf{y}$
vector outer	def outer (y: VectoD): MatriD	x outer y	$\mathbf{x} \otimes \mathbf{y}$
matrix mult	def * (y: MatriD): MatriD	x * y	$X\,Y$
matrix dot	def dot (y: MatriD): VectoD	x dot y	$X \cdot Y$
matrix mdot	def mdot (y: MatriD): MatriD	x mdot y	$X^t\,Y$
matrix vector	def * (y: VectoD): VectoD	x * y	$X\,\mathbf{y}$
matrix vector	def ** (y: VectoD): MatriD	x ** y	$X\,\text{diag}(\mathbf{y})$

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

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

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

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

2.2.11 Further Reading

1. Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares [3]
2. Matrix Computations [10]

2.3 Notational Conventions

With respect to random variables, vectors and matrices, the following notational conventions shown in Table 2.5 will be used in this book.

Table 2.5: Notational Conventions Followed

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

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

```
type FunctionS2S  = Double  => Double      // function of a scalar - Double
type FunctionV2S  = VectorD => Double      // function of a vector - VectorD
type FunctionV_2S = VectoD  => Double      // function of a vector - VectoD
type FunctionV2V  = VectorD => VectorD     // vector-valued function of a vector - VectorD
type FunctionV_2V = VectoD  => VectoD      // vector-valued function of a vector - VectoD
type FunctionM2M  = MatrixD => MatrixD     // matrix-valued function of a matrix - MatrixD
type FunctionM_2M = MatriD  => MatriD      // matrix-valued function of a matrix - MatriD
```

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'`. Mathemtically, the scalar-valued functions are denoted by a symbol, e.g., f.

$$\begin{aligned} &\texttt{S2S function} \quad f : \mathbb{R} \to \mathbb{R} \\ &\texttt{V2S function} \quad f : \mathbb{R}^n \to \mathbb{R} \\ &\texttt{V\_2S function} \quad f : \mathbb{R}^n \to \mathbb{R} \end{aligned}$$

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

$$\begin{aligned} &\texttt{V2V function} \quad \mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m \\ &\texttt{V\_2V function} \quad \mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m \\ &\texttt{M2M function} \quad \mathbf{f} : \mathbb{R}^{m \times n} \to \mathbb{R}^{p \times q} \\ &\texttt{M\_2M function} \quad \mathbf{f} : \mathbb{R}^{m \times n} \to \mathbb{R}^{p \times q} \end{aligned}$$

2.4 Model

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

$$y \;=\; f(\mathbf{x}; \mathbf{b}) \;+\; \epsilon \tag{2.41}$$

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

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

$$mse \;=\; \frac{1}{m}\,||\mathbf{y} - \hat{\mathbf{y}}||^2 \tag{2.42}$$

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

After a model is trained, its Quality of Fit (QoF) should be evaluated. One way to perform the evaluation is to train the model on the full dataset and test as well on the full dataset. For complex models with many parameters, *overfitting* will likely occur. Then its excellent evaluation is unlikely to be reproduced when the model is applied in the real-world. To avoid overly optimistic evaluations due to overfitting, it is common to divide a dataset $(X, \mathbf{y})$ into a training dataset and testing dataset where training is conducted on the training dataset $(X_r, \mathbf{y_r})$ and evaluation is done on the test dataset $(X_e, \mathbf{y_e})$. The conventions used in this book for the full, training and test datasets are shown in Table 2.6

Table 2.6: Convention for Datasets

Math Symbol	Code	Description
X	`x`	full data/input matrix
X_r	`x_r`	training data/input matrix (maybe full)
X_e	`x_e`	test data/input matrix (maybe full)
$\mathbf{y}$	`y`	full response/output vector
$\mathbf{y_r}$	`y_r`	training response/output vector (maybe full)
$\mathbf{y_e}$	`y_e`	test response/output vector (maybe full)

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

In ScalaTion, the `Model` trait severs as base trait for all the modeling techniques in the `analytics` package and its subpackages `classifier`, `clusterer`, `fda`, `forecaster`, and `recommeneder`.

`Model` **Trait**

Trait Methods:

```
trait Model extends Error

def train (x_r: MatriD, y_r: VectoD): Model
def eval (x_e: MatriD, y_e: VectoD): Model
def hparameter: HyperParameter
def parameter: VectoD
def report: String
```

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

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

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

Implementations of the `train` method take a training data/input matrix `x_r` and a training response/output vector `y_r` and optimize the parameter vector `b` to, for example, mimimize error or maximize likelihood. Implementations of the `eval` method take a test data/input matrix `x_e` and the corresponding test response/output vector `y_e` to compute errors and evaluate the Quality of Fit (QoF). Note that with cross-validation (to be explained later), there will be multiple training and test datasets created from one full dataset. Implementations of the `hparameter` method simply return the hyper-parameter vector `hparam`, while implementations of the `parameter` method simply return the optimized parameter vector `b`. (The `fname_` and `technique` parameters for `Regression` are the feature names and the solution/optimization technique used to estimate the parameter vector, respectively.)

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

3.2 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.

3.2.1 Relation Creation

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

```
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,
“ISDDDD”)

val mroad = Relation (“road”, Seq (“roadID”, “rdName”, “lanes”, “lat1”, “long1”, “lat2”, “long2”),
Seq (), 0, “ISIDDDD”)

val traffic = Relation (“traffic”, Seq (“time”, “sensorID”, “count”, “speed”), Seq (), Seq (0, 1), “TIID”)

val wsensor = Relation (“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.

3.2.2 Relation Population

There are several ways to populate the `Relation`s. A row/tuple can be added one at a time using `def add (tuple: Row)`. Population may also occur during relation construction (via a constructor or apply method). There are factory apply functions that take a file or URL as input.

For example to populate the `sensor` relation with information about Austin, Texas' traffic sensors stored in the file `austin_traffic_sensors.csv` the following line of code may be used.

```
val sensor = Relation (“sensor”, “austin_traffic_sensors.csv”)
```

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

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 $r \cup s$ compute the union of relations r and s). Note, the extended projection operator *eproject* (Π) provides a convenient mechanism for applying aggregate functions. It is often called after the *groupby* operator, in which case multiple rows will be returned. Multiple columns may be specified in *eproject* as well. There are also several varieties of *join* operators. As an alternative to using the Unicode symbol when they are Greek letters, the letter may be written out in English (*pi*, *sigma*, *rho*, *gamma*, *epi*, *omega*, *zeta*, *unzeta*).

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

Operator	**Unicode**	**Example**	**Return**
select	σ	$r.\sigma$ ("rdName", _ == "I285")	rows of r where rdName == "I285"
project	π	$r.\pi$ ("rdName", "lat1", "long1")	the rdName, lat1, and long1 columns of r
union	$\cup$	$r \cup q$	rows that are in r or q
minus	-	$r - q$	rows that are in r but not q
product	$\times$	$r \times t$	concatenation of each row of r with those of t
rename	ρ	$r.\rho$("r2")	a copy of r with new name $r2$
join	$\bowtie$	$r \bowtie s$	rows in natural join of r and s
intersect	$\cap$	$r \cap q$	rows that are in r and q
groupby	γ	$t.\gamma$ ("sensorId")	rows of t grouped by sensorId
eproject	Π	$t.\Pi$ (avg, "acount", "count")("sensorId")	the average of the count column of t
orderBy	ω	$t.\omega$ ("sensorId")	rows of t ordered by sensorId
compress	ζ	$t.\zeta$ ("count")	compress the count column of t
uncompress	Z	$t.Z$ ("count")	uncompress the count column of t

The extended projection operator *eproject* applies aggregate operators on aggregation columns (first arguments) and regular project on the other columns (second arguments). Typically it is called after the *groupby* operator.

$t.\gamma$ ("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.

$$\begin{aligned} & r \bowtie (\text{“}roadId\text{”}, \text{“}on\text{”}, s) && \text{equi} - \text{join} \\ & r \bowtie [\text{Int}](s, (\text{“}roadId\text{”}, \text{“}on\text{”}, \_ == \_)) && \text{theta join} \\ & t \ltimes (\text{“}time\text{”}, \text{“}time\text{”}, w) && \text{left outer join} \\ & t \rtimes (\text{“}time\text{”}, \text{“}time\text{”}, w) && \text{right outer join} \end{aligned}$$

3.2.3 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).

```
val austin = latLong2UTMxy (LatitudeLongitude (30.266667, -97.733333))
val alat = (austin._1 - 100000, austin._1 + 100000)
val along = (austin._2 - 100000, austin._2 + 100000)
traffic ⋈ sensor.σ [Double] (“latitude”, _ ∈ alat).σ [Double] (“longitude” _ ∈ along)
```

3.3 SQL-Like API

The SQL-Like API in ScalaTion provides many of the language constucts of SQL in a functional style.

3.3.1 Relation Creation

A `RelationSQL` object is created by invoking a constructor or factory apply function. For example, the following six `RelationSQL`s may be useful in a traffic forecasting study.

```
val sensor  = RelationSQL ("sensor", Seq ("sensorID", "model", "latitude", "longitude", "on"),
              null, 0, "ISDDI")
val road    = RelationSQL ("road", Seq ("roadID", "rdName", "lat1", "long1", "lat2", "long2"),
              null, 0, "ISDDDD")
val mroad   = RelationSQL ("road", Seq ("roadID", "rdName", "lanes", "lat1", "long1", "lat2", "long2"),
              null, 0, "ISIDDDD")
val traffic = RelationSQL ("traffic", Seq ("time", "sensorID", "count", "speed"),
              null, 0, "TIID")
val wsensor = RelationSQL ("sensor", Seq ("sensorID", "model", "latitude", "longitude"),
              null, 0, "ISDD")
val weather = RelationSQL ("weather", Seq ("time", "sensorID", "precipitation", "wind"),
              null, 0, "TIID")
```

`RelationSQL` Class

Class Methods:

```
@param name     the name of the relation
@param colName  the names of columns
@param col      the Scala Vector of columns making up the columnar relation
@param key      the column number for the primary key (< 0 => no primary key)
@param domain   an optional string indicating domains for columns (e.g., 'SD' = 'StrNum', 'Double')
@param fKeys    an optional sequence of foreign keys - Seq (column name, ref table name, ref column position)

class RelationSQL (name: String, colName: Seq [String], col: Vector [Vec],
                   key: Int = 0, domain: String = null, fKeys: Seq [(String, String, Int)] = null)
      extends Tabular with Serializable



def repr: Relation = r
def this (r: Relation) = this (r.name, r.colName, r.col, r.key, r.domain, r.fKeys)
def select (cName: String*): RelationSQL =
def join (r2: RelationSQL): RelationSQL =
def join (cName1: String, cName2: String, r2: RelationSQL): RelationSQL =
def join (cName1: Seq [String], cName2: Seq [String], r2: RelationSQL): RelationSQL =
def where [T: ClassTag] (cName: String, p: T => Boolean): RelationSQL =
def where2 [T: ClassTag] (p: Predicate [T]*): RelationSQL =
def groupBy (cName: String*): RelationSQL =
def orderBy (cName: String*): RelationSQL =
def reverseOrderBy (cName: String*): RelationSQL =
def union (r2: RelationSQL): RelationSQL =
def intersect (r2: RelationSQL): RelationSQL =
def intersect2 (r2: RelationSQL): RelationSQL =
def minus (r2: RelationSQL): RelationSQL =
def stack (cName1: String, cName2: String): RelationSQL =
def insert (rows: Row*)
def materialize ()
def exists: Boolean = r.exists
def toMatriD (colPos: Seq [Int], kind: MatrixKind = DENSE): MatriD =
def toMatriDD (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriD, VectorD) =
def toMatriDI (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriD, VectorI) =
def toMatriI (colPos: Seq [Int], kind: MatrixKind = DENSE): MatriI =
def toMatriI2 (colPos: Seq [Int] = null, kind: MatrixKind = DENSE): MatriI =
def toMatriII (colPos: Seq [Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriI, VectorI) =
def toVectorC (colPos: Int = 0): VectorC = r.toVectorC (colPos)
def toVectorC (colName: String): VectorC = r.toVectorC (colName)
def toVectorD (colPos: Int = 0): VectorD = r.toVectorD (colPos)
def toVectorD (colName: String): VectorD = r.toVectorD (colName)
def toVectorI (colPos: Int = 0): VectorI = r.toVectorI (colPos)
def toVectorI (colName: String): VectorI = r.toVectorI (colName)
def toVectorL (colPos: Int = 0): VectorL = r.toVectorL (colPos)
def toVectorL (colName: String): VectorL = r.toVectorL (colName)
def toVectorQ (colPos: Int = 0): VectorQ = r.toVectorQ (colPos)
def toVectorQ (colName: String): VectorQ = r.toVectorQ (colName)
def toVectorR (colPos: Int = 0): VectorR = r.toVectorR (colPos)
def toVectorR (colName: String): VectorR = r.toVectorR (colName)
def toVectorS (colPos: Int = 0): VectorS = r.toVectorS (colPos)
def toVectorS (colName: String): VectorS = r.toVectorS (colName)
def toVectorT (colPos: Int = 0): VectorT = r.toVectorT (colPos)
def toVectorT (colName: String): VectorT = r.toVectorT (colName)
def show (limit: Int = Int.MaxValue) { r.show (limit) }
def save () { r.save () }
```

3.4 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.

3.4.1 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.

3.4.2 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.linalgebra.Converter` object.

3.4.3 Identify Missing Values

Missing Values are common is real datasets. For some datasets, a question mark character '?' is used to indicate that a value is missing. In Comma Separated Value (CSV) files, repeated commas may indicate missing values, e.g., 10.1, 11.2,,,9.8. If zero or negative numbers are not valid for the application, these may be used to indicate missing values.

3.4.4 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`.

DistanceOutlier.rmOutlier (traffic.column ("speed"))

3.4.5 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. Imputation implementations are based on the `Imputation` trait in the `scalation.modeling` package.

`Imputation` **Trait**

Trait Methods:

```
trait Imputation

def setMissVal (missVal_ : Double) { missVal = missVal_ }
def setDist (dist_ : Int) { dist = dist_ }
def imputeAt (x: VectoD, i: Int): Double
def impute (x: VectoD, i: Int = 0): (Int, Double) = findMissing (x, i)
def imputeAll (x: VectoD): VectoD =
def impute (x: MatriD): MatriD =
def imputeCol (c: Vec, i: Int = 0): (Int, Any) =
```

ScalaTion currently supports the following imputation techniques:

1. `object ImputeRegression extends Imputation`: Use `SimpleRegression` on the instance index to estimate the next missing value.

2. `object ImputeForward extends Imputation`: Use the pevious value and slope to estimate the next missing value.

3. `object ImputeBackward extends Imputation`: Use the subsequent value and slope to estimate the pevious missing value.

4. `object ImputeMean extends Imputation`: Use the filtered mean to estimate the next missing value.

5. `object ImputeMovingAvg extends Imputation`: Use the moving average of the last 'dist' values to estimate the next missing value.

6. `object ImputeNormal extends Imputation`: Use the median of three Normally distributed, based on filtered mean and variance, random values to estimate the next missing value.

7. `object ImputeNormalWin extends Imputation`: Same as `ImputeNormal` except mean and variance are recomputed over a sliding window.

3.4.6 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.

3.4.7 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.

$$\text{traffic} \ltimes (0.01)(\text{“time”}, \text{“time”}, \text{weather}) \qquad\qquad \text{approximate left outer join}$$

The second operator ...

3.4.8 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.

val mat = $\pi_{\text{“time”},\text{“count”}}$ (traffic).toMatriD

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

val model1 = Regression (mat)
val model2 = ARIMA (mat)

By default in ScalaTion the rightmost columns are the response/output variables. As many of the modeling techniques have a single response variable, it will be assumed to in the last column. There are also contructors and factory apply functions that take explicit vector and matrix parameters, e.g., a matrix of predictor variables and a response vector.

3.5 Excercises

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.

Chapter 4

Prediction

As the name predictive analytics indicates, the purpose of techniques that fall in this category is to develop models to predict outcomes. For example, the distance a golf ball travels y when hit by a driver depends on several factors or inputs $\mathbf{x}$ such as club head speed, barometric pressure, and smash factor (how square the impact is). The models can be developed using a combination of data (e.g., from experiments) and knowledge (e.g., Newton's Second Law). The modeling techniques discussed in this technical report tend to emphasize the use of data more than knowledge, while those in the simulation modeling technical report emphasize knowledge.

Abstractly, a predictive model can generally be formulated using a prediction function f as follows:

$$y \ = \ f(\mathbf{x},\ t;\ \mathbf{b}) + \epsilon \tag{4.1}$$

where

- y is an response/output scalar,
- $\mathbf{x}$ is an predictor/input vector,
- t is a scalar representing time,
- $\mathbf{b}$ is the vector of parameters of the function, and
- ϵ represents remaining residuals/errors.

Both the response y and residuals/errors ϵ are treated as random variables, while the predictor/feature variables $\mathbf{x}$ may be treated as either random or deterministic depending on context. Depending on the goals of the study as well as whether the data are the product of controlled/designed experiments, the random or deterministic view may be more suitable.

The parameters $\mathbf{b}$ can be adjusted so that the predictive model matches the available data. Note, in the definition of a function, the *arguments* appear before the ";", while the *parameters* appear after. The residuals/errors are typically additive as shown above, but may also be multiplicative. Of course, the formulation could be generalized by turning the output/response into a vector $\mathbf{y}$ and the parameters into a matrix B.

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

$$y \;=\; f(\mathbf{x};\; \mathbf{b}) + \epsilon \tag{4.2}$$

Another way to look at such models, is that we are trying to estimate the conditional expectation of y given $\mathbf{x}$.

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

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

Given a dataset (m instances of data), each instance contributes to an overall residual/error vector $\boldsymbol{\epsilon}$. One of the simpler ways to estimate the parameters $\mathbf{b}$ is to minimize the size of the residual/error vector, e.g., its Euclidean norm. The square of this norm is the sum of squared errors (sse)

$$sse \;=\; ||\boldsymbol{\epsilon}||^2 \;=\; \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} \tag{4.3}$$

This corresponds to mininizing the raw mean square error ($mse = sse/m$). See the section on Generalized Linear Models for further development along these lines.

In ScalaTion, data are passed to the `train` function to train the model/fit the parameters $\mathbf{b}$. In the case of prediction, the `predict` function is used to predict values for the scalar response y.

A key question to address is the possible functional forms that f may take, such as the importance of time, the linearity of the function, the domains for y and $\mathbf{x}$, etc. We consider several cases in the subsections below.

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 an abstract class extending the `Predictor` trait, namely `PredictorMat`. These `PredictorMat` also extend the `Fit` class to enable Quality of Fit (QoF) evaluation.

`Predictor` **Trait**

Trait Methods:

```
trait Predictor extends Model

def getX: MatriD
def getY: VectoD
def residual: VectoD
def analyze (x_r: MatriD, y_r: VectoD, x_e: MatriD, y_e: VectoD): Predictor
def predict (z: VectoD): Double
def predict (z: VectoI): Double = predict (z.toDouble)
def predict (z: MatriD): VectoD
def forwardSel (cols: Set [Int], index_q: Int): (Int, Predictor)
def corrMatrix (xx: MatriD): MatriD = corr (xx.asInstanceOf [MatrixD])
def test (modelName: String, doPlot: Boolean = true)
```

The `Predictor` trait inherits the five methods from the `Model` trait: `train`, `eval`, `hparameter`, `parameter` and `report`. In addition, it defines the nine methods shown above (eight of which are to implemented by classes extending from the `Predictor` trait). The `getX` and `getY` methods simply return the data matrix and response vector that are used internally by the modeling technique (some techniques expand the original `x` and `y` passed into the constructor). The `residual` method returns the error vector (difference between actual and predicted values). The `analyze` method trains on the training dataset and evaluates on the test dataset. The `predict` method take a data vector (e.g., a new data instance) and predict its response. An overloaded `predict` method takes a matrix as input (with each row being an instance) and makes predictions for each row. The `forwardSel` method performs forward selection by adding the next most predictive variable/feature to an existing model. The `corrMatrix` method computes the correlation matrix for the column vectors in a matrix. The `test` method analyzes, reports and plots the actual response versus the predicted 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 $\mathbf{y}$. After the parameters $\mathbf{b}$ (an n vector) have been fit/estimated, the error vector $\boldsymbol{\epsilon}$ may be calculated. The basic QoF measures involve taking either ℓ_1 (Manhattan) or ℓ_2 (Euclidean) norms of the error vector as indicated in Table 4.1.

Table 4.1: Quality of Fit

error/residual	**absolute**	**ℓ_1 norm**	**squared**	**ℓ_2 norm**
sum	sum of absolute errors	$sae = \|\|\boldsymbol{\epsilon}\|\|_1$	sum of squared errors	$sse = \|\|\boldsymbol{\epsilon}\|\|_2^2$
mean	mean absolute error	$mae^0 = sae/m$	mean squared error	$mse^0 = sse/m$
unbiased mean	mean absolute error	$mae = sae/df$	mean squared error	$mse = sse/df$

Typically, if a model has m instances/rows in the dataset and n parameters to fit, the error vector will live in an $m - n$ dimensional space (ignoring issues related to the rank the data matrix). Note, if $n = m$, there may be a unique solution for the parameter vector $\mathbf{b}$, in which case $\boldsymbol{\epsilon} = \mathbf{0}$, i.e., the error vector lives in a 0-dimensional space. The *degrees of freedom* (for error) is the dimensionality of the space that the error vector lives in, namely, $df = m - n$.

`Fit` Class

Class Methods:

```
@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 (model/regression, error)

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

def resetDF (df_update: PairD)
def mse_ : Double = mse
def diagnose (e: VectoD, yy: VectoD, yp: VectoD, w: VectoD = null, ym_ : Double = noDouble)
def ll (ms: Double = mse0, s2: Double = sig2e): Double = m * (log (_2pi) + log (s2) + ms / s2)
def fit: VectoD = VectorD (rSq, rBarSq, sst, sse, mse0, rmse, mae,
                           df._1, df._2, fStat, aic, bic, smape)
def fitLabel: Seq [String] = Fit.fitLabel
def help: String = Fit.help
def summary (b: VectoD, stdErr: VectoD, vf: VectoD, show: Boolean = false): String =
```

The `Fit` class as well as other similiar classes extend the basic `QoF` trait.

`QoF` Trait

The `QoF` trait defines methods to determine basic Quality of Fit (QoF) measures.

Class Methods:

```
trait QoF

def diagnose (e: VectoD, yy: VectoD, yp: VectoD, w: VectoD = null, ym: Double = noDouble)
def fit: VectoD
def fitLabel: Seq [String]
def help: String
def f_ (z: Double): String = "%.5f".format (z)
def fitMap: Map [String, String] =
```

For modeling, a user chooses one the of classes (directly or indirectly) extending the trait `Predictor` (e.g., `Regression`) to instantiate an object. Next the `train` method would be typically called, followed by the `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. The quality of fit measures compute by the `diagnose` method in the `Fit` class are shown below.

```
    @param e    the m-dimensional error/residual vector (yy - yp)
    @param yy   the actual response/output vector to use (test/full)
    @param yp   the predicted response/output vector (test/full)
    @param w    the weights on the instances (defaults to null)
    @param ym_  the mean of the actual response/output vector to use (training/full)

    def diagnose (e: VectoD, yy: VectoD, yp: VectoD, w: VectoD = null, ym_ : Double = noDouble)
    {
        m = yy.dim                                          // size of response vector (test/full)
        if (m < 1) flaw ("diagnose", s"no responses to evaluate m = $m")
        if (e.dim  != m) flaw ("diagnose", s"e.dim  = ${e.dim}  != yy.dim = $m")
        if (yp.dim != m) flaw ("diagnose", s"yp.dim = ${yp.dim} != yy.dim = $m")

        val ln_m = log (m)                                  // natural log of m (ln(m))
        val mu   = yy.mean                                  // mean of yy (may be zero)
        val ym   = if (ym_ == noDouble) { if (DEBUG) println ("diagnose: test mean"); mu }
                   else                 { if (DEBUG) println ("diagnose: train mean"); ym_ }

        sse = e.normSq                                      // sum of squares for error
        if (w == null) {
            sst = (yy - ym).normSq                          // 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/MLE mean squared error
        rmse   = sqrt (mse0)                                // root mean squared error (RMSE)
//      nrmse  = rmse / mu                                  // normalized RMSE
```

```
        mae    = e.norm1 / m                                    // mean absolute error
        rSq    = ssr / sst                                      // coefficient of determination

        if (df._1 <= 0 || df._2 <= 0) flaw ("diagnose", s"degrees of freedom df = $df <= 0")
        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                              // NaN => check error from 'fisherCDF'
        if (sig2e == -1.0) sig2e = e.pvariance
        aic    = ll() + 2 * (df._1 + 1)                         // Akaike Information Criterion
        bic    = aic + (df._1 + 1) * (ln_m - 2)                 // Bayesian Information Criterion
        smape  = (e.abs / (yy.abs + yp.abs)).sum / m            // symmetric mean abs. percentage error / 100
//      nmae   = mae / mu                                       // normalized MAE (MAD/Mean Ratio)
    } // diagnose
```

Note, `~^` is the exponentiation operator provided in ScalaTion, where the first character is `~` to give the operator higher precedence than multiplication (*).
The sum of squares total (sst) measures the variability of the response y,

$$sst \;=\; ||\mathbf{y} - \mu_{\mathbf{y}}||^2 \;=\; \mathbf{y} \cdot \mathbf{y} - m\,\mu_{\mathbf{y}}^2 \;=\; \mathbf{y} \cdot \mathbf{y} - \frac{1}{m}\big[\sum y_i\big]^2 \tag{4.4}$$

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.

$$R^2 \;=\; \frac{ssr}{sst} \;\leq\; 1 \tag{4.5}$$

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: MatriD, protected val y: VectoD,
                             protected var fname: Strings, hparam: HyperParameter)
         extends Fit (y, x.dim2, (x.dim2 - 1, x.dim1 - x.dim2)) with Predictor
         // if not using an intercept df = (x.dim2, x.dim1-x.dim2)
         // correct by calling 'resetDF' method from `Fit`

def getX: MatriD = x
def getY: VectoD = y
def train (x_r: MatriD = x, y_r: VectoD = y): PredictorMat
def train2 (x_r: MatriD = x, y_r: VectoD = y): PredictorMat =
def eval (x_e: MatriD = x, y_e: VectoD = y): PredictorMat =
def eval (ym: Double, y_e: VectoD, yp: VectoD): PredictorMat = ???
def analyze (x_r: MatriD = x, y_r: VectoD = y,
def hparameter: HyperParameter = hparam
def parameter: VectoD = b
def report: String =
def summary: String =
def residual: VectoD = e
def predict (z: VectoD): Double = b dot z
def predict (z: MatriD = x): VectoD = VectorD (for (i <- z.range1) yield predict (z(i)))
def buildModel (x_cols: MatriD): PredictorMat
def forwardSel (cols: Set [Int], index_q: Int = index_rSqBar): (Int, PredictorMat) =
def forwardSelAll (index_q: Int = index_rSqBar, cross: Boolean = true): (Set [Int], MatriD) =
def backwardElim (cols: Set [Int], index_q: Int = index_rSqBar, first: Int = 1): (Int, PredictorMat) =
def backwardElimAll (index_q: Int = index_rSqBar, first: Int = 1, cross: Boolean = true):
    (Set [Int], MatriD) =
override def corrMatrix (xx: MatriD = x): MatriD = corr (x.asInstanceOf [MatrixD])
def vif (skip: Int = 1): VectoD =
def crossValidate (k: Int = 10, rando: Boolean = true): Array [Statistic] =
```

A brief discription of all these methods follows:

1. The `getX` method returns the actual data/input matrix used by the model. Some complex models expanded the columns in an initial data matrix to add for example quadratic or cross terms.

2. The `getY` method returns the actual response/output vector used by the model. Some complex models transform the initial response vector.

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

4. 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.

5. 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.

6. The `parameter` method returns the estimated parameters for the model.

7. 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).

8. The `report` method return a basic report on the model's hyper-parameters, parameters and overall quality of fit.

9. 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.

10. The `residual` method returns the difference between the actual and predicted response vectors. The residual indicates what the model has left to explain/account for (e.g., an ideal model will only leave the noise in the data unaccounted for).

11. The two `predict` methods use the paramater 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.

12. The `buildModel` method build a sub-model that is restricted to given columns of the data matrix.

13. The `forwardSel` method is used for forward selection of variables/features for inclusion into the model. At each step the variable that increases the predictive power of the model the most is selected. This method is called repeatedly in `forwardSelAll` to find "best" combination of features. Not guaranteed to find the optimal combination.

14. The `bakwardElim` method is used for backward elimination of variables/features from the model. At each step the variable that contributes the least to the predictor power of the model is eliminated. This method is called repeatedly in `bakwardElimAll` to find "best" combination of features. Not guaranteed to find the optimal combination.

15. The `corrMatrix` method returns the correlation matrix for the data/input matrix. High correlation between column vectors in the matrix may indicate *collinearity.*

16. The `vif` method returns the Variance Inflation Factors (VIFs) for each of the columns in the data/input matrix. High VIF scores may indicate *multi-collinearity.*

17. The `crossValidate` method implements 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.

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 $\mathbf{x}$ are not relevant, not collected in a useful range or the relationship is too complex for the modeling techniques you have applied.

4.2.1 Model Equation

Ignoring the predictor variables $\mathbf{x}$ gives the following simple model equation.

$$\boxed{y \;=\; b_0 + \epsilon} \tag{4.6}$$

It is just a constant term plus the error/residual term.

4.2.2 Training

The training dataset in this case case only consists of a response vector $\mathbf{y}$. The optimal solution for the parameter vector $\mathbf{b}$ is simple to compute. It can be shown that (see exercises) the optimal value for the parameter is the mean of the response vector.

$$\boxed{b_0 \;=\; \mu_{\mathbf{y}}} \tag{4.7}$$

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

```
def train (x_null: MatriD, y_r: VectoD): NullModel =
{
    b = VectorD (y_r.mean)                            // parameter vector [b0]
    this
} // train
```

After values for the model parameters are determined, it it important to assess the Quality of Fit (QoF). The `eval` method will compute the residual/error vector $\boldsymbol{\epsilon}$ and then call the `diagnose` method.

```
def eval (x_null: MatriD, y_e: VectoD): NullModel =
{
    val yp = VectorD.fill (y_e.dim)(b(0))             // y predicted for (test/full)
    e = y_e - yp                                      // compute residual/error vector e
    diagnose (e, y_e, yp)                             // compute diagnostics
} // 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.

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

`NullModel` **Class**

Class Methods:

```
@param y  the response/output vector

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

def getX: MatriD = null
def getY: VectoD = y
def train (x_null: MatriD, y_r: VectoD): NullModel =
def eval (x_null: MatriD, y_e: VectoD): NullModel =
def analyze (x_r: MatriD = null, y_r: VectoD = y,
def hparameter: HyperParameter = null
def parameter: VectoD = b
def report: String =
def residual: VectoD = e
def predict (z: VectoD): Double = b(0)
def predict (z: MatriD = null): VectoD = VectorD.fill (y.dim)(b(0))
```

4.2.3 Exercises

1. Determine the value for the parameter b_0 that minmizes the sum of squared errors $sse = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}$.

2. Let the response vector $\mathbf{y}$ be

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

   and execute the `NullModel`.

   For context, assume the corresponding predictor vector $\mathbf{y}$ is

   ```
   val x = VectorD (1, 2, 3, 4)
   ```

   Draw an xy plot of the data points. Give the value for the parameter vector $\mathbf{b}$. 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.analyze ()                                       // 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 is 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., $\mathbf{x} = [x_0]$ and there is only a single parameter that is the coefficient for x_0 in the model.

4.3.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b} = [b_0]$ in the following model/regression equation,

$$\boxed{y \;=\; b_0 x_0 + \epsilon} \tag{4.8}$$

where ϵ represents the residuals/errors (the part not explained by the model).

4.3.2 Training

A dateset may be collected for providing an estimate for parameter b_0. Given m data points, stored in an m-dimensional vector $\mathbf{x_0}$ and m response values, stored in an m-dimensional vector $\mathbf{y}$, we may obtain the following vector equation.

$$\mathbf{y} \;=\; b_0\mathbf{x_0} + \boldsymbol{\epsilon} \tag{4.9}$$

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

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

Since $\boldsymbol{\epsilon} \;=\; \mathbf{y} - b_0\mathbf{x_0}$, we may solve the following optimization problem:

$$\min_{b_0} \|\mathbf{y} \;-\; b_0\mathbf{x_0}\|$$

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

$$(\mathbf{y} \;-\; b_0\mathbf{x_0}) \cdot (\mathbf{y} \;-\; b_0\mathbf{x_0})$$

4.3.3 Optimization - Derviative

A function can be optimized using Calculus by taking the first derivative and setting it equal to zero. If the second derivative is positive (negative) it will be minimal (maximal). Taking the derivative w.r.t. b_0, $\dfrac{d}{db_0}$, using the derivative product rule (for dot products)

$$(\mathbf{f}\cdot\mathbf{g})' \;=\; \mathbf{f}'\cdot\mathbf{g} + \mathbf{f}\cdot\mathbf{g}' \tag{4.10}$$

and setting it equal to zero yields the following equation.

$$\frac{d\,sse}{db_0} \;=\; -\,2\mathbf{x_0}\cdot(\mathbf{y} \;-\; b_0\mathbf{x_0}) = 0$$

Therefore, the optimal value for the parameter b_0 is

$$\boxed{b_0 \;=\; \frac{\mathbf{x_0}\cdot\mathbf{y}}{\mathbf{x_0}\cdot\mathbf{x_0}}} \tag{4.11}$$

4.3.4 Example Calculation

Consider the following data points $\{(1,1),(2,3),(3,3),(3,4)\}$ and solve for the parameter (slope) b_0.

$$b_0 \;=\; \frac{[1,2,3,4]\cdot[1,3,3,4]}{[1,2,3,4]\cdot[1,2,3,4]} \;=\; \frac{32}{30} \;=\; \frac{16}{15}$$

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

$$\hat{\mathbf{y}} \;=\; \hat{\mathbf{y}} \;=\; predict(\mathbf{x_0}) \;=\; b_0\mathbf{x_0} \;=\; [1.067, 2.133, 3.200, 4.267]$$

Therefore, the error/residual vector is

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

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
@param hparam  the hyper-parameters (currently has none)

class SimplerRegression (x: MatriD, y: VectoD, fname_ : Strings = null)
                         hparam: HyperParameter = null
      extends PredictorMat (x, y, fname_, hparam) with NoFeatureSelectionMat

def train (x_r: MatriD, y_r: VectoD): SimplerRegression =
override def vif (skip: Int = 1): VectoD = VectorD (1.0)       // not a problem for this model
```

4.3.5 Exercises

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

```
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` for the directory structure. Copy the `scalation_mathstat` jar file `scalation_mathstat_2.12-1.6.jar` into your `lib` directory. Create a file called `SimplerRegression_exer_1.scala` in the `src/main/scala` directory. In the project's base ditrectory, type `sbt`. Within `sbt` type `compile` and then `run`.

2. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \le i < m\}$ and draw the line that is nearest to these points. What is the slope of this line. Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimplerRegression` class to obtain the $\mathbf{b} = [b_0]$ vector.

```
    // 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.analyze ()                                    // train and evaluate
    println (rg.report)

    val yp = rg.predict ()
    new Plot (x.col(0), y, yp, lines = true)         // black for y and red for yp
```

An alternative to using the above constructor `new SimplerRegression` is to use a factory function `SimplerRegression`. Substitute in the following lines of code to do this.

```
    val x = VectorD (1, 2, 3, 4)
    val rg = SimplerRegression (x, y, drp._1, drp._2)
    new Plot (x, y, yp, lines = true)
```

Note, `drp` stands for default remaining parameters, giving `null` values for `fname_` and `hparam`.

```
   val drp = (null, null)
```

3. Redo the last exercise using a spreadsheet by making columns for each vector: $\mathbf{x}_0$, $\mathbf{y}$, $\hat{\mathbf{y}}$, $\boldsymbol{\epsilon}$, and $\boldsymbol{\epsilon}^2$. 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 $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \leq i < m\}$ and draw the line that is nearest to these points and intersects the origin $[0, 0]$. What is the slope of this line? Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimplerRegression` class to obtain the $\mathbf{b} = [b_0]$ vector.

```
// 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.analyze ()                                    // train and evaluate
println (rg.report)

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 `rSq`)? Is this value expected? What does it say about this model? Try using different columns for the predictor variable.

6. Compute the second derivative w.r.t. b_0, $\dfrac{d^2 sse}{db_0^2}$. Under what conditions will it be positive?

4.4 Simple Regression

The `SimpleRegression` class supports simple linear regression. In this case, the predictor vector $\mathbf{x}$ consists of the constant one and a single variable x_1, i.e., $[1, x_1]$, so there are now two parameters $\mathbf{b} = [b_0, b_1]$ in the model.

4.4.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model,regression equation,

$$\boxed{y \ = \ \mathbf{b} \cdot \mathbf{x} + \epsilon \ = \ [b_0, b_1] \cdot [1, x_1] + \epsilon \ = \ b_0 + b_1 x_1 + \epsilon} \tag{4.12}$$

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

4.4.2 Training

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 $\mathbf{y}$, solve the following optimization problem,

$$\min_{\mathbf{b}} \|\boldsymbol{\epsilon}\|$$

Substituting $\boldsymbol{\epsilon} = \mathbf{y} - X\mathbf{b}$ yields

$$\min_{\mathbf{b}} \|\mathbf{y} - X\mathbf{b}\|$$

$$\min_{[b_0, b_1]} \|\mathbf{y} - [\mathbf{1}\, \mathbf{x_1}][\begin{smallmatrix} b_0 \\ b_1 \end{smallmatrix}]\|$$

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

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

$$(\mathbf{y} - (b_0 \mathbf{1} + b_1 \mathbf{x_1})) \cdot (\mathbf{y} - (b_0 \mathbf{1} + b_1 \mathbf{x_1}))$$

Since $\mathbf{x_0}$ is just $\mathbf{1}$, for simplicity we drop the subscript on $\mathbf{x_1}$.

$$(\mathbf{y} - (b_0 \mathbf{1} + b_1 \mathbf{x})) \cdot (\mathbf{y} - (b_0 \mathbf{1} + b_1 \mathbf{x}))$$

4.4.3 Optimization - Gradient

A function of several variables can be optimized using Vector Calculus by setting its gradient equal to zero and and solving the resulting system of equations. When the system of equations are linear, matrix factorization may be used, otherwise techniques from Nonlinear Optimization may be needed.

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

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

Partial Derivative w.r.t. b_0

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

$$\begin{gathered}
-2\mathbf{1} \cdot (\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})) = 0 \\
\mathbf{1} \cdot \mathbf{y} \,-\, \mathbf{1} \cdot (b_0\mathbf{1} + b_1\mathbf{x}) = 0 \\
b_0\mathbf{1} \cdot \mathbf{1} = \mathbf{1} \cdot \mathbf{y} \,-\, b_1\mathbf{1} \cdot \mathbf{x}
\end{gathered}$$

Since $\mathbf{1} \cdot \mathbf{1} = m$, b_0 may be expressed as

$$b_0 \;=\; \frac{\mathbf{1} \cdot \mathbf{y} \,-\, b_1\mathbf{1} \cdot \mathbf{x}}{m} \tag{4.13}$$

Partial Derivative w.r.t. b_1

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

$$\begin{gathered}
-2\mathbf{x} \cdot (\mathbf{y} \,-\, (b_0\mathbf{1} + b_1\mathbf{x})) = 0 \\
\mathbf{x} \cdot \mathbf{y} \,-\, \mathbf{x} \cdot (b_0\mathbf{1} + b_1\mathbf{x}) = 0 \\
b_0\mathbf{1} \cdot \mathbf{x} + b_1\mathbf{x} \cdot \mathbf{x} \;=\; \mathbf{x} \cdot \mathbf{y}
\end{gathered}$$

Multiplying by both sides by m produces

$$m\, b_0\mathbf{1} \cdot \mathbf{x} + m\, b_1\mathbf{x} \cdot \mathbf{x} \;=\; m\,\mathbf{x} \cdot \mathbf{y}$$

Substituting for $m\, b_0 = \mathbf{1} \cdot \mathbf{y} \,-\, b_1\mathbf{1} \cdot \mathbf{x}$ yields

$$\begin{gathered}
[\mathbf{1} \cdot \mathbf{y} \,-\, b_1\mathbf{1} \cdot \mathbf{x}]\mathbf{1} \cdot \mathbf{x} \,+\, m\, b_1\mathbf{x} \cdot \mathbf{x} \;=\; m\,\mathbf{x} \cdot \mathbf{y} \\
b_1[m\,\mathbf{x} \cdot \mathbf{x} - (\mathbf{1} \cdot \mathbf{x})^2] \;=\; m\,\mathbf{x} \cdot \mathbf{y} - (\mathbf{1} \cdot \mathbf{x})(\mathbf{1} \cdot \mathbf{y})
\end{gathered}$$

Solving for b_1 gives

$$b_1 \;=\; \frac{m\,\mathbf{x} \cdot \mathbf{y} - (\mathbf{1} \cdot \mathbf{x})(\mathbf{1} \cdot \mathbf{y})}{m\,\mathbf{x} \cdot \mathbf{x} - (\mathbf{1} \cdot \mathbf{x})^2} \tag{4.14}$$

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

4.4.4 Example Calculation

Consider again the problem from the last section where the data points are $\{(1,1),(2,3),(3,3),(3,4)\}$ and solve for the two parameters, (intercept) b_0 and (slope) b_1.

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

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

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 $\mu_{\mathbf{x}}$ and $\mu_{\mathbf{y}}$, it can be shown that the expression for b_0 shortens to

$$\boxed{b_0 \;=\; \mu_{\mathbf{y}} \,-\, b_1 \mu_{\mathbf{x}}} \tag{4.15}$$

  Draw a line through the following two points $[0, b_0]$ (the intercept) and $[\mu_{\mathbf{x}}, \mu_{\mathbf{y}}]$ (the center of mass). How does this line compare to the regression line.

