Introduction to Data Science
Using ScalaTion

John A. Miller
Department of Computer Science
University of Georgia

October 23, 2018
3 Data Management and Preprocessing

3.1 Analytics Databases

3.1.1 Columnar Relational Algebra API

3.1.2 SQL-Like API

3.1.3 Map-Reduce API

3.2 Preprocessing

3.2.1 Remove Identifiers

3.2.2 Convert String Columns to Numeric Columns

3.2.3 Identify Missing Values

3.2.4 Detect Outliers

3.2.5 Imputation Techniques

3.2.6 Preliminary Feature Selection

3.2.7 Align Multiple Time Series

3.2.8 Creating Vectors and Matrices

3.3 Exercises

4 Prediction

4.1 Predictor

4.1.1 Fit

4.1.2 PredictorMat

4.2 Null Model

4.2.1 Exercises

4.3 Simpler Regression

4.3.1 Exercises

4.4 Simple Regression

4.4.1 Exercises

4.5 Regression

4.5.1 Matrix Inversion Technique

4.5.2 LU Factorization Technique

4.5.3 Cholesky Factorization Technique

4.5.4 QR Factorization Technique

4.5.5 Singular Value Decomposition Technique

4.5.6 Use of Factorization in Regression

4.5.7 Model Assessment

4.5.8 Model Validation

4.5.9 Collinearity

4.5.10 Feature Selection

4.5.11 Exercises

4.5.12 Further Reading

4.6 Weighted Least Squares Regression

4.6.1 Exercises

4.7 Ridge Regression

4.7.1 Exercises

4.8 Lasso Regression

4.8.1 Exercises
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Temporal Models</td>
<td>159</td>
</tr>
<tr>
<td>9.1</td>
<td>State Space Models</td>
<td>160</td>
</tr>
<tr>
<td>9.2</td>
<td>Exponential Smoothing</td>
<td>161</td>
</tr>
<tr>
<td>9.3</td>
<td>Auto-Regressive (AR) Models</td>
<td>162</td>
</tr>
<tr>
<td>9.4</td>
<td>Moving-Average (MA) Models</td>
<td>163</td>
</tr>
<tr>
<td>9.5</td>
<td>ARMA</td>
<td>164</td>
</tr>
<tr>
<td>9.6</td>
<td>ARIMA</td>
<td>165</td>
</tr>
<tr>
<td>9.7</td>
<td>SARIMA</td>
<td>167</td>
</tr>
<tr>
<td>9.8</td>
<td>Recurrent Neural Networks (RNN)</td>
<td>168</td>
</tr>
<tr>
<td>9.8.1</td>
<td>Long Short Term Memory (LSTM) Networks</td>
<td>168</td>
</tr>
<tr>
<td>9.9</td>
<td>Temporal Convolutional Networks (TCN)</td>
<td>169</td>
</tr>
<tr>
<td>9.10</td>
<td>ODE Parameter Estimation</td>
<td>170</td>
</tr>
<tr>
<td>9.10.1</td>
<td>Non-Linear Least Squares (NLS)</td>
<td>170</td>
</tr>
<tr>
<td>9.10.2</td>
<td>Least Squares Approximation (LSA)</td>
<td>170</td>
</tr>
<tr>
<td>10</td>
<td>Spatial Models</td>
<td>171</td>
</tr>
<tr>
<td>10.1</td>
<td>Convolutional Neural Networks</td>
<td>172</td>
</tr>
<tr>
<td>11</td>
<td>Clustering</td>
<td>173</td>
</tr>
<tr>
<td>11.1</td>
<td>Clusterer</td>
<td>174</td>
</tr>
<tr>
<td>11.2</td>
<td>K-Means Clustering</td>
<td>175</td>
</tr>
<tr>
<td>11.3</td>
<td>Hierarchical Clustering</td>
<td>176</td>
</tr>
<tr>
<td>11.4</td>
<td>Markov Clustering</td>
<td>177</td>
</tr>
<tr>
<td>12</td>
<td>Dimensionality Reduction</td>
<td>179</td>
</tr>
<tr>
<td>12.1</td>
<td>Reducer</td>
<td>180</td>
</tr>
<tr>
<td>12.2</td>
<td>Principal Component Analytics</td>
<td>181</td>
</tr>
<tr>
<td>13</td>
<td>Functional Data Analysis</td>
<td>183</td>
</tr>
<tr>
<td>13.1</td>
<td>Basis Functions</td>
<td>184</td>
</tr>
<tr>
<td>13.2</td>
<td>Functional Smoothing</td>
<td>185</td>
</tr>
<tr>
<td>13.3</td>
<td>Functional Principal Component Analysis</td>
<td>186</td>
</tr>
<tr>
<td>13.4</td>
<td>Functional Regression</td>
<td>187</td>
</tr>
<tr>
<td>14</td>
<td>Optimization</td>
<td>189</td>
</tr>
<tr>
<td>14.1</td>
<td>Ensemble Learning</td>
<td>190</td>
</tr>
<tr>
<td>15</td>
<td>Optimization</td>
<td>191</td>
</tr>
<tr>
<td>15.1</td>
<td>Method of Lagrange Multipliers</td>
<td>192</td>
</tr>
<tr>
<td>15.2</td>
<td>Karush-Kuhn-Tucker Conditions</td>
<td>193</td>
</tr>
<tr>
<td>15.2.1</td>
<td>Active and Inactive Constraints</td>
<td>193</td>
</tr>
<tr>
<td>15.3</td>
<td>Quadratic Programming</td>
<td>194</td>
</tr>
<tr>
<td>15.4</td>
<td>Coordinate Descent</td>
<td>196</td>
</tr>
<tr>
<td>15.5</td>
<td>Gradient Descent</td>
<td>197</td>
</tr>
<tr>
<td>15.6</td>
<td>Stochastic Gradient Descent</td>
<td>198</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction to Data Science

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. Data is becoming too large for traditional approaches for statistical analysis or machine learning. 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. It contains four types of tools: predictors, classifiers, clusterers and reducers. A trait is defined for each type. The sister package graphalytics provides tools for performing graph analytics.

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).

More detailed development of the material in the report can be found in textbooks on statistical learning, such as “An Introduction to Statistical Learning” (ISL) [8] and “The Elements of Statistical Learning” (ESL) [6].
<table>
<thead>
<tr>
<th>Topic</th>
<th>SCALaTION</th>
<th>ISL</th>
<th>ESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathematical Preliminaries</td>
<td>Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data Management</td>
<td>Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prediction</td>
<td>Ch. 4</td>
<td>Ch. 3, 5, 6</td>
<td>Ch. 3</td>
</tr>
<tr>
<td>Classification</td>
<td>Ch. 5</td>
<td>Ch. 2, 5, 8</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Classification - Continuous</td>
<td>Ch. 6</td>
<td>Ch. 4, 8, 9</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Generalized Linear Models</td>
<td>Ch. 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Non-Linear Models</td>
<td>Ch. 8</td>
<td>Ch. 7</td>
<td>Ch. 11</td>
</tr>
<tr>
<td>Temporal Models</td>
<td>Ch. 9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spatial Models</td>
<td>Ch. 10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Clustering</td>
<td>Ch. 11</td>
<td>Ch. 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Dimensional Reduction</td>
<td>Ch. 12</td>
<td>Ch. 6, 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Functional Data Analysis</td>
<td>Ch. 13</td>
<td>Ch. 7</td>
<td>Ch. 5</td>
</tr>
<tr>
<td>Ensemble Learning</td>
<td>Ch. 14</td>
<td>-</td>
<td>Ch. 16</td>
</tr>
<tr>
<td>Optimization</td>
<td>Ch. 15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Parallel Computing</td>
<td>Ch. 16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Distributed Computing</td>
<td>Ch. 17</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1.1: Source Material Chapter Mappings
Chapter 2

Mathematical Preliminaries

This chapter serves as a quick review of the two principal mathematical foundations for data science, probability and linear algebra.
2.1 Probability

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

2.1.1 Probability Measure

A probability measure \( P \) can be defined axiomatically as follows:

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

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

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

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

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

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

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

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

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

2.1.2 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 \( D_y \).

\[
y \in D_y
\]

For \( A \subseteq D_y \) one can measure the probability of the random variable \( y \) taking on a value from the set \( A \). This is denoted by \( P(y \in A) \). For example, the probability of rolling a natural in dice (sum of 7 or 11 with two dice) is given by

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

It is often easier to examine the probability measure for a random variable in terms of a Cumulative Distribution Function (CDF). It measures the amount probability or mass accumulated over the domain up to and including the point \( y \). The color highlighted symbol \( y \) is the random variable, while \( y \) simply represents a value.

\[
F_y(y) = P(y \leq y)
\] (2.6)

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

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

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

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

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

2.1.4 Probability Mass Function

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

\[
p_y(y_i) = F_y(y_i) - F_y(y_{i-1})
\] (2.7)

This indicates the amount of mass/probability at point \( y_i \), i.e., the amount of accumulated mass at point \( y_i \) minus the amount of accumulated mass at the previous point \( y_{i-1} \). For one dice \( x_1 \), the pmf is

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

A second dice \( x_2 \) will have the same pmf. They both follow the Discrete Uniform Distribution. If the two random variables are added \( y = x_1 + x_2 \), the pmf for the random variable \( y \) (roll of two dice), \( p_y(y) \) is

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

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

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

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

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.

```scala
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.

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

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:

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

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

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

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

\[\mathbb{E}[y] = \sum_{y \in D_y} y p_y(y)\] (2.11)

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

### 2.1.7 Variance

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

\[\mathbb{V}[y] = \mathbb{E}[(y - \mathbb{E}[y])^2]\] (2.12)

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

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

That is, the variance of the one kilogram, two-meter rod is \( \frac{1}{3} \) kilogram meter\(^2\). Again, for probability to be viewed as mass, unit mass (one kilogram) must be used, so the answer may also be given as \( \frac{1}{3} \) meter\(^2\).

Similarly to interpreting the mean as the center of mass, the variance corresponds to the moment of inertia. The standard deviation is simply the square root of variance.

\[\text{SD}[y] = \sqrt{\mathbb{V}[y]}\] (2.13)

For the two-meter rod, the standard deviation is \( \sqrt{\frac{1}{3}} = 0.57735 \). 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

\[\text{C}[x, y] = \mathbb{E}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])]\] (2.14)

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

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

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

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

(2.15)

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

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

(2.16)

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

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

Taking the inverse yields the iCDF.

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

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

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

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

(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.

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

(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:

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

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

$$p_x(x) = \sum_{y \in D_y} p_{x,y}(x, y)$$  (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:

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

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

$$f_x(x) = \int_{y \in D_y} f_{x,y}(x, y) dy$$  (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:

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

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

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

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

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

2.1.13 Odds

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

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

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

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

### 2.1.14 Example Problems

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

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

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

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

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

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

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

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

where

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

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

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

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

\[
\frac{P(y \in \{7, 11\} \mid y \in \{5, 7\}) P(y \in \{5, 7\})}{P(y \in \{7, 11\})} = \frac{(3/5) \cdot (5/18)}{2/9} = \frac{3}{4}
\]
2. To illustrate the usefulness of Bayes Theorem, consider the following problem from John Allen Paulos that is hard to solve without it. Suppose you are given three coins, two fair and one counterfeit (always lands heads). Randomly select one of the coins. Let $x$ indicate whether the selected coin is fair (0) or counterfeit (1). What is the probability that you selected the counterfeit coin?

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

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

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

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

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

3. Suppose the speeds of cars on an interstate highway are Normally distributed with a mean at the speed limit of 70 mph (113 kph) and a standard deviation of 8 mph (13 kph). Create a sample of size $m = 100$ data points, using a Normal random variate generator. The population values for the mean $\mu$ and standard deviation $\sigma$ are typically unknown and need to estimated from the sample, hence the names sample mean $\hat{\mu}$ and sample standard deviation $\hat{\sigma}$. Show the generated sample, by plotting the data points and displaying a histogram.

```scala
val (mu, sig) = (70.0, 8.0) // population mean and standard deviation
val m = 100 // sample size
val t = VectorD.range (0, m) // time/index vector
val rvg = Normal (mu, sig * sig) // Normal random variate generator

val sample = new VectorD (m) // vector to hold sample
for (i <- sample.range) sample(i) = rvg.gen // sample from Normal distribution
val (mu_, sig_) = (sample.mean, sample.stddev) // sample mean and standard deviation
println (s"(mu_, sig_) = ($mu_, $sig_)")
new Plot (t, sample)
new Histogram (sample)
```


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

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

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

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

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

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

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

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

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

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

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

In other words,

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

This condition implies that \( \mu \) would likely be inside the following confidence interval.

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

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

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

```scala
println (s"1 - F(2) = \${1 - normalCDF (2)}")
```
The probability is one minus twice this value. If 1.96 is used instead of 2, what is the probability, expressed as a percentage.

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

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

being replace by the Student t-distribution

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

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

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

for (it <- 1 to iter) {
  val sample = new VectorD (m) // vector to hold sample
  for (i <- sample.range) sample(i) = rvgen.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
}

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?

2.1.15 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:

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

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

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

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

6. Use the \texttt{Uniform} random variate generator and the \texttt{Histogram} class to run experiments illustrating the Central Limit Theorem (CLT).

```scala
import scalation.linalgebra.VectorD
import scalation.stat.Histogram
import scalation.random.Uniform

object CLTTest extends App
{
  val rg = Uniform ()
  val x = VectorD (for (i <- 0 until 100000) yield rg.gen + rg.gen + rg.gen + rg.gen)
  new Histogram (x)
} // CLTTest object
```
7. Imagine you are a contestant on the *Let’s Make a Deal* game show and host, Monty Hall, asks you to select door number 0, 1 or 2, behind which are two worthless prizes and one luxury car. Whatever door you pick, he randomly opens one of the other non-car doors and asked if you want to stay with your initial choice or switch to the remaining door. What are the probabilities of winning if you (a) stay with your initial choice, or (b) switch to the other door? Finish the code below to validate your results.

```scala
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)
}
```

2.1.16 Further Reading

1. Probability and Mathematical Statistics [14]
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 $y$, the values for predictors/features are collected in a matrix $X$ and the parameters/coefficients $b$ are fit to the data.

2.2.1 Linear System of Equations

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

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

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

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

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

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

\[
y = Xb
\]

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

2.2.2 Matrix Inversion

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

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

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

\[
X = LU
\]

Then $LUb = y$, so multiplying both sides by $L^{-1}$ gives $Ub = L^{-1}y$. Taking an augmented matrix

\[
\begin{bmatrix}
 1 & 3 & 1 \\
 2 & 1 & 7
\end{bmatrix}
\]
and performing row operations to make it upper right triangular has the effect of multiplying by \( L^{-1} \). In this case, the first row multiplied by -2 is added to second row to give:

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

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

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

In cases where \( m > n \), the system may be overdetermined, and no solution will exist. Values for \( \mathbf{b} \) 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 \textsc{ScalaTion} are in lowercase, since by convention, uppercase indicates a type, not a variable. \textsc{ScalaTion} supports vectors and matrices in its \texttt{linalg}bra and \texttt{linalg}bra-gen packages. A commonly used operation is the dot (inner) product, \( \mathbf{x} \cdot \mathbf{y} \), or in \textsc{ScalaTion}, \( \texttt{x dot y} \).

### 2.2.3 Vector

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

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

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) (\( \mathbf{x} \ast \mathbf{y} \)), 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.

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

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

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

The norm of \( \mathbf{y} \) is \( \sqrt{2} \). If \( \theta \) 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 \( \theta \) is,
\[ \cos(\theta) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{1.1547}{1 \cdot \sqrt{2}} = 0.8165 \]

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

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

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

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

\[
\mu(x) = \mu_x = \frac{1}{m} \cdot x
\]

\[
\sigma^2(x) = \sigma_x^2 = \frac{(x - \mu_x) \cdot (x - \mu_x)}{m} = \frac{x \cdot x}{m} - \mu_x^2
\]

\[
\sigma(x, y) = \sigma_{x,y} = \frac{(x - \mu_x) \cdot (y - \mu_y)}{m} = \frac{x \cdot y}{m} - \mu_x \mu_y
\]

\[
\rho(x, y) = \rho_{x,y} = \frac{\sigma_{x,y}}{\sigma_x \sigma_y}
\]

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 \( \|\mathbf{1}\|^2 = 1 \cdot \mathbf{1} = m \). 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 \( u(t) \) be the location of an object (e.g., golf ball) in three dimensional space \( \mathbb{R}^3 \) at time \( t \),

\[
u(t) = [x(t), y(t), z(t)]
\]

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

\[
u(t) = \nu(0) + v(0) t + \frac{1}{2} a t^2
\]

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

### 2.2.5 Gradient

Consider the following function \( f \) of vector \( u = [x, y] \)

\[ f(u) = (x - 2)^2 + (y - 3)^2 \]
The gradient of function \( f \)

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

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 \( u = [2, 3] \) where \( f = 0 \). For more complex functions, repeatedly moving in the opposite direction to the gradient, may lead to finding a minimal value.

Vector operations are illustrated by the `VectoD` trait, which includes methods for size, indices, set, copy, filter, select, concatenate, vector arithmetic, power, square, reciprocal, abs, sum, mean variance, rank, cumulate, normalize, dot, norm, max, min, mag, argmax, argmin, indexOf, indexWhere, count, contains, sort and swap.

<table>
<thead>
<tr>
<th>op</th>
<th>vector op vector</th>
<th>vector op scalar</th>
<th>vector element op scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>def + (b: VectoD): VectoD</td>
<td>def + (s: Double): VectoD</td>
<td>def + (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>+=</td>
<td>def += (b: VectoD): VectoD</td>
<td>def += (s: Double): VectoD</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>def - (b: VectoD): VectoD</td>
<td>def - (s: Double): VectoD</td>
<td>def - (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>-=</td>
<td>def -= (b: VectoD): VectoD</td>
<td>def -= (s: Double): VectoD</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>def * (b: VectoD): VectoD</td>
<td>def * (s: Double): VectoD</td>
<td>def * (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>*=</td>
<td>def *= (b: VectoD): VectoD</td>
<td>def *= (s: Double): VectoD</td>
<td>-</td>
</tr>
<tr>
<td>/</td>
<td>def / (b: VectoD): VectoD</td>
<td>def / (s: Double): VectoD</td>
<td>def / (s: (Int, Double)): VectoD</td>
</tr>
<tr>
<td>/=</td>
<td>def /= (b: VectoD): VectoD</td>
<td>def /= (s: Double): VectoD</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.1: Vector Arithmetic Operations

2.2.6 Matrix

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

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

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

```scala
case class MatrixD(rows: Int, cols: Int, elements: DoubleArray) { 
  def + (b: VectoD): VectoD = b + this 
  def += (b: VectoD): VectoD = b += this 
  def -= (b: VectoD): VectoD = b -= this 
  def * (b: VectoD): VectoD = b * this 
  def *= (b: VectoD): VectoD = b *= this 
  def / (b: VectoD): VectoD = b / this 
  def /= (b: VectoD): VectoD = b /= this 
} 

val a = MatrixD((3, 2), 1, 2, 
                 3, 4, 
                 5, 6) 
```
Matrix $A$ will transform $\mathbf{u}$ vectors in $\mathbb{R}^2$ into $\mathbf{v}$ vectors in $\mathbb{R}^3$.

$$A\mathbf{u} = \mathbf{v}$$

(2.36)

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.

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

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.

**Matrix Addition and Subtraction**

Matrix addition val $c = \mathbf{a} + \mathbf{b}$

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

and matrix subtraction val $c = \mathbf{a} - \mathbf{b}$ are supported.

**Matrix Multiplication**

A frequently used operation in data science is matrix multiplication val $c = \mathbf{a} \ast \mathbf{b}$.

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

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 $B$.  

28
Matrix Transpose

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

```scala
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 $|A|$ ($\text{val } d = a.det$), 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 $A^tB = A \cdot B = a.t \ast b = a \text{ mdot } b$.

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

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

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

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

Table 2.2: Matrix Factorization Techniques

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

2.2.9 Internal Representation

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

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

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 `linalggebra_gen`, but they tend to run more slowly.

ScalaTion supports many operations involving matrices and vectors, including the following:
### Table 2.3: Types of Vectors and Matrices: Implementing Classes

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

### Table 2.4: Types of Vector and Matrix Products

<table>
<thead>
<tr>
<th>Product</th>
<th>Method</th>
<th>Example</th>
<th>in Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector dot</td>
<td>def dot (y: VectoD): Double</td>
<td>x dot y</td>
<td>x \cdot y</td>
</tr>
<tr>
<td>vector elementwise</td>
<td>def * (y: VectoD): VectoD</td>
<td>x * y</td>
<td>x \cdot y</td>
</tr>
<tr>
<td>vector outer</td>
<td>def outer (y: VectoD): MatriD</td>
<td>x outer y</td>
<td>x \otimes y</td>
</tr>
<tr>
<td>matrix mult</td>
<td>def * (y: MatriD): MatriD</td>
<td>x * y</td>
<td>X Y</td>
</tr>
<tr>
<td>matrix dot</td>
<td>def dot (y: MatriD): VectoD</td>
<td>x dot y</td>
<td>X \cdot Y</td>
</tr>
<tr>
<td>matrix mdot</td>
<td>def mdot (y: MatriD): MatriD</td>
<td>x mdot y</td>
<td>X' Y</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def * (y: VectoD): VectoD</td>
<td>x * y</td>
<td>X y</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def ** (y: VectoD): MatriD</td>
<td>x ** y</td>
<td>X \text{diag}(y)</td>
</tr>
</tbody>
</table>

### 2.2.10 Exercises

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

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

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

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

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

### 2.2.11 Further Reading

1. Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares [1]
2.3 Notational Conventions

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

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

Table 2.5: Notational Conventions Followed
Data Science relies on having large amounts of quality data. Collecting data and handling data quality issues are of utmost importance. Without support from a system or framework, this can be very time-consuming and error-prone. This chapter provides a quick overview of the support provided by ScalaTion for data management and preprocessing. Data management capabilities are provided by ScalaTion’s Time Series DataBase (TSDB). Preprocessing of data should be done before applying analytics techniques to ensure they are working on quality data. ScalaTion provides a variety of preprocessing techniques.
3.1 Analytics Databases

It is convenient to collect data from multiple sources and store the data in a database. Analytics databases are organized to support efficient data analytics. Multiple systems, including SCALATion’s TSDB, are built on top of columnar, main memory databases in order to provide high performance. SCALATion’s TSDB is a Time Series DataBase that has built-in capabilities for handling time series data. It is able to store non-time series data as well. It provides three Application Programming Interfaces (APIs) for convenient access to the data [?].

3.1.1 Columnar Relational Algebra API

The first API is a Columnar Relational Algebra that includes the standard operators of relational algebra plus those common to column-oriented databases. It consists of the Table trait and two implementing classes: Relation and MM_Relation. Persistence for Relation is provided by the save method, while MM_Relation utilizes memory-mapped files.