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

$$\boxed{b_1 \;=\; \frac{\sigma_{\mathbf{x},\mathbf{y}}}{\sigma^2_{\mathbf{x}}}} \tag{4.16}$$

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

4.4.5 Exploratory Data Analysis

As discussed in Chapter 1, *Exploratory Data Analysis* (EDA) should be performed after preprocessing the dataset. Once the response variable y is selected, a null model should be created to see in a plot where the data points lie compared to the mean. The code below shows how to do this for the AutoMPG dataset.

```
import ExampleAutoMPG.{t, x, y}

banner ("Plot response y vs. row index t")
val nm = new NullModel (y)
nm.analyze ()
println (nm.report)
val yp = nm.predict ()
new Plot (t, y, yp, "EDA: y and yp (red) vs. t", lines = true)
```

Next the relationships between the predictor variable x_j (the columns in input/data matrix X) should be compared. If two of the predictor variables are highly correlated, their individual effects on the response variable y may be indistinguishable. The correlations between the predictor variable, may be seen by examining the *correlation matrix*.

```
banner ("Correlation Matrix for columns of x")
println (nm.corrMatrix (x)
```

Although Simple Regression may be too simple for many problems/datasets, it should be used in *Exploratory Data Analysis* (EDA). A simple regression model should be created for each predictor variable x_j. The data points and the best fitting line should be plotted with y on the verticle axis and x_j on the horizonal axis. The data scientist should look for patterns/tendencies of y versus x_j, such as linear, quadratic, etc. When there is no relationship, the points will appear to be randomly and unformly positioned in the plane.

```
for (j <- x.range2) {
    banner (s"Plot response y vs. predictor variable x$j")
    val xj = x.col(j)
    val rg = SimpleRegression (xj, y, drp._1, drp._2)
    rg.analyze ()
    println (rg.report)
    val yp = rg.predict ()
    new Plot (xj, y, yp, s"EDA: y and yp (red) vs. x$j", lines = true)
} // for
```

`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
@param hparam  the hyper-parameters (currently has none)

class SimpleRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
                        hparam: HyperParameter = null)
      extends PredictorMat (x, y, fname_, hparam) with NoFeatureSelectionMat

def train (x_r: MatriD, y_r: VectoD): SimpleRegression =
override def vif (skip: Int = 1): VectoD = VectorD (1.0)      // not a problem for this model
```

4.4.6 Exercises

1. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \leq i < m\}$ and draw the line that is nearest to these points (i.e., that minimize $||\boldsymbol{\epsilon}||$). Using the formulas from developed in this section, what are the intercept and slope $[b_0, b_1]$ of this line.

   Also, pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimpleRegression` class to obtain the $\mathbf{b}$ vector.

```
// 4 data points:     constant x1
val x = new MatrixD ((4, 2), 1, 1,              // x 4-by-2 matrix
                             1, 2,
                             1, 3,
                             1, 4)
val y = VectorD (1, 3, 3, 4)                    // y vector

val rg = new SimpleRegression (x, y)            // create a SimpleRegression
rg.analyze ()                                   // train and evaluate
println (rg.report)
```

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 optimzation techniques may be applied. Two simple optimization techniques are *grid search* and *gradient descent.*

   For grid search, in a spreadsheet set up a 5-by-5 grid around the optimal point for $\mathbf{b}$, found in the previous problem. Compute values for sse for each point in the grid. Plot sse versus b_0 across the optimal point. Do the same for b_1.

   For gradient descent, pick a starting point $\mathbf{b}^0$, compute the gradient ∇sse and move $-\eta\nabla sse$ from $\mathbf{b}^0$ where η is the learning rate. Repeat for a few iterations. What is happening to the value of *see*.

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

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

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

3. From the X matrix and $\mathbf{y}$ vector, plot the set of data points $\{(x_{i1}, y_i) \,|\, 0 \le i < m\}$ and draw the line that is nearest to these points. What are the intercept and slope of this line. Pass the X matrix and $\mathbf{y}$ vector as arguments to the `SimpleRegression` class to obtain the $\mathbf{b}$ vector.

```
// 5 data points:       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.analyze ()                                    // train and evaluate
println (rg.report)

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 where multiple input variables are used to predict a value for the response variable. In this case, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ...x_k]$, so the parameter vector $\mathbf{b} = [b_0, b_1, \dots, b_k]$ has the same dimension as $\mathbf{x}$.

4.5.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... + b_k x_k + \epsilon} \tag{4.17}$$

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

4.5.2 Training

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

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

The goal is to minimize the distance between $\mathbf{y}$ and its estimate $\hat{\mathbf{y}}$. i.e., minimize the norm of residual/error vector.

$$\min_{\mathbf{b}} \|\boldsymbol{\epsilon}\|$$

Substituting $\boldsymbol{\epsilon} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} \;-\; X\mathbf{b}$ yields

$$\min_{\mathbf{b}} \|\mathbf{y} \;-\; X\mathbf{b}\|$$

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

$$(\mathbf{y} \;-\; X\mathbf{b}) \cdot (\mathbf{y} \;-\; X\mathbf{b})$$

$$(\mathbf{y} \;-\; X\mathbf{b})^t (\mathbf{y} \;-\; X\mathbf{b})$$

4.5.3 Optimization

Taking the gradient ∇sse with respect to the parameter vector $\mathbf{b}$ and setting it equal to the zero vector yields

$$-2X^t(\mathbf{y} \;-\; X\mathbf{b}) \;=\; \mathbf{0}$$

$$-2X^t\mathbf{y} \;+\; 2X^tX\mathbf{b} \;=\; \mathbf{0}$$

A more detailed derivation of this equation is given in section 3.4 of "Matrix Calculus: Derivation and Simple Application" [12]. Dividing the equation by 2 and moving the term involving $\mathbf{b}$ to the left side, results in the *Normal Equations*.

$$\boxed{X^t X\mathbf{b} \;=\; X^t\mathbf{y}} \tag{4.18}$$

Note: equivalent to minimizing the distance between $\mathbf{y}$ and $X\mathbf{b}$ is minimizing the sum of the squared residuals/errors (*Least Squares* method).

ScalaTion provides five techniques for solving for the parameter vector $\mathbf{b}$ based on the Normal Equations: Matrix Inversion, LU Factorization, Cholesky Factorization, QR Factorization and SVD Factorization.

4.5.4 Matrix Inversion Technique

Starting with the Normal Equations

$$X^t X\mathbf{b} \;=\; X^t\mathbf{y}$$

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

$$\boxed{\mathbf{b} \;=\; (X^tX)^{-1}X^t\mathbf{y}} \tag{4.19}$$

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

$$X^{\sim 1} \;=\; (X^tX)^{-1}X^t$$

Using the pseudo-inverse, the parameter vector $\mathbf{b}$ may be solved for as follows:

$$\mathbf{b} \;=\; X^{\sim 1}\mathbf{y}$$

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_pinv`) looks similar to the math.

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

A more robust approach is

```
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_pinv`, rather is directly solves for the parameters $\mathbf{b}$.

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

4.5.5 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^tX, an n-by-n matrix, is factored

$$X^tX = LU$$

where L is a lower left triangular n-by-n matrix and U is an upper right triangular n-by-n matrix. Then the normal equations may be rewritten

$$LU\mathbf{b} = X^t\mathbf{y}$$

Letting $\mathbf{w} = U\mathbf{b}$ allows the problem to solved in two steps. The first is solved by forward substitution to determine the vector $\mathbf{w}$.

$$L\mathbf{w} = X^t\mathbf{y}$$

Finally, the parameter vector $\mathbf{b}$ is determined by backward substitution.

$$U\mathbf{b} = \mathbf{w}$$

4.5.6 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

$$LL^t\mathbf{b} = X^t\mathbf{y}$$

Letting $\mathbf{w} = L^t\mathbf{b}$, we may solve for $\mathbf{w}$ using forward substitution

$$L\mathbf{w} = X^t\mathbf{y}$$

and then solve for $\mathbf{b}$ using backward substitution.

$$L^t\mathbf{b} = \mathbf{w}$$

4.5.7 QR Factorization Technique

A slightly slower, but even more robust technique is to use QR Factorization. Using this technique, the m-by-n X matrix can be factored directly, which increases the stability of the technique.

$$X = QR$$

where Q is an orthogonal m-by-n matrix and R matrix is a right upper triangular n-by-n matrix. Starting again with the Normal Equations,

$$X^t X \mathbf{b} \;=\; X^t \mathbf{y}$$

simply substitute QR for X.

$$(QR)^t QR\mathbf{b} \;=\; (QR)^t \mathbf{y}$$

Taking the transpose gives

$$R^t Q^t QR\mathbf{b} \;=\; R^t Q^t \mathbf{y}$$

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

$$R^t R\mathbf{b} \;=\; R^t Q^t \mathbf{y}$$

Multiply both sides by $(R^t)^{-1}$ yields

$$R\mathbf{b} \;=\; Q^t \mathbf{y}$$

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

$$X^{\sim 1} \;=\; R^{-1} Q^t$$

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

4.5.8 Singular Value Decomposition Technique

In cases where the rank of the data/input matrix X is not full or its multi-collinearity is high, a useful technique to solve for the parameters of the model is Singular Value Decomposition (SVD). Based on the derivation given in `http://www.ime.unicamp.br/~marianar/MI602/material%20extra/svd-regression-analysis.pdf`, we start with the equation estimating $\mathbf{y}$ as the product of the data matrix X and the parameter vector $\mathbf{b}$.

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

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

$$X \;=\; U\Sigma V^t$$

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

$$\mathbf{y} \;=\; U\Sigma V^t \mathbf{b}$$

Defining $\mathbf{d} = \Sigma V^t \mathbf{b}$, we may write

$$\mathbf{y} \;=\; U\mathbf{d}$$

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

$$\mathbf{d} \ = \ (U^tU)^{-1}U^t\mathbf{y}$$

Since $U^tU = I$, this reduces to

$$\mathbf{d} \ = \ U^t\mathbf{y}$$

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

$$\mathbf{b} \ = \ V\Sigma^{-1}\mathbf{d}$$

where Σ^{-1} is a diagonal matrix where elements on the main diagonal are the reciprocals of the singular values.

4.5.9 Use of Factorization in Regression

By default, ScalaTion uses QR Factorization to compute the pseudo-inverse $X^{\sim 1}$. 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`.

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

} // RegTechnique

import RegTechnique._
```

Based on the selected technique, the appropriate type of matrix factorization is performed. The first part of the code below constructs and returns a factorization object.

```
private def solver (xx: MatriD): Factorization =
{
    technique match {                                // select the factorization technique
    case QR       => new Fac_QR (xx, false)          // QR Factorization
    case Cholesky => new Fac_Cholesky (xx.t * xx)    // Cholesky Factorization
    case SVD      => new SVD (xx)                    // Singular Value Decomposition
    case LU       => new Fac_LU (xx.t * xx)          // LU Factorization
    case _        => new Fac_Inv (xx.t * xx)         // Inverse Factorization
    } // match
} // solver
```

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

```
def train (x_r: MatriD = x, y_r: VectoD = y): Regression =
{
    val fac = solver (x_r)                              // create selected factorization technique
    fac.factor ()                                       // factor the matrix, either X or X.t * X
    b = technique match {                               // solve for parameter/coefficient vector b
        case QR       => fac.solve (y_r)                // R * b = Q.t * y
        case Cholesky => fac.solve (x_r.t * y_r)        // L * L.t * b = X.t * y
        case SVD      => fac.solve (y_r)                // b = V * ^-1 * U.t * y
        case LU       => fac.solve (x_r.t * y_r)        // b = (X.t * X) \ X.t * y
        case _        => fac.solve (x_r.t * y_r)        // b = (X.t * X)^-1 * X.t * y
    } // match
    if (b(0).isNaN) flaw ("train", s"parameter b = $b")
    if (DEBUG) (s"train: parameter b = $b")
    this
} // train
```

After training, the `eval` method overridden for efficiency from `PredictorMat` does two things: First, the residual/error vector $\boldsymbol{\epsilon}$ is computed. Second, several quality of fit measures are computed by calling the `diagnose` method.

```
override def eval (x_e: MatriD = x, y_e: VectoD = y): Regression =
{
    val yp = x_e * b                                    // y predicted for x_e (test/full)
    e = y_e - yp                                        // compute residual/error vector e
    diagnose (e, y_e, yp)                               // compute diagnostics
    this
} // eval
```

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

4.5.10 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 df_r and left for error is df.

$$\begin{aligned} df_r &= n - 1 = k \\ df &= m - n \end{aligned}$$

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

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

`SimplerRegression` is a one extreme of model complexity, where $df = m-1$ and $df_r = 1$, so $r_{df} = m/(m-1)$ is close to one. For a more complicated model, say with $n = m/2$, r_{df} will be close to 2. This ratio can be used to adjust the Coefficient of Determination R^2 to reduce it with increasing number of parameters. This is called the Adjusted Coefficient of Determination $\hat{R}^2$

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

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

$$\begin{aligned} mse &= sse \,/\, df \\ msr &= ssr \,/\, df_r \end{aligned}$$

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 df_r degrees of freedom. Consequently, the ratio $\frac{msr}{mse}$ follows an F-distribution with (df_r, df) degrees of freedom. If this number exceeds the critical value, one can claim that the parameter vector **b** is not zero, implying the model is useful. More general quality of fit measures useful for comparing models are the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC).

In ScalaTion the several Quality of Fit (QoF) measures are computed by the `diagnose` method in the `Fit` class, see the section 4.1.

```
def diagnose (e: VectoD, yy: VectoD, yp : VectoD = null, w: VectoD = null)
```

4.5.11 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 $y_t - \mathbf{b} \cdot \mathbf{x}_t$. This can be repeated by iteratively letting t range from the first to the last row of data matrix X. For certain predictive analytics techniques such as Multiple Linear Regression, there are efficient ways to compute the test *sse* based on the leverage each point in the training set has [13].

***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.

```
val tsize = m / k                                                // test dataset size
for (l <- 0 until k) {
    x_e = x.slice (l * tsize, ((l+1) * tsize)                    // l-th test dataset
    x_r = 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 $\bar{R}^2$ are intending to more closely follow R^2_{cv} than R^2.

4.5.12 Collinearity

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

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

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

$$\mathbf{y} \;=\; b_0\mathbf{1} \;+\; b_1\mathbf{x}_{\text{-}1} \;+\; \dots \;+\; b_k\mathbf{x}_{\text{-}k} \;+\; \boldsymbol{\epsilon}$$

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 $\mathbf{y}$. This is the same example used for `SimpleRegression` in Exercise 2 with new variable x_2 added (i.e., $y = b_0 + b_1x_1 + b_2x_2 + \epsilon$).

```
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?

```
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, othersise -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 (VIF) function (or vif in ScalaTion). For a particular parameter b_j for the variable/predictor x_j, the function is evaluated as follows:

$$\text{vif}(b_j) \;=\; \frac{1}{1 - R^2(x_j)} \tag{4.20}$$

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 $\mathbf{x}_{\text{-}j}$) can be predicted by all x_l for $l \neq j$. Values above 10 may be considered problematic. In particular, the value for parameter b_j may be suspect, since its variance is inflated by $\text{vif}(b_j)$.

$$\hat{\sigma}^2(b_j) \;=\; \frac{mse}{k\,\hat{\sigma}^2(x_j)} \cdot \text{vif}(b_j) \tag{4.21}$$

See the exercises for details. Both `corr` and `vif` may be tested in ScalaTion using `RegressionTest4`.

One remedy to reduce collinearity/multi-collinearity is to eliminate the variable with the highest `corr`/`vif` function. Another is to use regularized regression such as `RidgeRegression` or `LassoRegression`.

4.5.13 Feature Selection

There may be predictor variables (features) in the model that contribute little in terms of their contributions to the model's ability to make predictions. The improvement to R^2 may be small and may make $\bar{R}^2$ or other quality of fit measures worse. An easy way to get a basic understanding is to compute the correlation of each predictor variable $\mathbf{x}_{:\mathbf{j}}$ (j^{th} column of matrix X) with the response vector $\mathbf{y}$. A more intuitive way to do this would be to plot the response vector $\mathbf{y}$ versus each predictor variable $\mathbf{x}_{:\mathbf{j}}$. 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 `forewordSel` method performs *forward selection* by adding the most predictive variable to the existing model, returning the variable to be added and a reference to the new model with the added variable/feature.

```
@param cols     the columns of matrix x currently included in the existing model
@param index_q  index of Quality of Fit (QoF) to use for comparing quality

def forwardSel (cols: Set [Int], index_q: Int = index_rSqBar): (Int, PredictorMat) =
```

Selecting the most predictive variable to add boils down to comparing on the basis of a Quality of Fit (QoF) measure. The default is the Adjusted Coefficient of Determination $\bar{R}^2$. The optional argument `index_q` indicates which QoF measure to use (defaults to `index_rSqBar`). 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.

```
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. This process is automated in the `forwardSelAll` method.

```
@param index_q  index of Quality of Fit (QoF) to use for comparing quality
@param cross    whether to include the cross-validation QoF measure

def forwardSelAll (index_q: Int = index_rSqBar, cross: Boolean = true): (Set [Int], MatriD) =
```

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.

```
@param cols     the columns of matrix x currently included in the existing model
@param index_q  index of Quality of Fit (QoF) to use for comparing quality
@param first    first variable to consider for elimination
                      (default (1) assume intercept x_0 will be in any model)

def backwardElim (cols: Set [Int], index_q: Int = index_rSqBar, first: Int = 1):
    (Int, PredictorMat) =
```

To start with a maximal model, set `cols = Set (0, 1, ..., k)` for a full model. As with `forwardSel` the `index_q` optional argument allows select from among the QoF measures. The last parameter `first` provides immunity from elimination for any variable/parameter that is less than `first` (e.g., to ensure that models include an intercept b_0, set `first` to one). The method will consider every variable/column from `first` until `x.dim2` in `cols` and pick the worst one for elimination.

```
    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. This process is automated in the `backwardElimAll` method.

```
    @param index_q  index of Quality of Fit (QoF) to use for comparing quality
    @param first    first variable to consider for elimination
    @param cross    whether to include the cross-validation QoF measure

    def backwardElimAll (index_q: Int = index_rSqBar, first: Int = 1, cross: Boolean = true):
        (Set [Int], MatriD) =
```

More advanced feature selection techniques include using genetic algorithms to find near optimal subsets of variables as well as techniques that select variables as part of the parameter estimation process, e.g., `LassoRegression`.

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:

```
    @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)

    def train (x_r: MatriD = x, y_r: VectoD = y): Regression =
    override def eval (x_e: MatriD = x, y_e: VectoD = y): Regression =
    override def predict (z: MatriD = x): VectoD = z * b
    def buildModel (x_cols: MatriD): Regression =
```

4.5.14 Exercises

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

$$A\mathbf{b} \ = \ \mathbf{z}$$

2. Solving a regression problem in ScalaTion simply involves creating a data/input matrix X and response/output vector $\mathbf{y}$ and then creating a `Regression` object upon which `analyze` (i.e., `train` and `eval`) and `report` methods are called. The Texas Temperature data-set below from `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, 30.867, 3000.0, 102.900,    // Pecos
                              1.0, 36.350, 3693.0, 102.083,    // Sherman
                              1.0, 30.300,  597.0,  97.700,    // Travis
                              1.0, 26.900,  315.0,  99.283,    // Zapata
                              1.0, 28.450,  459.0,  99.217,    // Lasalle
                              1.0, 25.900,   19.0,  97.433)    // Cameron

val y = VectorD (56.0, 48.0, 60.0, 46.0, 38.0, 46.0, 53.0, 46.0,
                 44.0, 41.0, 47.0, 36.0, 52.0, 60.0, 56.0, 62.0)

val rg = new Regression (x, y)                                 // create a Regression model
rg.analyze ()                                                  // train and evaluate
println (reg.report)
```

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 $\mathbf{z}$, 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 `forwrardSel` or `backwardElim`.

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

The source code for this example is at
`http://www.cs.uga.edu/~jam/scalation_1.6/src/main/scala/apps/analytics/TempRegression.scala` .

3. Gradient descent can be used for Multiple Linear Regression as well. For gradient descent, pick a starting point $\mathbf{b}^0$, compute the gradient ∇sse and move $-\eta \nabla sse$ from $\mathbf{b}^0$ where η is the learning rate. Write a Scala program that repeats this for several iterations for the above data. What is happening to the value of sse.

$$\nabla sse \;=\; -\,2X^t(\mathbf{y} - X\mathbf{b})$$

Substituting $\boldsymbol{\epsilon} = \mathbf{y} \,-\, X\mathbf{b}$, half ∇sse may be written as

$$-X^t\boldsymbol{\epsilon}$$

Starting with data matrix `x`, response vector `y` and parameter vector `b`, in ScalaTion, the calculations become

```
val yp = x * b                              // y predicted
val e  = y - yp                             // error
val g  = x.t * e                            // - gradient
b     += g * eta                            // update parameter b
val 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 know dataset that is available at multiple websites including the UCI Machine Learning Repository `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`. The response variable is the miles per gallon (`mpg`: continuous) while the predictor variables are `cylinders`: multi-valued discrete, `displacement`: continuous, `horsepower`: continuous, `weight`: continuous, `acceleration`: continuous, `model_year`: multi-valued discrete, `origin`: multi-valued discrete, and `car_name`: string (unique for each instance). Since the `car_name` is unique and obviously not causal, this variable is eliminated, leaving seven predictor variables. First compute the correlations between `mpg` (vector $\mathbf{y}$) and the seven predictor variables (each column vector $\mathbf{x_{:j}}$ in matrix X).

```
val correlation = y corr x_j
```

and then plot `mpg` versus each of the predictor variables. The source code for this example is at
`http://www.cs.uga.edu/~jam/scalation_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.

```
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 $\bar{R}^2$, 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.

9. The variance of the estimate of parameter b_j may be estimated as follows:

$$\hat{\sigma}^2(b_j) \;=\; \frac{mse}{k\,\hat{\sigma}^2(x_j)} \cdot \text{vif}(b_j)$$

Derive this formula. The standard error is the square root of this value. Use the estimate for b_j and its standard error to compute a t-value and p-value for the estimate. Run the AutoMPG model and explain these values produced by the `summary` method.

4.5.15 Further Reading

1. Introduction to Linear Regression Analysis, 5th Edition [15]

4.6 Ridge Regression

The `RidgeRegression` class supports multiple linear ridge regression. As with `Regression`, the predictor variables $\mathbf{x}$ are multi-dimensional $[x_1, \dots, x_k]$, as are the parameters $\mathbf{b} = [b_1, \dots, b_k]$. Ridge regression adds a penalty based on the ℓ_2 norm of the parameters $\mathbf{b}$ to reduce the chance of them taking on large values that may lead to less robust models. The penalty is not to be included on the intercept parameter b_0, as this would shift predictions in a way that would adversely affect the quality of the model.

4.6.1 Model Equation

Centering of the data allows the intercept to be removed from the model. The combined centering on both the predictor variables and the response variable takes care of the intercept, so it is not included in the model. Thus, the goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_1 x_1 \;+\; \cdots + b_k x_k \;+\; \epsilon} \tag{4.22}$$

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

4.6.2 Training

Center the dataset $(X, \mathbf{y})$ has the following effects: First, when the X matrix is centered, the intercept $b_0 = \mu_y$. Second, when $\mathbf{y}$ is centered, μ_y becomes zero, implying $b_0 = 0$. To rescale back to the original reposnse values, μ_y can be added back during prediction. Therefore, both the data/input matrix X and the response/output vector $\mathbf{y}$ should be *centered* (zero meaned).

The regularization of the model adds an ℓ_2-penalty on the parameters $\mathbf{b}$. The loss/objective function to minimize is now *sse* plus the penalty.

$$\boxed{f_{obj} \;=\; sse \;+\; \lambda \,\|\mathbf{b}\|^2 \;=\; \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon} \;+\; \lambda\, \mathbf{b} \cdot \mathbf{b}} \tag{4.23}$$

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

$$f_{obj} \;=\; (\mathbf{y} - X\mathbf{b}) \cdot (\mathbf{y} - X\mathbf{b}) \;+\; \lambda\, \mathbf{b} \cdot \mathbf{b}$$

4.6.3 Optimization

Fortunately, the quadratic nature of the penalty function allows it to be combined easily with the quadratic error terms, so that matrix factorization can still be used for finding optimal values for parameters.

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

$$\begin{aligned} -2X^t(\mathbf{y} \;-\; X\mathbf{b}) \;+\; 2\lambda\mathbf{b} \;&=\; \mathbf{0} \\ -X^t\mathbf{y} \;+\; X^tX\mathbf{b}) \;+\; \lambda\mathbf{b} \;&=\; \mathbf{0} \\ X^tX\mathbf{b} \;+\; \lambda\mathbf{b} \;&=\; X^t\mathbf{y} \end{aligned}$$

Since $\lambda\mathbf{b} = \lambda I\mathbf{b}$ where I is the n-by-n identity matrix, we may write

$$\boxed{(X^tX + \lambda I)\mathbf{b} \;=\; X^t\mathbf{y}} \tag{4.24}$$

Matrix factorization may now be used to solve for the parameters $\mathbf{b}$ in the modified Normal Equations.

4.6.4 Centering

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

```
val mu_x = x.mean                          // columnwise mean of x
val mu_y = y.mean                          // mean of y
val x_c  = x - mu_x                        // centered x (columnwise)
val y_c  = y - mu_y                        // centered y
```

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

```
val rrg = new RidgeRegression (x_c, y_c)
rrg.analyze ()
println (rrg.report)
```

Now, when making predictions, the new data vector $\mathbf{z}$ needs to be centered by subtracting `mu_x`. Then the `predict` method is called, after which the mean of $\mathbf{y}$ is added.

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

4.6.5 The λ Hyper-parameter

The value for λ 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 => OLS) in penalty
@param technique  the technique used to solve for b in (x.t*x + lambda*I)*b = x.t*y

class RidgeRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
                       hparam: HyperParameter  = RidgeRegression.hp,
                       technique: RegTechnique = Cholesky)
      extends PredictorMat (x, y, fname_, hparam)


def train (x_r: MatriD, y_r: VectoD): RidgeRegression =
def gcv (xx: MatriD = x, yy: VectoD = y): Double =
def buildModel (x_cols: MatriD): RidgeRegression =
```

4.6.6 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 λ and examine its effect on the parameter vector $\mathbf{b}$, 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 $\mathbf{y}$? 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 ℓ_2-penalty (Euclidean norm) like `RidgeRegression`, it uses and an ℓ_1-penalty (Manhattan norm). In `RidgeRegression` when b_j approaches zero, b_j^2 becomes very small and has little effect on the penalty. For `LassoRegression`, the effect based on $|b_j|$ will be larger, so it is more likely to set parameters to zero. See section 6.2.2 in [13] for a more detailed explantion on how `LassoRegression` can eliminate a variable/feature by setting its parameter/coefficient to zero.

4.7.1 Model Equation

As with `Regression`, the goal is to fit the parameter vector **b** in the model/regression equation,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... + b_k x_k + \epsilon} \tag{4.25}$$

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

4.7.2 Training

The regularization of the model adds an ℓ_1-penalty on the parameters **b**. The objective/loss function to minimize is now *sse* plus the penalty.

$$\boxed{f_{obj} \;=\; \frac{1}{2}\, sse \,+\, \lambda\, \|\mathbf{b}\|_1 \;=\; \frac{1}{2}\, \|\boldsymbol{\epsilon}\|^2 \,+\, \lambda\, \|\mathbf{b}\|_1} \tag{4.26}$$

where λ is the shrinkage parameter. Substituting $\boldsymbol{\epsilon} = \mathbf{y} - X\mathbf{b}$ yields

$$f_{obj} \;=\; \tfrac{1}{2}\, \|\mathbf{y} - X\mathbf{b}\|^2 + \lambda\, \|\mathbf{b}\|_1$$

Although similar to the ℓ_2 penalty used in Ridge Regression, it may often be more effective. Still, the ℓ_1 penalty for Lasso has a disadvantage that the absolute values in the ℓ_1 norm make the objective function non-differentiable. 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.

4.7.3 Optimization Stategies

Coordinate Descent

Alternative Direction Method of Multipliers

ScalaTion also uses the Alternative Direction Method of Multipliers (ADMM) [4] algorithm to optimize the **b** parameter vector. The algorithm for using ADMM for Lasso Regression is outlined in section 6.4 of [4]. 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.

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

For Lasso Regression, the f function will capture the loss function ($\frac{1}{2}\, sse$), while the g function will capture the ℓ_1 regularization, i.e.,

$$f(\mathbf{b}) = \tfrac{1}{2}\, \|\mathbf{y} - X\mathbf{b}\|^2 \quad , \quad g(\mathbf{z}) = \lambda\, \|\mathbf{z}\|_1$$

Therefore, the iterative step in the ADMM algorithm becomes

$$\begin{aligned}
\mathbf{b} &= (X^tX + \rho I)^{-1}(X^t\mathbf{y} + \rho(\mathbf{z} - \mathbf{u})) \\
\mathbf{z} &= S_{\lambda/\rho}(\mathbf{b} + \mathbf{u}) \\
\mathbf{u} &= \mathbf{u} + \mathbf{b} - \mathbf{z}
\end{aligned}$$

where S is the soft thresholding function and $\mathbf{u}$ is the Lagrangian vector. See `scalation.minima.LassoAdmm` for coding details.

4.7.4 The λ Hyper-parameter

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

`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'

class LassoRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
                       hparam: HyperParameter = LassoRegression.hp)
      extends PredictorMat (x, y, fname_, hparam)

def train (x_r: MatriD = x, y_r: VectoD = y): LassoRegression =
def buildModel (x_cols: MatriD): LassoRegression =
```

4.7.5 Exercises

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

```
    // 5 data points:             one   x_0    x_1
    val x = 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 λ and examine its effect on the parameter vector **b**, the quality of fit and predictions made.

```
    import LassoRegression.hp

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

3. Using the above dataset and the AutoMPG dataset, determine the effects of (a) centering the data ($\mu = 0$), (b) standardizing the data ($\mu = 0, \sigma = 1$).

```
    import MatrixTransforms._

    val x_n = normalize (x, (mu_x, sig_x))
    val y_n = y.standardize
```

4. 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 ℓ_2 and ℓ_1 penalties to try to combine the best features of both `RidgeRegression` and `LassoRegression`. Elastic Nets naturally includes two shrinkage parameters, λ_1 and λ_2. Is the additional complexity worth the benefits?

4.7.6 Further Reading

1. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers [4]

2. Feature Selection Using LASSO [8]

4.8 Transformed Regression

The `TranRegression` class supports transformed multiple linear regression. In this case, the predictor vector **x** is multi-dimensional $[1, x_1, ...x_k]$. In many cases, the relationship between the response scalar y and the predictor vector **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)
```

4.8.1 Model Equation

The goal then is to fit the parameter vector **b** in the transformed model/regression equation

$$\boxed{\text{tran}(y) \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + ... \, b_k x_k + \epsilon} \tag{4.27}$$

where ϵ 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).

```
Regression (x, y.map (tran), technique)
```

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

```
override def predict (z: VectoD): Double = itran (b dot z)
```

4.8.2 Example

Imagine a system where the rate of change of the response variable y with the predictor varaible x is proportional to ts current value y and is y_0 when $x = 0$.

$$\frac{dy}{dx} \;=\; gy$$

This *differential equation* can be solved by direct integration to obtain

$$\int \frac{dy}{y} \;=\; \int g\,dx$$

As the integral of $\frac{1}{y}$ is $\ln y$, integrating both sides gives

$$\ln y \;=\; gx + C$$

Solving for the contant C, produces

$$y \;=\; y_0 e^{gx}$$

When the growth factor g if positive, the system exhibits exponential growth, while when it is negative, it exhibits exponential decay. So far we have ignored noise. For previous modeling techniques we have assumed that noise is additive and typically normally distributed. For phenomena exhibiting exponential growth or decay, this may nor be the case. When the error is multiplicative, we may collect it into the exponent.

$$y \;=\; y_0 e^{gx+\epsilon}$$

Now applying a log transformation, will yield

$$\log(y) \;=\; log(y_0) + gx + \epsilon \;=\; b_0 + b_1 x + \epsilon$$

An alternative to using `TranRegression` is to use Exponential Regression `ExpRegression`, a form of Generalized Linear Model (see the exercises for a comparison).

4.8.3 Quality of Fit

For a fair comparison with other modeling techniques, the Quality of Fit (QoF) or overall diagnostics are based on the original response values, as provided by the usual `eval` method. It may be useful to study the Quality of Fit for the trantransformed response vector `y.map (tran)` as well via the `eval0` method.

`TranRegression` **Class**

Class Methods:

```
    @param x          the data/input matrix
    @param y          the response/output vector
    @param fname_     the feature/variable names
    @param hparam     the hyper-parameters (currently none)
    @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,
                          hparam: HyperParameter = null,
                          tran: FunctionS2S = log, itran: FunctionS2S = exp,
                          technique: RegTechnique = QR)
          extends Regression (x, y.map (tran), fname_, hparam, technique)

    def eval0 (x_e: MatriD = x, y_e: VectoD = getY): TranRegression =
    override def eval (x_e: MatriD = x, y_e: VectoD = y): TranRegression =
    override def analyze (x_r: MatriD = x, y_r: VectoD = getY,
                          x_e: MatriD = x, y_e: VectoD = y): PredictorMat =
    override def predict (z: VectoD): Double = itran (b dot z)
    override def predict (z: MatriD = x): VectoD = (z * b).map (itran)
```

Box-Cox transformations (see the last exercise) are provided in the class' companion object.

4.8.4 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.analyze ()
    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 the 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 (y2, yp2)
val ys3 = MatrixD (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 ("Regession")
val rg = new Regression (x, y)
rg.analyze ()
println (rg.report)
println (rg.summary)

banner ("TranRegession")
val trg = new TranRegression (x, y, sqrt _, sq _)
trg.analyze ()
println (rg.report)
println (rg.summary)
```

5. Compare `SimpleRegression`, `TranRegression` and `ExpRegression` on the beer foam dataset `www.tf.uni-kiel.de/matwis/amat/iss/kap_2/articles/beer_article.pdf`. The last two are similar, but `TranRegression` assumes multiplicative noise, while `ExpRegression` assumes additive noise, so they produce different predictions. Plot and compare the three predictions.

```
val x1 = VectorD (0, 15, 30, 45, 60, 75, 90, 105, 120, 150, 180, 210, 240, 300, 360)
val y  = VectorD (14.0, 12.1, 10.9, 10.0, 9.3, 8.6, 8.0, 7.5,
                   7.0,  6.2,  5.5, 4.5, 3.5, 2.0, 0.9)
val _1 = VectorD.one (x1.dim)
val x  = MatrixD (_1, x1)
```

6. Consider the following family of transformation functions.

$$f_{tran}(y) \;=\; \frac{y^\lambda - 1}{\lambda}$$

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

```
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.

4.9.1 Model Equation

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

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{x}' + \epsilon \;=\; b_0 + b_1 x_1 + b_2 x_1^2 + b_4 x_2 + b_5 x_2^2 + \epsilon} \tag{4.28}$$

where $\mathbf{x}' = [1, x_1, x_1^2, x_2, x_2^2]$ and ϵ represents the residuals (the part not explained by the model). The number of terms (nt) in the model increases linearly with the dimensionality of the space (n) according to the following formula:

$$nt \;=\; 2n + 1 \qquad e.g., \;\; nt \;=\; 5 \;\; \text{for} \;\; n = 2 \tag{4.29}$$

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 `forms` will take an unexpanded vector `v`, the number of variables, 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.

```
@param v   the vector/point (i-th row of x) for creating forms/terms
@param k   number of features/predictor variables (not counting intercept) [not used]
@param nt  the number of terms

override def forms (v: VectoD, k: Int, 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
```

The `allForms` function in the `ModelFactory` object calls this function for each row of the data matrix X to create the expanded matrix.

`QuadRegression` **Class**

Class Methods:

```
@param x_         the initial data/input matrix (before quadratic term expansion)
                      must not include an intercept column of all ones
@param y          the response/output vector
@param fname_     the feature/variable names
@param hparam     the hyper-parameters
@param technique  the technique used to solve for b in x.t*x*b = x.t*y

class QuadRegression (x_ : MatriD, y: VectoD, fname_ : Strings = null, hparam: HyperParameter = null,
                      technique: RegTechnique = QR)
      extends Regression (QuadRegression.allForms (x_), y, fname_, hparam, technique)
      with ExpandableForms

def expand (z: VectoD): VectoD = QuadRegression.forms (z, n0, nt)
def predict_ex (z: VectoD): Double = predict (expand (z))
```

The next few modeling techniques described in subsequent sections support the development of low-order multi-dimensional polynomials regression models.

4.9.2 Exercises

1. Perform Quadratic 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 * m * m)`.

   ```
   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`.

    ```
    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 Quadratic Regression with Cross-Terms

The `QuadXRegression` class extends `Regression` and adds cross-terms in addition to the quadratic-terms added by `QuadRegression`.

4.10.1 Model Equation

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

$$\boxed{y \ = \ \mathbf{b} \cdot \mathbf{x}' + \epsilon \ = \ b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1 x_2 + b_4 x_2 + b_5 x_2^2 + \epsilon} \tag{4.30}$$

where $\mathbf{x}' = [1, x_1, x_1^2, x_1x_2, x_2, x_2^2]$ and ϵ 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 ($n \rightarrow n+1$).

$$nt \ = \ \binom{n+2}{2} \ = \ \frac{(n+2)(n+1)}{2} \qquad e.g., \quad nt \ = \ 6 \ \text{ for } \ n = 2 \tag{4.31}$$

Such models generalize `QuadRegression` by introducing cross-terms, e.g., x_1x_2. Adding cross-terms makes the number of terms increase quadratically rather than linearly with the dimensionality. Consequently, multi-collinearity problems may be intensified and the need for feature selection, therefore, increases.

The addition of squared and cross-term columns is performed by functions in the companion object. The function `forms` will take an unexpanded vector v, the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```
@param v   the source vector/point for creating forms/terms
@param k   the number of features/predictor variables (not counting intercept)
@param nt  the number of forms/terms

override def forms (v: VectoD, k: Int, nt: Int): VectoD =
{
    val q = one (1) ++ v                     // augmented vector: [ 1., v(0), ..., v(k-1) ]
    val z = new VectorD (nt)                 // vector of all forms/terms
    var l = 0
    for (i <- 0 to k; j <- i to k) { z(l) = q(i) * q(j); l += 1 }
    z
} // forms
```

`QuadXRegression` **Class**

Class Methods:

```
@param x_          the m-by-n data/input matrix (original un-expanded)
@param y           the m response/output vector
@param fname_      the feature/variable names
@param hparam      the hyper-parameters
@param technique   the technique used to solve for b in x.t*x*b = x.t*y
```

```
class QuadXRegression (x_ : MatriD, y: VectoD, fname_ : Strings = null, hparam: HyperParameter = null,
                       technique: RegTechnique = QR)
      extends Regression (QuadXRegression.allForms (x_), y, fname_, hparam, technique)
      with ExpandableForms

def expand (z: VectoD): VectoD = QuadXRegression.forms (z, n0, nt)
def predict_ex (z: VectoD): Double = predict (expand (z))
```

4.10.2 Exercises

1. Perform Quadratic and QuadraticX 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` and `QuadXRegression`.

4.11 Cubic Regression

The `CubicRegression` class extends `Regression` and adds cubic-terms in addition to the quadratic-terms and cross-terms added by `QuadXRegression`.

4.11.1 Model Equation

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

$$y \;=\; \mathbf{b} \cdot \mathbf{x}' + \epsilon \;=\; b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1 x_2 + b_4 x_2 + b_5 x_2^2 + b_6 x_1^2 + b_7 x_2^3 + \epsilon \tag{4.32}$$

where $\mathbf{x}' = [1, x_1, x_1^2, x_1 x_2, x_2, x_2^2, x_1^3, x_2^3]$ and ϵ represents the residuals (the part not explained by the model). The number of terms (nt) in the model still increases quadratically with the dimensionality of the space (n) according to the formula for triangular numbers shifted by ($n \rightarrow n+1$) plus n for the cubic terms.

$$nt \;=\; \binom{n+2}{2} \;=\; \frac{(n+2)(n+1)}{2} + n \qquad e.g., \quad nt \;=\; 8 \;\text{ for }\; n = 2 \tag{4.33}$$

When $n = 10$, the number of terms and corresponding parameters $nt = 76$, whereas for `Regression`, `QuadRegression` and `QuaXdRegression` and order 2, it would 11, 21 and 66, respectively. Issues related negative degrees of freedom, overfitting and multi-collinearity will need careful attention.

The addition of squared, cross-term and cubic columns is performed by functions in the companion object. The function `forms` will take an unexpanded vector v, the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```
@param v   the source vector/point for creating forms/terms
@param k   the number of features/predictor variables (not counting intercept)
@param nt  the number of forms/terms

override def forms (v: VectoD, k: Int, nt: Int): VectoD =
{
    val z = new VectorD (k)
    for (i <- z.range) z(i) = v(i) ~^3
    QuadXRegression.forms (v, k, QuadXRegression.numTerms (k)) ++ z
} // forms
```

`CubicRegression` **Class**

Class Methods:

```
@param x_         the input vectors/points
@param y          the response vector
@param fname_     the feature/variable names
@param hparam     the hyper-parameters
@param technique  the technique used to solve for b in x.t*x*b = x.t*y

class CubicRegression (x_ : MatriD, y: VectoD, fname_ : Strings = null, hparam: HyperParameter = null,
                       technique: RegTechnique = QR)
```

```
        extends Regression (CubicRegression.allForms (x_), y, fname_, hparam, technique)
        with ExpandableForms

    def expand (z: VectoD): VectoD = CubicRegression.forms (z, n0, nt)
    def predict_ex (z: VectoD): Double = predict (expand (z))
```

4.12 Cubic Regression with Cross Terms

The `CubicXRegression` class extends `Regression` and adds cubic-cross-terms in addition to the quadratic-terms, cross-terms and cubic-terms added by `CubicRegression`.

4.12.1 Model Equation

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

$$y \;=\; \mathbf{b} \cdot \mathbf{x}' + \epsilon \;=\; b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1^3 + b_4 x_1 x_2 + b_5 x_1^2 x_2 + b_6 x_1 x_2^2 + b_7 x_2 + b_8 x_2^2 + b_9 x_2^3 + \epsilon \tag{4.34}$$

where $\mathbf{x}' = [1, x_1, x_1^2, x_1^3, x_1 x_2, x_1^2 x_2, x_1 x_2^2, x_2, x_2^2, x_1^3, x_2^3]$ and ϵ 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 ($n \to n+1$).

$$nt \;=\; \binom{n+3}{3} \;=\; \frac{(n+3)(n+2)(n+1)}{6} \qquad e.g., \quad nt \;=\; 10 \;\; \text{for} \;\; n = 2 \tag{4.35}$$

When $n = 10$, the number of terms and corresponding parameters $nt = 286$, whereas for `Regression`, `QuadRegression`, `QuaXdRegression` and `CubicRegression` and order 2, it would 11, 21, 66 and 76, respectively. Issues related negative degrees of freedom, overfitting and multi-collinearity will need careful attention.

The addition of squared, cross-term, cubic and cubic-cross-term columns is performed by functions in the companion object. The function `forms` will take an unexpanded vector v, the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```
@param v   the source vector/point for creating forms/terms
@param k   the number of features/predictor variables (not counting intercept)
@param nt  the number of forms/terms

override def forms (v: VectoD, k: Int, nt: Int): VectoD =
{
    val q = one (1) ++ v                    // augmented vector: [ 1., v(0), ..., v(k-1) ]
    val z = new VectorD (nt)                // vector of all forms/terms
    var l = 0
    for (i <- 0 to k; j <- i to k; h <- j to k) { z(l) = q(i) * q(j) * q(h); l += 1 }
    z
} // forms
```

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 $\mathbf{x}$ 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.

`CubicXRegression` **Class**

Class Methods:

```
@param x_         the input vectors/points
@param y          the response vector
@param fname_     the feature/variable names
@param hparam     the hyper-parameters
@param technique  the technique used to solve for b in x.t*x*b = x.t*y

class CubicXRegression (x_ : MatriD, y: VectoD, fname_ : Strings = null, hparam: HyperParameter = null,
                       technique: RegTechnique = QR)
      extends Regression (CubicXRegression.allForms (x_), y, fname_, hparam, technique)
      with ExpandableForms

def expand (z: VectoD): VectoD = CubicXRegression.forms (z, n0, nt)
def predict_ex (z: VectoD): Double = predict (expand (z))
```

4.12.2 Exercises

1. Perform Cubic and CubicX 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`, `QuadXRegression`, `CubicRegression` and `CubicXRegression` 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 $f(\mathbf{x})$ in n dimensions is $\binom{n+2}{2}$, by decomposing the function into its quadratic (both squared and cross), linear and constant terms,

   $$f(\mathbf{x}) = \mathbf{x}^t A\mathbf{x} + \mathbf{b}^t\mathbf{x} + c$$

   where A in an n-by-n matrix, $\mathbf{b}$ is an n-dimensional column vector and c is a scalar. Hint: A is symmetric, but the main diagonal is not repeated, and we are looking for unique terms (e.g., x_1x_2 and

x_2x_1 are treated as the same). Note, when $n = 1$, A and $\mathbf{b}$ become scalars, yielding the usual quadratic function $ax^2 + bx + c$.

4.13 Weighted Least Squares Regression

The `Regression_WLS` class supports weighted multiple linear regression. In this case, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ...x_k]$.

4.13.1 Model Equation

As before the model/regression equation is

$$y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_0 \;+\; b_1 x_1 \;+\; \ldots \;+\; b_k x_k + \epsilon$$

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

$$X^t X \mathbf{b} \;=\; X^t \mathbf{y}$$

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

$$w_i \;=\; \frac{1}{\sigma_i^2}$$

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

$$\mathbf{r} \;=\; \sqrt{|\boldsymbol{\epsilon}|}$$

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 $\mathbf{r}$ onto X to obtain the predictions $\hat{\mathbf{r}}$. See Exercise 1 for a comparison or the two methods `setWeights0` and `setWeights`.

$$w_i \;=\; \frac{n}{\hat{r}_i} \tag{4.36}$$

See [15] 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

$$X^t W X \mathbf{b} \;=\; X^t W \mathbf{y} \tag{4.37}$$

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

In summary, Weighted Least-Squares (WLS) is accomplished by reweighting and then using Ordinary Least Squares (OLS). See `http://en.wikipedia.org/wiki/Least_squares#Weighted_least_squares`.

`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]
```

4.13.2 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 advanrages and disadvantages.

2. Prove that reweighting the data matrix X and the response vector $\mathbf{y}$ and solving for the parameter vector $\mathbf{b}$ in the standard Normal Equations $X^tX\mathbf{b} = X^t\mathbf{y}$ gives the same result as not reweighting and solving for the parameter vector $\mathbf{b}$ in the Weighted Normal Equations $X^tWX\mathbf{b} = X^tW\mathbf{y}$.

3. Given an error vector $\boldsymbol{\epsilon}$, what does its covariance matrix $\mathbb{C}\left[\boldsymbol{\epsilon}\right]$ represent? How can it be estimated? What are its diagonal elements?

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?

4.14 Polynomial Regression

The PolyRegression class supports polynomial regression. In this case, $\mathbf{x}$ is formed from powers of a single parameter t, $[1, t, t^2, \ldots, t^k]$.

4.14.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_0 \;+\; b_1 t \;+\; b_2 t^2 \;+\; \ldots \;+\; b_k t^k \;+\; \epsilon$$

where ϵ represents the residuals (the part not explained by the model). Such models are useful when there is a nonlinear relationship between a response and a predictor variable, e.g., y may vary quadratically with t.

A training set now consists of two vectors, one for the m-vector $\mathbf{t}$ and one for the m-vector $\mathbf{y}$. An easy way to implement polynomial regression is to expand each t value into an $\mathbf{x}$ vector to form a data/input matrix X and pass it to the Regression class (multiple linear regression). The columns of data matrix X represent powers of the vector $\mathbf{t}$.

$$X \;=\; \left[\mathbf{1}, \mathbf{t}, \mathbf{t}^2, \ldots, \mathbf{t}^k\right]$$

In ScalaTion the vector $\mathbf{t}$ is expanded into a matrix X before calling Regression. The number of columns in matrix X is the order k plus 1 for the intercept.

```
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 [23]. This is done in ScalaTion by changing the raw flag to false.

PolyRegression Class

Class Methods:

```
    @see www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx
    @param t          the initial data/input m-by-1 matrix: t_i expands to x_i = [1, t_i, t_i^2, ... t_i^k]
    @param y          the response/ouput vector
    @param ord        the order (k) of the polynomial (max degree)
    @param fname_     the feature/variable names
    @param hparam     the hyper-parameters
    @param technique  the technique used to solve for b in x.t*x*b = x.t*y

    class PolyRegression (t: MatriD, y: VectoD, ord: Int, fname_ : Strings = null, hparam: HyperParameter = null,
                          technique: RegTechnique = Cholesky)
          extends Regression (PolyRegression.allForms (t, ord), y, fname_, hparam, technique)
```

```
      with ExpandableForms

  def expand (z: VectoD): VectoD = PolyRegression.forms (z, n0, nt)
  def predict (z: Double): Double = predict_ex (VectorD (z))
  def predict_ex (z: VectoD): Double = predict (expand (z))
```

4.14.2 Exercises

1. Generate two vectors **t** and **y** as follows.

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

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

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

4.15 Trigonometric Regression

The `TrigRegression` class supports trigonometric regression. In this case, **x** is formed from trigonometric functions of a single parameter t, $[1, \sin(\omega t), \cos(\omega t), \dots, \sin(k\omega t), \cos(k\omega t)]$.

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.

4.15.1 Model Equation

The goal is to fit the parameter vector **b** in the model/regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 \sin(\omega t) + b_2 \cos(\omega t) + \dots,\; b_{2k-1} \sin(k\omega t) + b_{2k} \cos(k\omega t) + \epsilon$$

where ω is the base angular displacement in radians (e.g., π) and ϵ represents the residuals (the part not explained by the model).

A training set now consists of two vectors, one for the m-vector **t** and one for the m-vector **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.

$$X \;=\; [\mathbf{1},\, \sin(\omega \mathbf{t}),\, \cos(\omega \mathbf{t}),\, \sin(2\omega \mathbf{t}),\, \cos(2\omega \mathbf{t}),\, \dots,\, \sin(k\omega \mathbf{t}),\, \cos(k\omega \mathbf{t})]$$

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

```
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:

```
    @param t          the initial data/input m-by-1 matrix: t_i expands to x_i
    @param y          the response/ouput vector
    @param ord        the order (k), maximum multiplier in the trig function (kwt)
    @param fname_     the feature/variable names
    @param hparam     the hyper-parameters
    @param technique  the technique used to solve for b in x.t*x*b = x.t*y

    class TrigRegression (t: MatriD, y: VectoD, ord: Int, fname_ : Strings = null, hparam: HyperParameter = null,
                          technique: RegTechnique = QR)
          extends Regression (TrigRegression.allForms (t, ord), y, fname_, hparam, technique)
          with ExpandableForms
```

```
def expand (z: VectoD): VectoD = TrigRegression.forms (z, n0, nt, w)
def predict (z: Double): Double = predict_ex (VectorD (z))
def predict_ex (z: VectoD): Double = predict (expand (z))
```

4.15.2 Exercises

1. Create a noisy cubic function and test how well `TrigRegression` can fit the data for various values of k (harmonics) generated from this function.

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

2. Make the noisy cubic function periodic and test how well `TrigRegression` can fit the data for various values of k (harmonics) generated from this function.

```
val noise = Normal (0.0, 10.0)
val t     = VectorD.range (0, 200)
val y     = new VectorD (t.dim)
for (i <- 0 until 5) {
    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?

4.16 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.

4.16.1 Handling Categorical Variables

Binary Variables

In the *binary case*, when a variable x_j may take on only two distinct values, e.g., Red or Black, then it may simply be encoded as 0 for Red and 1 for Black. Therefore, a single zero-one, encoded/dummy variable x_j, can be used to distinguish the two cases. For example, when $x_j \in \{Red, Black\}$, it would be replaced by one encoded/dummy variable, x_{j0} as shown in Table 4.3.

Table 4.2: Encoding a Binary Variable

x_j	**encoded** x_j	**dummy** x_{j0}
Red	0	0
Black	1	1

Categorical Variables

For the more general *categorical case*, when the number distinct values for a variable x_j is greater than two, simply encoding the j^{th} column may not be ideal. Instead multiple dummy variables should be used. The number of dummy variables required is one less than the number of distinct values n_{dv}. In one hot encoding, the number of dummy variables may be equal to the n_{dv}, however, this will produce a singular expanded data matrix X (i.e., perfect multi-collinearity).

First, the categorical variable x_j may be encoded using integer values as follows:

$$\text{encoded } \; x_j \; = \; 0, 1, \ldots, n_{dv} - 1$$

Next, for categorical variable x_j, create $n_{dv} - 1$ dummy variables $\{x_{jk} | k = 0, \ldots, n_{dv} - 2\}$ and use the following loop to set the value for each dummy variable.

```
for (k <- 0 until n_dv - 1) 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, $x_j \in \{English, French, German, Spanish\}$ would be replaced by three dummy variables, x_{j0}, x_{j1} and x_{j2}, as shown in Table 4.3.

Table 4.3: Conventional Encoding a Categorical Variable

x_j	**encoded** x_j	**dummy** x_{j0}**,** x_{j1}**,** x_{j2}
English	0	0, 0, 0
French	1	1, 0, 0
German	2	0, 1, 0
Spanish	3	0, 0, 1

Unfortunately, for the conventional encoding of a categorical variable, a dummy variable column will be identical to its square, which will result in singular matrix for `QuadRegression`. One solution is to exclude dummy variables in the column expansion done by `QuadRegression`. Alternatively, a more robust encoding such as the one given in Table 4.4 may be used.

Table 4.4: Robust Encoding a Categorical Variable

x_j	**encoded** x_j	**dummy** x_{j0}**,** x_{j1}**,** x_{j2}
English	0	1, 1, 1
French	1	2, 1, 1
German	2	1, 2, 1
Spanish	3	1, 1, 2

Conversion from strings to an integer encoding can be accomplished using the `map2Int` function in the `Converter` object within the `scalation.linalgebra` package. The vector of encoded integers `xe` can be made into a matrix using `MatrixI (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 done by the caller in cases where there are few categorical variables, by expanding the input data matrix before passing it to the Rgression 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.

4.16.2 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, $\mathbf{x}$ consists of $n_{dv}-1$ dummy variables d_k for $k \in \{1, n_{dv}-1\}$

$$y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_0 \;+\; b_1 d_1 \;+\; \ldots \;+\; b_l d_l \;+\; \epsilon$$

where $l = n_{dv} - 1$ and ϵ represents the residuals (the part not explained by the model). The dummy variables are binary and are used to determine the level/type of a categorical variable. See `http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf`.

In ScalaTion, the `ANOVA1` class is implemented using regular multiple linear regression. A data/input matrix X is build from columns corresponding to levels/types for the treatment vector $\mathbf{t}$. As with multiple linear regression, the $\mathbf{y}$ vector holds the response values. Multi-way Analysis of Variance may be performed using the more general `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 treatment/categorical variable vector
@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 ANOVA1 (t: VectoI, y: VectoD, fname_ : Strings = null, technique: RegTechnique = QR)
      extends Regression (VectorD.one (t.dim) +^: Variable.dummyVars (t), y, fname_, null, technique)
      with ExpandableVariable

def expand (t: VectoD, nCat: Int = 1): VectoD =
def predict_ex (zt: VectoD): Double = predict (expand (zt))
```

4.16.3 ANCOVA Implementation

When there is only one categorical/binary variable, $\mathbf{x}$ consists of the usual $k = n - 1$ continuous variables x_j. Assuming there is a single categorical variable, call it t, it will need to be expanded into $n_{dv} - 1$ additional dummy variables.

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

Therefore, the regression equation becomes the following:

$$y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_0 \;+\; b_1 x_1 \;+\; \ldots \;+\; b_k x_k \;+\; b_{k+1} d_0 \;+\; \ldots \;+\; b_{k+l} d_l \;+\; \epsilon \tag{4.38}$$

The dummy variables are binary and are used the determine the level of a categorical variable. See `http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx`.

In general, there may be multiple categorical variables and an expansion will be done for each such variable. Then the data for continuous variable are collected into matrix $X_{\_}$ and the values for the categorical variables are collected into matrix T.

In ScalaTion, 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 $\mathbf{y}$ 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)
      with ExpandableVariable

def expand (zt: VectoD, nCat: Int = nCatVar): VectoD =
def predict_ex (zt: VectoD): Double = predict (expand (zt))
```

4.16.4 Exercises

1. Use the `map2Int` function in the `Converter` object within the `scalation.linalgebra` package to convert the given strings into encoded integers. Turn this vector into a matrix and pass it into the `dummyVars` function to produce the dummy variable columns. Print out the values `xe`, `xm` and `xd`.

   ```
   val x1 = VectorS ("English", "French", "German", "Spanish")
   val (xe, map) = Converter.map2Int (x1)            // map strings to integers
   val xm = MatrixI (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 `map`.

2. Use the `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.

   ```
   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.analyze ()
println (arg.report)

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.

```
// 6 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)                          // new instance
val ze = VectorD (1.0, 20.0, 80.0, 2, 1)                       // encoded and expanded

val xt  = x ++^ t.toDouble
val rg  = new Regression (xt, y)                               // treated as ordinal
val anc = new ANCOVA (x, t, y)                                 // treated as categorical
```

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 $\mathbf{x}$.

$$y = \mathbb{E}\left[y|\mathbf{x}\right] + \epsilon \tag{5.1}$$

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

$$y^* = \text{argmax } P(y|\mathbf{x}) = \mathbb{M}\left[y|\mathbf{x}\right] \tag{5.2}$$

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: VectoI): (Int, String, Double)
def classify (z: VectoD): (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 =
def crossValidateRand (nx: Int = 10, show: Boolean = false): Double =
def fit (y: VectoI, yp: VectoI, k: Int = 2): VectoD =
def fitLabel: Seq [String] = Seq ("acc", "prec", "recall", "kappa")
def fitMap (y: VectoI, yp: VectoI, k: Int = 2): Map [String, String] =
def reset ()
```

For modeling, a user chooses one the of 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
    } // for
    sum / nx.toDouble                                        // return average accuracy
} // crossValidate
```

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.

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

The `crossValidateRand` method calls the following methods:

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

while the `crossValidate` method calls the following methods:

```
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
@param hparam  the hyper-parameters

abstract class ClassifierInt (x: MatriI, y: VectoI, protected var fn: Strings = null,
                              k: Int, protected var cn: Strings = null, hparam: HyperParameter)
         extends ConfusionFit (y, k) with Classifier

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 test (itest: Ints): Double =
def test (xx: MatriI, yy: VectoI): Double =
def eval (xx: MatriD, yy: VectoD = null): ClassifierInt =
def crossValidate (nx: Int = 10, show: Boolean = false): Array [Statistic] =
def crossValidateRand (nx: Int = 10, show: Boolean = false): Array [Statistic] =
def hparameter: HyperParameter = hparam
def report: String =
def classify (z: VectoD): (Int, String, Double) = classify (roundVec (z))
def classify (xx: MatriI = x): VectoI =
def calcCorrelation: MatriD =
def calcCorrelation2 (zrg: Range, xrg: Range): MatriD =
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 $fset \subset \{0, 1, \ldots, n-1\}$ 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 ScalaTion when $k = 2$, the confusion matrix C is configured as follows:

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

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 precicted class labels
@param k   the number class values

class ConfusionMat (y: VectoI, yp: VectoI, k: Int = 2)

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

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: MatriI, y: VectoI, 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: VectoD, 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 $\mathbf{y}$ within the training dataset. These probabilities are estimated based on the frequency ν (`nu` in the code) with which each class value occurs.

$$\nu(\mathbf{y} = c) \;=\; |\{i \,|\, y_i = c\}| \;=\; m_c$$

The right hand side is simply the size of the set containing the instance/row indices where $y_i = c$ for $c = 0, \dots, k-1$. The probability that random variable y equals c can be estimated by the number of elements in the vector $\mathbf{y}$ where y_i equals c divided by the total number of elements.

$$P(y = c) \;=\; \frac{\nu(\mathbf{y} = c)}{m} \;=\; \frac{m_c}{m} \tag{5.3}$$

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.,

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

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

Table 5.1: Class Fequency Vector

y	**0**	**1**
	5	9

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

Table 5.2: Class Probability Vector

y	**0**	**1**
	5/14	9/14

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

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:

```
    @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: VectoI, k: Int = 2, cn_ : Strings = null)
          extends ClassifierInt (null, y, null, k, cn_)

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

The `train` method for this modeling technique is very simple. It takes the parameter `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.

```
def train (itest: IndexedSeq [Int]): NullModel =
{
    val idx = 0 until m diff itest                // training dataset - opposite of tesing
    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 \n 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 `frequency` and `toProbability` are functions from the `Probability` object in the `scalation.analytics` package.

```
def frequency (x: VectoI, k: Int, idx_ : IndexedSeq [Int] = null): VectoI =
{
    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
 *  @see euclid.nmu.edu/~mkowalcz/cs495f09/slides/lesson004.pdf
 */
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
```

2. Build a `NullModel` classifier for the Breast Cancer problem (data in `breast-cancer.arff` file).

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.

$$\mathbf{x} = [x_0, \dots, x_{n-1}],$$

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

It may be complex and less robust to estimate $P(y|\mathbf{x})$ directly. Often it is easier to examine the conditional probability of $\mathbf{x}$ given y. This answers the question of how likely it is that the input data comes from a certain class y. Flipping the perspective can be done using Bayes Theorem.

$$P(y|\mathbf{x}) \;=\; \frac{P(\mathbf{x}|y)\,P(y)}{P(\mathbf{x})}$$

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

$$P(y|\mathbf{x}) \;\propto\; P(\mathbf{x}|y)\,P(y)$$

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

$$P(\mathbf{x}, y) \;=\; P(\mathbf{x}|y)\,P(y) \tag{5.4}$$

One could in principle represent the joint probability $P(\mathbf{x}, y)$ or the conditional probability $P(\mathbf{x}|y)$ in a matrix. Unfortunately, with 30 binary random variables, the matrix would have over one billion rows and exhibit issues with sparsity. Bayesian classifiers will factor the probability and use multiple matrices to represent the probabilities.

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 $P(\mathbf{x}|\,y)$ into the product of their conditional probabilities (independence rule).

$$P(\mathbf{x}|\,y) \;=\; \prod_{j=0}^{n-1} P(x_j|y)$$

Research has shown that even though the assumption that given response/class variable y, the x-variables are independent is often violated by a dataset, Naïve Bayes still tends to perform well [27]. Substituting this factorization in equation 5.4 yields

$$P(\mathbf{x}, y) \;=\; P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{5.5}$$

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 $D_y = \{0, \dots, k-1\}$ that maximizes the probability.

$$y^* \;=\; \underset{y \in \{0,\dots,k-1\}}{\operatorname{argmax}} \; P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{5.6}$$

5.6.2 Estimating Conditional Probabilities

For Integer-based classifiers $x_j \in \{0, 1, ..., vc_j - 1\}$ where vc_j is the value count for the j^{th} variable/feature (i.e., the number of distinct values). The Integer-based Naïve Bayes classifier is trained using an m-by-n data matrix X and an m-dimensional classification vector $\mathbf{y}$. Each data vector/row in the matrix is classified into one of k classes numbered $0, 1, \dots, k-1$. The frequency or number of instances where column vector $\mathbf{x}_{\text{-}j} = h$ and vector $\mathbf{y} = c$ is as follows:

$$\nu(\mathbf{x}_{\text{-}j} = h,\, \mathbf{y} = c) \;=\; |\{i \,|\, x_{ij} = h,\, y_i = c\}|$$

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

$$P(x_j = h \,|\, y = c) \;=\; \frac{\nu(\mathbf{x}_{\text{-}j} = h,\, \mathbf{y} = c)}{\nu(\mathbf{y} = c)} \tag{5.7}$$

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 `Outlook` random variable x_0 is shown in Table 5.3.

$$\nu(\mathbf{x}_{\text{-}0} = h,\, \mathbf{y} = c) \quad \text{for} \quad h \in \{0, 1, 2\},\; c \in \{0, 1\}$$

Table 5.3: JFT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash y$	**0**	**1**
0	2	3
1	0	4
2	3	2

The column sums in the above matrix are 5 and 9, repsectively. The corresponding *Conditional Probability Matrix/Table* (CPT) for random variable x_0, i.e., $P(x_0 = h \,|\, y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum.

Table 5.4: CPT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash y$	**0**	**1**
0	2/5	3/9
1	0	4/9
2	3/5	2/9

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

$$\nu(\mathbf{x}_{\text{-}3} = h,\, \mathbf{y} = c) \quad \text{for} \quad h \in \{0,1\},\ c \in \{0,1\}$$

Table 5.5: JFT for $\mathbf{x}_{\text{-}3}$

$x_3 \backslash y$	**0**	**1**
0	2	6
1	3	3

As expected, the column sums in the above matrix are again 5 and 9, repsectively. The corresponding *Conditional Probability Matrix/Table* for random variable x_0, i.e., $P(x_0 = h \,|\, y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum as shown in table 5.6

Table 5.6: CPT for $\mathbf{x}_{\text{-}3}$

$x_3 \backslash y$	**0**	**1**
0	2/5	6/9
1	3/5	3/9

Similar matrices/tables can be created for the other random variables: Temperature x_1 and Humdity 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:

$$P(x_j = h \,|\, y = c) \;=\; \frac{\nu(\mathbf{x}_{\text{-}j} = h,\, \mathbf{y} = c) \,+\, m_e/vc_j}{\nu(\mathbf{y} = c) \,+\, m_e}$$

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 5.7: CPT for $\mathbf{x}_{\_0}$ with $m_e = 1$

$x_0\backslash y$	**0**	**1**
0	7/18	10/30
1	1/18	13/30
2	10/18	7/30

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 disortion 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 (`p`).

```
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 hypermatix 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:

```
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.

5.6.5 The `classify` Method

A new instance can now be classified by simply matching its values with with those in the class probability table and conditional probability tables and multiplying all the entries. This is done for all k class values and the class with the highest product is chosen.

```
def classify (z: VectoI): (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.

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

5.6.6 Feature Selection

Suppose that x_1 and x_2 are not consisdered 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 5.8: Joint Data-Class Probability

P	**0**	**1**
y	5/14	9/14
z_0	3/5	2/9
z_3	3/5	3/9
$\mathbf{z}, y$	9/70	1/21

The two probabilities are approximately 0.129 for c = 0 (Do not Play) and 0.0476 for c = 1 (Play). The higher probability is for $c = 0$.

To perform feature selection in a systematic way ScalaTion provides 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: MatriI, y: VectoI, 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: VectoI): (Int, String, Double) =
def lclassify (z: VectoI): (Int, String, Double) =
protected def vlog (p: VectoD): VectoD = p.map (log (_))
def reset ()
def printConditionalProb ()
```

`NaiveBayes` **Class**

This class is the same as the one above, butuses 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: MatriI, y: VectoI, 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 probabilties for $z = [2, 2, 1, 1]$.

2. Use SCALATION's Integer-based `NaiveBayes` class to build a classifier for the `ExampleTennis` problem.

```
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 rcall of `NaiveBayes` on the full dataset to that of `NullModel`.

```
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)}")
} // for
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`.

```
println ("nb cv accu = " + nb.crossValidateRand (10, true))    // 10-fold cross-validation
```

5. Compare the confusion matrix, accuracy, precision and rcall 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,

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

we can obtain a better factored approximation (better than Naïve Bayes) by keeping the most important dependencies amongst the random variables. Each 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.

$$P(\mathbf{x}, y) \;=\; P(y) \prod_{j=0}^{n-1} P(x_j | x_{p(j)}, y)$$

Since the root x_r, has no x-parent, it can be factored out as special case.

$$P(\mathbf{x}, y) \;=\; P(y) P(x_r|y) \prod_{j \neq r} P(x_j | x_{p(j)}, y) \tag{5.8}$$

As with Naïve Bayes, the goal is to find an optimal value for the random variable y that maximizes the probability.

$$y^* \;=\; \underset{y \in D_y}{\operatorname{argmax}}\; P(y) P(x_r|y) \prod_{j=0}^{n-1} P(x_j | x_{p(j)}, y)$$

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

$$I(x_j; x_l) \;=\; \sum_{x_j} \sum_{x_l} p(x_j, x_l) \log \frac{p(x_j, x_l)}{p(x_j) p(x_l)} \tag{5.9}$$

The Conditional Mutual Information (CMI) between two random variables x_j and x_l given a third random variable y is

$$I(x_j; x_l | y) \;=\; \tag{5.10}$$

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 `ExampleTennis` 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 5.9: Parent Table

x_j	$x_{p(j)}$
x_0	x_3
x_3	null

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 5.10: Extended JFT for $\mathbf{x}_{\text{-}0}$

$x_0 \backslash x_3, y$	**0, 0**	**0, 1**	**1, 0**	**1, 1**
0	0	3	2	0
1	0	2	0	2
2	2	1	1	1

The column sums are 2, 6, 3, 3, repsectively. 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:

$$P(x_j = h \,|\, x_p = l, y = c) \;=\; \frac{\nu(\mathbf{x}_{\text{-}j} = h,\, \mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c)}{\nu(\mathbf{x}_{\text{-}p} = l,\, \mathbf{y} = c)} \tag{5.11}$$

Table 5.11: Extended CPT for $\mathbf{x}_{\_0}$

$x_0 \backslash x_3, y$	**0, 0**	**0, 1**	**1, 0**	**1, 1**
0	0	1/2	2/3	0
1	0	1/3	0	2/3
2	1	1/6	1/3	1/3

5.7.3 Smoothing

The analog of Laplace smoothing used in Naïve Bayes is the following.

$$P(x_j = h \,|\, x_p = l, y = c) \;=\; \frac{\nu(\mathbf{x}_{\_j} = h,\, \mathbf{x}_{\_p} = l,\, \mathbf{y} = c) \,+\, m_e/vc_j}{\nu(\mathbf{x}_{\_p} = l,\, \mathbf{y} = c) \,+\, m_e}$$

In Friedman's paper [9], he suggests using the marginal distribution rather than uniform (as shown above), which results in the following formula.

$$P(x_j = h \,|\, x_p = l, y = c) \;=\; \frac{\nu(\mathbf{x}_{\_j} = h,\, \mathbf{x}_{\_p} = l,\, \mathbf{y} = c) \,+\, m_e * mp_j}{\nu(\mathbf{x}_{\_p} = l,\, \mathbf{y} = c) \,+\, m_e}$$

where

$$mp_j \;=\; \frac{\nu(\mathbf{x}_{\_j})}{m}$$

5.7.4 The `classify` Method

As with `NaiveBayes`, 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.

```
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 nore efficient "subtractive" cross-validation.

TANBayes0 **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(l)
@param fn_  the names for all features/variables
@param k   the number of classes
@param cn_  the names for all classes
@param me   use m-estimates (me == 0 => regular MLE estimates)
@param vc   the value count (number of distinct values) for each feature

class TANBayes0 (x: MatriI, y: VectoI, fn_ : Strings = null, k: Int = 2, cn_ : Strings = null,
                 me: Double = me_default, protected var vc: Array [Int] = null)
      extends BayesClassifier (x, y, fn_, k, cn_)

def train (itest: IndexedSeq [Int]): TANBayes0 =
def computeParent (idx: IndexedSeq [Int])
override def getParent: VectoI = parent
protected def updateFreq (i: Int)
def maxSpanningTree (ch: Array [SET [Int]], elabel: Map [Pair, Double]): MinSpanningTree =
def computeVcp ()
def classify (z: VectoI): (Int, String, Double) =
def reset ()
def printConditionalProb ()
```

The TANBayes class is similar, but uses a more efficient cross-validation method.

5.7.6 Exercises

1. Use the Integer-based TANBayes to build classifiers for (a) the ExampleTennis problem and (b) the Breast Cancer problem (data in breast-cancer.arff file). Compare its accuracy to that of NullModel and NaiveBayes.

2. Re-engineer TANBayes to use correlation instead of conditional mutual information. Compare the results with the current 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 know as a Bayesian Network Classifier.

5.9.1 Bayesian Network Classifier

A Bayesian Network Classifier [2] is used to classify a discrete input data vector $\mathbf{x}$ by determining which of k classes has the highest Joint Probability of $\mathbf{x}$ and the response/outcome y (i.e., one of the k classes) of occurring.

$$P(y, x_0, x_1, \ldots, x_{n-1})$$

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,

$$y^* \;=\; \underset{y \in D_y}{\text{argmax}}\; P(y) \prod_{j=0}^{n-1} P(x_j | \mathbf{x}_{\mathbf{p}(j)}, y)$$

where $\mathbf{x}_{\mathbf{p}(j)}$ is the vector of features/variables that x_j is dependent on, i.e., its parents. In our model, each variable has dependency with the response variable y (a defacto parent). Note, some more general BN formulations do not distinguish one of the variables to be the response y as we do.

Conditional probabilities are recorded in tables referred to as Conditional Probability Tables (CPTs). Each variable will have a CPT and the number of columns in the table is governed by the number of other variables it is dependent upon. If this number is large, the CPT may become prohibitively large.

5.9.2 Structure Learning

For `TwoNANBayes` the parents of variable x_j are recoded in a vector $\mathbf{x}_{\mathbf{p}(j)}$ of length 0, 1 or 2. Although the restriction to at most 2 parents might seem limiting, the problem of finding the optimal structure is still NP-hard [5].

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 Markov Network

A Markov Network is a probabilistic graphical model where directionality/causality between random variables is not considered, only their bidirectional relationships. In general, let $\mathbf{x}$ be an n-dimensional vector of random variables.

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

Given a data instance $\mathbf{x}$, its likelihood of occurrence is given by the joint proabability.

$$P(\mathbf{x} = \mathbf{x})$$

In order to compute the joint proabability, it needs to be factored based on *conditional independencies.* These conditional independencies may be illustrated graphically, by creating a vertex for each random variable x_i and letting the structure of the graph reflect the conditional independencies,

$$x_i \perp x_k \,|\, \{x_j\}$$

such that removal of the vertices in the set $\{x_j\}$ will disconnect x_i and x_k in the graph. These conditional independencies may be exploited to factor the joint probability, e.g.,

$$P(x_i, x_k \,|\, \{x_j\}) = P(x_i \,|\, \{x_j\})\, P(x_k \,|\, \{x_j\})$$

When two random variables are directedly connected by an undirected edge (denoted $x_i - x_j$) they cannot to separated by removal of other vertices. Together they form an Undirected Graph $G(\mathbf{x}, E)$ where the vertex-set is the set of random variables $\mathbf{x}$ and the edge-set is defined as follows:

$$E \;=\; \{x_i - x_j | \, x_i \text{ and } x_j \text{ are not conditionally independent}\}$$

When the random variables are distributed in space, the Markov Network may from a grid, in which case the network is often referred to as a Markov Random Field (MRF).

5.10.1 Markov Blanket

A vertex in the graph x_i will be conditionally independent of all other vertices, except those in its Markov Blanket. The Markov Blanket for random variable x_i is simply the immediate neighbors of x_i in G:

$$B(x_i) \;=\; \{x_j | \, x_i - x_j \in E\} \tag{5.12}$$

The edges E are selected so that random variable x_i will be conditionally independent of any other $(k \neq i)$ random variable x_k that is not in its Markov blanket.

$$x_i \perp x_k \,|\, B(x_i) \tag{5.13}$$

5.10.2 Factoring the Joint Probability

Factorization of the joint probability is based on the graphical structure of G that reflects the conditional independencies. It has been shown (see the Hammersley-Clifford Theorem) that $P(\mathbf{x})$ may be factored according the set of maximal cliques[1] Cl in graph G.

$$P(\mathbf{x}) \;=\; \frac{1}{Z} \prod_{c \in Cl} \phi_c(\mathbf{x_c}) \tag{5.14}$$

For each clique c in the set Cl, a potential function $\phi_c(\mathbf{x_c})$ is defined. (Potential functions are non-negative functions that are used in place of marginal/conditional probabilities and need not sum to one; hence the normalizing constant Z).

Suppose a graph $G([x_0, x_1, x_2, x_3, x_4], E)$ has two maximal cliques, $Cl = \{[x_0, x_1, x_2], [x_2, x_3, x_4]\}$ then

$$P(\mathbf{x}) \;=\; \frac{1}{Z} \phi_0(x_0, x_1, x_2)\, \phi_1(x_2, x_3, x_4)$$

5.10.3 Exercises

1. Consider the random vector $\mathbf{x} \;=\; [x_0, x_1, x_2]$ with conditional independency

   $$x_0 \perp\; x_1 \,|\, x_2$$

   show that

   $$P(x_0, x_1, x_2) \;=\; P(x_2)\, P(x_0 \,|\, x_2)\, P(x_1 \,|\, x_2)$$

[1] a clique is a set of vertices that are fully connected

5.11 Decision Tree ID3

A Decision Tree (or Classification Tree) classifier [21, 20] will take an input vector $\mathbf{x}$ and classify it, i.e., give one of k class values to y by applying a set of decision rules configured into a tree. Abstractly, the decision rules may be viewed as a function f.

$$y \;=\; f(\mathbf{x}) \;=\; f(x_0, x_1, \ldots, x_{n-1}) \tag{5.15}$$

The `DecisionTreeID3` [17] class implements a Decision Tree classifier using the Iterative Dichotomiser 3 (ID3) algorithm. The classifier is trained using an m-by-n data matrix X and a classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, 1, \ldots, k-1$. Each column in the matrix represents a x-variable/feature (e.g., Humidity). The value count $\mathbf{vc}$ vector gives the number of distinct values per feature (e.g., 2 for Humidity).

5.11.1 Entropy

In decision trees, the goal is to reduce the disorder in decision making. Assume the decision is of the yes(1)/no(0) variety and consider the following decision/classification vectors: $\mathbf{y} = (1, 1, \ldots, 1, 1)$ or $\mathbf{y}' = (1, 0, \ldots, 1, 0)$. In the first case all the decisions are yes, while in the second, three are an equal number of yes and no decisions. One way to measure the level of disorder is Shannon entropy. To compute the Shannon entropy, first convert the m-dimensional decision/classification vector $\mathbf{y}$ into a k-dimensional probability vector $\mathbf{p}$. The `frequency` and `toProbability` functions in the `Probability` object may be used for this task (see `NullModel` from the last chapter).

For the two cases, $\mathbf{p} = (1, 0)$ and $\mathbf{p}' = (.5, .5)$, so computing the Shannon *entropy* $H(\mathbf{p})$,

$$H(\mathbf{p}) = -\sum_{i=0}^{k-1} p_i \, log_2(p_i) \tag{5.16}$$

we obtain $H(\mathbf{p}) = 0$ and $H(\mathbf{p}') = 1$, which indicate that there is no disorder in the first case and maximum disorder in the second case.

```
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.

$$H(\mathbf{p}) = -\sum_{i=0}^{k-1} p_i \, log_k(p_i)$$

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 [20]. For a deeper dive into entropy, relative entropy and mutual information see [6].

5.11.2 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

Table 5.12: Tennis Example

Day	$\mathbf{x_{-0}}$	$\mathbf{x_{-1}}$	$\mathbf{x_{-2}}$	$\mathbf{x_{-3}}$	$\mathbf{y}$
1	2	2	1	0	0
2	2	2	1	1	0
3	1	2	1	0	1
4	0	1	1	0	1
5	0	0	0	0	1
6	0	0	0	1	0
7	1	0	0	1	1
8	2	1	1	0	0
9	2	0	0	0	1
10	0	1	0	0	1
11	2	1	0	1	1
12	1	1	1	1	1
13	1	2	0	0	1
14	0	1	1	1	0

$$H(\mathbf{p}) \;=\; H(\tfrac{5}{14}, \tfrac{9}{14}) = -\tfrac{5}{14}\, log_2(\tfrac{5}{14}) - \tfrac{9}{14}\, log_2(\tfrac{9}{14}) \;=\; 0.9403$$

Recall that the features are Outlook x_0, Temp x_1, Humidity x_2, and Wind x_3. To reduce entropy, find the feature/variable that has the greatest impact on reducing disorder. If feature/variable j is factored into the decision making, entropy is now calculated as follows:

$$\sum_{v=0}^{vc_j-1} \frac{\nu(\mathbf{x_{:j}} = v\}}{m} H(\mathbf{p}_{\mathbf{x_{:j}}=v})$$

where $\nu(\mathbf{x_{:j}} = v\}$ is the frequency count of value v for column vector $\mathbf{x_{:j}}$ in matrix X. The sum is the weighted average of the entropy over all possible vc_j values for variable j.

To see how this works, let us compute new entropy values assuming each feature/variable is used, in turn, as the principal feature for decision making. Starting with feature $j = 0$ (Outlook) with values of Rain (0), Overcast (1) and Sunny (2), compute the probability vector and entropy for each value and weight them by how often that value occurs.

$$\sum_{v=0}^{2} \frac{\nu(\mathbf{x_{-0}} = v)}{m} H(\mathbf{p}_{\mathbf{x_{-0}}=v})$$

For $v = 0$, we have 2 no (0) cases and 3 yes (1) cases $(\mathbf{2}-, \mathbf{3}+)$, for $v = 1$, we have $(\mathbf{0}-, \mathbf{4}+)$ and for $v = 2$, we have $(\mathbf{3}-, \mathbf{2}+)$.

$$\frac{\nu(\mathbf{x_{-0}} = 0)}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=0}) + \frac{\nu(\mathbf{x_{-0}} = 1)}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=1}) + \frac{\nu(\mathbf{x_{-0}} = 2)}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=2})$$

$$\tfrac{5}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=0}) + \tfrac{4}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=1}) + \tfrac{5}{14} H(\mathbf{p}_{\mathbf{x_{-0}}=2})$$

We are left with computing three entropy values:

$$\begin{aligned}
H(\mathbf{p}_{\mathbf{x_{-0}}=0}) &= H(\tfrac{2}{5}, \tfrac{3}{5}) = -\tfrac{2}{5}\, log_2(\tfrac{2}{5}) - \tfrac{3}{5}\, log_2(\tfrac{3}{5}) &= 0.9710 \\
H(\mathbf{p}_{\mathbf{x_{-0}}=1}) &= H(\tfrac{0}{4}, \tfrac{4}{4}) = -\tfrac{0}{4}\, log_2(\tfrac{0}{4}) - \tfrac{4}{4}\, log_2(\tfrac{4}{4}) &= 0.0000 \\
H(\mathbf{p}_{\mathbf{x_{-0}}=2}) &= H(\tfrac{3}{5}, \tfrac{2}{5}) = -\tfrac{3}{5}\, log_2(\tfrac{3}{5}) - \tfrac{2}{5}\, log_2(\tfrac{2}{5}) &= 0.9710
\end{aligned}$$

The weighted average is then 0.6936, so that the drop in entropy (also called information gain) is 0.9403 - 0.6936 = 0.2467. As shown in Table 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

j	**Variable/Feature**	**Entropy**	**Entropy Drop**
0	Outlook	0.6936	0.2467
1	Temperature	0.9111	0.0292
2	Humidity	0.7885	0.1518
3	Wind	0.8922	0.0481

Hence, Outlook ($j = 0$) should be chosen as the principal feature for decision making. As the entropy is too high, make a tree with Outlook (0) as the root and make a branch for each value of Outlook: Rain (0), Overcast (1), Sunny (2). Each branch defines a sub-problem.

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 $\mathbf{x_{-0}} = 0$.

Table 5.14: Sub-problem for node x_0 and branch 0

Day	$\mathbf{x_{-1}}$	$\mathbf{x_{-2}}$	$\mathbf{x_{-3}}$	$\mathbf{y}$
4	1	1	0	1
5	0	0	0	1
6	0	0	1	0
10	1	0	0	1
14	1	1	1	0

If we select Wind ($j = 3$) as the next variable, we obtain the following cases: For $v = 0$, we have $(\mathbf{0}-, \mathbf{3}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x_{-3}}=0} \;=\; (\tfrac{0}{5}, \tfrac{3}{5}) \quad H(\mathbf{p}_{\mathbf{x_{-3}}=0}) = 0$$

For $v = 1$, we have $(\mathbf{2}-, \mathbf{0}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x_{-3}}=1} \;=\; (\tfrac{2}{5}, \tfrac{0}{5}) \quad H(\mathbf{p}_{\mathbf{x_{-3}}=1}) = 0$$

If we stop expanding the tree at this point, we have the following rules.

```
if x0 = 0 then
   if x3 = 0 then yes
   if x3 = 1 then no
if x0 = 1 then yes
if x0 = 2 then no
```

The overall entropy can be calculated as the weighted average of all the leaf nodes.

$$\tfrac{3}{14}\cdot 0 + \tfrac{2}{14}\cdot 0 + \tfrac{4}{14}\cdot 0 + \tfrac{5}{14}\cdot .9710 \;=\; .3468$$

Sub-problem $x_0 = 2$

Note that if $x_0 = 1$, the entropy for this case is already zero, so this node need not be split and remains as a leaf node. There is still some uncertainty left when $x_0 = 2$, so this node may be split. The sub-problem for Outlook: Rain (2) see Table 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

Day	$\mathbf{x_{-1}}$	$\mathbf{x_{-2}}$	$\mathbf{x_{-3}}$	$\mathbf{y}$
1	2	1	0	0
2	2	1	1	0
8	1	1	0	0
9	0	0	0	1
11	1	0	1	1

It should be obvious that $\mathbf{y} = \mathbf{1} - \mathbf{x_{-2}}$. For $v = 0$, we have $(\mathbf{0}-, \mathbf{2}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x_{-2}}=0} \;=\; (\tfrac{0}{5}, \tfrac{2}{5}) \quad H(\mathbf{p}_{\mathbf{x_{-3}}=0}) = 0$$

For $v = 1$, we have $(\mathbf{3}-, \mathbf{0}+)$, so the probability vector and entropy are

$$\mathbf{p}_{\mathbf{x_{-2}}=1} \;=\; (\tfrac{3}{5}, \tfrac{0}{5}) \quad H(\mathbf{p}_{\mathbf{x_{-3}}=0}) = 0$$

At this point, the overall entropy is zero and the decsiion tree is the following (shown as a pre-order traveral 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.