Relation Creation

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

```scala
val sensor = Relation ("sensor", Seq ("sensorID", "model", "latitude", "longitude", "on"), Seq (), 0, "ISDDI")
val road = Relation ("road", Seq ("roadID", "rdName", "lat1", "long1", "lat2", "long2"), Seq (), 0, "ISDDDDD")
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.
Relation Population

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

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

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

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

Columnar Relational Algebra Operators

Table 3.1 shows the thirteen operators supported (the first six are considered fundamental). Operator names as well as Unicode symbols may be used interchangeably (e.g., `r union s` or `r ∪ s` compute the union of relations `r` and `s`). Note, the extended projection operator `eproject` provides a convenient mechanism for applying aggregate functions. It is often called after the `groupby` operator, in which case multiple rows will be returned. Multiple columns may be specified in `eproject` as well. There are also several varieties of `join` operators. As an alternative to using the Unicode symbol when they are Greek letters, the letter may be written out in English (`pi`, `sigma`, `rho`, `gamma`, `epi`, `omega`, `zeta`, `unzeta`).

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

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

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

```
t.γ (“sensorId”).Π (avg, “acount”, “count”) (“sensorId”)
```

In addition to the natural join shown in Table 3.1 the SCALATION TSDB also supports equi-join, general theta join, left outer join, and right outer join, as shown below.
\[ r \bowtie ("roadId", \"on\", s) \quad \text{equi-join} \]
\[ r \bowtie \{ \text{Int} \}(s, ("roadId", \"on\", \_ == \_)) \quad \text{theta join} \]
\[ t \times (\"time\", \"time\", w) \quad \text{left outer join} \]
\[ t \times (\"time\", \"time\", w) \quad \text{right outer join} \]

**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).

   ```scala
case class LatitudeLongitude(latitude: Double, longitude: Double)

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 \bowtie sensor.σ[Double] ("latitude", \_ ∈ alat).σ[Double] ("longitude", \_ ∈ along)
```

3.1.2 **SQL-Like API**

3.1.3 **Map-Reduce API**
3.2 Preprocessing

Using the SCALATION TSDB, data scientists may write queries that extract data from one or more columnar relations. These data are used to create vectors and matrices that may be passed to various analytics techniques. Before the vectors and matrices are created the data need to be preprocessed to improve data quality and transform the data into a form more suitable for analytics.

3.2.1 Remove Identifiers

Any column that is unique (e.g., a primary key) with arbitrary values should be removed before applying a modeling/analytics technique. For example, an employee ID in a Neural Network analysis to predict salary could result in a perfect fit. Upon knowing the employee ID, the salary is a known. As the ID itself (e.g., ID = 1234567) is arbitrary, such a model has little value.

3.2.2 Convert String Columns to Numeric Columns

In SCALATION, columns with strings (of type StrNum) should be converted to integers. For displaying final results, however, is often useful to convert the integers back to the original strings. The capabilities are provided by the mapToInt function in the scalation.linalgebra.Converter object.

3.2.3 Identify Missing Values

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

3.2.4 Detect Outliers

Data points that are considered outliers may happen because of errors or highly unusual occurrences. For example, suppose a dataset records the times for members of a football team to run a 100-yard dash and one of the recorded values is 3.2 seconds. This is an outlier. Some analytics techniques are less sensitive to outliers, e.g., \( L_1 \) Regression, while others, e.g., \( L_2 \) Regression, are more sensitive. Detection of outliers suffers from the obvious problems of being too strict (in which case good data may be thrown away) or too lenient (in which case outliers are passed to an analytics technique). One may choose to handle outliers separately, or turn them into missing values, so that both outliers and missing values may be handled together.

SCALATION currently provides the following techniques for outlier detection: so many standard deviation units from the mean, DistanceOutlier; the smallest and largest percent values, QuantileOutlier; and an expansion multiplier beyond the middle two quartiles, QuartileXOutlier. For example, the following function will turn outliers in missing values, by reassigning the outliers to noDouble, SCALATION’s indicator of a missing value of type Double.

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

The two main ways to handle missing values are (1) throw them away, or (2) use imputation to replace them with reasonable guesses. When there is a gap in time series data, imputation may be used for short gaps, but is unlikely to be useful for long gaps. This is especially true when imputation techniques are simple. The alternative could be to use an advanced modeling technique like SARIMA for imputation, but then results of a modeling study using SARIMA are likely to be biased.

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

3.2.6 Preliminary Feature Selection

Before selecting a modeling/analytics technique, certain columns may be thrown away. Examples include columns with too many missing values or columns with near zero variance.

3.2.7 Align Multiple Time Series

When the data include multiple time series, there are likely to be time alignment problems. The frequency and/or phase may not be in agreement. For example, traffic count data may be recorded every 15 minutes and phased on the hour, while weather precipitation data may be collected every 30 minutes and phased to 10 minutes past the hour.

ScalaTion supports the following alignments techniques: (1) approximate left outer join and (2) dynamic time warping. The first operator will perform a left outer join between two relations based on their time (\texttt{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.

$$\text{traffic} \bowtie_{(0.01)}(\text{“time”}, \text{“time”}, \text{weather})$$

$$\text{approximate left outer join}$$

The second operator ...

3.2.8 Creating Vectors and Matrices

Once the data have been preprocessed, columns may be projected out to create a matrix that may be passed to analytics/modeling techniques.

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

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

$$\text{val model1} = \text{Regression (mat)}$$
$$\text{val model2} = \text{ARIMA (mat)}$$
By default in SCALAION the rightmost columns are the response/output variables. As many of the modeling techniques have a single response variable, it will be assumed to in the last column. There are also constructors and factory apply functions that take explicit vector and matrix parameters, e.g., a matrix of predictor variables and a response vector.

### 3.3 Exercises

1. Load the auto_mpg.csv dataset into an auto_mpg relation. Perform the preprocessing steps above to create a cleaned-up relation auto_mpg2 and produce a data matrix called auto_mat from this relation. Print out the correlation matrix for auto_mat. Which columns have the highest correlation? To predict the miles per gallon mpg which columns are likely to be the best predictors.

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 $x$ such as club head speed, barometric pressure, and smash factor (how square the impact is). The models can be developed using a combination of data (e.g., from experiments) and knowledge (e.g., Newton’s Second Law). The modeling techniques discussed in this technical report tend to emphasize the use of data more than knowledge, while those in the simulation modeling technical report emphasize knowledge.

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

$$ y = f(x; t; b) + \epsilon $$

where

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

The parameters can be adjusted so that the predictive model matches available data. Note, in the definition of a function, the arguments appear before the “;”, while the parameters appear after. The input/predictor vector could be random as well, but could also be controlled.designed and thus treated as deterministic. The residuals/errors are typically additive as shown above, but may also be multiplicative. Of course, the formulation could be generalized by turning the output/response into a vector $y$ and the parameters into a matrix $B$.

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

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

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

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

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

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

See the section on Generalized Linear Models for further development along these lines.

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

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

The Predictor trait provides a common framework for several predictor classes such as SimpleRegression.

Trait Methods:

trait Predictor

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

4.1.1 Fit

The related Fit class provides a common framework for quality of fit measures.

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 (regression, error)

class Fit (y: VectoD, n: Int, val df: (Double, Double) = (0.0, 0.0))

def mse_ : Double = mse
def diagnose (e: VectoD, w: VectoD = null, yp: VectoD = null)
def fit: VectoD = VectorD (rSq, sst, sse, mse0, rmse, mae, 
  df._2, rBarSq, fStat, aic, bic)
def fitLabel: Seq [String] = Seq ("rSq", "sst", "sse", "mse0", "rmse", "mae", 
  "df", "rBarSq", "fStat", "aic", "bic")
def fitMap: Map [String, String] = 
def sumCoeff (b: VectoD, stdErr: VectoD = null): String =
def summary (b: VectoD, stdErr: VectoD = null): String =

For modeling, a user chooses one of the classes extending the trait Predictor (e.g., Regression) to instantiate an object. Next the train method would be typically called, followed by the eval method, which computes the residual/error vector and calls the diagnose method. Then the fitMap method would be called.
to return quality of fit statistics computed by the diagnose method. Some of the quality of fit measures are shown below, see the Fit class for details.

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

Note, ^ is the exponentiation operator provided in 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,

\[ sst = y \cdot y - \frac{1}{m} \left[ \sum y_i \right]^2 \]

while the sum of squares regression (ssr = sst − sse) measures the variability captured by the model, so the coefficient of determination measures the fraction of the variability captured by the model.

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

Values for $R^2$ would be nonnegative, unless the model is so bad that actually adds variability.

### 4.1.2 PredictorMat

Commonly, the Predictor trait and the Fit class are used to together in most models. The PredictorMat abstract class merges them.

```scala
abstract class PredictorMat (protected val x: MatrixD, protected val y: VectorD)
  extends Fit (y, x.dim2, (x.dim2-1, x.dim1-x.dim2)) with Predictor with Error
```

**Class Methods:**

- `@param x` the input/data m-by-n matrix
  
- `@param y` the response m-vector

**abstract class** PredictorMat (protected val x: MatrixD, protected val y: VectorD)

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

```scala
def train (yy: VectorD): PredictorMat
def train (): PredictorMat = train (y)
def eval ()
override def eval (xx: MatrixD, yy: VectorD)
def summary ()
def predict (z: VectorD): Double = b dot z
def predict (z: MatrixD): VectorD = z * b
```
def crossValidate (algor: (MatriD, VectoD) => PredictorMat, k: Int = 10, 
  rando: Boolean = true): Array [Statistic] =
def crossVal (k: Int = 10, rando: Boolean = true)

A few models use the PredictorVec abstract class rather than the PredictorMat class.
4.2 Null Model

The ModelNull class implements the simplest type of predictive modeling technique. If all else fails it may be reasonable to simply guess that \( y \) will take on its expected value or mean.

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

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

\[
y = b_0 + \epsilon
\]

The optimal solution for the parameter vector \( b \) is simple to compute. In SCALATiON it requires just one line of code inside the \texttt{train} method.