5.11.3 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.

5.11.4 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
```

```
class DecisionTreeID3 (x: MatriI, y: VectoI, fn_: Strings = null, k: Int = 2, cn_: Strings = null,
                       private var vc: Array [Int] = null, td: Int = -1)
      extends ClassifierInt (x, y, fn_, k, cn_)

def frequency (dset: Array [(Int, Int)], value: Int): (Double, VectoI, VectoD) =
def gain (f: Int, path: List [(Int, Int)]): (Double, VectoI) =
def train (itest: IndexedSeq [Int]): DecisionTreeID3 =
def calcEntropy (listOfLeaves: ArrayBuffer [LeafNode]): Double =
def buildTree (path: List [(Int, Int)], depth: Int): FeatureNode =
def prune (threshold: Double, fold: Int = 5): DecisionTreeID3 =
def compareModel (folds: Int, threshold: Double) =
def printTree ()
def classify (z: VectoI): (Int, String, Double) =
def reset ()
```

5.11.5 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.

```
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 decisions trees.

```
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.

5.12 Hidden Markov Model

A Hidden Markov Model (HMM) provides a natural way to study a system with an internal state and external observations. One could image looking at a flame and judging the temperature (internal state) of the flame by its color (external observation). When this is treated as a discrete problem, an HMM may be used; whereas, as a continuous problem, a Kalman Filter may be used (see section FIX). For HMMs, we assume that the internal state is unknown (hidden), but may be predicted by from the observations.

Consider two discrete-valued, discrete-time stochastic processes. The first process represents the internal state of a system

$$\{x_t : t \in \{0, \dots T-1\}\}$$

while the second process represents corresponding observations of the system

$$\{y_t : t \in \{0, \dots T-1\}\}$$

The internal state influences the observations. In a deterministic setting, one might imagine

$$y_t = f(x_t)$$

Unfortunately, since both x_t and y_t are both stochastic processes, their trajectories need to be described probabilistically. For tractability and because it often suffices, the assumption is made that the state x_t is only significantly influenced by its previous state x_{t-1}.

$$P(x_t | x_{t-1}, x_{t-2}, x_0) = P(x_t | x_{t-1})$$

In other words, the transitions from state to state are governed by a *discrete-time Markov chain* and characterized by *state-transtion probability matrix* $A = [a_{ij}]$, where

$$a_{ij} = P(x_t = j | x_{t-1} = i)$$

The influence of the state upon the observation is also characterized by *emission probability matrix* $B = [b_{kj}]$, where

$$b_{jk} = P(y_t = k | x_t = j)$$

is the conditional probability of the observation being k when the state is j. This represents a second simplifying assumption that the observation is effectively independent of prior states or observations. To predict the evolution of the system, it is necessary to characterize the initial state of the system x_0.

$$\pi_j = P(x_t = j)$$

The dynamics of an HMM model is thus represented by two matrices A, B and an *intial state probability vector* $\boldsymbol{\pi}$.

5.12.1 Example Problem

Let the system under study be a lane of road with a sensor to count traffic flow (number of vehicles passing the sensor in a five minute period). As a simple example, let the state of the road be whether or not there is an accident ahead. In other words, the state of road is either 0 (**N**o-accident) or 1 (**A**ccident). The only information avialable is the traffic counts and of course historical information for training an HMM model. Suppose the chance of an accident ahead is 10%.

$$\boldsymbol{\pi} \;=\; [0.9, 0.1]$$

From historical information, two transition probabilities are estimated: the first is for the transition from no accident to accident which is 20%; the second from accident to no-accident state (i.e., the accident has been cleared) which is 50% (i.e., probability of one half that the accident will be cleared by the next time increment). The number of states $n = 2$. Therefore, the state-transition probability matrix A is

$$\begin{bmatrix} 0.8 & 0.2 \\ 0.5 & 0.5 \end{bmatrix}$$

As A maps states to state, A is an n-by-n matrix.

Clearly, the state will influence the traffic flow (tens of cars per 5 minutes) with possible values of 0, 1, 2, 3. The number of observed values $m = 4$. Again from historical data the emission probability matrix B is estimated to be

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.2 & 0.2 & 0.1 \end{bmatrix}$$

As B maps states to observed values, B is an n-by-m matrix.

One question to address is, given a time series (observations sequence), what corresponding sequence of states gives the highest probability of occurrence to the observed sequences.

$$\mathbf{y} \;=\; [3, 3, 0]$$

This may be done by computing the joint probability $P(\mathbf{x}, \mathbf{y})$

$$\begin{aligned}
P(NNN, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{00} \cdot b_{00} = 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.8 \cdot 0.1 = 0.009216 \\
P(NNA, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{01} \cdot b_{10} = 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.2 \cdot 0.5 = 0.011520 \\
P(NAN, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{01} \cdot b_{13} \cdot a_{10} \cdot b_{00} = 0.9 \cdot 0.4 \cdot 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.1 = 0.000360 \\
P(NAA, \mathbf{y}) &= \pi_0 \cdot b_{03} \cdot a_{01} \cdot b_{13} \cdot a_{11} \cdot b_{10} = 0.9 \cdot 0.4 \cdot 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.5 = 0.001800 \\
P(ANN, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{00} \cdot b_{00} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.8 \cdot 0.1 = 0.000160 \\
P(ANA, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{01} \cdot b_{10} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.2 \cdot 0.5 = 0.000200 \\
P(AAN, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{10} \cdot b_{00} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.1 = 0.000025 \\
P(AAA, \mathbf{y}) &= \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{11} \cdot b_{10} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.5 = 0.000125
\end{aligned}$$

The state giving the highest probability is $\mathbf{x} = NNA$. The marginal probability of the observed sequence $P(\mathbf{y})$ can be computed by summing over all eight states.

$$P(\mathbf{y}) \;=\; \sum_{\mathbf{x}} P(\mathbf{x}, \mathbf{y}) \;=\; 0.023406$$

The algorithms given in the subsections below are adapted from [25]. For these algorithms, we divide the time series/sequence of obseverations into two parts (past and future).

$$\mathbf{y^{t-}} \;=\; [y_0, y_1, y_t]$$
$$\mathbf{y^{t+}} \;=\; [y_{t+1}, y_{t+2}, y_{T-1}]$$

They allow one to calculate (1) the probability of arriving in a state at time t with observations $\mathbf{y^{t-}}$, (2) the conditional probability of seeing future observations $\mathbf{y^{t+}}$ from a given state at time t, and (3) the conditional probability of being in a state at time t given all the observations $\mathbf{y}$.

5.12.2 Forward Algorithm

For longer observation sequences/time series, the approach of summing over all possible state vectors (of which there are n^T) will become intractable. Much of the computation is repetitive anyway. New matrices A, B and Γ are defined to save such intermediate calculations.

The forward algorithm (α-pass) computes the A matrix. The probability of being in state j at time t having observations up to time t is given by

$$\alpha_{tj} \;=\; P(x_t = j, \mathbf{y} = \mathbf{y^{t-}})$$

Computation of α_{tj} may be done efficiently using the following recurrence.

$$\alpha_{tj} \;=\; b_{j,y_t} \sum_{i=0}^{n-1} \alpha_{t-1,i}\, a_{ij} \;=\; b_{j,y_t}\, [\boldsymbol{\alpha}_{t-1} \cdot \boldsymbol{a}_{:j}]$$

To get to state j at time t, the system must transition from some state i at time $t-1$ and at time t emit the value y_t. These values may be saved in a T-by-n matrix $\mathrm{A} = [\alpha_{tj}]$ and efficiently computed by moving forward in time.

```
def forwardEval0 (): MatrixD =
{
    for (j <- rstate) alp(0, j) = pi(j) * b(j, y(0))
    for (t <- 1 until tt) {
        for (j <- rstate) {
            alp(t, j) = b(j, y(t)) * (alp(t-1) dot a.col(j))
        } // for
    } // for
    alp
 } // forwardEval0
```

The marginal probability is now simply the sum of the elements in the last row of the α matrix

$$P(\mathbf{y}) \;=\; \sum_{j=0}^{n} \alpha_{T-1,j}$$

ScalaTion also provides a `forwardEval` method that uses scaling to avoid underflow.

5.12.3 Backward Algorithm

The backward algorithm (β-pass) computes the B matrix. The conditional probability of having future observations after time t ($\mathbf{y} = \mathbf{y^{t+}}$) given the current state $x_t = i$ is

$$\beta_{ti} \;=\; P(\mathbf{y} = \mathbf{y^{t+}} | x_t = i)$$

Computation of β_{tj} may be done efficiently using the following recurrence.

$$\beta_{ti} \;=\; \sum_{j=0}^{n-1} a_{ij}\, b_{j,y_{t+1}} \beta_{t+1,j}$$

From state i at time t, the system must transition to some state j at time $t+1$ and at time $t+1$ emit the value y_{t+1}. These values may be saved in a T-by-n matrix $\mathrm{B} = [\beta_{ti}]$ and efficiently computed by moving backward in time.

```
def backwardEval0 (): MatrixD =
{
    for (i <- rstate) bet(tt-1, i) = 1.0
    for (t <- tt-2 to 0 by -1) {
        for (i <- rstate) {
            bet(t, i) = 0.0
            for (j <- rstate) bet(t, i) += a(i, j) * b(j, y(t+1)) * bet(t+1, j)
        } // for
    } // for
    bet
 } // backwardEval0
```

ScalaTion also provides a `backwardEval` method that uses scaling to avoid underflow.

5.12.4 Viterbi Algorithm

The Viterbi algorithm (γ-pass) computes the Γ matrix. The conditional probability of the state at time t being i, given all observations ($\mathbf{y} = \mathbf{y}$) is

$$\gamma_{ti} \;=\; P(x_t = i | \mathbf{y} = \mathbf{y})$$

As α_{ti} captures the probability up to time t and β_{ti} captures the probability after time t, the conditional probability may be calculated as follows:

$$\gamma_{ti} \;=\; \frac{\alpha_{ti}\beta_{ti}}{P(\mathbf{y})}$$

In ScalaTion, the $\Gamma = [\gamma_{ti}]$ matrix is calculated using the Hadamard product.

```
def gamma (alp: MatriD, bet: MatriD): MatriD = (alp ** bet) / probY (alp)
```

The conditional probability of being in state i at time t and transitioning to state j at time $t+1$ given all observations ($\mathbf{y} = \mathbf{y}$) is

$$\gamma_{tij} \;=\; P(x_t = i, x_{t+1} = j | \mathbf{y} = \mathbf{y})$$

Note, this equation is not defined for the last time point $T-1$, since there is no next state. Getting to state i at time t is characterized by α_{ti}, the probability of the state transitioning to from i to j is characterized a_{ij}, the probability of emitting y_{t+1} from state j at time $t+1$ is $b_{j,y_{t+1}}$, and finally going from state j to the end is characterized by $\beta_{t+1,j}$.

$$\gamma_{tij} \;=\; \frac{\alpha_{ti} a_{ij} b_{j,y_{t+1}} \beta_{t+1,j}}{P(\mathbf{y})}$$

The Viterbi Algorithm `viterbiDecode` computes the Γ matrix (`gam` in code) from scaled versions of `alp` and `bet`. It also computes the $\Gamma = [\gamma_{tij}]$ tensor (`gat` in code).

5.12.5 Training

The `train` method will call `forwardEval`, `backwardEval` and `viterbiDecode` to calculate updated values for the A, B and Γ matrices as well as for the Γ tensor. These values are used to `reestimate` the π, A and B parameters.

```
def train (itest: Ints): HiddenMarkov =
{
    var oldLogPr = 0.0
    for (it <- 1 to MIT) {
        val logPr = logProbY (true)
        if (logPr > oldLogPr) {
            oldLogPr = logPr
            forwardEval ()
            backwardEval ()
            viterbiDecode ()
            reestimate ()
        } else {
            println (s"train: HMM model converged after $it iterations")
            return this
        } // if
    } // for
    println (s"train: HMM model did not converged after $MIT iterations")
    this
} // train
```

The training loop will terminate early when there is no improvement to $P(\mathbf{y})$. To avoid underflow $-\log(P(\mathbf{y})$ is used.

5.12.6 Reestimation of Parameters

The parameters for an HMM model are π, A and B and may be adjusted to maximize the probability of seeing the observation vector $P(\mathbf{y})$. Since $\alpha_{0i} = \pi_i b_{i,y(0)}$, we can reestimation π as follows:

$$\pi_i \;=\; \frac{\alpha_{0i}}{b_{i,y_0}} \;=\; \gamma_{0i}$$

The A matrix can re-estimated as follows:

$$a_{ij} \;=\; \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{ti}}$$

Similarly, the B matrix can re-estimated as follows:

$$b_{ik} \;=\; \frac{\sum_{t=0}^{T-1} I_{y_t=k}\, \gamma_{ti}}{\sum_{t=0}^{T-1} \gamma_{ti}}$$

The detailed derivations are left to the exercises.

`HiddenMarkov` Class

Class Methods:

```
@param y    the observation vector/observed discrete-valued time series
@param m    the number of observation symbols/values {0, 1, ... m-1}
@param n    the number of (hidden) states in the model
@param cn_  the class names for the states, e.g., ("Hot", "Cold")
@param pi   the probabilty vector for the initial state
@param a    the state transition probability matrix (n-by-n)
@param b    the observation probability matrix (n-by-m)

class HiddenMarkov (y: VectoI, m: Int, n: Int, cn_ : Strings = null,
                    private var pi: VectoD = null,
                    private var a:  MatriD = null,
                    private var b:  MatriD = null)
      extends ClassifierInt (null, y, null, n, cn_)

    override def size: Int = n
    def parameter: VectoD = pi
    def parameters: (MatriD, MatriD) = (a, b)
    def jointProb (x: VectoI): Double =
    def forwardEval0 (): MatriD =
    def probY (scaled: Boolean = false): Double =
    def logProbY (scaled: Boolean = false): Double =
    def backwardEval0 (): MatriD =
    def gamma (): MatriD = (alp ** bet) / probY ()
    def forwardEval (): MatriD =
    def getC: VectoD = c
    def backwardEval (): MatriD =
    def viterbiDecode (): MatriD =
    def reestimate ()
    def train (itest: Ints): HiddenMarkov =
    override def report: String =
    def classify (z: VectoI): (Int, String, Double) =
    def reset () {}
```

5.12.7 Exercises

1. Show that for $t \in \{0, \dots T-2\}$,

$$\gamma_{ti} \;=\; \sum_{j=0}^{n-1} \gamma_{tij}$$

2. Show that

$$\pi_i \;=\; \frac{\alpha_{0i}}{b_{i,y_0}} \;=\; \gamma_{0i}$$

3. Show that

$$a_{ij} \;=\; \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{ti}}$$

4. Show that

$$b_{ik} \;=\; \frac{\sum_{t=0}^{T-1} I_{y_t=k}\, \gamma_{ti}}{\sum_{t=0}^{T-1} \gamma_{ti}}$$

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. Techniquely, 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
@param hparam  the hyper-parameters

abstract class ClassifierReal (x: MatriD, y: VectoI, protected var fn: Strings,
                               k: Int, protected var cn: Strings, hparam: HyperParameter)
         extends ConfusionFit (y, k) with Classifier

def vc_default: Array [Int] = Array.fill (n)(2)
def size: Int = m
def test (itest: Ints): Double =
def test (xx: MatriD, yy: VectoI): Double =
def eval (xx: MatriD, yy: VectoD = null): ClassifierReal =
def crossValidate (nx: Int = 10, show: Boolean = false): Array [Statistic] =
def crossValidateRand (nx: Int = 10, show: Boolean = false): Array [Statistic] =
def hparameter: HyperParameter = hparam
def report: String =
def classify (z: VectoI): (Int, String, Double) = classify (z.toDouble)
def classify (xx: MatriD = x): VectoI =
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 Naïve Bayes Classifier, which is the most commonly used such classifier for continuous input data. The classifier is trained using a data matrix X and a classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, 1, \dots, k-1$.

Class probabilities are calculated based on the population of each class in the training-set. Relative probabilities are computed by multiplying these by values computed using conditional density functions based on the Normal (Gaussian) distribution. The classifier is naïve, because it assumes feature independence and therefore simply multiplies the conditional densities.

Starting with main results from the section on Naïve Bayes (equation 4.5),

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

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^* that maximizes the product of the conditional densities multiplied by the class probability.

$$y^* \;=\; \underset{y \in \{0,\dots,k-1\}}{\text{argmax}} \; P(y) \prod_{j=0}^{n-1} f(x_j|y) \tag{6.1}$$

Although the formula assumes the conditional independence of x_js, the technique can be applied as long as correlations are not too high.

Using the Gaussian assumption, the conditional density of x_j given y, is approximated by estimating the two parameters of the `Normal` distribution,

$$x_j|y \;\sim\; Normal(\mu_c, \sigma_c^2)$$

where class $c \in \{0, 1, \dots, k-1\}$, $\mu_c = \mathbb{E}[x|y = c]$ and $\sigma_c^2 = \mathbb{V}[x|y = c])$. Thus, the conditional density function is

$$f(x_j|y = c) \;=\; \frac{1}{\sqrt{2\pi}\sigma_c} e^{-\frac{(x-\mu_c)^2}{2\sigma_c^2}}$$

Class probabilities $P(y = c)$ may be estimated as $\frac{m_c}{m}$, where m_c is the frequency count of the number of occurrences of c in the class vector $\mathbf{y}$. Conditional densities are needed for each of the k class values, for each of the n variables (each x_j) (i.e., kn are needed). Corresponding means and variances may be estimated as follows:

$$\hat{\mu}_{cj} \;=\; \frac{1}{m_c} \sum_{i=0}^{m-1} (x_{ij}|y_i = c)$$

$$\hat{\sigma}_{cj}^2 \;=\; \frac{1}{m_c - 1} \sum_{i=0}^{m-1} ((x_{ij} - \hat{\mu}_{cj})^2|y_i = c)$$

Using conditional density (cd) functions estimated in the `train` function (see code for details), an input vector $\mathbf{z}$ 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 whether they should pass quality control based on *curvature* and `diameter` tolerances. See `people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html` for details.

```
    // features/variable:
    // x1: curvature
    // x2: diameter
    // y:  classification: pass (0), fail (1)
    //                             x1    x2    y
    val xy = new MatrixD ((7, 3), 2.95, 6.63, 0,     // joint data matrix
                                  2.53, 7.79, 0,
                                  3.57, 5.65, 0,
                                  3.16, 5.47, 0,
                                  2.58, 4.46, 1,
                                  2.16, 6.22, 1,
                                  3.27, 3.52, 1)

    val 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 $\mathbf{x}$ is two-dimensional $[1, x_1]$. Again, the goal is to fit the parameter vector $\mathbf{b}$ in the regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} + \epsilon \;=\; b_0 + b_1 x_1 + \epsilon$$

where ϵ represents the residuals (the part not explained by the model). This looks like simple linear regression, with the difference being that the response variable y is binary ($y \in \{0, 1\}$). Since y is binary, minimizing the distance, as was done before, may not work well. First, instead of focusing on $y \in \{0, 1\}$, we focus on the conditional probability of success $p_y(\mathbf{x}) \in [0, 1]$, i.e.,

$$p_y(\mathbf{x}) \;=\; P(y = 1|\mathbf{x})$$

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/a412dfbd88b3db70b74b`). Try using `mpg` to predict/classify the car's engine as either `V`-shaped(0) or `S`traight(1), as in V-6 or S-4. First, use SimpleRegression to predict $p_y(\mathbf{x})$ where y is `V/S` and x_1 is `mpg`, ($\mathbf{x} = [1, x_1]$). Plot y versus x_1 and then add a vector to the plot for the predicted values for p_y. Utilizing simple linear regression to predict $p_y(\mathbf{x})$ would correspond to the following equation.

$$p_y(\mathbf{x}) \;=\; b_0 + b_1 x_1$$

6.3.2 Logistic Function

The linear relationship between y and x_1 may be problematic, in the sense that there is likely to be a range of rapid transition before which loan default is likely and after which loan repayment is likely. Similarly, there is rapid transition from `S`(1) to `V`(0) as `mpg` increases. This suggests that some "S-curve" function such as the *logistic* function may be more useful. The standard logistic function (sigmoid function) is

$$\text{logistic}(z) \;=\; \frac{1}{1 + e^{-z}} \;=\; \frac{e^z}{1 + e^z} \tag{6.2}$$

Letting $z = b_0 + b_1 x_1$, we obtain

$$p_y(\mathbf{x}) \;=\; \text{logistic}(b_0 + b_1 x_1) \;=\; \frac{e^{b_0 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}} \tag{6.3}$$

6.3.3 Logit Function

The goal now is to transform the right hand side into the usual linear form (i.e., $\mathbf{b} \cdot \mathbf{x}$).

$$p_y(\mathbf{x}) \;=\; \frac{e^{\mathbf{b} \cdot \mathbf{x}}}{1 + e^{\mathbf{b} \cdot \mathbf{x}}}$$

Multiplying through by $1 + e^{\mathbf{b}\cdot\mathbf{x}}$ gives

$$p_y(\mathbf{x}) + e^{\mathbf{b}\cdot\mathbf{x}} p_y(\mathbf{x}) \;=\; e^{\mathbf{b}\cdot\mathbf{x}}$$

Solving for $e^{\mathbf{b}\cdot\mathbf{x}}$ yields

$$e^{\mathbf{b}\cdot\mathbf{x}} \;=\; \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})}$$

Taking the natural logarithm of both sides gives

$$\ln \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} \;=\; \mathbf{b}\cdot\mathbf{x} \;=\; b_0 + b_1 x_1 \tag{6.4}$$

where the function on the left hand side is called the *logit* function.

$$\text{logit}(p_y(\mathbf{x})) \;=\; \mathbf{b}\cdot\mathbf{x} \;=\; b_0 + b_1 x_1 \tag{6.5}$$

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 $\mathbf{b}$ (treated as a random vector) will impact the quality of the model. Define a function of $\mathbf{b}$ that will be maximized when the parameters are ideally set to generate the observed data.

6.3.5 Likelihood Function

We can think of this function as the likelihood of $\mathbf{b}$ given the predictor vector $\mathbf{x}$ and the response variable y.

$$L(\mathbf{b}|\mathbf{x}, y)$$

In this case, $y \in \{0, 1\}$, so if we estimate the likelihood for a single data instance (or row), we have

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; p_y(\mathbf{x})^y \, (1 - p_y(\mathbf{x}))^{1-y} \tag{6.6}$$

If $y = 1$, then $L = p_y(\mathbf{x})$ and otherwise $L = 1 - p_y(\mathbf{x})$. These are the probabilities for the two outcomes for a Bernoulli random variable (and equation 6.5 concisely captures both).

For each instance $i \in \{0, \ldots, m-1\}$, a similar factor is created. These are multiplied together for all the instances (in the dataset, or training or testing). The likelihood of $\mathbf{b}$ given the predictor matrix X and and the response vector $\mathbf{y}$ is then

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; \prod_{i=0}^{m-1} p_y(\mathbf{x}_i)^{y_i} \, (1 - p_y(\mathbf{x}_i))^{1-y_i} \tag{6.7}$$

6.3.6 Log-likelihood Function

To reduce round-off errors, a log (e.g., natural log, ln) is taken

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \ln(p_y(\mathbf{x_i})) + (1 - y_i)\ln(1 - p_y(\mathbf{x_i}))$$

This is referred as the log-likelihood function. Collecting y_i terms give

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \ln\frac{p_y(\mathbf{x_i})}{1 - p_y(\mathbf{x_i})} + \ln(1 - p_y(\mathbf{x_i}))$$

Substituting $\mathbf{b} \cdot \mathbf{x_i}$ for logit$(p_y(\mathbf{x_i}))$ gives

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \, \mathbf{b} \cdot \mathbf{x_i} + \ln(1 - p_y(\mathbf{x_i}))$$

Now substituting $\dfrac{e^{\mathbf{b}\cdot\mathbf{x_i}}}{1 + e^{\mathbf{b}\cdot\mathbf{x_i}}}$ for $p_y(\mathbf{x_i})$ gives

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \, \mathbf{b} \cdot \mathbf{x_i} - \ln(1 + e^{\mathbf{b}\cdot\mathbf{x_i}}) \tag{6.8}$$

Multiplying the log-likelihood by -2 makes the distribution approximately Chi-square [?].

$$-2l \;=\; -\,2 \sum_{i=0}^{m-1} y_i \, \mathbf{b} \cdot \mathbf{x_i} - \ln(1 + e^{\mathbf{b}\cdot\mathbf{x_i}})$$

Or since $\mathbf{b} = [b_0, b_1]$,

$$-2l \;=\; -\,2 \sum_{i=0}^{m-1} y_i(b_0 + b_1 x_{i1}) - \ln(1 + e^{b_0 + x_{i1}})$$

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.

$$-\,2l \;=\; -\,2 \sum_{i=0}^{m-1} y_i \beta_i - \beta_i - \ln(e^{-\beta_i} + 1) \tag{6.9}$$

6.3.7 Computation in ScalaTion

The computation of $-2l$ is carried out in ScalaTion via the `ll` method. It loops through all instances computing β_i (`bx` in the code) and summing all the terms given in equation 6.9.

```
def ll (b: 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
```

6.3.8 Making a Decision

So far, `SimpleLogisticRegression` is a model for predicting $p_y(\mathbf{x})$. In order to use this for binary classification a decision needs to be made: deciding on either 0 (no) or 1 (yes). A natural way to do this is to choose 1 when $p_y(\mathbf{x})$ exceeds 0.5.

```
override def classify (z: VectoD): (Int, String, Double) =
{
    val p_y = sigmoid (b dot z)
    val c = if (p_y > cThresh) 1 else 0
    (c, cn(c), p_y)
} // classify
```

In some cases, this may results in imbalance between false positives and false negatives. Quality of Fit (QoF) measures may improve by tuning the classification/descision threshold `cThresh`. Decreasing the threshold pushes false negatives to false positives. Increasing the threshold does the opposite. Ideally, the tuning of the threshold will also push more cases into the diagonal of the confusion matrix and minimize errors. Finally, in some cases it may be more important to reduce one more than the other, false negatives vs. false positives (see the exercises).

`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 */ }
```

6.3.9 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 `ll` function. Use the `x` matrix (`one`, `mpg`) and `y` vector (`V/S`) from `SimpleLogisticRegressionTest`.

```
// 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, 1, 0, 0, 0, 0, 0,
                 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1)
```

3. Compare the effectiveness of `SimpleLogisticRegression` versus `SimpleRegression` on the mtcars dataset. See `SimpleLogisticRegressionTest`.

4. If the treatment for a disease is risky and consequences of having the disease are minimal, would you prefer to focus on reducing false positives or false negatives?

5. If the treatment for a disease is safe and consequences of having the disease may be severe, would you prefer to focus on reducing false positives or false negatives?

6.4 Logistic Regression

The `LogisticRegression` class supports logistic regression. In this case, $\mathbf{x}$ may be multi-dimensional $[1, x_1, \dots, x_k]$. Again, the goal is to fit the parameter vector $\mathbf{b}$ in the regression equation

$$y \;=\; \mathbf{b} \cdot \mathbf{x} \;+\; \epsilon \;=\; b_0 \;+\; b_1 x_1 \;+\; \dots \;+\; b_k x_k \;+\; \epsilon$$

where ϵ represents the residuals (the part not explained by the model). This looks like multiple linear regression. The difference being that the response variable y is binary ($y \in \{0, 1\}$). Since y is binary, minimizing the distance, as was done before may not work well. First, instead of focusing on $y \in \{0, 1\}$, we focus on the conditional probability of success $p_y(\mathbf{x}) \in [0, 1]$, i.e.,

$$p_y(\mathbf{x}) \;=\; P(y = 1|\mathbf{x})$$

Still, $p_y(\mathbf{x})$ is bounded, while $\mathbf{b} \cdot \mathbf{x}$ is not. We therefore, need a transformation, such as the logit transformation, and fit $\mathbf{b} \cdot \mathbf{x}$ to this function. Treating this as a GZLM problem,

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

$$g(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x}$$

we let the link function $g \;=\;$ logit.

$$\text{logit}(\mu(\mathbf{x})) \;=\; \ln \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} \;=\; \mathbf{b} \cdot \mathbf{x}$$

This is the logit regression equation. Second, instead of minimizing the sum of squared errors, we wish to maximize the likelihood of predicting correct outcomes. For the ith training case $\mathbf{x_i}$ with outcome y_i, the likelihood function is based on the Bernoulli distribution.

$$p_y(\mathbf{x}_i)^{y_i} (1 - p_y(\mathbf{x}_i))^{1-y_i}$$

The overall likelihood function is the product over all m cases. The equation is the same as 6.6 from the last section.

$$L(\mathbf{b}|\mathbf{x}, y) \;=\; \prod_{i=0}^{m-1} p_y(\mathbf{x}_i)^{y_i} \, (1 - p_y(\mathbf{x}_i))^{1-y_i} \tag{6.10}$$

Following the same derivation steps, will give the same log-likelihood that is in equation 6.7.

$$l(\mathbf{b}|\mathbf{x}, y) \;=\; \sum_{i=0}^{m-1} y_i \, \mathbf{b} \cdot \mathbf{x_i} - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x_i}}) \tag{6.11}$$

Again, multiplying the log-likelihood function by -2 makes the distribution approximately Chi-square.

$$-2l \;=\; -\,2 \sum_{i=0}^{m-1} y_i \, \mathbf{b} \cdot \mathbf{x_i} - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x_i}})$$

The likelihood can be maximized by minimizing $-2l$, which is a non-linear function of the parameter vector $\mathbf{b}$. Various optimization techniques may be used to search for optimal values for $\mathbf{b}$. Currently, ScalaTion uses BFGS, a popular general-purpose QuasiNewton NLP solver. Other possible optimizers include LBFGS and IRWLS. For a more detailed derivation, see `http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf`.

`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: MatriD, y: VectoI, fn_ : Strings = null,
                          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]): LogisticRegression =
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 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 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 [13] section 4.6.2.

2. Use `Logistic Regression` to classify whether a customer will purchase caraavan insurance. The `Caravan` dataset is in the ISLR library, see [13] section 4.6.6.

6.5 Simple Linear Discriminant Analysis

The `SimpleLDA` class support Linear Discriminant Analysis which is useful for multiway classification of continuously valued data. The response/classification variable can take on k possible values, $y \in \{0, 1, \ldots, k-1\}$. 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 is 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.

$$x|y \ \sim \ Normal(\mu_c, \sigma^2)$$

where class $c \in \{0, 1, \ldots, k-1\}$, $\mu_c = \mathbb{E}\,[x|y=c]$ and σ^2 is the pooled variance (weighted average of $\mathbb{V}\,[x|y=c]$). Thus, the conditional density function is

$$f(x|y=c) \ = \ \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}$$

Substituting into eqaution 6.10 gives

$$P(y|x) \ = \ \frac{\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}\, P(y)}{f(x)} \tag{6.13}$$

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 proabability expressed in equation 6.11.

$$P(y|x) \ \propto \ \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}\, P(y)$$

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).

$$\ln(P(y|x)) \;\propto\; \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} \;+\; ln(P(y)) \tag{6.14}$$

The right hand side functions in 4.12 are linear in x and are called discriminant functions $\delta_c(x)$.

Given training data vectors $\mathbf{x}$ and $\mathbf{y}$, define $\mathbf{x_c}$ (or `xc` in the code) to be the vector of all x_i values where $y_i = c$ and let its length be denoted by m_c. Now the k means may be estimated as follows:

$$\hat{\mu}_c \;=\; \frac{\mathbf{1}\cdot\mathbf{x_c}}{m_c}$$

The common variance my be estimated using a pooled variance estimator.

$$\hat{\sigma}^2 \;=\; \frac{1}{m-k}\sum_{c=0}^{k-1} ||\mathbf{x_c} - \mu_c||^2$$

Finally, $\frac{m_c}{m}$ can be used to estimate $P(y)$.

These can easily be translated into ScalaTion code. Most of the calculations are done in the **`train`** method. It estimates the class probability vector **`py`**, the group means vector **`mu`** and the pooled variance. The vectors **`term1`** and **`term2`** capture the x-term (μ_c/σ^2) and the constant term ($\mu_c^2/2\sigma^2 - ln(P(y))$) in equation 6.12.

```
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 **`classify`** method simply multiplies the first by **`z(0)`** and subtracts the second. Then it finds the **`argmax`** of the **`delta`** vector to return the class with the maximum **`delta`**, which corresponds the most probable classification.

$$y^* \;=\; \text{argmax}_c \frac{z\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + ln(P(y)) \tag{6.15}$$

```
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 (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.SimpleLDATest2`.

6.6 Linear Discriminant Analysis

Like `SimpleLDA`, the `LDA` class support Linear Discriminant Analysis that is used for multiway classification of continuously valued data. Similarly, the response/classification variable can take on k possible values, $y \in \{0, 1, \ldots, k-1\}$. Unlike `SimpleLDA`, this class is intended for cases where the feature vector $\mathbf{x}$ is multi-dimensional. The classification $y = c$ is chosen to maximize the conditional probability of class y given the n-dimensional data/feature vector $\mathbf{x}$.

$$P(y|\mathbf{x}) \;=\; \frac{f(\mathbf{x}|y)\,P(y)}{f(\mathbf{x})} \tag{6.16}$$

where

$$f(\mathbf{x}) \;=\; \sum_{c=0}^{k-1} f(\mathbf{x}|y=c)P(y=c)$$

In the multi-dimensional case, $\mathbf{x}|y$ has a multivariate Gaussian distribution, $Normal(\boldsymbol{\mu}_c, \Sigma)$, where $\boldsymbol{\mu}_c$ are the mean vectors $\mathbb{E}\,[\mathbf{x}|y=c]$ and Σ is the common covariance matrix (weighted average of $\mathbb{C}\,[\mathbf{x}|y=c]$. The conditional density function is given by

$$f(\mathbf{x}|y=c) \;=\; \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^t\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c)}$$

Dropping factors independent of c and multiplying by $P(y=c)$ gives

$$f(\mathbf{x}|y=c)P(y=c) \;\propto\; e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^t\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c)}P(y=c)$$

Taking the natural logarithm

$$\ln(P(y|x)) \;\propto\; -\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^t\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_c) + \ln(P(y=c))$$

The discriminant functions are obtained by multiplying out and again dropping terms independing of c.

$$\delta_c(\mathbf{x}) \;=\; \mathbf{x}^t\Sigma^{-1}\boldsymbol{\mu}_c - \frac{\boldsymbol{\mu}_c{}^t\Sigma^{-1}\boldsymbol{\mu}_c}{2} + \ln(P(y=c)) \tag{6.17}$$

As in the last section, the means for each class c ($\boldsymbol{\mu}_c$), the common covariance matrix (Σ), and the class probabilities ($P(y)$) must be estimated.

Class Methods:

```
@param x    the real-valued training/test data vectors stored as rows of a matrix
@param y    the training/test classification vector, where y_i = class for row i of the matrix x
@param fn_  the names for all features/variables
@param k    the number of classes (k in {0, 1, ...k-1}
@param cn_  the names for all classes

class LDA (x: MatrixD, y: VectoI, fn_ : Strings = null, k: Int = 2, cn_ : Strings = null)
      extends ClassifierReal (x, y, fn_, k, cn_)

def corrected_cov (xc: MatriD): MatriD = (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 `KNN_Classifier` class is used to classify a new vector $\mathbf{z}$ into one of k classes $y \in \{0, 1, \ldots, k-1\}$. It works by finding its κ-nearest neighbors to the point $\mathbf{z}$. These neighbors essentially vote according to their classification. The class with the most votes is selected as the classification of vector $\mathbf{z}$. Using a distance metric, the κ vectors nearest to $\mathbf{z}$ are found in the training data, which are stored row-wise in data matrix X. The corresponding classifications are given in vector $\mathbf{y}$, such that the classification for vector $\mathbf{x_i}$ is given by y_i.

In ScalaTion to avoid the overhead of calling `sqrt`, the square of the Euclidean distance is used (although other metrics can easily be swapped in). The squared distance from vector $\mathbf{x}$ to vector $\mathbf{z}$ is then

$$d(\mathbf{x}) \;=\; d(\mathbf{x}, \mathbf{z}) \;=\; ||\mathbf{x} - \mathbf{z}||^2$$

The distance metric is used to collect the κ nearest vectors into set $top_\kappa(\mathbf{z})$, such that there does not exists any vector $\mathbf{x_j} \notin top_\kappa(\mathbf{z})$ that is closer to $\mathbf{z}$.

$$top_\kappa(\mathbf{z}) \;=\; \{\mathbf{x_i} | i \in \{0, \ldots, \kappa - 1\} \text{ and } \nexists(\mathbf{x_j} \notin top_\kappa(\mathbf{z}) \text{ and } d(\mathbf{x_j}) < d(\mathbf{x_i})\}$$

In case of ties for the most distant point to include in $top_\kappa(\mathbf{z})$ one could pick the first point encountered or the last point. A less biased approach would be to randomly break the tie.

Now $y(top_\kappa(\mathbf{z}))$ can be defined to be the vector of votes from the members of the set, e.g., $y(top_3(\mathbf{z})) = [1, 0, 1]$. The ultimate classification is then simply the mode (most frequent value) of this vector (e.g., 1 in this case).

$$y^* \;=\; \text{mode } \mathbf{y}(top_\kappa(\mathbf{z}))$$

6.7.1 Lazy Learning

Training in the `KNN_Classifier` class is lazy, i.e., the work is done in the `classify` method, rather than the `train` method.

```
override def classify (z: VectoD): (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 κ $\mathbf{x}$ vectors closest to the given vector $\mathbf{z}$. This method updates `topK` by replacing the most distant $\mathbf{x}$ vector in `topK` with a new one if it is closer. Each element in the `topK` array is a tuple `(j, d(j))` indicating which vector and its distance from $\mathbf{z}$. Each of these selected vectors will have their vote taken, voting for the class for which it is 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: MatriD, y: VectoI, fn_ : Strings = null, k: Int = 2,
                      cn_ : Strings = null, kappa: Int = 3)
      extends ClassifierReal (x, y, fn_, k, cn_)

def distance (x: VectoD, z: VectoD): Double = (x - z).normSq
def kNearest (z: VectoD)
def train (itest: IndexedSeq [Int]): KNN_Classifier =
override def classify (z: VectoD): (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,     // joint data matrix
                               2, 4, 1,
                               3, 4, 1,
                               4, 4, 1,
                               5, 3, 0,
                               6, 3, 1,
                               7, 2, 0,
                               8, 2, 0,
                               9, 1, 0,
                              10, 1, 0)
```

2. Under what circumstances would one expect a `KNN_Classifier` to perform better than LogisticRegression?

3. How could `KNN_Classifier` be adpated to work for prediction problems?

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-by-n data matrix X and an n-dimensional classification vector $\mathbf{y}$. Each data vector in the matrix is classified into one of k classes numbered $0, \ldots, k-1$. Each column in the matrix represents a feature (e.g., Humidity). The value count **vc** vector gives the number of distinct values per feature (e.g., 2 for Humidity).

Depending on the data type of a column, ScalaTion's implementation of C4.5 works like ID3 unless the column is continuous. A column is flagged `isCont` if it is continuous or relatively large ordinal. For a column that `isCont`, values for the feature are split into a left group and a right group based upon whether they are $\leq$ or $>$ an optimal threshold, respectively.

Candidate thresholds/split points are all the mid points between all column values that have been sorted. The threshold giving the maximum entropy drop (or *gain*) is the one that is chosen.

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.

```
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
/** 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.
 *  The 'Cont' version uses continuous values for Temperature and Humidity,
 *  @see sefiks.com/2018/05/13/a-step-by-step-c4-5-decision-tree-example
 */
object ExampleTennisCont
{
    // combined data matrix [ x | y ]
    // dataset ----------------------------------------------------------------
    // x0: Outlook:     Rain (0),   Overcast (1), Sunny (2)
    // x1: Temperature: Continuous
    // x2: Humidity:    Continuous
    // x3: Wind:        Weak (0),   Strong (1)
    // y:  the response/classification decision
    // variables/features:          x0    x1     x2      x3     y
    val xy = 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
```

```
                                        1,    72,    90,     1,    1,       // day 12
                                        1,    81,    75,     0,    1,       // day 13
                                        0,    71,    80,     1,    0)       // day 14

    val fn    = Array ("Outlook", "Temp", "Humidity", "Wind")               // feature names
    val isCon = Array (false, true, true, false)                            // continuous feature flag
    val cn    = Array ("No", "Yes")                                         // class names for y
    val k     = cn.size                                                     // number of classes
} // ExampleTennisCont object
```

As with the ID3 algorithm, the C4.5 algorithm picks x_0 as the root node. This feature is not continuous and has three branches. Branch `b0` will lead to a node where as before x_3 is chosen. Branch `b1` will lead to a leaf node. Finally, branch `b2` will lead to a node where continuous feature x_2 is chosen.

Sub-problem $x_0 = 2$

Note that if $x_0 = 0$ or 1, the algorithm works like ID3. However, there is still some uncertainty left when $x_0 = 2$, so this node may be split and it turn out the split will involve continuous feature x_2. The sub-problem for Outlook: Rain (2) see Table 6.1 is defined as follows: Take all five cases/rows in the data matrix X for which $x_0 = 2$.

Table 6.1: Sub-problem for node x_0 and branch 2

Day	**x_1**	**x_2**	**x_3**	**y**
1	85	85	0	0
2	80	90	1	0
8	72	95	0	0
9	69	70	0	1
11	75	70	1	1

The distinct values for feature x_2 in sorted order are the following: [70.0 ,85.0, 90.0, 95.0]. Therefore, the candidate threshold/split points for continuous feature x_2 are their midpoints: [77.5, 87.5, 92.5]. Threshold 77.5 yields (0-, 2+) on the left and (3-, 0+) on the right, 87.5 yields (1-, 2+) on the left and (2-, 0+) on the right, and 92.5 yields (2-, 2+) on the left and (1-, 0+) on the right. Clearly, the best threshold value is 77.5. Since a continuous feature splits elements into low (left) and high (right) groups, rather than branching on all possible values, the same continuous feature may be chosen again by a descendant node.

`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 constrint)

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: MatriD, y: VectoI, 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 (): MatriD =
def selectSubFeatures (subSample: MatriD): (MatrixD, VectorI) =
def train (testStart:Int, testEnd:Int)
def classify (z: VectoD): (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.

ZZ

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

class SupportVectorMachine (x: MatriD, y: VectoI, fn_ : Strings = null, cn_ : Strings = Array ("-", "+")
      extends ClassifierReal (x, y, fn_, 2, cn_)

def l_D (a: VectoD): Double =
def g (a: VectoD): Double = a dot y
def find_w ()
def find_b ()
def train ()
def fit: (VectoD, Double) = (w, b)
def classify (z: VectoD): Int = (signum (w dot z + b)).toInt
```

Chapter 7

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 $\mu(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}]$ and an error term (or multiplying in the case of multiplicative errors).

$$y = \mu(\mathbf{x}) + \epsilon$$

In the second step, the mean function is related to a linear combination of the predictor variables, i.e., $\mathbf{b} \cdot \mathbf{x}$

$$g(\mu(\mathbf{x})) = \mathbf{b} \cdot \mathbf{x}$$

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.

Model Type	**Response Type (y)**	**Link Function**	**Residual Distribution**
Logistic Regression	binary $\{0, 1\}$	logit	Bernoulli Distribution
Poisson Regression	integer $\{0, \ldots, \infty\}$	ln	Poisson Distribution
Exponential Regression	continuous $[0, \infty)$	ln or reciprocal	Exponential Distribution
General Linear Model (GLM)	continuous $(-\infty, \infty)$	identity	Normal Distribution

Table 7.1: Types of Generalized Linear Models

See `http://idiom.ucsd.edu/~rlevy/lign251/fall2007/lecture_13.pdf` and `http://link.springer.com/article/10.1023%2FA%3A1022436007242#page-1`. for additional details.

Since the response variable for Logistic Regression is defined on finite domains, it has been placed under Classification (see the 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) [26]

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 ϵ.

$$y \;=\; \mu(\mathbf{x})\,\epsilon$$

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).

$$g(\mu(\mathbf{x})) \;=\; \ln(\mu(\mathbf{x})) \;=\; \mathbf{b}\cdot\mathbf{x}$$

Expanding the dot product and using the inverse link function yields the following:

$$\mu(\mathbf{x}) \;=\; e^{\mathbf{b}\cdot\mathbf{x}} \;=\; e^{b_0 \,+\, b_1 x_1 \,+\, \ldots \,+\, b_k x_k}$$

The residuals $\epsilon_i = y_i/\mu(\mathbf{x_i})$ are distributed Exponential(1), so

$$f(y_i/\mu(\mathbf{x_i})) \;=\; \frac{1}{\mu(\mathbf{x_i})}\, e^{-y_i/\mu(\mathbf{x_i})}$$

Therefore, the likelihood function for Exponential Regression is as follows:

$$L \;=\; \prod_{i=0}^{m-1} \frac{1}{\mu(\mathbf{x_i})}\, e^{-y_i/\mu(\mathbf{x_i})}$$

Substituting for $\mu(\mathbf{x_i})$ gives

$$L \;=\; \prod_{i=0}^{m-1} e^{-\mathbf{b}\cdot\mathbf{x_i}}\, e^{-y_i/e^{\mathbf{b}\cdot\mathbf{x_i}}}$$

Taking the natural logarithm gives the log-likelihood function.

$$LL \;=\; \sum_{i=0}^{m-1} -\mathbf{b}\cdot\mathbf{x_i} - \frac{y_i}{e^{\mathbf{b}\cdot\mathbf{x_i}}}$$

See `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 λ may be defined as follows:

$$f(y; \lambda) \;=\; \frac{\lambda^y}{y!} e^{-\lambda}$$

Again, treating this as a GZLM problem,

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

$$g(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x}$$

The link function g for Poisson Regression is the ln (natural logarithm) function.

$$\ln(\mu(\mathbf{x})) \;=\; \mathbf{b} \cdot \mathbf{x}$$

The residuals ϵ_i are distributed according to the Poisson distribution.

$$\frac{\mu(\mathbf{x_i})^{y_i}}{y_i!} e^{-\mu(\mathbf{x_i})}$$

Therefore, the likelihood function for Poisson Regression is as follows:

$$L \;=\; \prod_{i=0}^{m-1} \frac{\mu(\mathbf{x_i})^{y_i}}{y_i!} e^{-\mu(\mathbf{x_i})}$$

Taking the natural logarithm gives the log-likelihood function.

$$LL \;=\; \sum_{i=0}^{m-1} y_i \ln(\mu(\mathbf{x_i}) - \mu(\mathbf{x_i}) - \ln(y_i!)$$

Substituting $\mu(\mathbf{x_i}) \;=\; e^{\mathbf{b} \cdot \mathbf{x_i}}$ yields the following:

$$LL \;=\; \sum_{i=0}^{m-1} y_i \mathbf{b} \cdot \mathbf{x_i} - e^{\mathbf{b} \cdot \mathbf{x_i}} - \ln(y_i!)$$

Since the last term is independent of the parameters, removing it will not affect the optimization.

$$LL_2 \;=\; \sum_{i=0}^{m-1} y_i \mathbf{b} \cdot \mathbf{x_i} - e^{\mathbf{b} \cdot \mathbf{x_i}}$$

See `http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf` for more details.

Example Problem:

Class Methods:

```
@param x   the input/data matrix augmented with a first column of ones
@param y   the integer response vector, y_i in {0, 1, ... }
@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 `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 θ_k be a threshold for splitting variable x_j, we may split the rows in the X matrix into left and right groups.

$$\text{left}_k(X) = \{\mathbf{x_i} | x_{ij} \le \theta_k\} \tag{8.1}$$

$$\text{right}_k(X) = \{\mathbf{x_i} | x_{ij} > \theta_k\} \tag{8.2}$$

For splitting variable x_j, the threshold θ_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.

```
val x = new MatrixD ((10, 1), 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val y = VectorD (5.23, 5.7, 5.91, 6.4, 6.8, 7.05, 8.9, 8.7, 9.0, 9.05)
```

In this case, $\theta_0 = 6.5$ divides the dataset into

$$\text{left}_0(X) = \{1, 2, 3, 4, 5, 6\}$$
$$\text{right}_0(X) = \{7, 8, 9, 10\}$$

with means $\mu_0(y) = 6.18$ (left) and $\mu_1(y) = 8.91$ (right). Further spliting may occur on x_0 (or x_j for multidimensional examples). If we let the maximum tree depth be two, we obtain the following four regions, corresponding to the four leaf nodes,

```
Root (-Inf, Inf]
    Node x0 in (-Inf, 6.5]
        Leaf x0 in (-Inf, 3.5]
        Leaf x0 in (3.5, 6.5]
    Node x0 in (6.5, Inf]
        Leaf x0 in (6.5, 8.5]
        Leaf x0 in (8.5, Inf]
```

with means $\mu_0(y) = 5.61$, $\mu_1(y) = 6.75$, $\mu_2(y) = 8.80$ and $\mu_3(y) = 9.03$. Each internal (non-leaf) node will have a threshold. They are $\theta_0 = 6.5$, $\theta_1 = 3.5$ and $\theta_2 = 8.5$.

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 *indictor function*,

$$\mathbb{I}_l(\mathbf{x}) \;=\; x_j \;\in\; (a_l, b_l] \tag{8.3}$$

which simply indicates (false/true) whether variable x_j is in the interval $(a_l, b_l]$. Now define $\mathbb{I}^*_l(\mathbf{x})$ as the product of the indicator funtions from leaf l until (not including) the root of the tree,

$$\mathbb{I}^*_l(\mathbf{x}) \;=\; \prod_{h \in \text{anc}(l)} \mathbb{I}_h(\mathbf{x}) \tag{8.4}$$

where anc(l) is the set of ancestors of leaf node l (inclusive of l, exclusive of root). Since only one of these $\mathbb{I}^*$ indicator functions can be true for any given $\mathbf{x}$ vector, we may concisely express the regression tree model as follows:

$$y \;=\; \sum_{l \,\in\, leaves} \mathbb{I}^*_l(\mathbf{x})\, \mu_l(y) \;+\; \epsilon \tag{8.5}$$

Thus, given a predictor vector $\mathbf{x}$, predicting a value for the response variable y corresponds to taking the mean y-value of the vectors in $\mathbf{x}$'s composite region (the intersection of regions from the leaf until the root). 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 θ_k that minimizes the sum of the MSEs.

$$\min_{\theta_k} \text{mse}(\text{left}_k(X)) + \text{mse}(\text{right}_k(X)) \tag{8.6}$$

Possible values for θ_k are the values between any two consecutive values in vector $\mathbf{x}_{:\mathbf{j}}$ sorted. This will allow any possible split of $\mathbf{x}_{:\mathbf{j}}$ 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 consective values, i.e., 5.5, 10.5 and 11.5. A straitforward way to implement determing the next variable x_j and its threshold θ_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 [7]

`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
```

```
@param maxDepth     the depth limit for tree
@param curDepth     current depth
@param branchValue  parameter used to record the branchValue for the tree node
@param thres        parameter used to record the threshold for the tree's parent node
@param feature      parameter used to record the feature for the tree's parent node

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 and Neural Networks

9.1 Non-Linear Regression

The `NonLinRegression` class supports non-linear regression. In this case, $\mathbf{x}$ can be multi-dimensional $[1, x_1, ...x_k]$ and the function f is non-linear in the parameters $\mathbf{b}$.

9.1.1 Model Equation

As before, the goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$\boxed{y \ = \ f(\mathbf{x}; \mathbf{b}) + \epsilon} \tag{9.1}$$

where ϵ 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.

$$y \ = \ (b_0 + b_1 x_1)/(b_2 + x_1) + \epsilon$$

9.1.2 Training

A training dataset consisting of m input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector $\mathbf{b}$. Given an input matrix X consisting of m input vectors and an output vector $\mathbf{y}$ consisting of m output values, minimize the distance between the target output vector $\mathbf{y}$ and the predicted output vector $f(X; \mathbf{b})$.

$$min_{\mathbf{b}} \|\mathbf{y} - f(X; \mathbf{b})\| \tag{9.2}$$

Again, it is convenient to minimize the dot product of the error with itself,

$$(\mathbf{y} - f(X; \mathbf{b})) \cdot (\mathbf{y} - f(X; \mathbf{b})) \tag{9.3}$$

9.1.3 Optimization

For non-linear regression, a Least-Squares (minimizing the residuals) method can be used to fit the parameter vector $\mathbf{b}$. Unlike the linear case (where one simply sets the gradient to zero), since the formula is non-linear in $\mathbf{b}$, Non-Linear Programming (NLP) is used to minimize the sum of squares error (sse). A user defined function taking a vector of inputs $\mathbf{x}$ and a vector of parameters $\mathbf{b}$,

```
f: (VectoD, VectoD) => Double
```

is passed as a class parameter. This function is used to create a predicted output value z_i for each input vector $\mathbf{x_i}$. 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 yields sse.

```
def sseF (b: VectoD): 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**.

```
val bfgs = new QuasiNewton (sseF)          // minimize sse using NLP
b        = bfgs.solve (b_init)             // estimate for b from optimizer
```

For more information see `http://www.bsos.umd.edu/socy/alan/stats/socy602_handouts/kut86916_ch13.pdf`.

`NonLinRegression` **Class**

Class Methods:

```
    @param x       the data/input matrix augmented with a first column of ones
    @param y       the response/output vector
    @param f       the non-linear function f(x, b) to fit
    @param b_init  the initial guess for the parameter vector b
    @param fname_  the feature/variable names
    @param hparam  the hyper-parameters (currently has none)

    class NonLinRegression (x: MatriD, y: VectoD, f: FunctionP2S,
                            b_init: VectorD, fname_ : Strings = null,
                            hparam: HyperParameter = null)
          extends PredictorMat (x, y, fname_, hparam) with NoFeatureSelectionMat

    def sseF (b: VectoD): Double =
    def train (x_r: MatriD = x, y_r: VectoD = y): NonLinRegression =
    override def predict (z: VectoD): Double = f(z, b)
```

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 $\mathbf{x}$, 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).

9.2.1 Model Equation

The weights on the edges are analogous to the parameter vector $\mathbf{b}$ in regression. The output y has an associated parameter vector $\mathbf{b}$, where parameter value b_j is the edge weight connecting input node x_j with output node y.

Recall the basic multiple regression model (equation 4.1).

$$y \ = \ \mathbf{b} \cdot \mathbf{x} + \epsilon \ = \ b_0 + b_1 x_1 + ... b_{n-1} x_{n-1} + \epsilon$$

We now take the linear combination of the inputs, $\mathbf{b} \cdot \mathbf{x}$, and apply an activation function f.

$$\boxed{y \ = \ f(\mathbf{b} \cdot \mathbf{x}) + \epsilon \ = \ f(\sum_{j=0}^{n-1} b_j x_j) + \epsilon} \tag{9.4}$$

9.2.2 Training

Given several input vectors and output values (e.g., in a training dataset), optimize/fit the weights $\mathbf{b}$ connecting the layers. After training, given an input vector $\mathbf{x}$, the net can be used to predict the corresponding output value y.

A training dataset consisting of m input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight vector $\mathbf{b}$. Given an input matrix X consisting of m input vectors and an output vector $\mathbf{y}$ consisting of m output values, minimize the distance between the actual/target output vector $\mathbf{y}$ and the predicted output vector $\hat{\mathbf{y}}$,

$$\boxed{\hat{\mathbf{y}} \ = \ \mathbf{f}(X\mathbf{b})} \tag{9.5}$$

where $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$ is the vectorized version of the activation function f. The vectorization may occur over the entire training set or more likely, an iterative algorithm may work with a group/batch of instances at a time. In other words, the goal is to minmize some norm of the error vector.

$$\boxed{\boldsymbol{\epsilon} \ = \ \mathbf{y} - \hat{\mathbf{y}} \ = \ \mathbf{y} - \mathbf{f}(X\mathbf{b})} \tag{9.6}$$

Using the Euclidean (ℓ_2) norm, we have

$$\min_{\mathbf{b}} \|\mathbf{y} - \mathbf{f}(X\mathbf{b})\|$$

As was the case with regression, it is convenient to minimize the dot product of the error with itself ($||\boldsymbol{\epsilon}||^2 = \boldsymbol{\epsilon} \cdot \boldsymbol{\epsilon}$). In particular, we aim to minimize half of this value, half *sse* (*hse*).

$$hse(\mathbf{b}) \;=\; \frac{1}{2}\,(\mathbf{y} - \mathbf{f}(X\mathbf{b})) \cdot (\mathbf{y} - \mathbf{f}(X\mathbf{b})) \tag{9.7}$$

9.2.3 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.

$$\nabla hse(\mathbf{b}) \;=\; \frac{\partial hse}{\partial \mathbf{b}} \;=\; \left[\frac{\partial hse}{\partial b_0}, \dots, \frac{\partial hse}{\partial b_{n-1}}\right]$$

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 $\mathbf{u} = X\mathbf{b}$ (the pre-activation response) allows equation 9.4 to be simplied to

$$hse \;=\; \frac{1}{2}\,(\mathbf{y} - \mathbf{f}(\mathbf{u})) \cdot (\mathbf{y} - \mathbf{f}(\mathbf{u})) \tag{9.8}$$

The chain rule from vector calculus to be applied is

$$\frac{\partial hse}{\partial b_j} \;=\; \frac{\partial hse}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial b_j} \tag{9.9}$$

The first partial derivative is

$$\boxed{\frac{\partial hse}{\partial \mathbf{u}} \;=\; -\,\mathbf{f}'(\mathbf{u})\,(\mathbf{y} - \mathbf{f}(\mathbf{u}))} \tag{9.10}$$

where the first part of the r.h.s. is $\mathbf{f}'(\mathbf{u})$ which is the derivative of $\mathbf{f}$ with respect to vector $\mathbf{u}$ and the second part is the difference between the actual and predicted output/response vectors. The two vectors are multiplied together, elementwise.

The second partial derivative is

$$\boxed{\frac{\partial \mathbf{u}}{\partial b_j} \;=\; \mathbf{x_{:j}}} \tag{9.11}$$

where $\mathbf{x_{:j}}$ is the j^{th} column of matrix X (see Exercises 4, 5 and 6 for details).

The dot product of the two partial derivatives gives

$$\frac{\partial hse}{\partial b_j} \;=\; -\,\mathbf{x_{:j}} \cdot \mathbf{f}'(X\mathbf{b})(\mathbf{y} - \mathbf{f}(X\mathbf{b}))$$

Since the error vector $\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{f}(X\mathbf{b})$, we may simplify the expression.

$$\frac{\partial hse}{\partial b_j} \;=\; -\, \mathbf{x_{:j}} \cdot \mathbf{f}'(X\mathbf{b})\, \boldsymbol{\epsilon} \tag{9.12}$$

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 hse.

The $\boldsymbol{\delta}$ Vector

It is helpful especially for multi-layer neural networks to define the delta vector $\boldsymbol{\delta}$ as follows:

$$\boxed{\boldsymbol{\delta} \;=\; \frac{\partial hse}{\partial \mathbf{u}} \;=\; -\, \mathbf{f}'(X\mathbf{b})\, \boldsymbol{\epsilon}} \tag{9.13}$$

It multiplies the derivative of $\mathbf{f}$ 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

$$\frac{\partial hse}{\partial b_j} \;=\; \mathbf{x_{:j}} \cdot \boldsymbol{\delta} \tag{9.14}$$

Note, if we consider a single instance $(\mathbf{x_i}, y_i)$, equation 9.11 becomes

$$\frac{\partial hse}{\partial b_j} \;=\; -\, x_{ij} f'(\mathbf{x_i} \cdot \mathbf{b})\, \epsilon_i \;=\; x_{ij}\delta_i$$

Example

Assume that the activation function is the sigmoid function. Starting with the parameter/weight vector $\mathbf{b} = [.1, .2, .1]$, compute the m-dimensional vectors, $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$, for Exercise 7. With these parameters, the predicted output/response vector $\hat{\mathbf{y}}$ may be computed in two steps: The first step computes the response, pre-activation $X\mathbf{b}$. The second step takes this vector and applies the activation function to each of its elements. This requires looking ahead to the section on *activation functions*. The sigmoid(t) function is $[1 + e^{-t}]^{-1}$.

$$\begin{aligned} \hat{\mathbf{y}} \;&=\; \text{sigmoid(X}\mathbf{b}) \;=\; \text{sigmoid}([.1, .15, .2, .2, .25, .3, .3, .35, .4]) \\ &=\; [.5249, .5374, .5498, .5498, .5621, .5744, .5744, .5866, .5986] \end{aligned}$$

The error vector $\boldsymbol{\epsilon}$ is simply the difference between the actual and predicted output/response vectors.

$$\begin{aligned} \boldsymbol{\epsilon} \;&=\; \mathbf{y} - \hat{\mathbf{y}} \\ &[.5000, .3000, .2000, .8000, .5000, .3000, 1.0000, .8000, .5000] \;- \\ &[.5249, .5374, .5498, .5498, .5621, .5744, .5744, .5866, .5986] \;= \\ &[-.0249, -.2374, -.3498, .2501, -.0621, -.2744, .4255, .2133, -.0986] \end{aligned}$$

To compute the delta vector $\boldsymbol{\delta}$, we must look ahead to get the derivative of the activation function.

$$\text{sigmoid}'(\text{t}) \;=\; \text{sigmoid}(\text{t})\,[1 - \text{sigmoid}(\text{t})]$$

Therefore, since sigmoid(X**b**) = $\hat{\mathbf{y}}$

$$\begin{aligned}\boldsymbol{\delta} \;&=\; -\,\hat{\mathbf{y}}\,(\mathbf{1} - \hat{\mathbf{y}})\,\boldsymbol{\epsilon} \\ &\quad [.0062, .0590, .0865, -.0619, .0153, .0670, -.1040, -.0517, .0237]\end{aligned}$$

Forming the Gradient

Combining the partial derivatives in equation 9.11 into an n-dimensional vector (i.e., the gradient) yields

$$\nabla hse(\mathbf{b}) \;=\; \frac{\partial hse}{\partial \mathbf{b}} \;=\; -\,X^t[\mathbf{f}'(X\mathbf{b})\,\boldsymbol{\epsilon}] \;=\; X^t\boldsymbol{\delta} \tag{9.15}$$

Since many optimizers such as gradient-descent, move in the direction opposite to the gradient by a distance governed by the learning rate η (alternatively step size), the following term should be added to the weight/parameter vector **b**.

$$X^t\,[\mathbf{f}'(X\mathbf{b})\,\boldsymbol{\epsilon}]\,\eta \;=\; -\,X^t\boldsymbol{\delta}\,\eta \tag{9.16}$$

The right hand side is an n-by-m matrix, m vector product yielding an n vector result. The factor in brackets, $[\mathbf{f}'(X\mathbf{b})\,\boldsymbol{\epsilon}]$, 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 η, the parameter/weight vector **b** is updated as follows:

$$\boxed{\mathbf{b} \;=\; \mathbf{b} \,-\, X^t\,\boldsymbol{\delta}\,\eta} \tag{9.17}$$

9.2.4 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, limit) before training.
@param stream  the random number stream to use
@param limit   the maximum value for any weight

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.

```
def setWeights (w0: VectoD) { b = w0 }
```

9.2.5 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 8). This activation function is usually not intended for neural nets with more layers, since theorectically 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 [14]. For these activation functions the outputs in the $\mathbf{y}$ 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 9.1: Activation Functions: Identity, Rectified Linear Unit, Leaky Rectified Linear Unit, Exponential Linear Unit, Sigmoid, Hyperbolic Tangent, Gaussian

Name	**Function** $u = f(t)$	**Domain**	**Range**	**Derivative** $f'(t)$	**Inverse** $t = f^{-1}(u)$
id	t	$\mathbb{R}$	$\mathbb{R}$	1	u
reLU	$\max(0, t)$	$\mathbb{R}$	$\mathbb{R}^+$	$I_{t>0}$	u for $u > 0$
lreLU	$\max(\alpha t, t),\ \alpha < 1$	$\mathbb{R}$	$\mathbb{R}$	$\text{if}_{t<0}(\alpha, 1)$	$\min(\frac{u}{\alpha}, u)$
eLU	$\text{if}_{t<0}(\alpha(e^t - 1), t)$	$\mathbb{R}$	$\mathbb{R}$	$\text{if}_{t<0}(f(t) + \alpha, 1)$	$\text{if}_{t<0}(\ln(\frac{u}{\alpha} + 1), u)$
sigmoid	$[1 + e^{-t}]^{-1}$	$\mathbb{R}$	$(0, 1)$	$f(t)[1 - f(t)]$	$-\ln(\frac{1}{u} - 1)$
tanh	$\tanh(t)$	$\mathbb{R}$	$(-1, 1)$	$1 - f(t)^2$	$.5 \ln\left(\frac{1+u}{1-u}\right)$
gaussian	e^{-t^2}	$\mathbb{R}$	$(0, 1]$	$-2te^{-t^2}$	$\sqrt{-\ln(u)}$

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

$$\frac{\partial hse}{\partial \mathbf{b}} \;=\; -\,X^t[\mathbf{f}(X\mathbf{b})[\mathbf{1} - \mathbf{f}(X\mathbf{b})]\,\boldsymbol{\epsilon}] \;=\; X^t\,\boldsymbol{\delta}$$

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 η. Therefore, the parameter/weight vector $\mathbf{b}$ is adjusted as follows:

$$\mathbf{b} \;=\; \mathbf{b} \,-\, X^t\,\boldsymbol{\delta}\,\eta$$

Assuming the learning rate $\eta = 1$ and taking the $\boldsymbol{\delta}$ vector from the example, the update to parameter/weight vector $\mathbf{b}$ is

$$X^t\,\boldsymbol{\delta}\,\eta \;=\; [.0402, -.1218, .1886]$$

Consequently, the updated value for the parameter/weight vector $\mathbf{b}$ is

$$\mathbf{b} \;=\; [.1, .2, .1] \,-\, [.0402, -.1218, .1886] \;=\; [.0597, .3218, -.0886]$$

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 η 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 η which multiplies the gradient. Setting it too low, slows convergence, while setting it too high can cause oscillation. In ScalaTion, the learning rate η (`eta` in the code) is a hyper-parameter that defaults to 0.1, but is easily adjusted, e.g.,

```
Optimizer.hp ("eta") = 0.05
```

The `train0` method contains the main training loop that is shown below. Inside the loop, new values `yp` are predicted, from which an error vector `e` is determined. This is used to calculate the delta vector `d`, which along `x.t` and `eta` are used to update the parameter/weight vector `b`.

```
def train0 (x_r: MatriD = x, y_r: 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 = f0.fV (x_r * b)                            // predicted output vector yp = f(Xb)
        e      = y_r - yp                                   // error vector for y
        val d  = -f0.dV (yp) * e                            // delta vector for y
        b     -= x_r.t * d * eta                            // update the parameters/weights

        val sse = sseF (y, f0.fV (x_r * 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 `f0.fV` is the vectorization of the activation function `f0.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**.

```
def vectorize (f: FunctionS2S): FunctionV_2V = (x: VectoD) => x.map (f(_))
val fV = vectorize (f)
```

The function `f0.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.
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.

Table 9.2: Correspondence between Code and Boxed Equations

Code	**Equation**	**Equation Number**
`yp = f0.fV (x * b)`	$\hat{\mathbf{y}} = \mathbf{f}(X\mathbf{b})$	9.2
`e = yy - yp`	$\boldsymbol{\epsilon} = \mathbf{y} - \hat{\mathbf{y}}$	9.3
`d = -f0.dV (yp) * e`	$\boldsymbol{\delta} = -\mathbf{f}'(X\mathbf{b})\boldsymbol{\epsilon}$	9.10
`b -= x.t * d * eta`	$\mathbf{b} = \mathbf{b} - X^t\boldsymbol{\delta}\eta$	9.14

A perceptron can be considered to be a special type of non-linear or transformed regression, see Exercise 9. The `Perceptron` class defaults to the `f_sigmoid` Activation Function Family (`AFF`), which is defined in the `ActivationFun` object.

```
val f_sigmoid = AFF (sigmoid, sigmoidV, sigmoidM, sigmoidDV, sigmoidDM, (0, 1))
```

In general, the `AFF` for family `f0` contains the following:

```
val f0 = AFF (f, fV, fM, dV, dM, (lb, ub))
```

The first three are the activation function in scalar, vector and matrix forms, respectively, the next two are the derivative of the activation function in vector and matrix forms, respectively, and the last is a tuple giving the lower and upper bounds for the range of the activation function. Note, if the actual response vector $\mathbf{y}$ is outside the bounds, it will be impossible for the predicted response vector $\hat{\mathbf{y}}$ to approximate it, so rescaling will be necessary.

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.

Essentially, parameter optimization in perceptrons involves using/calculating several vectors as summarized in Table 9.3 where n is the number of parameters and m is the number of instances used at a particular point in the iterative optimization algorithm, for example, corresponding to the total number of instances in a training set for Gradient Descent or the number of instances in a batch (subsample) for Stochastic Gradient Descent.

Table 9.3: Vectors used in Perceptrons

Vector	**Space**	**Formula**	**Description**
$\mathbf{x_i}$	$\mathbb{R}^n$	given	the i^{th} row of the input/data matrix
$\mathbf{x_{:j}}$	$\mathbb{R}^m$	given	the j^{th} column of the input/data matrix
$\mathbf{b}$	$\mathbb{R}^n$	given	the parameter vector (updated per iteration)
$\mathbf{u}$	$\mathbb{R}^m$	$X\mathbf{b}$	the pre-activation vector
$\mathbf{y}$	$\mathbb{R}^m$	given	the actual output/response vector
$\hat{\mathbf{y}}$	$\mathbb{R}^m$	$\mathbf{f}(\mathbf{u})$	the predicted output/response vector
$\boldsymbol{\epsilon}$	$\mathbb{R}^m$	$\mathbf{y} - \hat{\mathbf{y}}$	the error/residual vector
$\boldsymbol{\delta}$	$\mathbb{R}^m$	$-\mathbf{f}'(\mathbf{u})\boldsymbol{\epsilon}$	the negative-slope-weighted error vector
$[b_0] \oplus \mathbf{w}$	$\mathbb{R}^n$	$\mathbf{b}$	concatenation of bias scalar and weight vector

---

Class Methods:

```
@param x       the data/input m-by-n matrix (data consisting of m input vectors)
@param y       the response/output m-vector (data consisting of m output values)
@param fname_  the feature/variable names
@param hparam  the hyper-parameters for the model/network
@param f0      the activation function family for layers 1->2 (input to output)
@param itran   the inverse transformation function returns responses to original scale

class Perceptron (x: MatriD, y: VectoD,
                  fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp,
                  f0: AFF = f_sigmoid, val itran: FunctionV_2V = null)
      extends PredictorMat (x, y, fname_, hparam)

def setWeights (w0: VectoD)
def reset (eta_ : Double) { eta = eta_ }
def train0 (x_r: MatriD = x, y_r: VectoD = y): Perceptron =
def train (x_r: MatriD = x, y_r: VectoD = y): Perceptron =
override def train2 (x_r: MatriD = x, y_r: VectoD = y): Perceptron =
def trainSwitch (which: Int, x_r: MatriD = x, y_r: VectoD = y): Perceptron =
override def predict (z: VectoD): Double = f0.f (b dot z)
override def predict (z: MatriD = x): VectoD = f0.fV (z * b)
def buildModel (x_cols: MatriD): Perceptron =
```

The `train0` method uses Gradient Descent with a simple stopping rule and a non-adaptive learning rate. A better optimzer 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.

9.2.6 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.

```
// 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, 30.867, 3000.0, 102.900,    // Pecos
                                             1.0, 36.350, 3693.0, 102.083,    // Sherman
                                             1.0, 30.300,  597.0,  97.700,    // Travis
                                             1.0, 26.900,  315.0,  99.283,    // Zapata
                                             1.0, 28.450,  459.0,  99.217,    // Lasalle
                                             1.0, 25.900,   19.0,  97.433)    // Cameron

val y = VectorD (56.0, 48.0, 60.0, 46.0, 38.0, 46.0, 53.0, 46.0,
                 44.0, 41.0, 47.0, 36.0, 52.0, 60.0, 56.0, 62.0)

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 y_0, y_1 and y_2. Create a perceptron for each output variable.

4. Use the following formula for matrix-vector multiplication

$$\mathbf{u} \;=\; X\mathbf{b} \;=\; \sum_j b_j \mathbf{x_{:j}}$$

to derive the formula for the following partial derivative

$$\frac{\partial \mathbf{u}}{\partial b_j} \;=\; \mathbf{x_{:j}}$$

5. Given equation 9.5, the formula for the objective/cost function $hse : \mathbb{R}^m \to \mathbb{R}$ expressed in terms of the pre-activation vector $\mathbf{u} = X\mathbf{b}$ and the vectorized activation function $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$,

$$hse(\mathbf{u}) \;=\; \frac{1}{2}\,(\mathbf{y} - \mathbf{f}(\mathbf{u})) \cdot (\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

derive equation 9.7, the formula for the gradient of hse with respect to $\mathbf{u}$.

$$\nabla hse \;=\; \frac{\partial hse}{\partial \mathbf{u}} \;=\; -\,\mathbf{f}'(\mathbf{u})(\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

Hint: Take the gradient, $\frac{\partial hse}{\partial \mathbf{u}}$, using the product rule $(d_1 \cdot f_2 + f_1 \cdot d_2)$.

$$\frac{\partial hse}{\partial \mathbf{u}} \;=\; -\,\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}} \cdot (\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

where $f_1 = f_2 = \mathbf{y} - \mathbf{f}(\mathbf{u})$ and $d_1 = d_2 = -\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$. Next, assuming $\frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$ is a diagonal matrix, show that the above equation can be rewritten as

$$\frac{\partial hse}{\partial \mathbf{u}} \;=\; -\,\mathbf{f}'(\mathbf{u})\,(\mathbf{y} - \mathbf{f}(\mathbf{u}))$$

where $\mathbf{f}'(\mathbf{u}) = [f'(u_0), \dots, f'(u_{m-1})]$ and the two vectors, $\mathbf{f}'(\mathbf{u})$ and $\mathbf{y} - \mathbf{f}(\mathbf{u})$, are multiplied, element-wise.

6. Show that the m-by-m Jacobian matrix, $\mathbb{J}\,\mathbf{f}(\mathbf{u}) = \frac{\partial \mathbf{f}(\mathbf{u})}{\partial \mathbf{u}}$, is a diagonal matrix, i.e.,

$$\mathbb{J}\,\mathbf{f}(\mathbf{u}) \;=\; \left[\frac{\partial f_i(\mathbf{u})}{\partial u_j}\right] \;=\; 0 \text{ if } i \neq j$$

where $f_i = f$ the scalar activation function. Each diagonal element is the derivative of the activation function applied to the i^{th} input, $f'(u_i)$. See the section on Vector Calculus in Chapter 2 that discusses Gradient Vectors, Jacobian Matrices and Hessian Matrices.

7. Show the first 10 iterations that update the parameter/weight matrix **b** that is initialized to $[.1, .2, .1]$. Use the following combined input-output matrix. Let the perceptron use the default sigmoid function.

```
// 9 data points:         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,  1.0,  0.3,
                               1.0,  1.0,  0.0,  1.0,
                               1.0,  1.0,  0.5,  0.8,
                               1.0,  1.0,  1.0,  0.5)

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 **bup** and the new value for weight/parameter vector **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 $f'(\mathbf{u})$ is the one vector, $\mathbf{1}$. Plug this into equation 9.12 to obtain the following result.

$$\frac{\partial hse}{\partial \mathbf{b}} \;=\; -\,X^t[\mathbf{1}\,\boldsymbol{\epsilon}] \;=\; -\,X^t(\mathbf{y} - X\mathbf{b})$$

   Setting the gradient equal to zero, now yields $X^tX\mathbf{b} = X^t\mathbf{y}$, the Normal Equations.

9. Show that a `Perceptron` with an invertible activation function f is similar to `TranRegression` with tranform f^{-1}. Explain any differences in the parameter/weight vector $\mathbf{b}$ and the sum of squared errors sse. Use the sigmoid activation function and the AutoMPG dataset and make the following two plots (using `PlotM`): `y`, `ypr`, `ypt` vs. `t` and `y`, `ypr`, `ypp` vs. `t`, where `y` is the actual response/output, `ypr` is the prediction from `Regression`, `ypt` is the prediction from `TranRegression` and `ypp` is the prediction from `Perceptron`.

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, Extreme Learning Machines and Multi-Variate Regression. It serves the same role that `PredictorMat` has for the regression modeling techniques.

9.3.1 Model Equation

For modeling techniques extending this abstract class, the model equation takes an input vector $\mathbf{x}$, pre-multiplies it by the transpose of the parameter matrix B, applies a function $\mathbf{f}$ to the resulting vector and adds an error vector $\boldsymbol{\epsilon}$,

$$\mathbf{y} \;=\; \mathbf{f}(B \cdot \mathbf{x}) + \boldsymbol{\epsilon} \;=\; \mathbf{f}(B^t\mathbf{x}) + \boldsymbol{\epsilon} \tag{9.18}$$

where

- $\mathbf{y}$ is an n_y-dimensional output/response random vector,
- $\mathbf{x}$ is an n_x-dimensional input/data vector,
- B is an n_x-by-n_y parameter matrix,
- $\mathbf{f} : \mathbb{R}^{n_y} \to \mathbb{R}^{n_y}$ is a function mapping vectors to vectors, and
- $\boldsymbol{\epsilon}$ is an n_y-dimensional residual/error random vector.

For Multi-Variate Regression, $\mathbf{f}$ is the identity function.

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.

$$Y \;=\; \mathbf{f}(XB) + E \tag{9.19}$$

where X is an m-by-n_x data/input matrix, Y is an m-by-n_y response/output matrix, B is an n_x-by-n_y parameter matrix, $\mathbf{f}$ is a function mapping one m-by-n_y matrix to another, and E is an m-by-n_y residual/error matrix. Note, a bold function symbol $\mathbf{f}$ is used is used to denote a function mapping either vectors to vectors (as was the case in the Model Equation subsection) or matrices to matrices (as is the case here).

$$\mathbf{f} : \mathbb{R}^{m \times n_y} \to \mathbb{R}^{m \times n_y}$$

If one is interested in refering to the k^{th} component (or column) of the output, equation 9.16 becomes.

$$\mathbf{y}_{:\mathbf{k}} \;=\; \mathbf{f}(X\mathbf{b}_{:\mathbf{k}}) + \boldsymbol{\epsilon}_{:\mathbf{k}} \tag{9.20}$$

Recall that $\mathbf{y}_\mathbf{k}$ indicates the k^{th} row of matrix Y, while $\mathbf{y}_{:\mathbf{k}}$ indicates the k^{th} column.

Analogous to other predictive modeling techniques, `PredictorMat2` constructor takes four arguments: the data/input matrix `x`, the response/output matrix `y`, the feature/variable names `fname`, and the hyper-parameters for the model/network `hparam`.

`PredictorMat2` **Class**

Class Methods:

```
@param x       the m-by-nx data/input matrix (data consisting of m input vectors)
@param y       the m-by-ny response/output matrix (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

abstract class PredictorMat2 (x: MatriD, y: MatriD,
                              protected var fname: Strings, hparam: HyperParameter)
         extends Predictor with Error

def getX: MatriD = x
def getY: VectoD = y(0)
def getYY: MatriD = y
def reset (eta_ : Double) { eta = eta_ }
def train0 (x_r: MatriD = x, y_r: MatriD = y): PredictorMat2
def train (x_r: MatriD = x, y_r: MatriD = y): PredictorMat2
def train (x_r: MatriD, y_r: VectoD): PredictorMat2 = train (x_r, MatrixD (y_r))
def train2 (x_r: MatriD = x, y_r: MatriD = y): PredictorMat2 = train (x_r, y_r)
def trainSwitch (which: Int, x_r: MatriD = x, y_r: MatriD = y): PredictorMat2 =
def resetDF (df_update: PairD)
def eval (x_e: MatriD = x, y_e: VectoD = y.col(0)): PredictorMat2 =
def eval (x_e: MatriD, y_e: MatriD): PredictorMat2 =
def analyze (x_r: MatriD = x, y_r: VectoD = y(0),
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 buildModel (x_cols: MatriD): PredictorMat2
def forwardSel (cols: Set [Int], index_q: Int = index_rSqBar): (Int, PredictorMat2) =
def forwardSelAll (index_q: Int = index_rSqBar): (Set [Int], MatriD) =
def backwardElim (cols: Set [Int], index_q: Int = index_rSqBar, first: Int = 1):
    (Int, PredictorMat2) =
def vif (skip: Int = 1): VectoD =
def predict (z: VectoD): Double = predictV (z)(0)
def predict (z: MatriD = x): VectoD = predictV (z)(0)
def predictV (z: VectoD): VectoD
def predictV (z: MatriD = x): MatriD
def crossValidate (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
```

`NetParam` **Class**

A model producing multiple output variables will have parameters as weight matrices. They may also have bias vectors. To unify these cases, ScalaTion utilizes the `NetParam` case class for holding a weight matrix along with an optional bias vector. Linear algebra like operators are provided for convenience, e.g., the `*:` allows one to write x *: p, corresponding to the mathematical expression XP where X is the input/data matrix and P holds the parameters. If the bias `b` is null, this is just matrix multiplication.

```
def *: (x: MatriD): MatriD = x * w + b
```

Inside, the `x` is multiplied by the weight matrix `w` and the bais vector `b` is added. Note, the `*:` is right associative since the `NetParam` object is on right (see `NeuralNet_2L` for an example of its usage).

Class Methods:

```
@param w  the weight matrix
@param b  the optional bias/intercept vector (null => not used)

case class NetParam (w: MatriD, var b: VectoD = null)

def copy: NetParam = NetParam (w.copy, b.copy)
def set (c: NetParam) { w.set (c.w); b.set (c.b()) }
def set (cw: MatriD, cb: VectoD = null)
def += (c: NetParam) { w += c.w; b += c.b }
def += (cw: MatriD, cb: VectoD = null)
def -= (c: NetParam) { w -= c.w; b -= c.b }
def -= (cw: MatriD, cb: VectoD = null)
def * (x: MatriD): MatriD = x * w + b
def *: (x: MatriD): MatriD = x * w + b
def dot (x: VectoD): VectoD = (w dot x) + b
override def toString: String = s"b.w = $w \n b.b = $b"
```

9.4 Two-Layer Neural Networks

The `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 $\mathbf{x}$, while the outputs are given by the output vector $\mathbf{y}$. Each input x_j is associated with an input node in the network, while each output y_k is associated with an output node in the network, 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.

9.4.1 Model Equation

The weights on the edges are analogous to the parameter vector $\mathbf{b}$ in regression. Each output variable y_k, has its own parameter vector $\mathbf{b_{:k}}$. These are collected as column vectors into a parameter/weight matrix B, where parameter value b_{jk} is the edge weight connecting input node x_j with output node y_k.

After training, given an input vector $\mathbf{x}$, the network can be used to predict the corresponding output vector $\mathbf{y}$. The network predicts an output/response value for y_k by taking the weighted sum of its inputs and passing this sum through activation function f.

$$y_k \;=\; f(\mathbf{b_{:k}} \cdot \mathbf{x}) + \epsilon_k \;=\; f\big(\sum_{j=0}^{n_x-1} b_{jk} x_j\big) + \epsilon_k$$

The model equation for `NeuralNet_2L` can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f}(B \cdot \mathbf{x}) + \boldsymbol{\epsilon} \;=\; \mathbf{f}(B^t \mathbf{x}) + \boldsymbol{\epsilon}} \tag{9.21}$$

9.4.2 Training

Given several input vectors and output vectors in a training dataset ($i = 0, \ldots, m-1$), the goal is to optimize/fit the paramters/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 $\hat{Y}$,

$$\hat{Y} \;=\; \mathbf{f}(XB) \tag{9.22}$$

This will minimize the error matrix $E = Y - \hat{Y}$

$$\min_B \|Y - \mathbf{f}(XB)\|_F \tag{9.23}$$

where $\|\cdot\|_F$ is the Frobenius norm, X is a m-by-n_x matrix, Y is a m-by-n_y matrix, and B is a n_x-by-n_y matrix. Other norms may be used as well, but the square of the Frobenius norm will give the overall sum of squared errors sse.