```scala
val b = VectorD(y.mean)
```

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

```scala
def eval (yy: VectoD = y)
{
    val e = yy.mean - b(0)
    diagnose (yy)
} // eval
```

The coefficient of determination \( R^2 \) for the null is 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 \texttt{predict} method is simply.

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

---

**Class Methods:**

@param y the response vector

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

```scala
def train (yy: VectoD = y): NullModel =
def eval ()
def predict (z: VectoD): Double = b(0)
def predict (z: MatriD): VectoD = { val yp = new VectorD (z.dim1); yp.set (b(0)); yp
```
4.2.1 Exercises

1. Let the response vector $y$ be

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

   and execute the `NullModel`.

   For context, assume the corresponding predictor vector $x$ is

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

   Draw an $xy$ plot of the data points. Give the value for the parameter vector $b$. Show the error distance for each point in the plot. Compare the sum of squares error $sse$ with the sum of squares total $sst$. What is the value for the coefficient of determination $R^2$?

2. Using `SCALATION`, analyze the `NullModel` for the following response vector $y$.

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

3. Execute the `NullModel` on the `Auto_MPG` dataset. See `apps.analytics.Auto_MPG_Regression`. What is the quality of the fit (e.g., $R^2$ or $rSq$)? Is this value expected? Is it possible for a model to perform worse than this?
4.3 Simpler Regression

The `SimplerRegression` class supports simpler linear regression. In this case, the predictor vector $x$ consists of a single variable $x_0$, i.e., $x = [x_0]$. The goal is to fit the parameter vector $b = [b_0]$ in the regression equation

$$ y = b_0 x_0 + \epsilon $$

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

A dataset or training set may be collected for providing an estimate for parameter $b_0$. Given $m$ data points, stored in an $m$-dimensional vector $x_0$ and $m$ response values, stored in an $m$-dimensional vector $y$, we may obtain the following vector equation.

$$ y = b_0 x_0 + \epsilon $$

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

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

Since $\epsilon = y - b_0 x_0$, we may solve the following optimization problem:

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

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

$$ (y - b_0 x_0) \cdot (y - b_0 x_0) $$

Taking the derivative $\frac{d}{db_0}$ using the derivative product rule

$$ (fg)' = f'g + fg' $$

and setting it equal to zero yields the following equation.

$$ -2x_0 \cdot (y - b_0 x_0) = 0 $$

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

$$ b_0 = \frac{x_0 \cdot y}{x_0 \cdot x_0} $$

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

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

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

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

Therefore, the error/residual vector is

$$ \epsilon = y - \hat{y} = [1,3,3,4] - [1.067, 2.133, 3.200, 4.267] = [-0.067, 0.867, -0.2, -0.267]. $$
Note, that this model has no intercept. This makes the solution for the parameter very easy, but may make the model less accurate. This is remedied in the next section. Since no intercept really means the intercept is zero, the regression line will go through the origin.

Class Methods:

@param x the input/data matrix
@param y the response vector

class SimplerRegression (x: MatriD, y: VectoD)
    extends PredictorMat (x, y)

def train (yy: VectoD = y): SimplerRegression =
def crossVal (k: Int = 10, rando: Boolean = true)

4.3.1 Exercises

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

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

object SimplerRegression_exer_1 extends App {
    val x0 = VectorD (1, 2, 3, 4)
    val y = VectorD (1, 3, 3, 4)
    val b0 = VectorD.range (0, 50) / 25.0
    val sse = new VectorD (b0.dim)
    for (i <- b0.range) {
        val e = ?
        sse(i) = e dot e
    } // for
    new Plot (b0, sse, lines = true)
} // SimplerRegression_exer_1 object
```

Where do you think the minimum occurs?

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

49
2. From the $X$ matrix and $y$ vector, plot the set of data points $\{(x_i, y_i) | 0 \leq i < m\}$ and draw the line that is nearest to these points. What is the slope of this line? Pass the $X$ matrix and $y$ vector as arguments to the `SimplerRegression` class to obtain the $b = [b_0]$ vector.

```scala
// 4 data points: x0
val x = new MatrixD((4, 1), 1.0, // x 4-by-1 matrix
                    2.0,
                    3.0,
                    4.0)
val y = VectorD(1.0, 3.0, 3.0, 4.0) // y vector
val rg = new SimplerRegression(x, y) // create a SimplerRegression
rg.train().eval() // train on data and evaluate
println("coefficient = " + rg.coefficient) // parameter values
println("fitMap = " + rg.fitMap) // quality of fit
```

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

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

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

```scala
// 5 data points: x0
val x = new MatrixD((5, 1), 0.0, // x 5-by-1 matrix
                    1.0,
                    2.0,
                    3.0,
                    4.0)
val y = VectorD(2.0, 3.0, 5.0, 4.0, 6.0) // y vector
val rg = new SimplerRegression(x, y) // create a SimplerRegression
rg.train().eval() // train on data and evaluate
println("coefficient = " + rg.coefficient) // parameter values
println("fitMap = " + rg.fitMap) // quality of fit
val z = VectorD(5.0) // predict y for one point
val yp = rg.predict(z) // y-predicted
println(s"predict ($z) = $yp")
```
5. Execute the `SimplerRegression` on the `Auto_MPG` dataset. See `apps.analytics.Auto_MPG_Regression`. 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.
4.4 Simple Regression

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

\[
-2x \cdot (y - (b_0 1 + b_1 x)) = 0
\]

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

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

Multiplying by \( m \) and substituting for \( mb_0 = 1 \cdot y - b_1 1 \cdot x \) yields
\[
\begin{align*}
\mathbf{1} \cdot y - b_1 \mathbf{1} \cdot \mathbf{x} \mathbf{1} \cdot \mathbf{x} + mb_1 \mathbf{x} \cdot \mathbf{x} &= m\mathbf{x} \cdot \mathbf{y} \\
\mathbf{b}_1[m\mathbf{x} \cdot \mathbf{x} - (\mathbf{1} \cdot \mathbf{x})^2] &= m\mathbf{x} \cdot \mathbf{y} - (\mathbf{1} \cdot \mathbf{x})(\mathbf{1} \cdot \mathbf{y})
\end{align*}
\]

Solving for \( b_1 \) gives

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

The \( b_0 \) parameter gives the intercept, while the \( b_1 \) parameter gives the slope of the line that best fits the data points. Consider again the problem from the last section where the data points are \{(1, 1), (2, 3), (3, 3), (3, 4)\} and solve for the two parameters, (intercept) \( b_0 \) and (slope) \( b_1 \).

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

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

### Class Methods:

@param \( \mathbf{x} \) the input/data matrix augmented with a first column of ones
@param \( \mathbf{y} \) the response vector

class SimpleRegression (x: MatrixD, y: VectorD)
  extends PredictorMat (x, y)

  def train (yy: VectorD = y): SimpleRegression =
  def crossVal (k: Int = 10, rando: Boolean = true)

4.4.1 Exercises

1. Using the definition for mean from section 2.2.1 for \( \mu_x \) and \( \mu_y \), show that the expression for \( b_0 \) shortens to

\[
b_0 = \mu_y - b_1 \mu_x
\]

Using the definitions for covariance \( \sigma_{x,y} \) and variance \( \sigma_x^2 \) from section 2.2.1, show that the expression for \( b_1 \) shortens to

\[
b_1 = \frac{\sigma_{x,y}}{\sigma_x^2}
\]

2. From the \( \mathbf{X} \) matrix and \( \mathbf{y} \) vector, plot the set of data points \{ \( (x_i, y_i) \mid 0 \leq i < m \) \} and draw the line that is nearest to these points. What are the intercept and slope of this line. Pass the \( \mathbf{X} \) 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), 1.0, 1.0, // x 4-by-2 matrix
                  1.0, 2.0,
                  1.0, 3.0,
                  1.0, 4.0)
val y = VectorD (1.0, 3.0, 3.0, 4.0) // y vector
val rg = new SimpleRegression (x, y) // create a SimpleRegression
rg.train ().eval () // train on data and evaluate
println ("coefficient = " + rg.coefficient) // parameter values
println ("fitMap = " + rg.Map) // quality of fit

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

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

4. Execute the SimplerRegression on the Auto MPG dataset. See apps.analytics.Auto MPG.Regression. What is the quality of the fit (e.g., \(R^2\) or rSq)? Is this value expected? Try using different columns for the predictor variable.
4.5 Regression

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

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

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

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

\[
\mathbf{y} = \mathbf{Xb} + \epsilon
\]

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

\[
\min_{\mathbf{b}} \| \epsilon \|
\]

Substituting \( \epsilon = \mathbf{y} - \mathbf{Xb} \) yields

\[
\min_{\mathbf{b}} \| \mathbf{y} - \mathbf{Xb} \|
\]

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

\[
(\mathbf{y} - \mathbf{Xb}) \cdot (\mathbf{y} - \mathbf{Xb})
\]

\[
(\mathbf{y} - \mathbf{Xb})^t(\mathbf{y} - \mathbf{Xb})
\]

Taking the gradient with respect to the parameter vector \( \mathbf{b} \) and setting it equal to the zero vector yields

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

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

A more detailed derivation of this equation is given in section 3.4 of “Matrix Calculus: Derivation and Simple Application” [7]. Dividing the equation by 2 and moving the term involving \( \mathbf{b} \) to the left side, results in the Normal equations.

\[
\mathbf{X}^t\mathbf{Xb} = \mathbf{X}^t\mathbf{y}
\]

Note: equivalent to minimizing the distance between \( \mathbf{y} \) and \( \mathbf{Xb} \) 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.
4.5.1 Matrix Inversion Technique

Starting with the Normal Equations

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

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

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

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

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

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

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

The pseudo-inverse can be computed by first multiplying \( X \) by its transpose. Gaussian Elimination can be used to compute the inverse of this, which can be then multiplied by the transpose of \( X \). In SCALATION, the computation for the pseudo-inverse (\texttt{x.pinv}) looks similar to the math.

\[
\text{val x.pinv = (x.t * x).inverse * x.t}
\]

A more robust approach is

\[
\text{val fac = new Fac.Inv (x.t * x).factor ()}
\text{val x.pinv = fac.factors._2 * x.t}
\]

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

\[
\text{val b = fac.solve (x.t * y)}
\]

4.5.2 LU Factorization Technique

Lower, Upper Decomposition works like Matrix Inversion, except that is just reduces the matrix to zeroes below the diagonal, so it tends to be faster and less prone to numerical instability. First the product \( X^t X \), an \( n \)-by-\( n \) matrix, is factored

\[ X^t X = LU \]

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

\[ LU b = X^t y \]

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

\[ Lw = X^t y \]

Finally, the parameter vector \( b \) is determined by backward substitution.

\[ Ub = w \]
4.5.3 Cholesky Factorization Technique

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

$$X^tX = LL^t$$

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

$$LL^tb = X^ty$$

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

$$Lw = X^ty$$

and then solve for $b$ using backward substitution.

$$L^tb = w$$

4.5.4 QR Factorization Technique

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

$$X = QR$$

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

$$X^tXb = X^ty$$

simply substitute $QR$ for $X$.

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

Taking the transpose gives

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

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

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

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

$$Rb = Q^ty$$

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

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

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

57
4.5.5 Singular Value Decomposition Technique

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

$$ y = Xb $$

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

$$ X = UΣV^t $$

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

$$ y = UΣV^t b $$

Defining $d = ΣV^t b$, we may write

$$ y = Ud $$

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

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

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

$$ d = U^t y $$

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

$$ b = VΣ^{-1}d $$

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

4.5.6 Use of Factorization in Regression

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

import RegTechnique._

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

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

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

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

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

After training, the eval method inherited from PredictorMat does two things: First, the residual/error
vector ϵ is computed. Second, several quality of fit measures are computed by calling the diagnose method.

def eval ()
{
    e = y - x * b // compute residual/error vector e
    diagnose (e) // compute diagnostics
}

} // RegTechnique
4.5.7 Model Assessment

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

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

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

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

This ratio can be used to adjust the Coefficient of Determination $R^2$ to reduce it with increasing number of parameters. This is called the Adjusted Coefficient of Determination $\hat{R}^2$

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

Dividing $sse$ and $ssr$ by their respective degrees of freedom gives

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

The ratio $\frac{msr}{mse}$ can be shown to follow the $F$-distribution. More general quality of fit measures useful for comparing models is the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC). In SCALATION the following quantities are computed.

```scala
df_r = k // degrees of freedom for regression model
df = m - n // degrees of freedom left for error
r_df = (m - 1) / df // ratio of degrees of freedom
rBarSq = 1 - (1-rSq) * r_df // R-bar-squared (adjusted R-squared)
mse_ = sse / df // mean square error
msr_ = ssr / df_r // mean square regression
fStat = msr_ / mse_ // F statistic
aic = m * log (sse) - m * log (m) + 2 * n // Akaike Information Criterion (AIC)
bic = aic + (k+1) * (log (m) - 2) // Bayesian Information Criterion (BIC)
```

4.5.8 Model Validation

Data is 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 a 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 is available for training. Also, picking a different half may result in substantially different quality measures.

This leads to two guiding principles: First, the majority of the data should be used for training. Second, multiple testing should be done. In general, conducting real-world tests of a model can be difficult. There are, however, strategies that attempt to approximate such testing. Two simple and commonly used strategies are the following: Leave-One-Out and Cross-Validation. In both cases, a dataset is divided into a training dataset and a testing dataset.

**Leave-One-Out**

When fitting the parameters $b$ the more data available in the training set, in all likelihood, the better the fit. The Leave-One-Out strategy takes this to the extreme, by splitting the dataset into a training set of size $m - 1$ and test set of size 1 (e.g., row $t$ in data matrix $X$). From this, a test error can be computed $y_t - b \cdot x_t$. This can be repeated by iteratively letting $t$ range from the first to the last row of data matrix $X$. For certain predictive analytics techniques such as Multiple Linear Regression, there are efficient ways to compute the test $sse$ based on the leverage each point in the training set has. [8].

**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.

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

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.
4.5.9 Collinearity

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

$$y = Xb + \epsilon$$

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

$$y = b_0 1 + b_1 x_1 + \ldots b_k x_k + \epsilon$$

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.

```scala
println("corr (x) = " + corr (x))
```

The `corr` function is provided by the `scalation.stat.StatVector` object.

Even if no particular entry in the correlation matrix is high, a column in the matrix may still be nearly a linear combination of other columns. This is the problem of multi-collinearity. This can be checked by computing the Variance Inflation Factor function `vif`. For a particular parameter $b_j$ for the variable/predictor $x_j$, the function is evaluated as follows:

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

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

Both `corr` and `vif` may be tested in SCALATION using `RegressionTest4`.

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

4.5.10 Feature Selection

There may be predictor variables (features) in the model that contribute little in terms of their contributions to the model’s ability to make predictions. The improvement to $R^2$ may be small and may make $\bar{R}^2$ or other quality of fit measures worse. 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.

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

```scala
def forwardSel (cols: Set[Int]): (Int, VectoD, VectoD)
```

To start with a minimal model, set `cols = Set (0)` for an intercept-only model. Then iteratively add the variable suggested in the method’s return triple.

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.
def backwardElim (cols: Set[Int]): (Int, VectoD, VectoD)

To start with a maximal model, set cols = Set (0, 1, ..., k) for a full model. Then iteratively remove
the variable suggested in the method's return triple.

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

Class Methods:

@param x  the input/data m-by-n matrix
  (augment with a first column of ones to include intercept in model)
@param y  the response m-vector
@param technique  the technique used to solve for b in x.t * x * b = x.t * y

class Regression (x: MatriD, y: VectoD, technique: RegTechnique = QR)
  extends PredictorMat (x, y)

def train (yy: VectoD = y): Regression =
def forwardSel (cols: Set[Int]): (Int, VectoD, VectoD) =
def backwardElim (cols: Set[Int]): (Int, VectoD, VectoD) =
def vif: VectoD =
def crossVal (k: Int = 10, rando: Boolean = true)

4.5.11 Exercises

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

   \[ Ab = z \]

2. Solving a regression problem in SCALATION simply involves creating a data/design matrix X and
   response vector y and then creating a Regression object upon which train, eval, coefficient and
   fitMap 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

63
More details about the coefficients including standard error, \( t \)-values and \( p \)-values are shown by the \texttt{summary} method.

\begin{verbatim}
println (rg.summary ())
\end{verbatim}

Finally, a given new data vector \( \mathbf{z} \), the \texttt{predict} method may be used to predict its response value.

\begin{verbatim}
val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println (s"predict ($z) = \${rg.predict (z)}")
\end{verbatim}

Feature selection may be carried out by using either \texttt{forwardSel} or \texttt{backwardElim}.

\begin{verbatim}
println ("reduced mod fit = " + rg.backwardElim ())
\end{verbatim}

The source code for this example is at
\begin{verbatim}
\end{verbatim}

3. Use Regression analysis on the AutoMPG dataset. This is a well know dataset that is available at multiple websites including the UCI Machine Learning Repository\begin{verbatim}
http://archive.ics.uci.edu/ml/datasets/Auto+MPG
\end{verbatim} The response variable is the miles per gallon (\texttt{mpg}: continuous) while the predictor variables are \texttt{cylinders}: multi-valued discrete, \texttt{displacement}: continuous, \texttt{horsepower}: continuous, \texttt{weight}: continuous, \texttt{acceleration}: continuous, \texttt{model\_year}: multi-valued discrete, \texttt{origin}: multi-valued discrete, and \texttt{car\_name}: string (unique for each instance). Since the \texttt{car\_name} is unique and obviously not causal, this variable is eliminated, leaving seven predictor variables.

The source code for this example is at
\begin{verbatim}
\end{verbatim}
4. Examine the collinearity of the column vectors in the AutoMPG dataset.

5. 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 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?

6. Compare model assessment and model validation. Compute $sse$, $mse$ and $R^2$ for the full and best AutoMPG models trained on the entire data set. Compare this with the results of Leave-One-Out, 5-fold Cross-Validation and 10-fold Cross-Validation.

### 4.5.12 Further Reading

1. Introduction to Linear Regression Analysis, 5th Edition [9]
4.6 Weighted Least Squares Regression

The Regression\textsubscript{WLS} class supports weighted multiple linear regression. In this case, the predictor vector \( \mathbf{x} \) is multi-dimensional \([1, x_1, \ldots, x_k]\). As before the regression equation is

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

where \( \epsilon \) 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.

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

Let us look at the error vector \( \epsilon = y - X\mathbf{b} \) 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 is done by performing unweighted regression of \( y \) onto \( X \) to obtain the error vector \( \epsilon \). It is used to compute a root absolute deviation vector \( \mathbf{r} \).

\[
\mathbf{r} = \sqrt{|\epsilon|}
\]

For a second time unweighted regression is performed regressing \( \mathbf{r} \) onto \( X \) to obtain the predictions \( \hat{\mathbf{r}} \).

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

These weights can be used to build a diagonal weight matrix \( W \) that factors into the Normal Equations

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

In ScalaTion, this is accomplished by computing a weight vector \( \mathbf{w} \) and taking its square root \( \omega = \sqrt{\mathbf{w}} \). The data matrix \( X \) is then reweighted by premultiplying it by \( \omega \) (\( rtW \) in the code), as if it is a diagonal matrix \( rtW \star: x \). The response vector \( y \) is reweighted using vector multiplication \( rtW \star y \). 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](http://en.wikipedia.org/wiki/Least_squares#Weighted_least_squares).

Class Methods:

```scala
class Regression_WLS [MatT <: MatriD, VecT <: VectoD] (xx: MatT, yy: VecT,
  technique: RegTechnique = QR, private var w: VectoD = null)
  extends Regression ({ setWeights (xx, yy, technique, w); reweightX (xx, w) },
  reweightY (yy, w), technique)
```

66
4.6.1 Exercises

1. Prove that reweighting the data matrix $X$ and the response vector $y$ and solving for the parameter vector $b$ in the standard Normal Equations $X^T X b = X^T y$ gives the same result as not reweighting and solving for the parameter vector $b$ in the Weighted Normal Equations $X^T W X b = X^T W y$.

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

3. 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?
4.7 Ridge Regression

The **RidgeRegression** class supports multiple linear ridge regression. In this case, \( x \) is multi-dimensional \([x_1, \ldots, x_k]\). Ridge regression adds a penalty based on the \( \ell_2 \) norm of the parameters \( b \) to reduce the chance of them taking on large values that may lead to less robust models. For better results, both the input data matrix \( X \) and the response vector \( y \) are centered (zero mean). The combined centering takes care of the intercept, so it is not included in the model. Thus, the goal is to fit the parameter vector \( b \) in the regression equation

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

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

\[
f_{obj} = sse + \lambda \| b \|^2 = e \cdot e + \lambda b \cdot b
\]

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

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

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

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

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

\[(X^t X + \lambda I)b = X^t y
\]

Matrix factorization may now be used to solve for the parameters \( b \) in the modified Normal Equations. The value for \( \lambda \) can be user specified (typically a small value) or chosen by a procedure like Generalized Cross-Validation (GCV).

**Class Methods:**

@param x the centered input/data m-by-n matrix NOT augmented with a column of ones
@param y the centered response m-vector
@param l_ the shrinkage (lambda) parameter (0 => OLS) in the penalty term ‘lambda * b dot b’
@param technique the technique used to solve for b in (x.t*x + lambda*I)*b = x.t*y

class RidgeRegression [MatT <: MatriD, VecT <: VectoD] (x: MatT, y: VecT, l_: Double = 0.1, technique: RegTechnique = Cholesky)
extends Predictor with Error

def xtx_l_I (l_: Double)
def train (yy: VectoD): RidgeRegression [MatT, VecT] =
4.7.1 Exercises

1. Why is it important to center (zero mean) both the data matrix $X$ and the response vector $y$?

2. Compare the results produced when matrix $x$ and vector $y$ are passed directly to `RidgeRegression` against the results when they are centered as below.

```scala
// 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)

// Compute centered (zero mean) versions of x, y and z
val mu_x = x.mean // columnwise mean of x
val mu_y = y.mean // mean of y
val x_c = center (x, mu_x) // centered x (columnwise)
val y_c = y - mu_y // centered y

val rrg = new RidgeRegression (x_c, y_c)
rrg.train ().eval ()
println ("coefficient = " + rrg.coefficient)
println ("fitMap = " + rrg.fitMap)
```

3. What is scale invariance and how does it relate to standardizing the data?
4.8 Lasso Regression

The LassoRegression class supports multiple linear regression using the Least absolute shrinkage and selection operator (Lasso) that constrains the values of the b parameters and effectively sets those with low impact to zero (thereby deselecting such variables/features).

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

$$f_{obj} = \frac{1}{2} sse + \lambda \|b\|_1 = \frac{1}{2} \|\epsilon\|^2 + \lambda \|b\|_1$$

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

$$f_{obj} = \frac{1}{2} \|y - Xb\|^2 + \lambda \|b\|_1$$

This is similar to the $\ell_2$ penalty in Ridge Regression. The $\ell_1$ penalty for Lasso has a disadvantage that the absolute values in the $\ell_1$ norm make the objective function non-differentiable. Therefore the straightforward strategy of setting the gradient equal to zero to develop appropriate modified Normal Equations that allow the parameters to be determined by matrix factorization will no longer work. Instead, the objective function needs to be minimized using a search based optimization algorithm.

ScalaTion uses the Alternative Direction Method of Multipliers (ADMM) [2] algorithm to optimize the $b$ parameter vector. The algorithm for using ADMM for Lasso Regression is outlined in section 6.4 of [2]. We follow their development closely, but change to the notation to that used herein. Optimization problems in ADMM form separate the objective function into two parts $f$ and $g$.