9.4.3 Optimization

As was the case with regression, it is convenient to minimize the dot product of the error with itself. We do this for each of the columns of the Y matrix to get the sse for each y_k and sum them up. The goal then is to simply minimize the objective function $sse(B)$. As in the Perceptron section, we work with half of the sum of squared errors sse (or hse). Summing the error over each column vector $\mathbf{y_{:k}}$ in matrix Y gives

$$hse(B) \;=\; \frac{1}{2} \sum_{k=0}^{n_y-1} (\mathbf{y_{:k}} - \mathbf{f}(X\mathbf{b_{:k}})) \cdot (\mathbf{y_{:k}} - \mathbf{f}(X\mathbf{b_{:k}})) \tag{9.24}$$

This nonlinear optimization problem may be solved by a variety of optimization techniques, including Gradient-Descent, Stochastic Gradient Descent or Stochastic Gradient Descent with Momentum.

Most optimizers require a derivative and ideally these should be provided in functional form (otherwise the optimizer will need to numerically approximate them). Again, for the sigmoid activation function,

$$\text{sigmoid}(t) \;=\; \frac{1}{1+e^{-t}}$$

the derivative is

$$\text{sigmoid}(t)[1 - \text{sigmoid}(t)]$$

To minimize the objective function given in equation 9.20, we decompose it into n_y functions.

$$hse(\mathbf{b_{:k}}) \;=\; \tfrac{1}{2}\,(\mathbf{y_{:k}} - \mathbf{f}(X\mathbf{b_{:k}})) \cdot (\mathbf{y_{:k}} - \mathbf{f}(X\mathbf{b_{:k}}))$$

Notice that this is the same equation as 9.4, just with subscripts on $\mathbf{y}$ and $\mathbf{b}$.

In Regression, we took the gradient and set it equal to zero. Here, gradients will need to be computed by the optimizer. The equations will be the same as given in the Perceptron section, just with subscripts added. The boxed equations from the Perceptron section become the following: The prediction vector for the k^{th} response/output is

$$\boxed{\hat{\mathbf{y}}_{\mathbf{:k}} \;=\; \mathbf{f}(X\mathbf{b_{:k}})} \tag{9.25}$$

The error vector for the k^{th} response/output is

$$\boxed{\boldsymbol{\epsilon}_{\mathbf{:k}} \;=\; \mathbf{y_{:k}} - \hat{\mathbf{y}}_{\mathbf{:k}} \;=\; \mathbf{y_{:k}} - \mathbf{f}(X\mathbf{b_{:k}})} \tag{9.26}$$

The delta vector for the k^{th} response/output is

$$\boxed{\boldsymbol{\delta}_{\mathbf{:k}} \;=\; \frac{\partial hse}{\partial \mathbf{u_k}} \;=\; -\,\mathbf{f}'(X\mathbf{b_{:k}})\,\boldsymbol{\epsilon}_{\mathbf{:k}}} \tag{9.27}$$

where $\mathbf{u_k} = X\mathbf{b_{:k}}$. The gradient with respect to the k^{th} parameter vector is

$$\frac{\partial hse}{\partial \mathbf{b_{:k}}} \;=\; -\,X^t[\mathbf{f}'(X\mathbf{b_{:k}})\,\boldsymbol{\epsilon}_{\mathbf{:k}}] \;=\; X^t\boldsymbol{\delta}_{\mathbf{:k}}$$

Finally, the update for the k^{th} parameter vector is

$$\boxed{\mathbf{b_{:k}} \;=\; \mathbf{b_{:k}} \;-\; X^t\,\boldsymbol{\delta}_{\mathbf{:k}}\,\eta} \tag{9.28}$$

Sigmoid Case

For the sigmoid function, $\mathbf{f}'(X\mathbf{b}_{:\mathbf{k}}) = \mathbf{f}(X\mathbf{b}_{:\mathbf{k}})[1 - \mathbf{f}(X\mathbf{b}_{:\mathbf{k}})]$, so

$$\frac{\partial hse}{\partial \mathbf{b}_{:\mathbf{k}}} \;=\; -\,X^t[\mathbf{f}(X\mathbf{b}_{:\mathbf{k}})[1 - \mathbf{f}(X\mathbf{b}_{:\mathbf{k}})]\,\boldsymbol{\epsilon}_{:\mathbf{k}}]$$

Again, moving in the direction opposite to the gradient by a distance governed by the learning rate η the following term should be added to the weight/parameter vector $\mathbf{b}_{:\mathbf{k}}$.

$$X^t\,[\mathbf{f}(X\mathbf{b}_{:\mathbf{k}})(1 - \mathbf{f}(X\mathbf{b}_{:\mathbf{k}}))\,\boldsymbol{\epsilon}_{:\mathbf{k}}]\,\eta \;=\; -\,X^t\boldsymbol{\delta}_{:\mathbf{k}}\,\eta \tag{9.29}$$

9.4.4 Matrix Version

Of course the boxed equations may be rewritten in matrix form. The m-by-n_y prediction matrix $\hat{Y}$ has a column for each output variable.

$$\boxed{\hat{Y} \;=\; \mathbf{f}(XB)} \tag{9.30}$$

The m-by-n_y negative of the error matrix E is the difference between the predicted and actual/target output/response.

$$\boxed{E \;=\; \hat{Y} - Y} \tag{9.31}$$

The m-by-n_y delta matrix Δ adjusts the error according to the slopes within $\mathbf{f}'(XB)$ and is the elementwise matrix (Hadamard) product of $\mathbf{f}'(XB)$ and E.

$$\boxed{\Delta \;=\; \mathbf{f}'(XB) \circ E} \tag{9.32}$$

In math, the Hadamard product is denoted by the $\circ$ operator, while in ScalaTion it is denoted by the `**` operator. Finally, the n_x-by-n_y parameter matrix B is updated by $-X^t\Delta\eta$.

$$\boxed{B \;=\; B - X^t\Delta\eta} \tag{9.33}$$

The corresponding code in the `train0` method is shown below:

```
def train0 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_2L =
{
    var sse0 = Double.MaxValue                                // hold prior value of sse
    for (epoch <- 1 to maxEpochs) {                           // iterate over each epoch
        val yp = f0.fM (x_r *: b)                             // Yp = f0(XB)
        ee     = yp - y_r                                     // negative of error matrix
        val d  = f0.dM (yp) ** ee                             // delta matrix for yp
        b     -= x_r.t * d * eta                              // update 'b' parameters

        val sse = sseF (y_r, f0.fM (x_r *: b))
        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
```

```
        this
    } // train0
```

Note: `f0.fM` is the matrix version of the activation function and it is created using the `matrixize` high-order function that takes a vector function as input.

```
def matrixize (f: FunctionV_2V): FunctionM_2M = (x: MatriD) => x.map (f(_))
val fM = matrixize (fV)
```

Similary, `f0.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 η 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 f0      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,
                    f0: AFF = f_sigmoid, val itran: FunctionV_2V = null)
      extends PredictorMat2 (x, y, fname_, hparam)

def parameters: NetParams = Array (b)
def train0 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_2L =
def train (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_2L =
override def train2 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_2L =
def buildModel (x_cols: MatriD): NeuralNet_2L =
def predictV (z: VectoD): VectoD = f0.fV (b dot z)
def predictV (z: MatriD = x): MatriD = f0.fM (b * z)
```

9.4.5 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 Aggrregate
// Output Variables (3):
// 1. SLUMP (cm)
// 2. FLOW (cm)
// 3. 28-day Compressive STRENGTH (Mpa)
```

Create a `NeuralNet_2L` model to predict values for the three outputs y_0, y_1 and y_2. Compare with the results of using three `Perceptron`s.

2. Create a `NeuralNet_2L` model to predict values for the one output for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`.

3. Were the results in for `AutoMPG` dataset the same for `Perceptron` and `NeuralNet_2L`? Please explain. In general, is a `NeuralNet_2L` equivalent to n_y `Perceptron`s?

4. Draw a `NeuralNet_2L` with $n_x = 4$ input nodes and $n_y = 2$ output nodes. Label the eight edges with weights from the 4-by-2 weight matrix $B = [b_{jk}]$. Write the two model equations, one for y_0 and one for y_1. Combine these two equations into one vector equation for $\mathbf{y} = [y_0, y_1]$. Given column vector $\mathbf{x} = [1, x_1, x_2, x_3]$, express $\hat{\mathbf{y}} = \mathbf{f}(B^t\mathbf{x})$ at the scalar level.

9.5 Three-Layer Neural Networks

The NeuralNet_3L class supports 3-layer (input, hidden and output) Neural Networks. The inputs into a Neural Net are given by the input vector $\mathbf{x}$, while the outputs are given by the output vector $\mathbf{y}$. Between these two layers is a single hidden layer, whose intermediate values will be denoted by the vector $\mathbf{z}$. Each input x_j is associated with an input node in the network, while each output y_k is associated with an output node in the network. 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 $B = [b_{hk}]_{n_z \times n_y}$.

9.5.1 Model Equation

The model equation for NeuralNet_3L can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f_1}(B \cdot \mathbf{f_0}(A \cdot \mathbf{x})) + \boldsymbol{\epsilon} \;=\; \mathbf{f_1}(B^t \mathbf{f_0}(A^t \mathbf{x})) + \boldsymbol{\epsilon}} \tag{9.34}$$

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 $\mathbf{f_0}$ vectorized activation function. The outermost matrix-vector product multiplies the transpose of the n_z-by-n_y matrix B by the n_z-by-1 vector results, producing an n_y-by-1 vector, which is passed into the $\mathbf{f_1}$ vectorized activation function.

Intercept/Bias

As before, one may include an intercept in the model (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. ScalaTion supports this with the NetParam case class.

```
@param w  the weight matrix
@param b  the optional bias/intercept vector (null => not used)

case class NetParam (w: MatriD, var b: VectoD = null)

    def dot (x: VectoD): VectoD = (w dot x) + b
```

Following this approach, there is no need for the special nodes and the `dot` product is re-defined to add the bias `b` to the regular matrix-vector `dot` product (`w dot x`). Note, the NetParam class defines several other methods as well. The vector version of the predict method in NeuralNet_3L uses this `dot` product to make predictions.

```
def predictV (v: VectoD): VectoD = f1.fV (b dot f0.fV (a dot v))
```

9.5.2 Training

Given a training dataset made up of an m-by-n_x input matrix X and an m-by-n_y output matrix Y, training consists of making a prediction $\hat{Y}$,

$$\boxed{\hat{Y} \;=\; \mathbf{f_1}(\mathbf{f_0}(XA)B)} \tag{9.35}$$

and determining the error in prediction $E = Y - \hat{Y}$ with the goal of minimizing the error.

$$\min_B \|Y - \mathbf{f_1}(\mathbf{f_0}(XA)B)\|_F \tag{9.36}$$

Training involves an interative procedure (e.g., stochastic gradient descent) that adjusts parameter values (for weights and biases) to minimize an objective/loss 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 `NetParam` A and `NetParam` B. Roughly as outlined in section 3 of [19], the training can be broken into four steps:

1. Compute predicted values for output $\hat{\mathbf{y}}$ and compare with actual values $\mathbf{y}$ to determine the error $\mathbf{y} - \hat{\mathbf{y}}$.

2. Back propagate the adjusted error to determine the amount of correction needed at the output layer. Record this as vector $\boldsymbol{\delta}^1$.

3. Back propagate the correction to the hidden layer and determine the amount of correction needed at the hidden layer. Record this as vector $\boldsymbol{\delta}^0$.

4. Use the delta vectors, $\boldsymbol{\delta}^1$ and $\boldsymbol{\delta}^0$, to makes updates to `NetParam` A and `NetParam` B, i.e., the weights and biases.

9.5.3 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 `NetParam`s 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 $\hat{\mathbf{y}}$ are calculated. First values for the hidden layer $\mathbf{z}$ are calculated, where the values for hidden node h, z_h, is given by

   $$z_h \;=\; f_0(\mathbf{a_{:h}} \cdot \mathbf{x}) \qquad \text{for } h = 0, \ldots, n_z - 1$$

   where f_0 is the first activation function (e.g., sigmoid), $\mathbf{a_{:h}}$ is column-h of the A weight matrix, and $\mathbf{x}$ is an input vector for a training sample/instance (row in the data matrix). Typically, several samples (referred to as a *batch*) are used in each step. Next, the values computed at the hidden layer are used to produce predicted outputs $\hat{\mathbf{y}}$, where the value for output node k, $\hat{y}_k$, is given by

   $$\hat{y}_k \;=\; f_1(\mathbf{b_{:k}} \cdot \mathbf{z}) \qquad \text{for } k = 0, \ldots, n_y - 1$$

where the second activation function f_1 may be the same as (or different from) the one used in the hidden layer and $\mathbf{b_{:k}}$ is column-k of the B weight matrix. Now the difference between the actual and predicted output can be calculated by simply subtracting the two vectors, or elementwise, the error for the k^{th} output, ϵ_k, is given by

$$\epsilon_k \ = \ y_k - \hat{y}_k \qquad \text{for } k = 0, \dots, n_y - 1$$

Obviously, for subsequent iterations, the updated/corrected weights rather than the initial random weights are used.

2. Back propagate from output layer: Given the computed error vector $\boldsymbol{\epsilon}$, the delta/correction vector $\boldsymbol{\delta}^1$ for the output layer may be calculated, where for output node k, δ^1_k is given by

$$\boxed{\delta^1_k \ = \ -\, f_1'(\mathbf{b_{:k}} \cdot \mathbf{z})\, \epsilon_k} \qquad \text{for } k = 0, \dots, n_y - 1 \tag{9.37}$$

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

$$\frac{\partial hse}{\partial b_{hk}} \ = \ z_h \delta^1_k \tag{9.38}$$

3. Back propagate from hidden layer: Given the delta/correction vector $\boldsymbol{\delta}^1$ from the output layer, the delta vector for the hidden layer $\boldsymbol{\delta}^0$ may be calculated, where for hidden node h, δ^0_h is given by

$$\boxed{\delta^0_h \ = \ f_0'(\mathbf{a_{:h}} \cdot \mathbf{x})\, [\mathbf{b_h} \cdot \boldsymbol{\delta}^1]} \qquad \text{for } h = 0, \dots, n_z - 1 \tag{9.39}$$

This equation is parallel to the one given for δ^1_k in that an error-like factor multiplies the derivative of the activation function. In this case, the error-like factor is the weighted average of the δ^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.

$$\mathbf{b_h} \cdot \boldsymbol{\delta}^1 \ = \ \sum_{k=0}^{n_y - 1} b_{hk}\, \delta^1_k$$

The partial derivative of hse with respect to the weight connecting input node j with hidden node h, a_{jh}, is given by

$$\frac{\partial hse}{\partial a_{jh}} \ = \ x_j \delta^0_h \tag{9.40}$$

4. Update weights: The weight matrices A and B, connecting input to hidden and hidden to output layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate η which moderates the adjustments to the weights.

$$b_{hk} \;=\; b_{hk} \;-\; z_h\, \delta^1_k\, \eta \tag{9.41}$$

$$a_{jh} \;=\; a_{jh} \;-\; x_j\, \delta^0_h\, \eta \tag{9.42}$$

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.

9.5.4 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 optimzer 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 $\mathbf{f_0}$ to the matrix product XA. The predicted output values $\hat{Y}$ are similarly computed by applying the second matrixized activation function $\mathbf{f_1}$ to the matrix product ZB.

$$\boxed{Z \;=\; \mathbf{f_0}(XA)} \tag{9.43}$$

$$\boxed{\hat{Y} \;=\; \mathbf{f_1}(ZB)} \tag{9.44}$$

2. The negative of the error matrix E is just the difference between the predicted and actual/target values.

$$\boxed{E \;=\; \hat{Y} \,-\, Y} \tag{9.45}$$

3. This information is sufficient to calculate delta matrices: Δ^1 for adjusting B and Δ^0 for adjusting A. The Δ^1 matrix is the elementwise matrix (Hadamard) product of $\mathbf{f}_1'(ZB)$ and E.

$$\boxed{\Delta^1 \ = \ \mathbf{f}_1'(ZB) \circ E} \tag{9.46}$$

The Δ^0 matrix is the elementwise matrix (Hadamard) product of $\mathbf{f}_0'(XA)$ and $\Delta^1 B^t$.

$$\boxed{\Delta^0 \ = \ \mathbf{f}_0'(XA) \circ (\Delta^1 B^t)} \tag{9.47}$$

4. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate η) for updating the parameter/weight matrices, A and B.

$$\boxed{B \ = \ B \, - \, Z^t \Delta^1 \, \eta} \tag{9.48}$$

$$\boxed{A \ = \ A \, - \, X^t \Delta^0 \, \eta} \tag{9.49}$$

The corresponding SCALATION code for the `train0` method is show below.

```
def train0 (x_r: MatriD = x, y_r: MatriD = 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 (x_r *: a)                              // Z  = f0(XA)
        var yp = f1.fM (z *: b)                                // Yp = f1(ZB)
        ee     = yp - y_r                                      // negative of the error matrix
        val d1 = f1.dM (yp) ** ee                              // delta matrix for yp
        val d0 = f0.dM (z) ** (d1 * b.w.t)                     // delta matrix for z
        a -= (x_r.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_r, f1.fM (f0.fM (x_r *: a)) *: b)
        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
        this
    } // for
    this
} // train0
```

For stochastic gradient descent in the `Optimizer` object, the inner loop divides the training dataset into `nB` *batches*. A batch is a randomly selected group/batch of rows. Each batch (`ib`) is passed to the `updateWeight (x(ib), y(ib))` method that updates the A and B parameter/weight matrices.

Neural networks may be used for prediction/regression as well as classification problems. For prediction/regression, the number of output nodes would corresponding to the number of responses. For example, in the `ExampleConrete` example there are three response columns, requiring three instances of `Regression` or one instance of `NeuralNet_3L`. Three separate `NeuralNet_3L` instances each with one output node could be used as well. Since some activation functions have limited ranges, it is common practice for these types of problems to let the activation function in the last layer be identity `id`. If this is not done, response columns need to be re-scaled based on the training dataset. Since the testing dataset may have values outside this range, this approach may not be ideal.

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.

$$f_i(\mathbf{t}) = \frac{e^{t_i}}{\mathbf{1} \cdot e^{\mathbf{t}}} \qquad \text{for } i = 0, \ldots, n-1$$

9.5.5 Example Error Calculation Problem

Draw a 3-layer (input, hidden and output) Neural Network (with sigmoid activation), where the number of nodes per layer are $n_x = 2, n_z = 2$ and $n_y = 1$.

Input to Hidden Layer Parameters

Initialize bias vector $\mathbf{a^b}$ to $[.1, .1]$ and weight matrix A (n_x-by-n_z) to

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

Hidden to Output Layer Parameters

Initialize bias vector $\mathbf{b^b}$ to $[.1]$ and weight matrix B (n_z-by-n_y) to

$$\begin{bmatrix} .5 \\ .6 \end{bmatrix}$$

Compute the Error for the First Iteration

Let $\mathbf{x} = [x_0, x_1] = [2, 1]$ and $y_0 = .8$, and then compute the error $\epsilon_0 = y_0 - \hat{y_0}$, by feeding the values from vector $\mathbf{x}$ forward. First compute values at the hidden layer for $\mathbf{z}$.

$$\begin{aligned}
z_h &= f_0(\mathbf{a_{:h}} \cdot \mathbf{x} + a^b_h) \\
z_0 &= f_0(\mathbf{a_{:0}} \cdot \mathbf{x} + a^b_0) \\
z_0 &= f_0([2.0, 1.0] \cdot [0.1, 0.3] + 0.1) \\
z_0 &= f_0(0.6) \;=\; 0.645656 \\
z_1 &= f_0(\mathbf{a_{:1}} \cdot \mathbf{x} + b^b_0) \\
z_1 &= f_0([2.0, 1.0] \cdot [0.2, 0.4] + 0.1) \\
z_1 &= f_0(0.9) \;=\; 0.710950
\end{aligned}$$

One may compute the values for sigmoid activation function as follows:

```
println (ActivationFun.sigmoidV (VectorD (0.6, 0.9)))
```

Then compute predicted values at the output layer for $\mathbf{\hat{y}}$.

$$
\begin{aligned}
y_k &= f_1(\mathbf{b_{:k}} \cdot \mathbf{z} + b^b_k) \\
y_0 &= f_1(\mathbf{b_{:0}} \cdot \mathbf{z} + b^b_0) \\
y_0 &= f_1([0.5, 0.6] \cdot [0.645656, 0.71095] + 0.1) \\
y_0 &= f_1(0.749398) \;=\; 0.6790458
\end{aligned}
$$

Therefore, the error is

$$
\begin{aligned}
\epsilon_0 &= y_0 - \hat{y_0} \\
\epsilon_0 &= 0.8 - 0.6790458 \;=\; 0.1209542
\end{aligned}
$$

`NeuralNet_3L` **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 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 = hp,
                     f0: AFF = f_tanh, f1: AFF = f_id,
                     val itran: FunctionV_2V = null)
       extends PredictorMat2 (x, y, fname_, hparam)        // sets eta in parent class

def compute_df_m (n: Int): Int = (nx + 2) * n
def parameters: NetParams = Array (a, b)
def train0 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_3L =
def train (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_3L =
override def train2 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_3L =
def buildModel (x_cols: MatriD): NeuralNet_3L =
def predictV (v: VectoD): VectoD = f1.fV (b dot f0.fV (a dot v))
def predictV (v: MatriD = x): MatriD = f1.fM (b * f0.fM (a * v))
def showEstat ()
```

9.5.6 Exercises

1. **Delta Vectors**: For the example error calculation problem, calculate the $\boldsymbol{\delta}^1 = [\delta_0^1]$ vector. Explain equation 9.31 (with explicit weights and biases included) for computing δ_k^1 and use it for the calculation.

$$\delta_k^1 \;=\; -\, f_1'(\mathbf{b_{:k}} \cdot \mathbf{z} + b_k^b)\, \epsilon_k$$

   Calculate the $\boldsymbol{\delta}^0 = [\delta_0^0, \delta_1^0]$ vector. Explain equation 9.33 (with explicit weights and biases included) for computing δ_h^0 and use it for the calculation.

$$\delta_h^0 \;=\; f_0'(\mathbf{a_{:h}} \cdot \mathbf{x} + a_h^b)\, [\mathbf{b_h} \cdot \boldsymbol{\delta}^1]$$

2. **Parameter Update Equations**: Use the $\boldsymbol{\delta}_1$ vector to update weight matrix B, i.e., for each row h,

$$\mathbf{b_h} \;=\; \mathbf{b_h} - z_h \boldsymbol{\delta}_1 \eta$$

   and update the bias vector $\mathbf{b^b}$ as follows:

$$\mathbf{b^b} \;=\; \mathbf{b^b} - \boldsymbol{\delta}_1 \eta$$

   Use the $\boldsymbol{\delta}_0$ vector to update weight matrix A, i.e., for each row j,

$$\mathbf{a_j} \;=\; \mathbf{a_j} - x_j \boldsymbol{\delta}_0 \eta$$

   and update the bias vector $\mathbf{a^b}$ as follows:

$$\mathbf{a^b} \;=\; \mathbf{a^b} - \boldsymbol{\delta}_0 \eta$$

3. Derive equation 9.32 for the partial derivative of hse w.r.t. b_{hk},

$$\frac{\partial hse}{\partial b_{hk}} \;=\; z_h \delta_k^1$$

   by defining pre-activation value $v_k = \mathbf{b_{:k}} \cdot \mathbf{z}$ and applying the following chain rule:

$$\frac{\partial hse}{\partial b_{hk}} \;=\; \frac{\partial hse}{\partial v_k} \frac{\partial v_k}{\partial b_{hk}}$$

4. Explain the formulations for the two delta matrices.

$$\begin{aligned}\Delta^1 &= \mathbf{f}_1'(ZB) \circ E \\ \Delta^0 &= \mathbf{f}_0'(XA) \circ (\Delta^1 B^t)\end{aligned}$$

5. 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.

6. Create a `NeuralNet_3L` model to predict values for the one output for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`, (c) `NeuralNet_2L`.

9.6 Multi-Hidden Layer Neural Networks

The `NeuralNet_XL` class supports basic x-layer (input, {hidden} and output) Neural Networks. For example a four layer neural network with have four layers of nodes with (one input layer numbered 0, two hidden layers numbered 1 and 2, and one output layer numbered 3). Note, since the input layer's purpose is just to funnel the input into the model, it is also common to refer to such a neural network as a three layer network. This has the advantage that number layers now corresponds to the number parameter/weight matrices. In ScalaTion, the number of active layers is denoted by `nl` (which in this case equals 3). Since arrays of matrices are used in the ScalaTion 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

```
    type NetParams = IndexedSeq [NetParam]
```

9.6.1 Model Equation

The equations for `NeuralNet_XL` are the same as those used for `NeuralNet_3L`, except that the calculations are repeated layer by layer in a forward direction for prediction. The model equation for a four layer `NeuralNet_XL` can written in vector form as follows:

$$\boxed{\mathbf{y} \;=\; \mathbf{f_2}(B_2 \cdot \mathbf{f_1}(B_1 \cdot \mathbf{f_0}(B_0 \cdot \mathbf{x}))) + \boldsymbol{\epsilon}} \tag{9.50}$$

where B_l is the `NetParam` (weight matrix and bias vector) connecting layer l to layer $l+1$ and $\mathbf{f_l}$ is the vectorized activation function at layer $l+1$.

9.6.2 Training

As before, the training dataset consists of an m-by-n_x input matrix X and an m-by-n_y output matrix Y. During training, the predicted values $\hat{Y}$ are compared to actual/target values Y,

$$\boxed{\hat{Y} \;=\; \mathbf{f_2}(\mathbf{f_1}(\mathbf{f_0}(XB_0)B_1)B_2)} \tag{9.51}$$

to compute an error matrix $E = Y - \hat{Y}$, to be minimized.

$$\min_B \|Y - \mathbf{f_2}(\mathbf{f_1}(\mathbf{f_0}(XB_0)B_1)B_2)\|_F \tag{9.52}$$

Corrections based on these errors are propogated backward through the network to improve the parameter estimates (weights and biases) layer by layer.

9.6.3 Optimization

The seven boxed equations from the previous section become six due to unification of the last two. As before, the optimizers compute a predicted output matrix and then take differences between the actual/target values and these predicted values to compute an error matrix. These computed matrices are then used to compute delta matrices that form the basis for updating the weight matrices. Again for simplicity, biases are ignore in the equations below, but are taken care of in the code through the `NetParam` abstraction. See the exercises for details.

1. The values are feed forward through the network, layer by layer. For layer l, these values are stored in matrix Z_l. The first layer is the input, so $Z_0 = X$. For the rest of the layers, Z_{l+1} equals the result of activation function $\mathbf{f}_l$ being applied to the product of the previous layer's Z_l matrix times its parameter matrix B_l.

$$\boxed{Z_0 \;=\; X} \tag{9.53}$$

   For each layer l in the forward direction:

$$\boxed{Z_{l+1} \;=\; \mathbf{f}_l(Z_l B_l)} \tag{9.54}$$

2. The negative of the error matrix E is just the difference between the predicted and actual/target values, where $\hat{Y} = Z_{nl}$.

$$\boxed{E \;=\; \hat{Y} - Y} \tag{9.55}$$

3. This information is sufficient to calculate delta matrices Δ^l. For the last layer:

$$\boxed{\Delta^{nl-1} \;=\; \mathbf{f}'_{nl-1}(Z_{nl-1}B_{nl-1}) \circ E} \tag{9.56}$$

   For the rest of layers in the backward direction with l being decremented:

$$\boxed{\Delta^l \;=\; \mathbf{f}'_l(Z_l B_l) \circ (\Delta^{l+1} B^t_{l+1})} \tag{9.57}$$

4. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate η) for updating the parameter/weight matrices, B_l for each layer l.

$$\boxed{B_l \;=\; B_l \;-\; Z^t_l \Delta^l \,\eta} \tag{9.58}$$

The implementation of the `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.

```
def train0 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_XL =
{
    var sse0 = Double.MaxValue                                 // hold prior value of sse
    val z    = Array.ofDim [MatriD] (nl+1); z(0) = x_r         // storage: activations f(Z_l B_l)
    val d    = Array.ofDim [MatriD] (nl)                       // storage: deltas

    for (epoch <- 1 to maxEpochs) {                            // iterate over each epoch
        for (l <- layers) z(l+1) = f(l).fM (z(l) *: b(l))      // feedforward and store activations
        ee      = z.last - y_r                                 // 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.normFSq
            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
        } // for
        this
    } // train0
```

The `train0` method is kept simple to facilitate understanding. In practice, the `train` and `train2` methods should be used.

9.6.4 Number of Nodes in Hidden Layers

The array `nz` gives the number of nodes for each of the hidden layers. If the user does not specify the number of nodes for each hidden layer, then based on the number of input nodes `nx` and number of output nodes `ny`, defaults are utilized according to one of two rules. The comments should be switched to change rules.

```
//  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 `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.

9.6.5 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

class NeuralNet_XL (x: MatriD, y: MatriD,
                    private var nz: Array [Int] = null,
                    fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp,
                    f :  Array [AFF] = Array (f_tanh, f_tanh, f_id),
                    val itran: FunctionV_2V = null)
      extends PredictorMat2 (x, y, fname_, hparam)         // sets eta in parent class

def compute_nz (nx: Int): Array [Int] =
def compute_nz (nx: Int, ny: Int): Array [Int] =
def compute_df_m (n: Array [Int]): Int =
def parameters: NetParams = b
def train0 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_XL =
def train (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_XL =
override def train2 (x_r: MatriD = x, y_r: MatriD = y): NeuralNet_XL =
def buildModel (x_cols: MatriD): NeuralNet_XL =
def getNetParam (layer: Int = 1) = b(layer)
def predictV (v: VectoD): VectoD =
def predictV (v: MatriD = x): MatriD =
```

When a neural network is being used to for prediction/regression problems, it is often the case the there will be a single output node. ScalaTion provides the `NeuralNet_XL1` class for this.

`NeuralNet_XL1` **Class**

Class Methods:

```
@param x       the m-by-nx data/input matrix (training data having m input vectors)
@param y       the m response/output vector (training data having m output valaues)
@param nz_     the number of nodes in each hidden layer, e.g., Array (9, 8)
@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

class NeuralNet_XL1 (x: MatriD, y: VectoD,
                    nz_ : Array [Int] = null,
                    fname_ : Strings = null, hparam: HyperParameter = hp,
                    f: Array [AFF] = Array (f_tanh, f_tanh, f_id),
                    itran: FunctionV_2V = null)
      extends NeuralNet_XL (x, MatrixD (y), nz_, fname_, hparam, f, itran)
```

9.6.6 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 propogation 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 propogation 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`.

6. **Tuning the Hyper-Parameters**: The learning rate η (`eta` in the code) needs frequent tuning. ScalaTion as with most packages has limited auto-tuning of the learning rate. Tune the other hyper-parameters for the `AutoMPG` dataset.

```
/** hyper-parameters for tuning the optimization algorithms - user tuning
 */
val hp = new HyperParameter
hp += ("eta", 0.1, 0.1)                 // learning/convergence rate
hp += ("bSize", 20, 20)                 // mini-batch size, common range 10 to 30
hp += ("maxEpochs", 500, 500)           // maximum number of epochs/iterations
hp += ("patience", 4.0, 4.0)            // number of subsequent upward steps before stopping
hp += ("lambda", 0.01, 0.00)            // regularization hyper-parameter

// example adjustments - to be done before creating the neural network model
hp("eta")       = 0.05
hp("bSize")     = 25
hp("maxEpochs") = 200
hp("patience")  = 5
hp("lambda")    = 0.02
```

Note, in other packages `patience` is number of upward steps, not the number of subsequent upward steps.

7. **Tuning the Network Architecture**: The architecture of the neural network can be tuned by (1) changing the number of layers, (2) changing the number of nodes in each hidden layer, and (3) changing the activation functions. Tune the architecture for the `AutoMPG` dataset. The number of layers and number of nodes in each layer should only be increased when there is non-trivial improvement.

9.7 Transfer Learning

The first sets of transformations (from input to the first hidden layer and between the first and second hidden layers) in a Neural Network allow nonlinear effects to be created to better capture characteristics of the system under study. These are taken in combination in later latter layers of the Neural Network. The thought is that related problems share similar nonlinear effects. In other words, two datasets on related problems used to train two Neural Networks are likely to develop similar nonlinear effects at some point in their training. If this is case, the traininhg of the first Neural Netork could expedite the training of the second Neural Network. Some research has show the Quality of Fit (QoF) may also be enhanced [**?**] as well.

The easiest way to imagine this is to have two four-layer Neural Networks, say with 30 inputs for the first and 25 for the second. Let the first hidden layer have 50 nodes and third have 20 nodes, with an output layer for the single output/response value. The only difference in node count is in the input layer. For the first Neural Network, the sizes of the parameter/weight matrices are 30-by-50, 50-by-20 and 20-by-1. The only difference in the second Neural Network is that the first matrix is 25-by-50. After the first Neural Network is trained, its second matrix (50-by-20) along with its associated bias vector could be transferred to the second Neural Network. When training starts for the second Neural Network, random initialization is skipped for this matrix (and its associated bias vector). A training choice is whether to freeze this layer or allow its values to be adjusted during back-propogation.

`NeuralNet_XLT` **Class**

Class Methods:

```
@param x         the m-by-nx input matrix (training data consisting of m input vectors)
@param y         the m output vector (training data consisting of m output integer values)
@param nz        the number of nodes in each hidden layer, e.g., Array (9, 8) => sizes 9 and 8
@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 l_tran    the first layer to be transferred in
@param transfer  the saved network parameters from layers of a related neural network
@param itran     the inverse transformation function returns responses to original scale

class NeuralNet_XLT (x: MatriD, y: MatriD,
                     nz: Array [Int], fname_ : Strings = null,
                     hparam: HyperParameter = hp,
                     f: Array [AFF] = Array (f_tanh, f_tanh, f_id),
                     l_tran: Int = 1, transfer: NetParams = null,
                     itran: FunctionV_2V = null)
      extends NeuralNet_XL (x, y, nz, fname_, hparam, f, itran)

def trim (bl: NetParam, tl: NetParam): NetParam =
```

9.7.1 Exercises

1.

9.8 Extreme Learning Machines

An Extreme Learning Machine (ELM) may be viewed as three-layer Neural Network with the first set of fixed-parameters (weights and biases) frozen. The values for these parameters may be randomly generated or transferred in from a Neural Network. With the first set of fixed-parameters frozen, only the second set of parameters needs to be optimized. When the second activation function is the identity function (`id`), the optimization problem is the same as for the Regression problem, so that matrix factorization may be used to train the model. This greatly reduces the training time over Neural Networks. Although the first set of fixed-parameters is not optimized, nonlinear effects are still created at the hidden layer and there may be enough flexibility left in the second set of parameters to retain some of the advantages of Neural Networks.

9.8.1 Model Equation

A relatively simple form of Extreme Learning Machine in ScalaTion is `ELM_3L1`. It allows for a single output/response variable y and multiple input/predictors variables $\mathbf{x} = [x_0, x_1, \dots x_{n-1}]$. The number of nodes per layer are n_x for input, n_z for hidden and $n_y = 1$ for output. The second activation function $\mathbf{f_1}$ is implicitly `id` and be left out.

The model equation for `ELM_3L1` can written in vector form as follows,

$$\boxed{y \;=\; \mathbf{b} \cdot \mathbf{f_0}(A^t\mathbf{x})) + \boldsymbol{\epsilon}} \tag{9.59}$$

where A is the first layer `NetParam` consisting of an n_x-by-n_z weight matrix and an n_z bias vector. In ScalaTion, these parameters are initilized as follows (see exercises for details):

```
private var a = new NetParam (weightMat3 (n, nz, s),
                              weightVec3 (nz, s))
```

The second layer is simply an n_z weight vector $\mathbf{b}$. The first layer's weights and biases A are frozen, while the second layer parameters $\mathbf{b}$ are optimized.

9.8.2 Training

Given a dataset $(X, \mathbf{y})$, training will be used to adjust values of the parameter vector $\mathbf{b}$. The objective is to minimize the distance between the actual and predicted response vectors.

$$\boxed{f_{obj} \;=\; ||\mathbf{y} - \mathbf{b} \cdot \mathbf{f_0}(XA)||} \tag{9.60}$$

9.8.3 Optimization

An optimal value for parameter vector $\mathbf{b}$ may be found using `Regression`.

```
def train (x_r: MatriD = x, y_r: VectoD = y): ELM_3L1 =
{
    val z = f0.fM (x_r *: a)                                         // Z  = f(XA)
    val rg = new Regression (z, y_r)                                 // Regression
    rg.train ()
    b = rg.parameter
```

```
        this
    } // train
```

Extreme Learning Machines typically are competitive with lower order polynomial regression and may have fewer parameters than `CubicXRegression`.

`ELM_3L1` **Class**

Class Methods:

```
    @param x        the m-by-n input matrix (training data consisting of m input vectors)
    @param y        the m output vector (training data consisting of m output scalars)
    @param nz       the number of nodes in hidden layer (-1 => use default formula)
    @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 itran    the inverse transformation function returns responses to original scale

class ELM_3L1 (x: MatriD, y: VectoD, private var nz: Int = -1,
               fname_ : Strings = null, hparam: HyperParameter = null,
               f0: AFF = f_tanh, val itran: FunctionV_2V = null)
      extends PredictorMat (x, y, fname_, hparam)

    def compute_df_m (nz_ : Int): Int = nz_
    def parameters: VectoD = b
    def train (x_r: MatriD = x, y_r: VectoD = y): ELM_3L1 =
    def buildModel (x_cols: MatriD): ELM_3L1 =
    override def predict (v: VectoD): Double = b dot f0.fV (a dot v)
    override def predict (v: MatriD = x): VectoD = f0.fM (a * v) * b
```

When the second activation function is not `id`, then the optimization of the second set of parameters works like it does for Transformed Regression. Also, when multiple outputs are needed, the `ELM_3L` may be used.

9.8.4 Exercises

1. Create an `ELM_3L1` model to predict values for the `AutoMPG` dataset. Compare with the results of using the following models: (a) `Regression`, (b) `Perceptron`, (c) `NeuralNet_2L`, (d) `NeuralNet_3L`, (e) `NeuralNet_XL`

2. Time each of the six model given above using ScalaTion's `time` method (`import scalation.util.time`).

```
        def time [R] (block: => R): R =
```

This method can time the execution of any block of code (`time { block }`).

3. Compare the following stategies for intializing `NetParam` A (weights and biases for the first layer). ...

Chapter 10

Time-Series/Temporal Models

Time-Series/Temporal Models are used in order to make forecasts into the future. Applications are numerous and include weather, sales projections, energy consumption, economic indicators, financial instruments and traffic conditions. So far, predictive models have been of the form

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

where $\mathbf{x}$ is the vector of predictive variables/features, y is the response variable and ϵ is the residual/error.

In order to fit the parameters $\mathbf{b}$, m samples are collected into a data/input matrix X and a response/output vector $\mathbf{y}$. The samples are treated as though they are independent. In many case, such as data collected over time, they are often not independent. For example, the current Gross Domestic Product (GDP) will likely show 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.,

$$y_t = f(y_{t-1}, \dots, y_{t-p}; \boldsymbol{\phi}) + \epsilon_t \tag{10.1}$$

where $\boldsymbol{\phi}$ is a vector of parameters analogous to $\mathbf{b}$.

Of course, other predictor variables $\mathbf{x}$ 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 obejct 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
```

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 recorded in the Lake Level Times-series Dataset. (see `cran.r-project.org/web/packages/fpp/fpp.pdf`):

```
val m = 98
val t = VectorD.range (0, m)
val y = VectorD (580.38, 581.86, 580.97, 580.80, 579.79, 580.39, 580.42, 580.82, 581.40, 581.32,
                 581.44, 581.68, 581.17, 580.53, 580.01, 579.91, 579.14, 579.16, 579.55, 579.67,
                 578.44, 578.24, 579.10, 579.09, 579.35, 578.82, 579.32, 579.01, 579.00, 579.80,
                 579.83, 579.72, 579.89, 580.01, 579.37, 578.69, 578.19, 578.67, 579.55, 578.92,
                 578.09, 579.37, 580.13, 580.14, 579.51, 579.24, 578.66, 578.86, 578.05, 577.79,
                 576.75, 576.75, 577.82, 578.64, 580.58, 579.48, 577.38, 576.90, 576.94, 576.24,
                 576.84, 576.85, 576.90, 577.79, 578.18, 577.51, 577.23, 578.42, 579.61, 579.05,
                 579.26, 579.22, 579.38, 579.10, 577.95, 578.12, 579.75, 580.85, 580.41, 579.96,
                 579.61, 578.76, 578.18, 577.21, 577.13, 579.10, 578.25, 577.91, 576.89, 575.96,
                 576.80, 577.68, 578.38, 578.52, 579.74, 579.31, 579.89, 579.96)
```

First plot this dataset and then look at its Auto-Correlation Function (ACF).

```
new Plot (t, y, null, "Plot of y vs. t", true)
```

The Auto-Correlation Function (ACF) measures how much the past can influence a forecast. If the forecast is for time t, then the past time points are $t-1, \dots, t-p$. The k^{th} lag auto-covariance (auto-correlation), γ_k (ρ_k) is the covariance (correlation) of y_t and y_{t-k}.

$$\gamma_k = \mathbb{C}\left[y_t, y_{t-k}\right] \tag{10.2}$$

$$\rho_k = \text{corr}(y_t, y_{t-k}) \tag{10.3}$$

Note that $\gamma_0 = \mathbb{V}\left[y_t\right]$ 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.slice (2, m)` and `y.slice (0, m-2)`. In ScalaTion, the ACF is computed using the `acf` function from the `ForecasterVec` companion object.

```
import ForecasterVec._

val (sig2, acv, acr) = acf (y)
val zero = new VectorD (acr.dim)
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 ϕ_j

$$y_t \;=\; \delta \;+\; \phi_1 y_{t-1} \;+\; ...\; \phi_p y_{t-p} \;+\; \epsilon_t$$

where $\delta = \mu(1 - \mathbf{1} \cdot \boldsymbol{\phi})$ and the error/noise is represented by ϵ_t. ϵ_t represents noise that shocks the system at each time step. We require that $\mathbb{E}\left[\epsilon_t\right] = 0$, $\mathbb{V}\left[\epsilon_t\right] = \sigma_\epsilon^2$, 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 μ, $z_t = y_t - \mu$.

$$z_t \;=\; \phi_1 z_{t-1} \;+\; ...\; \phi_p z_{t-p} \;+\; \epsilon_t \tag{10.4}$$

Notice that since z_t is zero-centered, the formulas for the mean, variance and covariance are simplified.

$$\mathbb{E}\left[z_t\right] \;=\; 0 \tag{10.5}$$

$$\mathbb{V}\left[z_t\right] \;=\; \mathbb{E}\left[z_t^2\right] \tag{10.6}$$

$$\mathbb{C}\left[z_t, z_{t-k}\right] \;=\; \mathbb{E}\left[z_t z_{t-k}\right] \;=\; \gamma_k \tag{10.7}$$

10.2.1 AR(1) Model

When the future is mainly dependent only on the most recent value, e.g., ρ_1 is high and rest ρ_2, ρ_3, etc. are substantially lower, then an AR(1) model may be sufficient.

$$z_t \;=\; \phi_1 z_{t-1} \;+\; \epsilon_t \tag{10.8}$$

An estimate for the parameter ϕ_1 may be determined from the ACF. Take equation 10.8 and multiply it by z_{t-k} $(k = 1, 2, \ldots, p)$.

$$z_t z_{t-k} \;=\; \phi_1 z_{t-1} z_{t-k} \;+\; \epsilon_t z_{t-k}$$

Taking the expected value of the above equation gives,

$$\mathbb{E}\left[z_t z_{t-k}\right] \;=\; \phi_1 \mathbb{E}\left[z_{t-1} z_{t-k}\right] \;+\; \mathbb{E}\left[\epsilon_t z_{t-k}\right]$$

Using the definition in Equation 10.7 for γ_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 ϵ_t. Now dividing by γ_0 yields

$$\rho_k \;=\; \phi_1 \rho_{k-1} \tag{10.9}$$

An estimate for parameter ϕ_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.

$$y_t \;=\; \delta \;+\; \phi_1 z_{t-1} \;+\; \phi_2 z_{t-2} \;+\; \dots \;+\; \phi_p z_{t-p} \;+\; \epsilon_t$$

Subtracting the mean μ from the response y_t gives

$$z_t \;=\; \phi_1 z_{t-1} \;+\; \dots \;+\; \phi_p z_{t-p} \;+\; \epsilon_t \tag{10.11}$$

Multiplying by z_{t-k} and then taking the expectation produces

$$\gamma_k \;=\; \phi_1 \gamma_{k-1} \;+\; \phi_2 \gamma_{k-2} \;+\; \dots \;+\; \phi_p \gamma_{k-p}$$

Dividing by γ_0 yields

$$\rho_k \;=\; \phi_1 \rho_{k-1} \;+\; \phi_2 \rho_{k-2} \;+\; \dots \;+\; \phi_p \rho_{k-p} \tag{10.12}$$

Equation 10.12 contains p unknowns and by letting $k = 1, 2, \dots, p$, it can be used to generate p equations, or one matrix equation.

$$\begin{aligned}
\rho_1 &= \phi_1 \rho_0 &+\; \phi_2 \rho_1 &\;+\; \dots \;+\; \phi_p \rho_{p-1} \\
\rho_2 &= \phi_1 \rho_1 &+\; \phi_2 \rho_0 &\;+\; \dots \;+\; \phi_p \rho_{p-2} \\
\dots \\
\rho_p &= \phi_1 \rho_{p-1} &+\; \phi_2 \rho_{p-2} &\;+\; \dots \;+\; \phi_p \rho_0
\end{aligned}$$

These are the Yule-Walker equations (often ρ_0 is removed since it equals 1). Letting $\boldsymbol{\rho}$ be the p-dimensional vector of lag auto-correlations and $\boldsymbol{\phi}$ be the p-dimensional vector of parameters/coefficients, we may concisely write

$$\boldsymbol{\rho} \;=\; \left[\rho_{|i-j|}\right]_{i,j} \boldsymbol{\phi} \;=\; P\,\boldsymbol{\phi} \tag{10.13}$$

where P is a p-by-p symmetric Toeplitz matrix with ones on the main diagonal. One way to solve for the parameter vector $\boldsymbol{\phi}$ is to take the inverse (or use related matrix factorization techniques).

$$\boldsymbol{\phi} \;=\; P^{-1}\,\boldsymbol{\rho} \tag{10.14}$$

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
```

The parameter/coefficient vector $\boldsymbol{\phi}$ can be estimated from values in the ACF. In ScalaTion, the coefficients $\boldsymbol{\phi}$ are estimated using the Durbin-Levinson algorithm and extracted from the p^{th} row of the ψ (`psi`) matrix.

Define ψ_{kj} to be ϕ_j for an AR(k) model. Letting k range up to p allows the ϕ_j parameters to be calculated. Letting k range up to the maximum number of lags (`ml`) allows the Partial Auto-Correlation Function (PACF) to be computed.

```
private var phi: VectoD = null                // AR(p) parameters/coefficients
private var psi = new MatrixD (ml+1, ml+1)    // psi matrix (ml = max lags)
```

Invoke the **`durbinLevinson`** method [18] passing in the auto-covariance vector γ (`g`) and the maximum number of lags (`ml`). From 1 up to the maximum number of lags, iteratively compute the following:

$$\begin{aligned}
\psi_{kk} &= \frac{\gamma_k - \Sigma_{j=1}^{k-1} \psi_{k-1,j}\, \gamma_{k-j}}{r_{k-1}} \\
\psi_{kj} &= \psi_{k-1,j} - \psi_{kk}\, \psi_{k-1,k-j} \\
r_k &= r_{k-1}(1 - \psi_{kk}^2)
\end{aligned}$$

```
private def durbinLevinson (g: VectoD, ml: Int): MatriD =
{
   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)
   } // for
} // durbinLevinson
```

The **`train`** method simply calls the **`durbinLevinson`** method to create the Ψ matrix. The parameter/coefficient vector corresponds to the p^{th} row of the Ψ matrix, while the PACF is found in the main diagonal of the Ψ matrix.

```
def train (yy: VectoD = 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 Ψ matrix can be used along with ACF to select the appropropriate type of model. As k increases, the ρ_k will decrease and adding more parameters/coefficients will become of little help. Deciding where to cut off ρ_k from the ACF is somewhat arbitrary, but ψ_{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 `train` method, the AR(p) model can be used for forecasting.

```
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
```

The `forecast` method takes the parameter `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 `steps` gets larger.

```
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
```

`AR` **Class**

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
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 Time-series Dataset. Set parameter ϕ_1 to ρ_1, the first lag auto-correlation. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$ and for `k <- 1 until y.dim`

$$\begin{aligned} z_t &= y_t - \mu \\ \hat{y}_t &= \rho_1 z_{t-1} + \mu \end{aligned}$$

   Plot $\hat{y}_t$ and y_t versus t.

2. Consider the following AR(2) Model.

$$z_t \;=\; \phi_1 z_{t-1} + \phi_2 z_{t-2} \;+\; \epsilon_t$$

   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 ϕ_1 and ϕ_2. Solve for ϕ_1 and ϕ_2 in terms of the first and second lag auto-correlations, ρ_1 and ρ_2. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$, $\hat{y}_1 = y_1$ and for `k <- 2 until y.dim`

$$\begin{aligned} z_t &= y_t - \mu \\ \hat{y}_t &= \phi_1 z_{t-1} + \phi_2 z_{t-2} + \mu \end{aligned}$$

   Plot $\hat{y}_t$ and y_t versus t.

3. Use the ScalaTion class `AR` to develop Auto-Regressive Models for $p = 1, 2, 3$, for the Lake Level Time-series Dataset. Plot $\hat{y}_t$ and y_t versus t for each model. Also, compare the first two models with those developed in the previous exercises.

4. Generate a dataset for an AR(p) model as follows:

```
val 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.

```
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.

```
ar.plotFunc (ar.acF, "ACF")
ar.plotFunc (ar.pacF, "PACF")
```

Choose a value for p and retrain the model.

```
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.

$$\gamma_k = \mathbb{C}\left[y_t, y_{t-k}\right]$$

When y_t is covariance stationary, the covariance is only determined by the lag between the variables and not where they occur in the time series, e.g., $\mathbb{C}[y_4, y_3] = \mathbb{C}[y_2, y_1] = \gamma_1$. Covariance stationary also requires that $\mathbb{E}[y_t] = \mu$, i.e., the mean is time invariant. Repeat the previous exercise, but generate a time series from a process that is not covariance stationary. What can be done to transform such a process into a covariance stationary process?

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 days errors or shocks. In the simplest case,

$$y_t \;=\; \mu \;+\; \theta_1 \epsilon_{t-1} + \epsilon_t$$

To make the data zero-centered, again let $z_t = y_t - \mu$

$$z_t \;=\; \theta_1 \epsilon_{t-1} + \epsilon_t \tag{10.15}$$

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 θ_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 10.1: MA(q) Sample Process (to one decimal place)

t	y_t	z_t	ϵ_{t-1}	$\hat{z}_t = \theta_1 \epsilon_{t-1}$	$\hat{y}_t$	ϵ_t
0	100.0	0.0	-	-	-	0.0
1	110.0	10.0	0.0	0.0	100.0	10.0
2	120.0	20.0	10.0	8.0	108.0	12.0
3	110.0	10.0	12.0	9.6	109.6	0.4
4	90.0	-10.0	0.4	0.3	100.3	-10.3

The question of AR versus MA may be viewed as which column z_t or ϵ_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 influencial shocks to the economy. In such cases, an MA model may provide better forecasts than an AR model.

10.3.1 MA(q) Model

The model equation for an MA(q) model includes the past q values of ϵ_t.

$$y_t \;=\; \mu \;+\; \theta_1 \epsilon_{t-1} \;+\; \ldots \;+\; \theta_q \epsilon_{t-q} \;+\; \epsilon_t$$

Zero-centering the data $z_t = y_t - \mu$ produces

$$z_t \;=\; \theta_1 \epsilon_{t-1} \;+\; \ldots \;+\; \theta_q \epsilon_{t-q} \;+\; \epsilon_t \tag{10.16}$$

In order to estimate the parameter vector $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_q]$, we would like to develop a system of equations like the Yule-Walker equation. Proceding likewise, the auto-covariance function for MA(q) is

$$\gamma_k \;=\; \mathbb{C}\left[z_t, z_{t-k}\right] \;=\; \mathbb{E}\left[z_t, z_{t-k}\right]$$

$$\gamma_k \;=\; \mathbb{E}\left[(\theta_1\epsilon_{t-1} \;+\; \ldots \;+\; \theta_q\epsilon_{t-q} \;+\; \epsilon_t)(\theta_1\epsilon_{t-k-1} \;+\; \ldots \;+\; \theta_q\epsilon_{t-k-q} \;+\; \epsilon_t)\right] \tag{10.17}$$

Letting $k = 0$ will give the variance $\gamma_0 = \mathbb{V}\left[z_t\right]$

$$\gamma_0 \;=\; \mathbb{E}\left[(\theta_1\epsilon_{t-1} \;+\; \ldots \;+\; \theta_q\epsilon_{t-q} \;+\; \epsilon_t)(\theta_1\epsilon_{t-1} \;+\; \ldots \;+\; \theta_q\epsilon_{t-q} \;+\; \epsilon_t)\right]$$

Since the noise shocks are indendent, i.e., $\mathbb{C}\left[\epsilon_t, \epsilon_u\right] = 0$ unless $t = u$, many of the terms in the product drop out.

$$\gamma_0 \;=\; \theta_1^2\mathbb{E}\left[{\epsilon_{t-1}}^2\right] \;+\; \ldots \;+\; \theta_q^2\mathbb{E}\left[{\epsilon_{t-q}}^2\right] \;+\; \mathbb{E}\left[{\epsilon_t}^2\right]$$

The variance of the noise shocks is defined, for any t, to be $\mathbb{V}\left[\epsilon_t\right] = \mathbb{E}\left[{\epsilon_t}^2\right] = \sigma_\epsilon^2$, so

$$\gamma_0 \;=\; (\theta_1^2 \;+\; \ldots \;+\; \theta_q^2 \;+\; 1)\,\sigma_\epsilon^2 \tag{10.18}$$

For $k \in [1, \ldots, q]$, similar equations will be created, only with parameter index swifted so the noise shocks match up.

$$\gamma_k \;=\; (\theta_1\theta_{k+1} \;+\; \ldots \;+\; \theta_q\theta_{q-k} \;+\; \theta_k)\,\sigma_\epsilon^2 \tag{10.19}$$

As before the Auto-Correlation Function (ACF) is simply γ_k/γ_0,

$$\rho_k \;=\; (\theta_1\theta_{k+1} \;+\; \ldots \;+\; \theta_q\theta_{q-k} \;+\; \theta_k)\,\frac{\sigma_\epsilon^2}{\gamma_0} \tag{10.20}$$

Notice that when $k > q$, the ACF will be zero. This is because only the last q noise shocks are included in the model, so any earlier noise shocks before that are forgotten. MA processes will tend to exhibit a more rapid drop off of the ACF compared to the slow decay for AR processes.

Unfotunately, the system of equations that can be generated from equation 10.20 are nonlinear. Consequently, training is more difficult and less efficient.

10.3.2 Training

Training for MA(1)

Training for Moving Average models is easiest to understand for thr case of a single parameter θ_1. In general, training is about errors and so rearranging equation 10.15 gives the following:

$$e_t \;=\; z_t - \theta_1 e_{t-1} \tag{10.21}$$

Given a value for parameter θ_1, this is a recursive equation that can be used to compute subsequent errors from previous ones. Unfortunately, the equation cannot be used to compute the first error e_0. One approach to deal with indeterminancy of e_0 is to assume (or condition on) it being 0.

$$e_0 \;=\; 0 \tag{10.22}$$

Note, this affects more than the first error, since the first affects the second and so on. Next, we may compute a sum of squared errors, in this case called Conditional Sum of Squared Errors (denoted in ScalaTion as `csse`.

$$csse \;=\; \sum_{t=0}^{m-1} e_t^2 \;=\; \sum_{t=1}^{m-1} (z_t - \theta_1 e_{t-1})^2 \tag{10.23}$$

One way to find a near optimal value for θ_1 is to minimize the Conditional Sum of Squared Error.

$$\text{argmin}_{\theta_1} \sum_{t=1}^{m-1} (z_t - \theta_1 e_{t-1})^2 \tag{10.24}$$

As the parameter $\theta_1 \in (-1, 1)$, an optimal value minimizing $csse$ may be found using Grid Search. A more efficient approach is to use the Newton method for optimization.

$$\theta_1^i \;=\; \theta_1^{i-1} - \frac{d\,csse}{d\theta_1} / \frac{d^2 csse}{d\theta_1^2} \tag{10.25}$$

where

$$\frac{d\,csse}{d\theta_1} \;=\; -\,2 \sum_{t=1}^{m-1} e_{t-1}(z_t - \theta_1 e_{t-1})$$

$$\frac{d^2\,csse}{d\theta_1^2} \;=\; 2 \sum_{t=1}^{m-1} e_{t-1}^2$$

Substituting gives

$$\theta_1^i \;=\; \theta_1^{i-1} + \sum_{t=1}^{m-1} e_{t-1}(z_t - \theta_1 e_{t-1}) / \sum_{t=1}^{m-1} e_{t-1}^2 \tag{10.26}$$

10.3.3 Exercises

1. For an MA(1) model, solve for θ_1 using Equation 10.20.

$$\gamma_0 \;=\; (1 \;+\; \theta_1^2)\,\sigma_\epsilon^2$$

$$\rho_1 \;=\; (\theta_1)\,\frac{\sigma_\epsilon^2}{\gamma_0} \;=\; \frac{\theta_1}{1 + \theta_1^2}$$

   Solve for θ_1 in terms of ρ_1.

2. Develop an MA(1) model for the Lake Level Time-series Dataset using the solution you derived in the previous question. Plot y_t and $\hat{y}_t$ vs. t.

3. Use ScalaTion to assess the quality of an MA(1) model versus an MA(2) model for the Lake Level Time-series Dataset.

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.

$$y_t \;=\; \mu \;+\; \theta_1 \epsilon_{t-1} \;+\; ... +\; \theta_q \epsilon_{t-q} \;+\; \epsilon_t$$

A combined p^{th}-order Auto-Regressive, q^{th}-order Moving Average ARMA(q) model predicts the next value y_t from both a weighted combination of prior values and the combined effects of prior noise/disturbances.

$$y_t \;=\; \delta \;+\; \phi_1 y_{t-1} \;+\; ... +\; \phi_p y_{t-p} \;+\; \theta_1 \epsilon_{t-1} \;+\; ... +\; \theta_q \epsilon_{t-q} \;+\; \epsilon_t \tag{10.27}$$

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.

10.4.1 Selection Based on ACF and PACF

The Auto-correlation Function (ACF) and partial auto-correlation (PACF) may be used for chosing values for the hyper-parameters p and q. When the PACF as a function of p drops in value toward zero, one may select a value of for p to be in this region. Similarly, when the ACF as a function of q drops in value toward zero, one may select a value of for q to be in this region. See the exercise to explore why.

`ARMA` Class

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 =
```

10.4.2 Exercises

1. Plot the ACF for the Lake Level Time-series Dataset and use it to pick a value for q. Assess the quality of an MA(q) with this value for q. Try it for q being one lower and one higher.

2. Plot the PACF for the Lake Level Time-series Dataset and use it to pick a value for p. Assess the quality of an AR(p) with this value for p. Try it for q being one lower and one higher.

3. Using the selected values for p and q from the two previous exercises, assess the quality of an ARMA(p, q). Try the four possibilities around the point (p, q).

4. For an ARMA (p, q) model, explain why the Partial Auto-correlation Function (PACF) is useful in in chosing a value for the p AR hyper-parameter.

5. For an ARMA (p, q) model, explain why the Auto-correlation Function (ACF) is useful in in chosing a value for the q MA hyper-parameter.

10.5 ARIMA

1.

10.6 ARIMAX

10.7 SARIMA

10.8 Exponential Smoothing

10.9 Dynamic Linear Models

As with a Hidden Markov Model (HMM), a Dynamic Linear Model (DLM) may be used to represent a system in terms of two stochatic processes, the state of system at time t, $\mathbf{x}_t$ and the observed values from measurements of the system at time t, $\mathbf{y}_t$. The main difference is that the state and its observation are treated as continuous quantities. For time series analysis, it is natural to treat time as discrete values.

For a basic DLM, the dynamics of the system are described by two equations: The *State Equation* indicates how the next state vector $\mathbf{x}_t$ is dependent on the previous state vector $\mathbf{x}_{t-1}$ and a process noise vector $\mathbf{w}_t \sim N(0, Q)$

$$\mathbf{x}_t \;=\; A\mathbf{x}_{t-1} + \mathbf{w}_t \tag{10.28}$$

where Q is the covariance matrix for the process noise. If the dynamics are deterministic, then the covariance matrix is zero, otherwise it can capture uncertainity in the relationships between the state variables (e.g., simple models of the flight of a golf ball often ignore the effects due to the spin on the golf ball).

The *Observation/Measurement Equation* indicates how at time t, the observation vector $\mathbf{y}_t$ is dependent on the current state $\mathbf{x}_t$ and a measurement noise vector $\mathbf{v}_t \sim N(0, R)$

$$\mathbf{y}_t \;=\; C\mathbf{x}_t + \mathbf{v}_t \tag{10.29}$$

where R is the covariance matrix for the measurement noise/error. The process noise and measurement noise are assumed to be independent of each other. The state transition matrix A indicates the linear relationships between the state variables, while the C matrix establishes linear relationships between the state of system and its obsevations/measurements.

10.9.1 Example: Traffic Sensor

Consider the operation of a road sensor that records traffic flow (vehicles per 15 minutes) and average speed (km per hour). Let $\mathbf{x}_t = [x_{t0}, x_{t1}]$ be the flow of vehicles (x_{t0}) and their average speed (x_{t1}) at time t. Assume that the flow is high enough that it can be treated as a continuous quanity and that the covariance matrices are diagonal (uncertainty of flow and speed are independent). The dynamics of the system then may be described by the following state equations (see equation 10.22):

$$\begin{aligned} x_{t0} &= a_{00}x_{t-1,0} + a_{01}x_{t-1,1} + w_{t0} \\ x_{t1} &= a_{10}x_{t-1,0} + a_{11}x_{t-1,1} + w_{t1} \end{aligned}$$

The sensor tries to capture the dynamics of the system, but depending on the quality of the sensor there will be measurement errors. The observation/measurement variables $\mathbf{y}_t = [y_{t0}, y_{t1}]$ may correspond to the state variables in a one-to-one correspondence or by some linear relationship. The observation of the system then may be described by the following observation equations (see equation 10.23):

$$\begin{aligned} y_{t0} &= c_{00}x_{t0} + c_{01}x_{t1} + v_{t0} \\ y_{t1} &= c_{10}x_{t0} + c_{11}x_{t1} + v_{t1} \end{aligned}$$

Further assume that estimates for the A and C parameters of the model have been found (see the subsection on Training).

$$\begin{aligned} x_{t0} &= 0.9x_{t-1,0} + 0.2x_{t-1,1} + w_{t0} \\ x_{t1} &= -0.4x_{t-1,0} + 0.8x_{t-1,1} + w_{t1} \end{aligned}$$

These state equations suggest that the flow will be a high percentage of the previous flow, but that higher speed suggests increasing flow. In addition, the speed is based on the previous speed, by higher flow suggests that speeds may be decreasing (e.g., due to congestion).

$$\begin{aligned} y_{t0} &= 1.0x_{t0} - 0.1x_{t1} + v_{t0} \\ y_{t1} &= -0.1x_{t0} + 1.0x_{t1} + v_{t1} \end{aligned}$$

These observation equations suggest that higher speed makes it more likely for a vehicle to pass the sensor without being counted and higher flow makes under-estimation of speed to be greater.

10.9.2 Kalman Filter

A Kalman Filter is a Dynamic Linear Model that incorporates an outside influence on the system. If a driving force or control is applied to the system, an additional term $B\mathbf{u}_t$ is added to the state equation [?],

$$\mathbf{x}_t = A\mathbf{x}_{t-1} + B\mathbf{u}_t + \mathbf{w}_t \tag{10.30}$$

where $\mathbf{u}_t$ is the control vector and the B matrix establishes as linear relationships between the control vector and the state vector. The observation equation remains the same.

$$\mathbf{y}_t = C\mathbf{x}_t + \mathbf{v}_t$$

The process noise $\mathbf{w}_t$ and the measurement noise $\mathbf{v}_t$ also remain the same. The Kalman Filter model, therefore includes five matrices.

Table 10.2: Matrices Used in Kalman Filter Model

matrix	**dimensions**	**description**
A	n-by-n	state transition matrix
B	n-by-l	state-control matrix
C	m-by-n	measurement-state matrix
Q	n-by-n	process noise covariance matrix
R	m-by-m	measurement noise covariance matrix

The first three matrices are named A, B and C here to maintain consistency with the rest of the models, but are often named F, G and H in the literature. The Kalman Filter model at time t includes three vectors:

Table 10.3: Vectors Used in Kalman Filter Model

vector	dimension	description
x	n	state vector
u	l	control vector
y	m	measurement vector

10.9.3 Training

The main goal of training is to minimize the error in estimating the state. At time t, a new measurement y_t becomes available. The errors before and after this event are the differences between the actual state x_t and the estimated state before $\hat{x}_t^-$ and after $\hat{x}_t$ [**?**].

$$\begin{aligned} e_t^- &= x_t - \hat{x}_t^- \\ e_t &= x_t - \hat{x}_t \end{aligned}$$

Since $\mathbf{w}_t$ has a zero mean, the covariances of the before and after errors may be computed as expectations of their outer products.

$$\mathbb{C}\left[e_t^-\right] = \mathbb{E}\left[e_t^- \otimes {e_t}^-\right] \tag{10.31}$$

$$\mathbb{C}\left[e_t\right] = \mathbb{E}\left[e_t \otimes e_t\right] \tag{10.32}$$

The essential insight by Kalman was that the after estimate should be the before estimate adjusted by a weighted difference between the actual measured value y_t and its before estimate $C\hat{x}_t^-$.

$$\hat{x}_t = \hat{x}_t^- + K[y_t - C\hat{x}_t^-] \tag{10.33}$$

The n-by-m K matrix is called the Kalman gain. If the actual measurement is very close to its predicted value, little adjustment to the predicted state value is needed. On the other hand, when there is a disagreement, the adjustment based upon the measurement should be tempered based upon the reliablity of the measurement. A small gain will dampen the adjustment, while a high gain may result in large adjustments. The trick is to find the optimal gain K.

Using a Minimum Variance Unbiased Estimator (MVUE) for parameter estimation for a Kalman Filter means that the trace of the error covariance matrix should be minimized (see exercises for details).

$$\mathbb{V}\left[||e_t||\right] = \mathbb{E}\left[||e_t||^2\right] = \text{trace } \mathbb{C}\left[e_t\right] \tag{10.34}$$

Plugging equation (10.27) into (10.26) gives the following optimization problem:

$$\min \text{ trace } \mathbb{E}\left[(\hat{x}_t^- + K[y_t - C\hat{x}_t^-) \otimes (\hat{x}_t^- + K[y_t - C\hat{x}_t^-)\right] \tag{10.35}$$

10.9.4 Exercises

1. For a DLM, consider the case where $m = n = 1$. Equations (10.22) and (10.23) become

$$\begin{aligned} x_t &= a x_{t-1} + w_t \\ y_t &= c x_t + v_t \end{aligned}$$

where $w_t \sim N(0, \sigma_q^2)$ and $v_t \sim N(0, \sigma_r^2)$. Compare this model with an AR(1) model.

2. For the Traffic Sensor Example, let $Q = \sigma_q^2 I$ and $R = \sigma_r^2 I$. Develop a DLM model using ScalaTion and try low, medium and high values for the variances σ_q^2 and σ_r^2 (9 combinations). Let the initial state of the system be $x_{00} = 100.0$ vehicles per 15 minutes and $x_{01} = 100.0$ km per hour. How does the relative amount of process and measurement error affect the dynamics/observation of the syetem?

3. Consider the state and observation equations given in the Traffic Sensor Example and assume that the state equations are deterministic (no uncertainty in system, only in its observation). Reduce the DLM to a simpler type of time series model. Explain.

4. Use the Traffic Sensor Dataset (`traffic.csv`) to estimate values for the 2-by-2 covariance matrices Q and R.

5. Use the Traffic Sensor Dataset (`traffic.csv`) to estimate values for the parameters of a DLM model, i.e., for the A and C 2-by-2 matrices.

6. Suppose that fog negatively affects traffic and speed. Use the Traffic Sensor with Fog Dataset (`traffic_fog.csv`) to estimate values for the 2-by-2 covariance matrices Q and R.

7. Use the Traffic Sensor with Fog Dataset (`traffic_fog.csv`) to estimate values for the parameters of a Kalman Filter model, i.e., for the A, B and C 2-by-2 matrices.

8. Show that if $\hat{\mathbf{y}}$ is an unbiased estimator for $\mathbf{y}$ (i.e., $\mathbb{E}[\hat{\mathbf{y}}] = \mathbb{E}[\mathbf{y}]$) that the minimum error variance $\mathbb{V}[||\mathbf{y} - \hat{\mathbf{y}}||]$ is

$$\mathbb{E}\left[||\mathbf{y} - \hat{\mathbf{y}}||^2\right] \;=\; \text{trace}\; \mathbb{E}\left[(\mathbf{y} - \hat{\mathbf{y}}) \otimes ((\mathbf{y} - \hat{\mathbf{y}})\right]$$

9. Explain why minimizing the trace of the covariance $\mathbb{C}[e_t]$ leads to optimal Kalman gain K.

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

$$\begin{aligned} y &= \mathbf{x}(t) + \epsilon \\ \frac{d\mathbf{x}(t)}{dt} &= f(\mathbf{x}(\mathbf{t});\ \mathbf{b}) \end{aligned}$$

10.12.1 Non-Linear Least Squares (NLS)

10.12.2 Least Squares Approximation (LSA)

10.13 Spatial Models

10.14 Convolutional Neural Networks

Chapter 11

Clustering

Clustering is related to classification, except that specific classes are not prescribed. Instead data points (vectors) are placed into clusters based on some similarity or distance metric (e.g., Euclidean or Manhattan distance). It is also related to prediction in the sense that a predictive model may be associated with each cluster. Points in a cluster, are according to some metric, closer to each other than to points not in their cluster. Closeness or similarity may be defined in terms of ℓ_p distance $||\mathbf{x} - \mathbf{z}||_p$, correlation $\rho(\mathbf{x}, \mathbf{z})$, or cosine $\cos(\mathbf{x}, \mathbf{z})$. Abstractly, we may represents any of these by distance $d(\mathbf{x}, \mathbf{z})$. In ScalaTion, the function `dist` in the package object computes the square of Euclidean distance between two vectors, but may easlily be changed (e.g., `(u - v).norm1` for Manhattan distance).

```
def dist (u: VectoD, v: VectoD): Double =
{
    (u - v).normSq     // squared Euclidean norm used for efficiency, may use other norms
} // dist
```

Consider a general modeling equation, where the parameters $\mathbf{b}$ are estimated based on a dataset $(X, \mathbf{y})$.

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

Rather than trying to approximate the function f over the whole data domain, one might think that given point $\mathbf{z}$, that points similar to (or close to) $\mathbf{z}$, might be more useful in making a prediction $f(\mathbf{z})$.

A simple way to do this would be to find the κ-nearest neighbors to point $\mathbf{z}$,

$$top_\kappa(\mathbf{z}) \ = \ \{\mathbf{x_i} \in X \,|\, \mathbf{x_i} \text{ is among the } \kappa \text{ closest points to } \mathbf{z}\}$$

and simply average the responses or y-values.

Instead of surveying the responses from the κ-nearest neighbors, one could instead survey an entire group of similar points and take their averaged response (or utilize a linear model where each cluster c has its own parameters $\mathbf{b_c}$). The groups may be pre-computed and the averages/parameters can be maintained for each group. The groups are make by clustering the points in the X matrix into say k groups.

Clustering will partition the m-points $\{\mathbf{x_i} \in X\}$ into k groups/clusters. Group membership is based on closeness or similarity between the points. Commonly, algorithms form groups by establishing a centroid (or center) for each group/cluster. Centroids may defined as means (or medians) of the points in the group. In this way the data matrix X is partitioned into k submatrices

$$\{X_c \,|\, c \in \{0, \dots, k-1\}\}$$

each with centroid $\boldsymbol{\xi}_c = \mu(X_c)$. Typically, point $\mathbf{x_i}$ is in cluster c because it is closer to its centriod than any other centroid, i.e.,

$$\mathbf{x_i} \in X_c \implies d(\mathbf{x_i}, \boldsymbol{\xi}_c) \le d(\mathbf{x_i}, \boldsymbol{\xi}_h)$$

Define the *cluster assignment function* ξ to take a point $\mathbf{x_i}$ and assign it to the cluster with the closest centroid $\boldsymbol{\xi}_c$, i.e.,

$$\xi(\mathbf{x_i}) \;=\; c$$

The goal becomes to find an optimal cluster assignment function by minimizing the following objective/cost function:

$$\min_\xi \; \sum_{i=0}^{m} d(\mathbf{x_i}, \boldsymbol{\xi}_{\xi(\mathbf{x_i})})$$

If the distance d is $||\mathbf{x_i} - \boldsymbol{\xi}_{\xi(\mathbf{x_i})}||_2^2$ (the default in ScalaTion), then above sum may be viewed as a form of sum of squared errors (sse).

If one knew the optimal centroids ahead of time, finding an optimal cluster assignment function ξ would be trivial and would take $O(kmn)$ time. Unfortunately, k centroids must be initially chosen, but then as assignments are made, the centroids will move, causing assignments to need re-evaluation. The details vary by clustering algorithm, but it is useful to know that finding an optimal cluster assignment function is $\mathcal{NP}$-hard [**?**].

Other factors that can be considering in forming clusters include, balancing the size of clusters and maximizing the distance between clusters.

11.1 KNN_Predictor

Similar to the `KNN_Classifier` class, the `KNN_Predictor` class makes predictions based on individual predictions of its κ-nearest neighbors. For prediction, its function is analogous to using clustering for prediction and will be compared in the exercises in later sections of this chapter.

Training in `KNN_Predictor` is lazy and is done in the `predict` method, based on the following equation:

$$\hat{y} \;=\; \frac{1}{\kappa}\, \mathbf{1} \cdot \mathbf{y}(top_\kappa(\mathbf{z})) \tag{11.1}$$

Given point $\mathbf{z}$, find κ points that are the closest, sum there response values y, and return the average.

```
override def predict (z: VectoD): Double =
{
    kNearest (z)                                // set top-kappa to kappa nearest
    var sum = 0.0
    for (i <- 0 until kappa) sum = y(topK(i)._1)   // sum the individual predictions
    val yp = sum / kappa                        // divide to get average
    reset ()                                    // reset topK
    yp                                          // return the predicted value
} // predict
```

The `kNearest` method is same as the one in `KNN_Classifier`.

`KNN_Predictor` **Class**

Class Methods:

```
@param x       the vectors/points of predictor data stored as rows of a matrix
@param y       the response value for each vector in x
@param fname_  the names for all features/variables
@param hparam  the number of nearest neighbors to consider

class KNN_Predictor (x: MatriD, y: VectoD,
                     fname_ : Strings = null, hparam: HyperParameter = KNN_Predictor.hp)
      extends PredictorMat (x, y, fname_, hparam)

def distance (x: VectoD, z: VectoD): Double = (x - z).normSq
def train (yy: VectoD = y): KNN_Predictor = this
override def eval (xx: MatriD = x, yy: VectoD = y): KNN_Predictor =
override def predict (z: VectoD): Double =
def reset ()
def forwardSel (cols: Set [Int], adjusted: Boolean): (Int, VectoD, VectoD) =
def backwardElim (cols: Set [Int], adjusted: Boolean, first: Int): (Int, VectoD, VectoD) =
def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =
```

Note, the `forwardSel` and `backwardElim` methods are not relevant and just throw exceptions if called. Also, the `train` method has nothing to do, so it need not be called.

11.1.1 Exercises

1. Apply `KNN_Predictor` to the following combined data matrix.

```
//                              x1 x2  y
val xy = new MatrixD ((10, 3), 1, 5, 1,       // joint data matrix
                               2, 4, 1,
                               3, 4, 1,
                               4, 4, 1,
                               5, 3, 0,
                               6, 3, 1,
                               7, 2, 0,
                               8, 2, 0,
                               9, 1, 0,
                              10, 1, 0)

val knn = KNN_Predictor (xy)
val (x, y) = PredictorMat.pullResponse (xy)
val yp = knn.predict (x)
knn.eval (x, y)                               // due to lazy/late training
println (knn.report)
new Plot (xy.col(0), y, yp, lines = true)
```

11.2 Clusterer

The `Clusterer` trait provides a common framework for several clustering algorithms.

`Clusterer` **Trait**

Trait Methods:

```
trait Clusterer

def name_ (nm: Strings) { _name = nm }
def name (c: Int): String =
def setStream (s: Int) { stream = s }
def train (): Clusterer
def cluster: Array [Int]
def csize: VectoI
def centroids: MatriD
def initCentroids (): Boolean = false
def calcCentroids (x: MatriD, to_c: Array [Int], sz: VectoI, cent: MatriD)
def classify (z: VectoD): Int
def distance (u: VectoD, cn: MatriD, kc_ : Int = -1): VectoD =
def sse (x: MatriD, to_c: Array [Int]): Double =
def sse (x: MatriD, c: Int, to_c: Array [Int]): Double =
def sst (x: MatriD): Double =
def checkOpt (x: MatriD, to_c: Array [Int], opt: Double): Boolean = sse (x, to_c) <= opt
```

For readability, names may be given to clusters (see `name` and `name_`). To obtain a new (and likely different) cluster assignment, `setStream` method may be called to change the random number stream. The `train` methods in the implementing classes will take a set of points (vectors) and apply iterative algorithms to find a "good" cluster assignment function. The `cluster` method may be called after `train` to see the cluster assignments. The centroids are returned as rows in a matrix by calling `centroids`, whose cluster sizes are given by `csize`. The `initCentroid` method initializes the centroids, while the `calcCentroids` calculates the centroids based in the points contained in the each cluster.

```
def calcCentroids (x: MatriD, to_c: Array [Int], sz: VectoI, cent: MatriD)
{
    cent.set (0.0)                                       // set cent matrix to all zeros
    for (i <- x.range1) {
        val c   = to_c(i)                                // x_i currently assigned to cluster c
        cent(c) = cent(c) + x(i)                         // add the next vector in cluster
    } // for
    for (c <- cent.range1) cent(c) = cent(c) / sz(c)     // divide to get averages/means
} // calcCentroids
```

Given a new point/vector `z`, the `classify` method will indicate which cluster it belongs to (in the range 0 to k-1). The distances between a point and the centroids is computed by the `distance` method. The objective/cost function is defined to be the sum of squared errors (`sse`). If the cost of an optimal solution is known, `checkOpt` will return true if the cluster assignment is optimal.

11.3 K-Means Clustering

The `KMeansClustering` class clusters several vectors/points using k-means clustering. The user selects the number of clusters desired (k). The algorithm will partition the points in X into k clusters. Each cluster has a centroid (mean) and each data point $\mathbf{x_i} \in X$ is placed in the cluster whose centroid it is nearest to.

11.3.1 Initial Assignment

There are two ways to intialize the algorithm: Either (1) randomly assign points to k clusters or (2) randomly pick k points as initial centroids. Technique (1) tends to work better and is the primary technique used in ScalaTion. Using the primary technique, the first step is to randomly assign each point $\mathbf{x_i}$ to a cluster.

$$\xi(\mathbf{x_i}) \;=\; \text{random integer from } \{0, \ldots, k-1\}$$

In ScalaTion, this is carried out by the `assign` method, that uses the `Randi` random integer generator. It also uses multiple counters for determining the size `sz` of each cluster (i.e., the number of points in each cluster).

```
protected def assign ()
{
    val ran = new Randi (0, k-1, s)          // for random integers: 0, ..., k-1
    for (i <- x.range1) {
        to_c(i) = ran.igen                   // randomly assign x(i) to a cluster
        sz(to_c(i)) += 1                     // increment size of that cluster
    } // for
} // assign
```

See the exercises for more details on the second technique for initializing clusters/centroids.

Handling Empty Clusters

If any cluster turns out to be empty, move a point from another cluster. In ScalaTion this is done by removing a point from the largest cluster and adding it to the empty cluster. This is performed by the `fixEmptyClusters` method.

```
protected def fixEmptyClusters ()
```

After the `assign` and `fixEmptyClusters` methods have been called, the data matrix X will be logically partitioned into k non-empty submatrices X_c with cluster c having n_c (`sz(c)`) points/rows.

Calculating Centroids

The next step is to calculate the centroids using the `calcCentroids` method. For cluster c, the centroid is the vector mean of the rows in submatrix X_c.

$$\boldsymbol{\xi}_c \;=\; \frac{1}{n_c} \sum_{\mathbf{x_i} \in X_c} \mathbf{x_i}$$

ScalaTion iterates over all points and based on their cluster assignment adds them to one of the k centroids (stored in the `cent` matrix). After the loop, these sums are divided by the cluster sizes `sz` to get means. The `calcCentroids` method is defined in the base trait `Clusterer`.

11.3.2 Reassignment of Points to Closest Clusters

After initialization, the algorithm iteratively reassigns each point to the cluster containing the closest centroid. The algorithm stops when there are no changes to the cluster assignments. For each iteration, each point $\mathbf{x_i}$ needs to be re-evaluated and moved (if need be) to the cluster with the closest centroid. Reassignment is based on taking the argmin of all the distances to the centroids with ties going to the current cluster.

$$\xi(\mathbf{x_i}) \;=\; \text{argmin}_c \, d(\mathbf{x_i}, \boldsymbol{\xi}_c)$$

In ScalaTion, this is done by the `reassign` method which iterates over each $\mathbf{x_i} \in X$ computing the distance to each of k centroids. The cluster (`c2`) with the closest centroid is found using the `argmin` method. The distance to `c2`'s centroid is then compared to the distance to its current cluster `c1`'s centroid, and if the distance to `c2`'s centroid is less, $\mathbf{x_i}$ will be moved and a `done` flag will be set to `false`, indicating that during this reassignment phase at least one change was made.

```
protected def reassign (): Boolean =
{
    var done = true                                // done indicates no changes
    for (i <- x.range1) {                          // standard order for index i
        val c1 = to_c(i)                           // c1 = current cluster for point x_i
        if (sz(c1) > 1) {                          // if size of c1 > 1
            val d  = distance (x(i), cent)         // distances to all centroid
            val c2 = d.argmin ()                   // c2 = cluster with closest centroid to x_i
            if (d(c2) < d(c1)) {                   // if closest closer than current
                sz(c1) -= 1                        // decrement size of current cluster
                sz(c2) += 1                        // increment size of new cluster
                to_c(i) = c2                       // reassign point x_i to cluster c2
                done    = false                    // changed clusters => not done
                if (immediate) return false        // optionally return after first change
            } // if
        } // if
    } // for
    done
} // reassign
```

The exercises explore a change to this algorithm by having it return after the first change.

11.3.3 Training

The `train` method simply uses these methods until the `reassign` method returns true (internally the `done` flag is true). The method is set up to work for this and derived classes. It assigns points to clusters and then either initilizes/picks centroids or calculates centroids from the first cluster assignment. Inside the loop, `reassign` and `calcCentroid` are called until there is no change to the cluster assignment. After the loop, an exception is thrown if there are any empty clusters (a useful safe-guard since this method is used by derived classes). Finally, if post-processing is to be performed (`post = true`), then the `swap` method is called. This method will swap two points in different clusters, if the swap results in a lower sum of squared errror (sse).