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

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

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

Therefore, the iterative step in the ADMM algorithm becomes

$$b = (X'X + \rho I)^{-1}(X'y + \rho(z - u))$$
$$z = S_{\lambda/\rho}(b + u)$$
$$u = u + b - z$$

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

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

Class Methods:

@param x the input/data m-by-n matrix
@param y the response vector
@param l_ the initial value for the regularization weight parameter (lambda)
class LassoRegression [MatT <: MatrixD, VecT <: VectorD] (x: MatT, y: VecT, 
    l_: Double = 0.01) 
  extends Predictor with Error

def f (yy: VectorD)(b: VectorD): Double =
def train (yy: VectorD): LassoRegression [MatT, VecT] =
def train (): LassoRegression [MatT, VecT] = train (y)
def eval (yy: VectorD = y)
override protected def diagnose (yy: VectorD)
override def fit: VectorD = super.fit.asInstanceOf [VectorD] ++ VectorD (rBarSq, fStat, aic, bic)
override def fitLabels: Seq [String] = super.fitLabels ++ Seq ("rBarSq", "fStat", "aic", "bic")
def predict (z: VectorD): Double = b dot z
def predict (z: MatrixT): VectorD = z * b
def report ()

4.8.1 Exercises

1. Compare LassoRegression the with Regression that uses backward selection/elimination for feature selection. What are the advantages and disadvantages of each for feature selection.

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

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

4.8.2 Further Reading

1. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers [2]

4.9 Transformed Regression

The **TranRegression** class supports transformed multiple linear regression. In this case, the predictor vector \( \mathbf{x} \) is multi-dimensional \([1, x_1, ... x_k]\). In many cases, the relationship between the response scalar \( y \) and the predictor vector \( \mathbf{x} \) is not linear. There are many possible functional relationships that could apply, but four obvious choices are the following:

1. The response grows exponentially versus a linear combination of the predictor variable.
2. The response grows quadratically versus a linear combination of the predictor variable.
3. The response grows as the square root of a linear combination of the predictor variable.
4. The response grows logarithmically versus a linear combination of the predictor variable.

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

\[(\log, \exp), (\sqrt{x}, ^2), (^2, \sqrt{x}), (\exp, \log)\]

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

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

where \( \epsilon \) represents the residuals (the part not explained by the model) and \( \text{transform} \) is the function (defaults to log) used to transform the response \( y \).

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** \((x, y, \text{map (transform)}, \text{technique})\)

The inverse transform is then applied in the **predict** method.

**override def predict** \((z: \text{VectoD}): \text{Double} = \text{transInv (b dot z)}\)

**Class Methods:**

- **@param x** the data/design matrix
- **@param y** the response vector
- **@param transform** the transformation function (defaults to log)
- **@param transInv** the inverse transformation function to rescale predictions to original y scale (defaults to exp)
- **@param technique** the technique used to solve for \( \mathbf{b} \) in \( \mathbf{x}^T \mathbf{x} \mathbf{b} = \mathbf{x}^T \mathbf{y} \)

**class TranRegression** \([\text{MatT} <: \text{MatriD}, \text{VecT} <: \text{VectoD}] (x: \text{MatT}, y: \text{VecT},
transform: \text{FunctionS2S} = \log, \text{transInv: FunctionS2S} = \exp,
technique: \text{RegTechnique} = \text{QR})\)
extends Regression (x, y.map (transform), technique)

override def eval (yy: VectoD = y) =
override def predict (z: VectoD): Double = transInv (b dot z)
override def predict (z: MatT): VectoD = (z * b).map (transInv)

4.9.1 Exercises

1. Compare the quality of fit resulting from Regression versus TransRegression.

```scala
val cap = 40
val cap_rng = 0 until cap
val (m, n) = (cap * cap, 3)
val std = 10.0
val err = Normal (0.0, std)
val x = new MatrixD (m, n)
val y = new VectorD (m)
for (i <- cap_rng; j <- cap_rng) x(cap * i + j) = VectorD (1.0, 2 * i, 3 * j)
for (k <- x.range1) y(k) = abs (std + sq (2 * x(k, 0) + 3 * x(k, 1)) + err.gen)
val rg = new Regression (x, y)
rg.train ().eval ()
val trg = new TranRegression (x, y, sqrt _, sq _)
trg.train ().eval ()
println ("coefficient = " + rg.coefficient)
println ("rg.fitMap = " + rg.fitMap)
println ("trg.fitMap = " + trg.fitMap)
```

2. Consider the following family of transformation functions.

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

where \(\lambda\) determines the power function on \(y\), e.g., 0.5 for \(\sqrt{x}\) and 2.0 for \(x^2\). What is the inverse function? Try various Box-Cox transformations (values for \(\lambda\) for the above problem.

```scala
TranRegression (x, y, lambda)
```
4.10 Polynomial Regression

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

\[
y = b \cdot x + \epsilon = b_0 + b_1 t + b_2 t^2 + \ldots + b_k t^k + \epsilon
\]

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

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

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

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

\[
\text{val } x = \text{new } \text{MatrixD}(t.\text{dim}, 1 + k) \\
\text{for } (i <- t.\text{range}) x(i) = \text{expand}(t(i)) \\
\text{val } rg = \text{new } \text{Regression}(x, y, \text{technique})
\]

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

Class Methods:

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

class PolyRegression (t: VectoD, y: VectoD, k: Int, technique: RegTechnique = Cholesky)  
  raw: Boolean = true,  
  extends Predictor with Error

\[
\text{def } \text{expand}(t: \text{Double}): \text{VectoD} = \\
\text{def } \text{orthogonalize}(x: \text{MatrixD}): (\text{MatrixD}, \text{MatrixD}) = \\
\text{def } \text{orthoVector}(v: \text{VectoD}): \text{VectoD} = \\
\text{def } \text{train}(yy: \text{VectoD}): \text{Regression} [\text{MatrixD}, \text{VectoD}] = \text{rg}.\text{train}(yy) \\
\text{def } \text{train}(): \text{Regression} [\text{MatrixD}, \text{VectoD}] = \text{rg}.\text{train}() \\
\text{def } \text{eval}(yy: \text{VectoD} = y) \{ \text{rg}.\text{eval}(yy) \}
\]
override def coefficient: VectoD = rg.coefficient
override def residual: VectoD = rg.residual
override def fit: VectoD = rg.fit
override def fitLabels: Seq [String] = rg.fitLabels
def predict (z: Double): Double =
def predict (z: VectoD): Double = rg.predict (z)
def forwardSel (cols: Set [Int]): (Int, VectoD, VectoD) = rg.forwardSel (cols)
def backwardElim (cols: Set [Int]): (Int, VectoD, VectoD) = rg.backwardElim (cols)
def vif: VectoD = rg.vif
def corrMatrix: MatriD = corr (x)

4.10.1 Exercises

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

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

Test `new PolyRegression (t, y, 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.
4.11 Trigonometric Regression

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

A periodic function can be expressed as linear combination of trigonometric functions (sine and cosine functions) of increasing frequencies. Consequently, if the data points have a periodic nature, a trigonometric regression model may be superior to alternatives. The goal is to fit the parameter vector \( b \) in the regression equation

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

where \( \omega \) is the base angular displacement in radians (e.g., \( \pi \)) and \( \epsilon \) represents the residuals (the part not explained by the model).

A training set now consists of two vectors, one for the \( m \)-vector \( t \) 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/design matrix \( X \) and pass it to the Regression class (multiple linear regression). The columns of data matrix \( X \) represent sines and cosines at at multiple harmonic frequencies of the vector \( t \).

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

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

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

Class Methods:

```scala
class TrigRegression (t: VectoD, y: VectoD, k: Int, technique: RegTechnique = QR) extends Predictor with Error
    def expand (t: Double): VectoD =
    def train (): Regression [MatrixD, VectoD] = rg.train ()
    def eval (yy: VectoD = y)
    override def coefficient: VectoD = rg.coefficient
    override def residual: VectoD = rg.residual
    override def fit: VectoD = rg.fit
    override def fitLabels: Seq [String] = rg.fitLabels
```

76
def predict (z: Double): Double = rg.predict (expand (z))
def predict (z: VectoD): Double = rg.predict (z)
def forwardSel (cols: Set [Int]): (Int, VectoD, VectoD) = rg.forwardSel (cols)
def backwardElim (cols: Set [Int]): (Int, VectoD, VectoD) = rg.backwardElim (cols)
def vif: VectoD = rg.vif

### 4.11.1 Exercises

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

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

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

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

4. How does Trigonometric Regression relate to Fourier Series?
4.12 Response Surface Regression

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

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

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

where \( x' = [1, x_1, x_1^2, x_1 x_2, x_2, x_2^2] \) and \( \epsilon \) represents the residuals (the part not explained by the model). The number of terms (nt) in the model increases quadratically with the dimensionality of the space (n) according to the formula for triangular numbers shifted by (n → n + 1).

\[
nt = \frac{(n+1)(n+2)}{2} = 6 \text{ for } n = 2
\]

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

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

where \( x' = [1, x_1, x_1^2, x_1 x_2, x_1^2 x_2, x_1 x_2^2, x_2, x_2^2, x_2^3] \) and \( \epsilon \) represents the residuals (the part not explained by the model). Naturally, the number of terms in the model increases cubically with the dimensionality of the space (n) according to the formula for tetrahedral numbers shifted by (n → n + 1).

\[
nt = \frac{(n+1)(n+2)(n+3)}{6} = 10 \text{ for } n = 2
\]

Class Methods:

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

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

override def predict (z: VectoD): Double =

def numTerms (n: Int, cubic: Boolean) =
def allForms (x: MatriD, cubic: Boolean): MatriD =
def qForms (p: VectoD, nt: Int, n: Int): VectoD =
def cForms (p: VectoD, nt: Int, n: Int): VectoD =
4.12.1 Exercises

1. How does Response Surface Regression related to Response Surface Methodology?

2. Perform Response Surface Regression on the following data

```scala
//
val x = new MatrixD((20, 2),
  47.0, 85.4,
  49.0, 94.2,
  49.0, 95.3,
  50.0, 94.7,
  51.0, 89.4,
  48.0, 99.5,
  49.0, 99.8,
  47.0, 90.9,
  49.0, 89.2,
  48.0, 92.7,
  47.0, 94.4,
  49.0, 94.1,
  50.0, 91.6,
  45.0, 87.1,
  52.0, 101.3,
  46.0, 94.5,
  46.0, 87.0,
  46.0, 94.5,
  48.0, 90.5,
  56.0, 95.7)
// response BP
val y = VectorD(105.0, 115.0, 116.0, 117.0, 112.0, 121.0, 121.0, 110.0, 110.0, 114.0,
  114.0, 115.0, 114.0, 106.0, 125.0, 114.0, 106.0, 113.0, 110.0, 122.0)
```

3. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response.
4.13 ANOVA

An ANalysis Of VAriance (ANOVA) model may be developed using the ANOVA1 class. This type of model comes into play when input variables are binary or categorical. 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 level of fertilizer applied to a field.

In the categorical case, the number of dummy variables required is one less than the number of levels. If the treatment levels for variable \( t \) are \( \{1, 2, \ldots, l_{\text{max}}\} \), then the number of dummy variables is \( km = l_{\text{max}} - 1 \), so for \( (k \leftarrow 1 \text{ to } km) \) the \( k^{th} \) dummy variable is given by

\[
d_k = 1 \text{ if } k = t \\
0 \text{ otherwise}
\]

(4.2)

The ANOVA1 class in SCALATION only supports one categorical variable, so in general, \( x \) consists of \( km \) dummy variables \( d_k \) for \( k \in \{1, km\} \)

\[
y = b \cdot x + \epsilon = b_0 + b_1 d_1 + \ldots b_{km} d_{km} + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). The dummy variables are binary and are used to determine the level of a categorical variable. See [http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf](http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf) For example, dummy variables \( d_1, d_2 \) and \( d_3 \) can be used to specify four treatment levels, e.g.,

\[
(1, 0, 0) \implies \text{None}; (0, 1, 0) \implies \text{Low}; (0, 0, 1) \implies \text{Medium}; \text{ and } (0, 0, 0) \implies \text{High}.
\]

This encoding scheme works in the binary case as well. A single binary dummy variable \( d_1 \in \{0, 1\} \) can be used to distinguish the two cases, e.g., \( d_1 = 1 \implies \text{Off}; \text{ and } d_1 = 0 \implies \text{On} \).