```
def train (): KMeansClusterer =
{
    sz.set (0)                                                  // cluster sizes initialized to zero
    raniv = PermutedVecI (VectorI.range (0, x.dim1), stream)    // for randomizing index order
    assign ()                                                   // randomly assign points to clusters
    fixEmptyClusters ()                                         // move points into empty clusters
    if (! initCentroids ()) calcCentroids (x, to_c, sz, cent)   // pick points for initial centroids
    breakable { for (l <- 1 to MAX_ITER) {
        if (reassign ()) break                                  // reassign points (no change => break)
        calcCentroids (x, to_c, sz, cent)                       // re-calculate the centroids
    }} // for
    val ce = sz.indexOf (0)                                     // check for empty clusters
    if (ce != -1) throw new Exception (s"Empty cluster c = $ce")
    if (post) swap ()                                           // swap points to improve sse
    this
} // train
```

`KMeansClusterer` Class

The `KMeansClusterer` class and its derived classes take a data matrix, the desired number clusters and an array of flags as input parameters. The array of flags are used to make adjustments to the algorithms. For this class, there are two: `flags(0)` or `post` indicates whether to use post-processing, and `flags(1)` or `immediate` indicates whether return upon the first change in the `reassign` method.

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the array of flags used to adjust the algorithm
                  default: no post processing, no immediate return upon change

class KMeansClusterer (x: MatriD, k: Int, val flags: Array [Boolean] = Array (false, false))
      extends Clusterer with Error

def train (): KMeansClusterer =
def cluster: Array [Int] = to_c
def centroids: MatriD = cent
def csize: VectoI = sz
protected def assign ()
protected def fixEmptyClusters ()
protected def reassign (): Boolean =
protected def swap ()
def classify (z: VectoD): Int = distance (z, cent).argmin ()
def show (l: Int) { println (s"($l) to_c = ${to_c.deep} \n($l) cent = $cent") }
```

11.3.4 Exercises

1. Plot the following points.

```
//                              x0   x1
val x = new MatrixD ((6, 2), 1.0, 2.0,
                             2.0, 1.0,
                             4.0, 5.0,
                             5.0, 4.0,
                             8.0, 9.0,
                             9.0, 8.0)

new Plot (x.col(0), x.col(1), null, "x0 vs. x1")
```

   For `k = 3`, determine the optimal cluster assignment ξ. What is the sum of squared errors *sse* for this assignment?

2. Using the data from the previous exercise, apply the K-Means Clustering Algorithm by hand to complete the following cluster assignment function table. Let the number of clusters k be 3 (clusters 0, 1 and 2). The ξ^0 column is the initial random cluster assignment, while the next two columns represent the cluster assignments for the next two iterations.

Table 11.1: Cluster Assignment Function Table

point	(x_0, x_1)	ξ^0	ξ^1	ξ^2
0	(1, 2)	0	?	?
1	(2, 1)	2	?	?
2	(4, 5)	0	?	?
3	(5, 4)	1	?	?
4	(8, 9)	1	?	?
5	(9, 8)	2	?	?

3. The `test` function in the `Clusterer` object is used test various configurations of classes extending `Clusterer`, such as the `KMeansClusterer` class.

```
@param x    the data matrix holding the points/vectors
@param fls  the array of flags
@param alg  the clustering algorithm to test
@param opt  the known optimum for see (ignore if not known)

def test (x: MatriD, fls: Array [Boolean], alg: Clusterer, opt: Double = -1.0)
```

   Explain the meaning of each of the flags: `post` and `immediate`. Call the `test` function, passing in `x` and `k` from the last exercise. Also, let the value `opt` be the value determined in the last exercise. The `test` method will give the number of test cases out of `NTESTS` that are correct in terms of achieving the minimum *sse*.

4. The `primary` versus `secondary` techniques for initializing the clusters/centroids are provided by the `KMeansClusterer` class and the `KMeansClusterer2` class, respectively. Test the quality of these two techniques.

5. Show that the time complexity of the `reassign` method is $O(kmn)$. The time complexity of K-Means Clustering using Lloyd's Algorithm is simply the complexity of the `reassign` method times the number of iterations. In practice, the number of iterations tends to be small, but in the worst case only upper and lower bounds are known, see [1] for details.

6. Consider the objective/cost function given in ISL equation 10.11 in [13]. What does it measure and how does it compare to sse used in this book?

11.4 K-Means Clustering - Hartigan-Wong

An alternative to the Lloyd algorithm that often produces more tightly packed clusters is the Hartigan-Wong algorithm. Improvement is seen in the fraction of times that optimal clusters are formed as well as the reduction in sum of squared errors (sse). The change to the code is minimal in that only the `reassign` method needs to be overriden.

The basic difference is that rather than simply reassigning each point to the cluster with the closest centroid (the Lloyd algorithm), the Hartigan-Wong algorithm weights the distance by the relative changes in the number of points in a cluster. For example, if a point is to be moved into a cluster with 10 points currently, the weight would be 10/11. If the point is to stay in its present cluster with 10 points currently, the loss in removing it would be weighted as 10/9. The weighting scheme has two effects: First it makes it more likely to move a point out of its current cluster. Second it makes it more likely to join a small cluster.

Mathematically, the weighted distance d' to cluster c when the point $\mathbf{x_i} \notin X_c$ is given by

$$d'(\mathbf{x_i}, \boldsymbol{\xi}_c) = \frac{n_c}{n_c + 1} d(\mathbf{x_i}, \boldsymbol{\xi}_c) \tag{11.2}$$

When the point $\mathbf{x_i} \in X_c$, the weighted distance d' to cluster c is given by

$$d'(\mathbf{x_i}, \boldsymbol{\xi}_c) = \frac{n_c}{n_c - 1} d(\mathbf{x_i}, \boldsymbol{\xi}_c) \tag{11.3}$$

The code for the `reassign` method is similar to the one in `KMeansClusterer`, except that the private method `closestByR2` calculates weighted distances to return the closest centroid.

```
protected override def reassign (): Boolean =
{
    var done = true                                // done indicates no changes
    for (i <- raniv.igen) {                        // randomize order of index i
        val c1 = to_c(i)                           // c1 = current cluster for point x_i
        if (sz(c1) > 1) {                          // if size of c1 > 1
            val d  = distance2 (x(i), cent, c1)    // adjusted distances to all centroid
            val c2 = d.argmin ()                   // c2 = cluster with closest centroid to x_i
            if (d(c2) < d(c1)) {                   // if closest closer than current
                sz(c1) -= 1                        // decrement the size of cluster c1
                sz(c2) += 1                        // increment size of cluster c2
                to_c(i) = c2                       // reassign point x_i to cluster c2
                done = false                       // changed clusters => not done
                if (immediate) return false        // optionally return after first change
            } // if
        } // if
    } // for
    done
} // reassign
```

Besides switching from distance d to weighted distance d', the code also randomizes the index order and has the option of returning immediately after a change is made.

11.4.1 Adjusted Distance

The `distance2` method computes the adjusted distance of point `u` to all of the centroids `cent`, where `cc` is the current centroid that `u` is assigned to. Notice the inflation of distance when `c == cc`, and its deflation, otherwise.

```
def distance2 (u: VectoD, cent: MatriD, cc: Int): VectoD =
{
    val d = new VectorD (cent.dim1)
    for (c <- 0 until k) {
        d(c) = if (c == cc) (sz(c) * dist (u, cent(c))) / (sz(c) - 1)
               else         (sz(c) * dist (u, cent(c))) / (sz(c) + 1)
    } // for
    d
} // distance2
```

`KMeansClusteringHW` **Class**

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the flags used to adjust the algorithm

class KMeansClustererHW (x: MatriD, k: Int, flags: Array [Boolean] = Array (false, false))
      extends KMeansClusterer (x, k, flags)

protected override def reassign (): Boolean =
def distance2 (u: VectoD, cent: MatriD, cc: Int): VectoD =
```

11.4.2 Exercises

1. Compare `KMeansClustererHW` with `KMeansClusterer` for a variety of datasets, starting with the six points given in the last section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the *sse* over the `NTESTS` test cases.

11.5 K-Means++ Clustering

The `KMeansClustererPP` class clusters several vectors/points using a k-means++ clustering algorithm. The class may be derived from a K-Means clustering algorithm and in ScalaTion it is derived from the Hartigan-Wong algorithm (`KMeansClustererHW`). The innovation for `KMeansClustererPP` is to pick the initial centroids wisely, yet randomly. The wise part is to make sure points are well separated. The random part involves making a probability mass function (pmf) where points farther away from the current centroids are more likely to be selected as the next centroid. Picking the initial centroids entirely randomly leads to `KMeansClusterer2` which typically does not perform as well `KMeansClusterer`. However, maintaining randomness while giving preference to more distant points becoming the next centroid has been shown to work well.

11.5.1 Picking Initial Centroids

In order to pick k initial centoids, the first one, `cent(0)`, is chosen entirely randomly, using the `ranI` random variate generator object. The method call `ranI.igen` will pick one of the `m = x.dim1` points as the first centroid.

```
val ranI = new Randi (0, x.dim1-1, stream)       // uniform random integer generator
cent(0)  = x(ranI.igen)                          // pick first centroid uniformly at random
```

The rest of the centroids are chosen following a distance-derived discrete distribution, using the `ranD` random variate generator object. The probability mass function (pmf) for this discrete distribution is produced so that the probability of a point being selected as the next centroid is proportional to its distance to the closest existing centroid.

```
for (c <- 1 until k) {                           // pick remaining centroids
    val ranD = update_pmf (c)                    // update distance derived pmf
    cent(c)  = x(ranD.igen)                      // pick next centroid according to pmf
} // for
```

Each time a new centroid is chosen, the pmf must be updated as it is likely to be the closest centroid for some of the remaining as yet unchosen points. Given that the next centroid to selected is the c^{th} centroid, the `update_pmf` method will update the pmf and return a new distance-derived discrete distribution.

```
def update_pmf (c: Int): Discrete =
{
    for (i <- x.range1) pmf(i) = distance (x(i), cent, c).min ()   // shortest distances
    pmf /= pmf.sum                                                 // divide by sum
    Discrete (pmf, stream = (stream + c) % NSTREAMS)               // distance-derived generator
} // update_pmf
```

The `pmf` vector initially records the shortest distance from each point $\mathbf{x_i}$ to any of the existing already selected centroids $\{0, \ldots, c-1\}$. These distances are turned into probabilities by dividing by their sum. The `pmf` vector then defines a new distance-derived random generator that is returned.

`KMeansClustererPP` **Class**

Class Methods:

```
@param x      the vectors/points to be clustered stored as rows of a matrix
@param k      the number of clusters to make
@param flags  the flags used to adjust the algorithm

class KMeansClustererPP (x: MatriD, k: Int, flags: Array [Boolean] = Array (false, false))
      extends KMeansClustererHW (x, k, flags)

override def initCentroids (): Boolean =
def update_pmf (c: Int): Discrete =
```

11.5.2 Exercises

1. Compare `KMeansClustererPP` with `KMeansClustererHW` and `KMeansClusterer` for a variety of datasets, starting with the six points given in the `KMeansClusterer` section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the *sse* over the `NTESTS` test cases.

11.6 Clustering Predictor

The `ClusteringPredictor` class is used to predict a response value for new vector **z**. It works by finding the cluster that the point **z** would belong to. The recorded response value for y is then given as the predicted response. The per cluster recorded response value is the consensus (e.g., average) of the response values y_i for each member of the cluster. Training involves clustering the points in data matrix X and then computing each cluster's response. Assuming the closest centroid to **z** is $\boldsymbol{\xi}_c$, the predicted value $\hat{y}$ is

$$\hat{y} \;=\; \frac{1}{n_c} \sum_{\xi(\mathbf{x_i})=c} y_i \tag{11.4}$$

where n_c is the number points in cluster c and $\xi(\mathbf{x_i}) = c$ means that the i^{th} point is assigned to cluster c.

11.6.1 Training

The `train` method first clusters the points/rows in data matrix X by calling the `train` method of a clustering algorithms (e.g., `clust = KMeansClusterer (...)`). It then calls the `assignResponse` method to assign a consenus (average) response value for each cluster.

```
def train (yy: VectoD = y): ClusteringPredictor =
{
    clust.train ()
    val clustr = clust.cluster
    assignResponse (clustr)
    this
} // train
```

The computed consensus values are stored in `yclus`, so that the `predict` method may simply use the underlying clustering algorithm to classify a point **z** to indicate which cluster it belongs to. This is then used to index into the `yclus` vector.

```
override def predict (z: VectoD): Double = yclus (clust.classify (z))
```

`ClusteringPredictor` **Class**

Class Methods:

```
@param x       the vectors/points of predictor data stored as rows of a matrix
@param y       the response value for each vector in x
@param fname_  the names for all features/variables
@param hparam  the number of nearest neighbors to consider

class ClusteringPredictor (x: MatriD, y: VectoD,
                           fname_ : Strings = null, hparam: HyperParameter = ClusteringPredictor.hp)
      extends PredictorMat (x, y, fname_, hparam)

def train (yy: VectoD = y): ClusteringPredictor =
override def eval (xx: MatriD = x, yy: VectoD = y): ClusteringPredictor =
def classify (z: VectoD): Int =
override def predict (z: VectoD): Double = yclus (classify (z))
def reset ()
def forwardSel (cols: Set [Int], adjusted: Boolean): (Int, VectoD, VectoD) =
def backwardElim (cols: Set [Int], adjusted: Boolean, first: Int): (Int, VectoD, VectoD) =
def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =
```

11.6.2 Exercises

1. Apply `ClusteringPredictor` to the following combined data matrix.

```
//                             x0 x1  y
val xy = new MatrixD ((10, 3), 1, 5, 1,       // joint data matrix
                               2, 4, 1,
                               3, 4, 1,
                               4, 4, 1,
                               5, 3, 0,
                               6, 3, 1,
                               7, 2, 0,
                               8, 2, 0,
                               9, 1, 0,
                              10, 1, 0)

val cp = ClusteringPredictor (xy)
cp.train ().eval ()
val (x, y) = PredictorMat.pullResponse (xy)
val yp = cp.predict (x)
println (cp.report)
new Plot (xy.col(0), y, yp, lines = true)
```

   Compare its results to that of `KNN_Predictor`.

2. Compare `Regression`, `KNN_Predictor` and `ClusteringPredictor` on the `AutoMPG` dataset.

11.7 Hierarchical Clustering

One critique of K-Means Clustering is that the user choses the desired number of clusters (k) beforehand. With modern computing power, several values for k may be tried, so this is less of an issue now. There is, however, a clustering technique called Hierarchical Clustering where this a non-issue.

In SCALATION the `HierClusterer` class starts with each point in the data matrix X forming its own cluster (m clusters). For each iteration, the algorithm will merge two clusters into a one larger cluster, thereby reducing the number of clusters by one. The two clusters that are closest to each other are chosen as the clusters to merge. The `train` method is shown below.

```
def train (): HierClusterer =
{
    sz.set (0)                                      // initialize cluster sizes to zero
    initClusters ()                                 // make a cluster for each point

    for (kk <- x.dim1 until k by -1) {
        val (si, sj) = bestMerge (kk)               // find the 2 closest clusters
        clust += si | sj                            // add the union of sets i and j
        clust -= si                                 // remove set i
        clust -= sj                                 // remove set j
        if (DEBUG) println (s"train: for cluster (${kk-1}), clust = $clust")
    } // for

    finalClusters ()                                // make final cluster assignments
    calcCentroids (x, to_c, sz, cent)               // calculate centroids for clusters
    this
} // train
```

After reducing the number of clusters to the desired number k (which defaults to 2), final cluster assignments are made and centroids are calculated. Intermediate clustering results are available making it easier for the user to pick the desired number of clusters after the fact. The algorithm can be rerun with this value for k.

`HierClusterer` **Class**

Class Methods:

```
@param x  the vectors/points to be clustered stored as rows of a matrix
@param k  stop when the number of clusters equals k

class HierClusterer (x: MatriD, k: Int = 2)
      extends Clusterer with Error

def train (): HierClusterer =
def cluster: Array [Int] = to_c
def centroids: MatriD = cent
def csize: VectoI = sz
```

```
    def classify (z: VectoD): Int = distance (z, cent).argmin ()
```

11.7.1 Exercises

1. Compare `HierClusterer` with `KMeansClustererHW` and `KMeansClusterer` for a variety of datasets, starting with the six points given in the `KMeansClusterer` section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the sse over the `NTESTS` test cases.

2. K-Means Clustering techniques often tend to produce better clusters (e.g., lower sse) than Hierarchical Clustering techniques. For what types of datasets might Hierarchical Clustering be preferred?

3. What is the relationship between Hierarchical Clustering and Dendrograms?

11.8 Markov Clustering

The `MarkovClusterer` class implements a Markov Clustering Algorithm (MCL) and is used to cluster nodes in a graph. The graph is represented as an edge-weighted adjacency matrix (a non-zero cell indicates nodes i and j are connected).

The primary constructor takes either a graph (adjacency matrix) or a Markov transition matrix as input. If a graph is passed in, the normalize method must be called to convert it into a Markov transition matrix. Before normalizing, it may be helpful to add self loops to the graph. The matrix (graph or transition) may be either dense or sparse. See the MarkovClusteringTest object at the bottom of the file for examples.

`MarkovClusterer` **Class**

Class Methods:

```
@param t  either an adjacency matrix of a graph or a Markov transition matrix
@param k  the strength of expansion
@param r  the strength of inflation

class MarkovClusterer (t: MatriD, k: Int = 2, r: Double = 2.0)
      extends Clusterer with Error

def train (): MarkovClusterer =
def cluster: Array [Int] = clustr
def centroids: MatriD = throw new UnsupportedOperationException ("not applicable")
def csize: VectoI = throw new UnsupportedOperationException ("not applicable")
def addSelfLoops (weight: Double = 1.0)
def normalize ()
def processMatrix (): MatriD =
def classify (y: VectoD): Int = throw new UnsupportedOperationException ()
```

11.8.1 Exercises

1. Draw the directed graph obtained from the following adjacency matrix, where `g(i, j) == 1.0` means that a directed edge exists from node *i* to node *j*.

```
val g = new MatrixD ((12, 12),
    0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0,  0.0,
    0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0,  0.0,
    1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,  1.0,
    1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,  0.0,
    0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0,  1.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,  0.0)
```

Apply the MCL Algorithm to this graph and explain the significance of the resulting clusters.

```
    val mg = new MarkovClusterer (g)
    mg.addSelfLoops ()
    mg.normalize ()
    println ("result  = " + mg.processMatrix ())
    mg.train ()
    println ("cluster = " + mg.cluster.deep)
```

Chapter 12

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.

12.1 Reducer

The `Reducer` trait provides a common framework for several data reduction algorithms.

Trait Methods:

```
trait Reducer

def reduce (): MatriD
def recover (): MatriD
```

12.2 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 13

Functional Data Analysis

13.1 Basis Functions

13.2 Functional Smoothing

13.3 Functional Principal Component Analaysis

13.4 Functional Regression

Chapter 14

Simulation Models

14.1 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 $\mathbf{y}$.

$$\mathbf{y} \;=\; \mathbf{f}(\mathbf{x}, t)$$

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 [24].

- **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:

  - $\{e\} = a_e(\mathbf{s}(t))$ and
  - $\mathbf{s}(t') = \mathbf{d_e}(\mathbf{s}(t))$.

  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 $\mathbf{y} = [y_0 \ y_1]$ where y_0 indicates the horizontal distance traveled, while y_1 indicates the vertical height of the ball. Future positions $\mathbf{y}$ depends on the current position and time t. Using Newton's Second Law of Motion, $\mathbf{y}$ can be estimated by solving a system of Ordinary Differential Equations (ODEs) such as

  $$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t), \ \mathbf{y}(0) = \mathbf{y_0}.$$

  The `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.

14.2 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`.

```
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

} // Bank
```

14.2.1 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

class Model (name: String, m: Int, rv: Array [Variate], label: Array [String])

def simulate ()
def report
```

14.3 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.

14.3.1 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

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
    } // occur

} // Arrival class
```

For the Departure class, the occur method will check to see if there is another customer waiting in the queue and if so, schedule that customer's departure. It will then signal its own departure by updating the state; in this case decrementing nIn and incrementing nOut.

```
@param customer  the entity that departs, in this case a bank customer

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

} // Departure class
```

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 **t**ime in **q**ueue t_q_stat, the **t**ime in **s**ervice t_s_stat and the **t**ime in s**y**stem 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 place 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])
def animate (who: Identity, what: Value, color: Color,
             shape: Shape, from: Event, to: Event, at: Array [Double] = Array ())
```

The `animate` methods are used with Event Graphs (see the next section).

14.3.2 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 specify 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.

```
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,

```
() => 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 casual 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).

```
//:: 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

//:: 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 14.1.

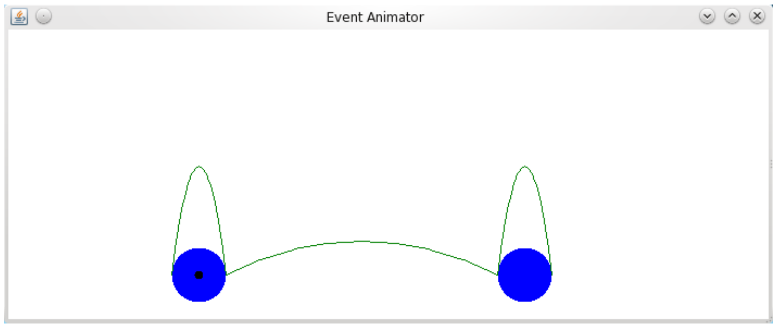

Figure 14.1: Event Graph Animation of a Bank.

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 `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

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

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])
```

CausalLink.scala

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

case class CausalLink (_name: String, director: Model, condition: () => Boolean, causedEvent: Event,
                       makeEvent: () => Event, delay: Variate, cancel: Boolean = false)
      extends Identity
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
```

14.4 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.

14.4.1 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
```

14.4.2 Signifiable.scala

The `Signifiable` trait defines standard messages sent between actors implementing process interaction simulations.

Class Methods:

```
trait Signifiable
```

14.4.3 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)
```

14.4.4 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 me made
@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,
def display ()
def act ()
```

14.4.5 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])
def display ()
def leave ()
```

14.4.6 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 ()
```

14.4.7 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 ()
```

14.4.8 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 (`SimActor`s) attempting to enter the queue are barred. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet). 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:

```
@param name  the name of the wait-queue
@param at    the location of the wait-queue (x, y, w, h)
@param cap   the capacity of the queue (defaults to unbounded)

class WaitQueue (name: String, at: Array [Double], cap: Int = Int.MaxValue)
      extends Queue [SimActor] with Component

def this (name: String, xy: Tuple2 [Double, Double], cap: Int)
def isFull: Boolean = length >= cap
def barred: Int = _barred
def display ()
def waitIn ()
def noWait ()
```

14.4.9 Junction.scala

The `Junction` class provides a connector between two transports/routes. Since `Line`s and `QCurve`s have limitation (e.g., hard to make a loop back), a junction may be needed.

Class Methods:

```
@param name  the name of the junction
@param director  the director controlling the model
@param jTime     the jump-time through the junction
@param at        the location of the junction (x, y, w, h)

class Junction (name: String, director: Model, jTime: Variate, at: Array [Double])
      extends Component

def this (name: String, director: Model, jTime: Variate, xy: Tuple2 [Double, Double])
def display ()
def move ()
```

14.4.10 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 $\implies$ open, red $\implies$ shut).

Class Methods:

```
@param name      the name of the gate
@param director  the model/container for this gate
@pram  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
```

14.4.11 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/transports in the route
@param from      the starting component
@param to        the ending component
@param motion    the speed/trip-time to move down the transports in the route
@param isSpeed   whether speed or trip-time is used for motion
@param angle     angle in radians of direction (0 => east, Pi/2 => north, Pi => west, 3Pi/2 => south)
@param bend      the bend or curvature of the route (0 => line)

class Route (name: String, k: Int, from: Component, to: Component,
             motion: Variate, isSpeed: Boolean = false,
             angle: Double = 0.0, bend: Double = 0.0)
      extends Component

override def at: Array [Double] = lane(0).at
def display ()
```

14.4.12 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])
def animate (who: Identifiable, what: Value, color: Color, shape: Shape,
             from: Component, to: Component, at: Array [Double] = Array ())
```

Appendices

Appendix A

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,

$$\begin{aligned} \text{minimize } & f(\mathbf{x}) \\ \text{subject to } & \mathbf{g}(\mathbf{x}) \le \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned}$$

where $f(\mathbf{x})$ is the objective function, $\mathbf{g}(\mathbf{x}) \le \mathbf{0}$ are the inequality constraints, and $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are the equality constraints. Consider the example below.

$$\begin{aligned} \text{minimize } & f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2 \\ \text{subject to } & \mathbf{g}(\mathbf{x}) = [x_1 - 3, x_2 - 1] \le \mathbf{0} \\ & h(\mathbf{x}) = x_1 - x_2 = 0 \end{aligned}$$

If we ignore all the constraints, the optimal solution is $\mathbf{x} = [4, 2]$ where $f(\mathbf{x}) = 0$, while enforcing the inequality constraints makes this solution infeasible. The new optimal solution is $\mathbf{x} = [3, 1]$ where $f(\mathbf{x}) = 2$. Finally, the optimal solution when all constraints are enforced is $\mathbf{x} = [1, 1]$ where $f(\mathbf{x}) = 10$. Note, for this example there is just one equality constraint that forces $x_1 = x_2$.

A.1 Gradient Descent

One the simplest algorithms for unconstrained optimiztion is Gradient Descent. Imagine you are in a mountain range at some point $\mathbf{x}$ with elevation $f(\mathbf{x})$. Your goal is the find the valley (or ideally the lowest valley). Look around (assume you cannot see very far) and determine the direction and magnitude of steepest ascent. This is the gradient.

Using the objective/cost function from the beginning of the chapter,

$$\text{minimize } f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2$$

the gradient of the objective function $\nabla f(\mathbf{x})$ is the vector formed by the partial derivatives $[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}]$

$$\nabla f(\mathbf{x}) \;=\; [2(x_1 - 4), 2(x_2 - 2)]$$

In its most elemental form the algorithm simply moves in the direction that is opposite to the gradient $-\nabla f(\mathbf{x})$ and a distance determined by the magnitude of the gradient. Unfortunately, at some points in the search space the magnitude of the gradient may be very large and moving that distance may result in divergence (you keep getting farther away from the valley). One solution is to temper the gradient by multiplying it by a learning rate η (tunable hyper-parameter typically smaller than one). Using a tuned learning rate, update your current location $\mathbf{x}$ as follows:

$$\mathbf{x} \;=\; \mathbf{x} \;-\; \eta \nabla f(\mathbf{x}) \tag{A.1}$$

Repeat this process until a stopping rule signals sufficient convergence. Examples of stopping rules include stop when the change to $\mathbf{x}$ or $f(\mathbf{x})$ becomes small or after the objective function has increased for too many consecutive iterations/steps.

A.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 sufficent 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 adpative learning rates or step sizes.

A.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 $hse(\mathbf{b})$

$$hse(\mathbf{b}) \;=\; (\mathbf{y} - f(X\mathbf{b}) \cdot (\mathbf{y} - f(X\mathbf{b})$$

in which case the gradient is

$$\nabla hse(\mathbf{b}) \;=\; -\,X^t[f'(X\mathbf{b})\,\boldsymbol{\epsilon}] \tag{A.2}$$

where $\boldsymbol{\epsilon} = \mathbf{y} - f(X\mathbf{b})$.

For each epoch, the parameter vector $\mathbf{b}$ is update according to equation 16.1.

$$\mathbf{b} \;=\; \mathbf{b} \,-\, \eta\nabla hse(\mathbf{b}) \tag{A.3}$$

A.1.3 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)                // vector to optimize
val eta = 0.1                            // 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)     // store x's trajectory
z(k-1) = x.copy
new Plot (z.col(0), z.col(1)
```

A.2 Stochastic Gradient Descent

Continuing for a `Perceptron`, starting with $hse(\mathbf{b})$

$$hse(\mathbf{b}) \;=\; (\mathbf{y} - f(X\mathbf{b}) \cdot (\mathbf{y} - f(X\mathbf{b})$$

an estimate of the gradient is computed for a limited number of instances (a 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

$$\nabla hse(\mathbf{b}) \;=\; -\,X[i_B]^t[f'(X[i_B]\mathbf{b})\,\boldsymbol{\epsilon}] \tag{A.4}$$

where $\boldsymbol{\epsilon} = \mathbf{y}[i_B] - f(X[i_B]\mathbf{b})$. Using the definition of the delta vector

$$\boldsymbol{\delta} \;=\; -\,f'(X[i_B]\mathbf{b})\,\boldsymbol{\epsilon}$$

the gradient becomes

$$\nabla hse(\mathbf{b}) \;=\; X[i_B]^t\boldsymbol{\delta}$$

For each epoch, n_B batches are created. For each batch, the parameter vector $\mathbf{b}$ is update according to equation 16.1, using that batch's estimate for the gradient.

$$\mathbf{b} \;=\; \mathbf{b} \,-\, \eta\nabla hse(\mathbf{b}) \;=\; \mathbf{b} \,-\, X[i_B]^t\boldsymbol{\delta}\eta \tag{A.5}$$

The corresponding ScalaTion code is in the `Optimizer_SGD` object.

```
def optimize (x: MatriD, y: VectoD, b: VectoD,
              eta_ : 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     = eta_                                       // 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())
```

```
                return (sse_best, epoch - UP_LIMIT)
            } // if
            if (epoch % ADJUST_PERIOD == 0) eta *= ADJUST_FACTOR    // adjust the learning rate
        } // for
```

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).

```
        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.

A.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**.

$$\mathbf{b_{up}} \;=\; \eta \nabla hse(\mathbf{b}) \;=\; X[i_B]^t \boldsymbol{\delta} \eta \tag{A.6}$$

Again for a `Perceptron`, one can include a momentum vector $\mathbf{m_o}$ to hold the weighted average of recent gradients. The momentum vector $\mathbf{m_o}$ is then simply the sum of the old momentum decayed by parameter β and $\mathbf{b_{up}}$.

$$\mathbf{m_o} \;=\; \beta \mathbf{m_o} + \mathbf{b_{up}} \tag{A.7}$$

Then the parameters are updated as follows:

$$\mathbf{b} \;=\; \mathbf{b} - \mathbf{m_o} \tag{A.8}$$

If β is zero, that algorithm behaves the same as Stochastic Gradient Descent. At the other extreme, if β is 1, there is no decay and all previous gradients will weigh in, so eventually the new gradient value will have little impact and the algorithm will become oblivious to its local environment. 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.

```
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
} // updateWeight
```

See the `Optimzer_SGDM` for more coding details.

A.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 muliplier λ. At optimality, the gradient of f should be orthogonal to the surface defined by the constraint $h(\mathbf{x}) = 0$, otherwise, moving along the surface in the opposite direction to the gradient ($-\nabla f(\mathbf{x})$ for minimization) would improve the solution. Since the gradient of h, $\nabla h(\mathbf{x})$, is orthogonal to the surface as well, this implies that the two gradients should only differ by a constant multiplier λ.

$$-\nabla f(\mathbf{x}) \;=\; \lambda \nabla h(\mathbf{x}) \tag{A.9}$$

In general, such problems can solved by defining the *Lagrangian*

$$L(\mathbf{x}, \boldsymbol{\lambda}) \;=\; f(\mathbf{x}) - \boldsymbol{\lambda} \cdot \mathbf{h}(\mathbf{x}) \tag{A.10}$$

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers. When there is a single equality constraint, this becomes

$$L(\mathbf{x}, \lambda) \;=\; f(\mathbf{x}) - \lambda\, h(\mathbf{x})$$

Taking the gradient of the Lagrangian w.r.t. $\mathbf{x}$ and λ yields a vector of dimension $n + 1$.

$$\nabla L(\mathbf{x}, \lambda) \;=\; [\, \nabla f(\mathbf{x}) - \lambda \nabla h(\mathbf{x}), h(\mathbf{x})\,]$$

Now we may try setting the gradient to zero and solving a system of equations.

A.4.1 Example Problem

The Lagrangian for the problem given at the beginning of the chapter is

$$L(\mathbf{x}, \lambda) \;=\; (x_1 - 4)^2 + (x_2 - 2)^2 - \lambda\, (x_1 - x_2)$$

Computation of the gradient $[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial \lambda}]$ of the Lagrangian yields the following three equations,

$$\begin{aligned} -2(x_1 - 4) &= \lambda \\ -2(x_2 - 2) &= -\lambda \\ x_1 - x_2 &= 0 \end{aligned}$$

The first two equations are from the gradient w.r.t. $\mathbf{x}$, while the third equation is simply the constraint itself $h(\mathbf{x}) = 0$. The equations may be rewritten in the following form.

$$\begin{aligned} 2x_1 + \lambda &= 8 \\ 2x_2 - \lambda &= 4 \\ x_1 - x_2 &= 0 \end{aligned}$$

This is a linear system of equations with 3 variables $[x_1, x_2, \lambda]$ and 3 equations that may be solved, for example, by LU Factorization. In this case, the last equation gives $x_1 = x_2$, so adding equations 1 and 2 yields $4x_1 = 12$. Therefore, the optimal value is $\mathbf{x} = [3, 3]$ with $\lambda = 2$ where $f(\mathbf{x}) = 2$.

Adding an equality constraint is addressed by adding another Lagrange mutiplier, e.g., 4 variables $[x_1, x_2, \lambda_1, \lambda_2]$ and 4 equations, two from the gradient w.r.t. $\mathbf{x}$ and one for each of the two constraints.

Linear systems of equations are generated when the objective function is at most quadratic and the constraints are linear. If this is not the case, a nonlinear system of equations may be generated.

A.5 Karush-Kuhn-Tucker Conditions

Introducing inequality constraints makes the situation is a little more complicated. A generalization of the Method of Lagrange Multipliers based on the Karush-Kuhn-Tucker (KKT) conditions is needed. For minimization, the KKT conditions are as follows:

$$-\nabla f(\mathbf{x}) \;=\; \boldsymbol{\alpha}\cdot\nabla\mathbf{g}(\mathbf{x}) + \boldsymbol{\lambda}\cdot\nabla\mathbf{h}(\mathbf{x}) \tag{A.11}$$

The original constraints must also hold.

$$\mathbf{g}(\mathbf{x}) \le \mathbf{0} \quad \text{and} \quad \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

Furthermore, the Lagrange multipliers for the inequality constraints $\boldsymbol{\alpha}$ are themselves constrained to be nonnegative.

$$\boldsymbol{\alpha} \ge \mathbf{0}$$

When the objective function is at most quadratic and the constraints are linear, the problem of finding an optimal value for $\mathbf{x}$ is referred to a Quadratic Programming. Many estimation/learning problems in data science are of this form. Beyond Quadratic Programming lies problems in Nonlinear Programming. Linear Programming (linear objective function and linear constraints) typically finds less use (e.g., Quantile Regression) in estimation/learning, so it will not be covered in this Chapter, although it is provided by ScalaTion.

A.5.1 Active and Inactive Constraints

A.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.

$$\begin{gathered}\text{minimize } f(\mathbf{x}) \\ \text{subject to } \mathbf{h}(\mathbf{x}) = \mathbf{0}\end{gathered}$$

where $f(\mathbf{x})$ is the objective function and $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are the equality constraints.

In penalty form, the constrained optimization problem becomes.

$$\text{minimize } f(\mathbf{x}) + \frac{\rho_k}{2} ||\mathbf{h}(\mathbf{x})||_2^2$$

where k is the iteration counter. The square of the Euclidean norm indicates to what degree the equality contraints are violated. Replacing the square of the Euclidean norm with the dot product gives.

$$\text{minimize } f(\mathbf{x}) + \frac{\rho_k}{2} \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{x})$$

The value of ρ_k increases (e.g., linearly) with k and thereby enforces the equality constraints more strongly with each iteration.

An alternative to minimizing $f(\mathbf{x})$ with a quadratic penalty is to minimize using the Augmented Lagrangian $L(\mathbf{x}, \rho_k, \boldsymbol{\lambda})$.

$$L(\mathbf{x}, \rho_k, \boldsymbol{\lambda}) = f(\mathbf{x}) + \frac{\rho_k}{2} \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{x}) - \boldsymbol{\lambda} \cdot \mathbf{h}(\mathbf{x}) \tag{A.12}$$

where $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers. After each iteration, the Lagrange multipliers are updated.

$$\boldsymbol{\lambda} = \boldsymbol{\lambda} - \rho_k \mathbf{h}(\mathbf{x})$$

This method allows for quicker convergence without the need for the penalty ρ_k to become as large (see the exercises for a comparison of the Augmented Lagranian Method with the Penalty Method). This method may be conbined 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 contraints.

A.6.1 Example Problem

Consider the problem given at the beginning of the chapter with the inequality constraint left out.

$$\begin{gathered}\text{minimize } f(\mathbf{x}) = (x_1 - 4)^2 + (x_2 - 2)^2 \\ \text{subject to } h(\mathbf{x}) = x_1 - x_2 = 0\end{gathered}$$

where $\mathbf{x} \in \mathbb{R}^2$, f is the objective function and h is the single equality contraint. The Augmented Lagrangian for this problem is

$$L(\mathbf{x}, \rho_k, \lambda) \;=\; (x_1 - 4)^2 + (x_2 - 2)^2 + \frac{\rho_k}{2}\,(x_1 - x_2)^2 - \lambda(x_1 - x_2) \tag{A.13}$$

The gradient of the Augmented Lagrangian $\nabla L(\mathbf{x}, \rho_k, \lambda)$ is made up of the following two partial derivatives.

$$\begin{aligned}
\partial/\partial x_1 &= 2(x_1 - 4) + \frac{\rho_k}{2}\,2(x_1 - x_2) - \lambda \\
\partial/\partial x_2 &= 2(x_2 - 2) - \frac{\rho_k}{2}\,2(x_1 - x_2) + \lambda
\end{aligned}$$

The Lagrange multiplier updates becomes

$$\lambda = \lambda - \rho_k\,(x_1 - x_2)$$

The code in the exercises tightly integrates the Gradient Descent algorithm with the Augmented Lagrangian method by updating the penalty and Lagrange multiplier during each iteration.

A.6.2 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

// 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.

A.7 Quadratic Programming

The `QuadraticSimplex` class solves Quadratic Programming (QP) problems using the Quadratic Simplex Algorithm. Given a constraint matrix A, constant vector $\mathbf{b}$, cost matrix Q and cost vector $\mathbf{c}$, find values for the solution/decision vector $\mathbf{x}$ that minimize the objective function $f(\mathbf{x})$, while satisfying all of the constraints, i.e.,

$$\begin{aligned}\text{minimize} \quad f(\mathbf{x}) &= \frac{1}{2}\mathbf{x} \cdot Q\mathbf{x} + \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} \quad \mathbf{g}(\mathbf{x}) &= A\mathbf{x} - \mathbf{b} \leq \mathbf{0}\end{aligned}$$

Before considering the type of optimization algorithm to use, we may simplify the problem by applying the KKT conditions.

$$-\nabla f(\mathbf{x}) \;=\; Q\mathbf{x} + \mathbf{c} \;=\; \boldsymbol{\alpha} \cdot \nabla g(\mathbf{x}) \;=\; \boldsymbol{\alpha} \cdot A$$

Adding the constraints gives the following n equations and $2m$ constraints:

$$\begin{aligned}Q\mathbf{x} + \mathbf{c} &= \boldsymbol{\alpha} \cdot A \\ A\mathbf{x} - \mathbf{b} &\leq \mathbf{0} \\ \boldsymbol{\alpha} &\geq \mathbf{0}\end{aligned}$$

These equations have two unknown vectors, $\mathbf{x}$ of dimension n and $\boldsymbol{\alpha}$ of dimension m.

The algorithm creates an simplex tableau. This implementation is restricted to linear constraints $A\mathbf{x} \leq \mathbf{b}$ 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)
 */
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 =
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 ()
```

A.8 Coordinate Descent

A.9 Conjugate Gradient

A.10 Quasi-Newton Method

A.11 Alternating Direction Method of Multipliers

A.12 Nelder-Mead Simplex

Appendix B

Parallel and Distributed Computing

B.1 MIMD - Multithreading

B.2 SIMD - Vector Instructions

B.3 Message Passing

B.4 Distributed Shared Memory

B.5 Microservices

B.6 Distributed Functional Programming

Bibliography

[1] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Symposium on Computational Geometry*, volume 6, pages 1–10, 2006.

[2] Concha Bielza and Pedro Larrañaga. Discrete Bayesian Network Classifiers: A Survey. *ACM Computing Surveys (CSUR)*, 47(1):5, 2014.

[3] Stephan Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares.* Cambridge University Press, 2018.

[4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[5] David Maxwell Chickering. Learning bayesian networks is np-complete. In *Learning from data*, pages 121–130. Springer, 1996.

[6] Thomas M. Cover and Joy A. Thomas. Entropy, Relative Entropy and Mutual Information. Technical report, 1991.

[7] Justin Domke. Statistical Machine Learning Notes: Trees. Technical report, Uinversity of Massachusetts, 2018.

[8] Valeria Fonti and Eduard Belitser. Feature Selection using LASSO. Technical report, VU Amsterdam, 2017.

[9] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, 4th Edition*, volume 3. JHU Press, 2013.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition.* Springer-Verlag New York, 2009.

[12] Pili Hu. Matrix Calculus: Derivation and Simple Application. Technical report, City University of Hong Kong, 2012.

[13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer-Verlag New York, 2013.

[14] Franco Manessi and Alessandro Rozza. Learning combinations of activation functions. *arXiv preprint arXiv:1801.09403*, 2018.

[15] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*, volume 821. John Wiley & Sons, 2012.

[16] Jeremy Orloff and Jonathan Bloom. Maximum Likelihood Estimates. Technical report, MIT, 2014.

[17] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[18] Suhasini Rao. A course in Time Series Analysis. Technical report, Texas A&M University, 2018.

[19] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

[20] Lior Rokack. Decision Trees. Technical report, Tel-Aviv University, 2015.

[21] Dan Roth. Decision Trees. Technical report, University of Illinois, 2016.

[22] Prasanna Sahoo. *Probability and Mathematical Statistics*. University of Louisville, Louisville, KY 40292, USA, 2013.

[23] Cosma Shalizi. Modern Regression. Technical report, Carneige-Mellon University, 2015.

[24] Gregory A Silver, John A Miller, Maria Hybinette, Gregory Baramidze, and William S York. An ontology for discrete-event modeling and simulation. *Simulation*, 87(9):747–773, 2011.

[25] Mark Stamp. A Revealing Introduction to Hidden Markov Models. Technical report, San Jose State University, 2018.

[26] Thaddeus Tarpey. Generalized Linear Models (GLM). Technical report, Wright State University, 2012.

[27] Harry Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.