In SCALATION, the ANOVA1 class is implemented using regular multiple linear regression. An data/design matrix \( X \) is build from columns corresponding to levels for the treatment vector \( t \). As with multiple linear regression, the \( y \) vector holds the response values. Multi-way Analysis of Variance is under development in the ANOVA class. Also, a more traditional implementation Anova, not following the GLM approach, is provided in the stat package.

Class Methods:

```
@param t the binary/categorical treatment variable vector
@param y the response vector
@param levels the number of treatment levels (1, ... levels)
@param technique the technique used to solve for \( b \) in \( x.t \cdot x \cdot b = x.t \cdot y \)

class ANOVA (t: VectorI, y: VectorD, levels: Int, technique: RegTechnique = QR)
    extends Predictor with Error

def assignDummyVar (lev: Int): VectorD =

def assignDummyVars (tt: VectorI = t)
```
def train (): Regression [MatrixD, VectoD] = rg.train ()
def eval (yy: VectoD = y) { rg.eval (yy) }
override def residual: VectoD = rg.residual
override def fit: VectoD = rg.fit
override def fitLabels: Seq [String] = rg.fitLabels
def predict (z: Int): Double = rg.predict (assignDummyVar (z))
def predict (z: VectoD): Double = rg.predict (z)
def forwardSel (cols: Set [Int]): (Int, VectoD, VectoD) = rg.forwardSel (cols)
def backwardElim (cols: Set [Int]): (Int, VectoD, VectoD) = rg.backwardElim (cols)
def vif: VectoD = rg.vif

4.13.1 Exercises

1. 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.

```scala
val t = VectorI (1, 1, 1, 2, 2, 2, 3, 3, 3) // treatment level data
val y = VectorD (755.0, 865.0, 815.0,
                    442.0, 420.0, 401.0,
                    282.0, 250.0, 227.0)

val levels = 3
val arg = new ANOVA1 (t, y, levels)
arg.train ().eval ()
println ("coefficient = " + arg.coefficient)
println ("fitMap = " + arg.fitMap)

banner ("test predictions")
val yp = new VectorD (y.dim)
for (i <- yp.range) yp(i) = arg.predict (t(i))
println (s" y = $y \n yp = $yp")
new Plot (t.toDouble, y, yp, "ANOVA1")
```
4.14 ANCOVA

An ANalysis of COVAriance (ANCOVA) model may be developed using the ANCOVA class. This type of model comes into play when input variables are mixed, i.e., some are continuous, while others are binary or categorical. In the binary case, a single dummy zero-one variable $d_1$ can be used to distinguish the two cases. In the categorical case, the number of dummy variables required is one less than the number of levels ($l + 1$). ScalaTion currently only supports one binary/categorical variable, so in general, $x$ consists of the usual $k$ continuous variables $x_j$, plus $l$ dummy variables $d_j$.

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

The dummy variables are binary and are used to determine the level of a binary or categorical variable. See [http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx](http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx). For example, dummy variables $d_1$ and $d_2$ can be used to specify three levels, e.g.,

$$(1, 0) \implies \text{Assistant}; (0, 1) \implies \text{Associate}; (0, 0) \implies \text{Full}.$$ 

Note, when $l = 0$, the model becomes multiple linear regression, while when $k = 0$, it becomes one-way analysis of variance.

In ScalaTion, ANCOVA is implemented using regular multiple linear regression. An augmented data/design matrix $X$ is built from $X_\cdot$ corresponding to the continuous variables with additional columns corresponding to levels for the treatment vector $t$. As with multiple linear regression, the $y$ vector holds the response values.

Example Problem:

Class Methods:

```scala
@param x_ the data/design matrix of continuous variables
@param t the treatment/categorical variable vector
@param y the response vector
@param levels the number of treatment levels (1, ... levels)
@param technique the technique used to solve for $b$ in $x . t \cdot x \cdot b = x . t \cdot y$

class ANCOVA (x_: MatrixD, t: VectorI, y: VectorD, levels: Int, technique: RegTechnique = QR)
  extends Predictor with Error
```

```scala
val x = new MatrixD (x_.dim1, x_.dim2 + levels - 1) // augmented data matrix
val rg = new Regression (x, y, technique) // regular multiple linear regression
```

```scala
def assignVars ()
def assignDummyVars ()
def train (): Regression [MatrixD, VectorD] = rg.train ()
def eval (yy: VectorD = y) { rg.eval (yy) }
```

```scala
override def coefficient: VectorD = rg.coefficient
```
override def residual: VectoD = rg.residual
override def fit: VectoD = rg.fit
override def fitLabels: Seq[String] = rg.fitLabels
def predict (z: VectoD): Double = rg.predict (z)
def forwardSel (cols: Set[Int]): (Int, VectoD, VectoD) = rg.forwardSel (cols)
def backwardElim (cols: Set[Int]): (Int, VectoD, VectoD) = rg.backwardElim (cols)
def vif: VectoD = rg.vif
4.15 General Linear Models

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

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

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

Example Problem:

Class Methods:

trait GLM

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

object GLM extends GLM
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.

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

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

\[
y = \mathbb{E}[y|x] + \epsilon
\]  \hspace{1cm} (5.1)

Others will focus on maximizing the conditional probability of \( y \) given \( x \), i.e., finding the conditional mode.

\[
y^* = \arg\max P(y|x) = \mathbb{M}[y|x]
\]  \hspace{1cm} (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 \texttt{train} function is still used, but the \texttt{classify} method replaces the \texttt{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 \( x \)). 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: Vector[Int]): (Int, String, Double)
  def classify (z: Vector[Double]): (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 actualVpredicted (y: Vector[Int], yp: Vector[Int]): Map[String, Double] =
  def reset ()

For modeling, a user chooses one of the classes extending the trait Classifier (e.g., DecisionTreeID3) to instantiate an object. Next the train method would be typically called. While the modeling techniques in the last chapter focused on minimizing errors, the focus in this chapter will be on minimizing incorrect classifications. Generally, this is done by dividing a dataset up into a training dataset and test dataset. A way to utilize one dataset to produce multiple training and test datasets is called cross-validation.

As discussed in the Model Validation section in the Prediction chapter, k-fold cross-validation is a useful general purpose strategy for examining the quality of a model. The first cross-validation method takes the number of folds k (nx in the software) and a show flag. It performs k iterations of training (train method) and testing (test method).

def crossValidate (nx: Int = 10, show: Boolean = false): Double =
{
  val testSize = size / nx // number of instances in test set
  var sum = 0.0
  for (it <- 0 until nx) {
    val testStart = it * testSize // test set start index (inclusive)
    val testEnd = testStart + testSize // test set end index (exclusive)
    train (testStart, testEnd) // train on opposite instances
    val acc = test (testStart, testEnd) // test on test set
    if (show) println (s"crossValidate: for it = $it, acc = $acc")
    sum += acc // accumulate accuracy
  } // for
  sum / nx.toDouble // return average accuracy

86
The second cross-validation method is more complicated, but usually preferred, since it randomizes the instances selected for the test dataset, so that patterns coincidental to the index are broken up.

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

The `crossValidateRand` method calls the following methods:

- `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

abstract class ClassifierInt (x: MatrixI, y: VectorI, fn: Array[String], k: Int, cn: Array[String])
extends Classifier with Error

def size: Int = m

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

ClassifierInt provides methods to determine the value count (vc) for the features/variables. A method to shift values in a vector toward zero by subtracting the minimum value. It has base implementations for test methods and methods for calculating correlations. Finally, the featureSelection method will eliminate features that have little positive impact on the quality of the model. Rather than considering all \( n \) features/variables, a proper subset \( fset \subset \{ 0, 1, \ldots n-1 \} \) 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 predicted class labels
@param k the number class values

```scala
class ConfusionMat (y: VectorI, yp: VectorI, k: Int = 2)

val conf = ...
def confusion: Matrix = conf
def pos_neg (con: Matrix = conf): (Double, Double, Double, Double) =
def accuracy: Double = conf.trace / conf.sum.toDouble

def prec_recl: (VectorD, VectorD, Double, Double) =

def f1_measure (prec: Double, recl: Double): Double = 2.0 * prec * recl / (prec + recl)
def kappa: Double =
```

89
### 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

```scala
abstract class BayesClassifier (x: Matrix, y: Vector, fn: Array[String], k: Int, cn: Array[String])
  extends ClassifierInt (x, y, fn, k, cn) with BayesMetrics

def toggleSmooth () { smooth = ! smooth }
def calcCMI (idx: IndexedSeq[Int], vca: Array[Int]): MatrixD =
def cmiJoint (p_y: VectorD, p_Xy: HMatrix3[Double], p_XZy: HMatrix5[Double]): MatrixD =
def getParent: Any = null
protected def updateFreq (i: Int) {}
def printClassProb () { println (s"ClassProb = $p_y") }
```
5.5 Null Model

The `NullModel` class implements a simple Classifier suitable for discrete input data. Corresponding to the Null Model in the Prediction chapter, one could imagine estimating probabilities for outcomes of a random variable \( y \). Given an instance, this random variable indicates the classification or decision to be made. For example, it may be used for a decision on whether or not to grant a loan request. The model may be trained by collecting a training dataset. Probabilities may be estimated from data stored in an \( m \)-dimensional response/classification vector \( y \) within the training dataset. These probabilities are estimated based on the frequency \( \nu \) with which each class value occurs.

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

The right hand side is simply the size of the set containing the instance/row indices where \( y_i = c \) for \( c = 0, \ldots, k-1 \). The probability that random variable \( y \) equals \( c \) can be estimated by the number of elements in the vector \( y \) where \( y_i \) equals \( c \) divided by the total number of elements.

\[
P(y = c) = \frac{\nu(y = c)}{m} = \frac{m_c}{m}
\]

(5.3)

Exercise 1 below is the well-known toy classification problem on whether to play tennis \((y = 1)\) or not \((y = 0)\) based on weather conditions. Of the 14 days \((m = 14)\), tennis was not played on 5 days and was played on 9 days, i.e.,

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

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

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

Table 5.1: Class Frequency Vector

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

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

Picking the maximum probability case, one should always predict that tennis will be played, i.e., \( y^* = 1 \). This modeling technique should outperform purely random guessing, since it factors in the relative frequency with which tennis is played. As with the `NullModel` for prediction, more sophisticated modeling techniques should perform better than this `NullModel` for classification. If they are unable to provide higher accuracy, they are of questionable value.
NullModel Class

Class Methods:

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

class NullModel (y: VectorI, k: Int, cn: Array[String])
  extends Classifier

  def size: Int = m
  def train (itest: IndexedSeq[Int]): NullModel =
  def classify (z: VectorI): (Int, String, Double) =
  def test (itest: IndexedSeq[Int]): Double =
  def reset () { nu_y.set (0) }

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 data - opposite of testing
    reset () // reset counter
    for (i <- idx) nu_y(y(i)) += 1 // tally frequency counts
    p_y = nu_y / idx.size.toDouble // probability vector for class 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.
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
}
```
5.6 Naive Bayes

The NaiveBayes class implements a Naive Bayes (NB) Classifier suitable for discrete input data. A Bayesian Classifier is a special case of a Bayesian Network where one of the random variables is distinguished as the basis for making decisions, call it random variable $y$, the class variable. The NullModel ignores weather conditions which are the whole point of the ExampleTennis exercise. For Naive Bayes, weather conditions (or other data relevant to decision making) are captured in an $n$-dimensional vector of random variables.

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

For the exercise, $n = 4$ where $x_0$ is Outlook, $x_1$ is Temperature, $x_2$ is Humidity, and $x_3$ is Wind. The decision should be conditioned on the weather, i.e., rather than computing $P(y)$, we should compute $P(y|x)$. Bayesian classifiers are designed to find the class (value for random variable $y$) that maximizes the conditional probability of $y$ given $x$.

It may be complex and less robust to estimate $P(y|x)$ directly. Often it is easier to examine the conditional probability of $x$ given $y$. This answers the question of how likely it is that the input data comes from a certain class $y$. Flipping the perspective can be done using Bayes Theorem.

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

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

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

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

\[ P(x, y) = P(x|y) P(y) \tag{5.4} \]

One could in principle represent the joint probability $P(x, y)$ or the conditional probability $P(x|y)$ in a matrix. Unfortunately, with 30 binary random variables, the matrix would have over one billion rows and exhibit issues with sparsity. Bayesian classifiers will factor the probability and use multiple matrices to represent the probabilities.

5.6.1 Factoring the Probability

A Bayesian classifier is said to be naïve, when it is assumed that the $x_j$’s are sufficiently uncorrelated to factor $P(x|y)$ into the product of their conditional probabilities (independence rule).

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

Research has shown that even though the assumption that given response/class variable $y$, the $x$-variables are independent is often violated by a dataset, Naive Bayes still tends to perform well [17]. Substituting this factorization in equation 5.4 yields

\[ P(x, y) = P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{5.5} \]
The classification problem then is to find the class value for \( y \) that maximizes this probability, i.e., let \( y^* \) be the \text{argmax} of the product of the class probability \( P(y) \) and all the conditional probabilities \( P(x_j | y) \). The \text{argmax} is the value in the domain \( D_y = \{0, \ldots, k-1\} \) that maximizes the probability.

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

### 5.6.2 Estimating Conditional Probabilities

For Integer-based classifiers \( x_j \in \{0, 1, \ldots, vc_j - 1\} \) where \( vc_j \) is the value count for the \( j^{th} \) variable/feature (i.e., the number of distinct values). The Integer-based Naive Bayes classifier is trained using an \( m \)-by-\( n \) data matrix \( X \) and an \( m \)-dimensional classification vector \( y \). Each data vector/row in the matrix is classified into one of \( k \) classes numbered \( 0, 1, \ldots, k-1 \). The frequency or number of instances where column vector \( x_j = h \) and vector \( y = c \) is as follows:

\[
\nu(x_j = h, y = c) = |\{i | x_{ij} = h, y_i = c\}|
\]

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

\[
P(x_j = h | y = c) = \frac{\nu(x_j = h, y = c)}{\nu(y = c)} \tag{5.7}
\]

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 \textit{Joint Frequency Matrices/Tables} (JFTs) and a \textit{Class Frequency Vector} (CFV). From these, it is straightforward to compute \textit{Conditional Probability Matrices/Tables} (CPTs) and a \textit{Class Probability Vector} (CPV).

#### ExampleTennis Problem

For the ExampleTennis problem, the \textit{Joint Frequency Matrix/Table} (JFT) for \textit{Outlook} random variable \( x_0 \) is shown in Table 5.3

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

The column sums in the above matrix are 5 and 9, respectively. The corresponding \textit{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 $x_0$

<table>
<thead>
<tr>
<th>$x_0 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>3/9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4/9</td>
</tr>
<tr>
<td>2</td>
<td>3/5</td>
<td>2/9</td>
</tr>
</tbody>
</table>

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

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

Table 5.5: JFT for $x_3$

<table>
<thead>
<tr>
<th>$x_3 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

As expected, the column sums in the above matrix are again 5 and 9, respectively. The corresponding **Conditional Probability Matrix/Table** for random variable $x_0$, i.e., $P(x_0 = h \mid y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum as shown in table 5.6.

Table 5.6: CPT for $x_3$

<table>
<thead>
<tr>
<th>$x_3 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>6/9</td>
</tr>
<tr>
<td>1</td>
<td>3/5</td>
<td>3/9</td>
</tr>
</tbody>
</table>

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

### 5.6.3 Laplace Smoothing

When there are several possible class values, a dataset may exhibit zero instances for a particular class. This will result in a zero in the CFV vector and cause a divide-by-zero error when computing CPTs. One way to avoid the divide-by-zero, is to add one ($m_e = 1$) fake instance for each class, guaranteeing no zeros in the CFV vector. If m-estimates are used, the conditional probability is adjusted slightly as follows:

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

where $m_e$ is the parameter used for the m-estimate. The term added to the numerator, takes the one (or $m_e$) instance(s) and adds uniform probability for each possible values for $x_j$ of which there are $vc_j$ of them.
Table 5.7 shows the result of adding 1/3 in the numerator and 1 in the denominator, (e.g., for \( h = 0 \) and \( c = 0 \), \((2 + 1/3)/(5 + 1) = 7/18\)).

Table 5.7: CPT for \( x_0 \) with \( m_e = 1 \)

<table>
<thead>
<tr>
<th>( x_0 _y )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7/18</td>
<td>10/30</td>
</tr>
<tr>
<td>1</td>
<td>1/18</td>
<td>13/30</td>
</tr>
<tr>
<td>2</td>
<td>10/18</td>
<td>7/30</td>
</tr>
</tbody>
</table>

Another problem is when a conditional probability in a CPT is zero. If any CPT has a zero element, the corresponding product for the column (where the CPV and CPTs are multiplied) will be zero no matter how high the other probabilities may be. This happens when the frequency count is zero in the corresponding JFT (see element (1, 0) in Table 5.3). The question now is whether this is due to the combination of \( x_0 = 1 \) and \( y = 0 \) being highly unlikely, or that the dataset is not large enough to exhibit this combination. Laplace smoothing guards against this problem as well.

Other values may be used for \( m_e \) as well. SCALATION uses a small value for the default \( m_e \) to reduce the distortion of the CPTs.

### 5.6.4 Hypermatrices

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

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

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

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

```scala
nu_y = new VectorI (2), nu_Xy = new HMatrix3 [Int] (2, 4, Array (3, 3, 2, 2))
```

### 5.6.5 The `classify` Method

A new instance can now be classified by simply matching its values with those in the class probability table and conditional probability tables and multiplying all the entries. This is done for all \( k \) class values and the class with the highest product is chosen.

```scala
def classify (z: VectorI): (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
}

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

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

### 5.6.6 Feature Selection

Suppose that \(x_1\) and \(x_2\) are not considered useful for classifying a day as to its suitability for playing tennis. For \(z = [2, 1]\), i.e., \(z_0 = 2\) and \(z_3 = 1\), the two relative probabilities are the following:

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

The two probabilities are approximately 0.129 for \(c = 0\) (Do not Play) and 0.0476 for \(c = 1\) (Play). The higher probability is for \(c = 0\).

To perform feature selection in a systematic way SCALATION provides an fset array that indicates the features/variables to be kept in the model. This array is assigned by calling the featureSelection method in the ClassifierInt abstract class.

### 5.6.7 Efficient Cross-Validation

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

**NaiveBayes0 Class**

**Class Methods:**

- @param x the integer-valued data vectors stored as rows of a matrix
- @param y the class vector, where \(y(l) = \text{class for row } l \text{ of the matrix } x, x(l)\)
@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: Array[String] = null, 
k: Int = 2, cn: Array[String] = Array("no", "yes"), 
    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, but uses an optimized cross-validation technique.

Class Methods:

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

class NaiveBayes (x: MatriI, y: VectoI, fn: Array[String] = null, 
k: Int = 2, cn: Array[String] = Array("no", "yes"), 
    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 ()

---

99
5.6.8 Exercises

1. Complete the *ExampleTennis* problem given in this section by creating CPTs for random variables \( x_1 \) and \( x_2 \) and then computing the relative probabilities for \( z = [2, 2, 1, 1] \).

2. Use SCALATIOn’s Integer-based *NaiveBayes* class to build a classifier for the *ExampleTennis* problem.

   ```scala
   import scalation.analytics.classifier.ExampleTennis._
   println ("Tennis Example")
   println ("xy = " + xy) // combined matrix [x | y]
   val nb = NaiveBayes (xy, fn, k, cn, null, 0) // create a classifier
   nb.train () // train the classifier
   val z = VectorI (2, 2, 1, 1) // new data vector
   println (s"classify ($z) = ${nb.classify (z)}") // classify z
   ```

3. Compare the confusion matrix, accuracy, precision and recall of *NaiveBayes* on the full dataset to that of *NullModel*.

   ```scala
   val x = xy.sliceCol (0, xy.dim2 - 1) // data matrix
   val y = xy.col (xy.dim2 - 1) // response/class label vector
   val yp = new VectorI (xy.dim1) // predicted class label vector
   for (i <- x.range1) {
     yp(i) = nb.classify (x(i))._1
     println (s"Use nb to classify (${x(i)}) = ${yp(i)}")
   } // 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*.

   ```scala
   println ("nb cv accu = " + nb.crossValidateRand (10, true)) // 10-fold cross-validation
   ```

5. Compare the confusion matrix, accuracy, precision and recall of *RoundRegression* on the full dataset to that of *NullModel*.

6. Perform feature selection on the *ExampleTennis* problem. Which feature/variable is removed from the model, first, second and third. Explain the basis for the *featureSelection* method’s decision to remove a feature.

7. Use the Integer-based *NaiveBayes* class to build a classifier for the Breast Cancer problem (data in *breast-cancer.arff* file). Compare its accuracy to that of *NullModel*.
### 5.7 Tree Augmented Naive Bayes

The `TANBayes` class implements a Tree Augmented Naive Bayes (TAN) Classifier suitable for discrete input data. Unlike Naive Bayes, a TAN model can capture more, yet limited dependencies between variables/features. In general, \(x_j\) can be dependent on the class \(y\) as well as one other variable \(x_{p(j)}\). Representing the dependency pattern graphically, \(y\) becomes a root node of a Directed Acyclic Graph (DAG), where each node/variable has at most two parents.

Starting with the joint probability given in equation 5.5,

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

we can obtain a better factored approximation (better than Naive 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 Naive Bayes structure where each \(x\) random variable has \(y\) as its parent.

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

Since the root \(x_r\), has no \(x\)-parent, it can be factored out as special case.

\[
P(x, y) = P(y)P(x_r|y) \prod_{j \neq r} P(x_j|x_{p(j)}, y)
\]

As with Naive Bayes, the goal is to find an optimal value for the random variable \(y\) that maximizes the probability.

\[
y^* = \arg\max_{y \in D_y} P(y)P(x_r|y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)
\]

#### Structure Learning

Naive 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 (CMI) (alternatively correlation).

\[
I_p(x_j, x_l|y) = \text{(5.9)}
\]

The steps involved in the structure learning algorithm for TAN Bayes are the following:

1. Compute the CMI \(I_p(x_j, x_l|y)\) 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, SCALA'Tion’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.

**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>
<thead>
<tr>
<th>( x_j )</th>
<th>( x_p(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>null</td>
</tr>
</tbody>
</table>

In this case, the only modification to the CPV and CPTs from the Naïve Bayes solution, is that the JFT and CPT for \( x_0 \) are extended. The extended Joint Frequency Table (JFT) for \( x_0 \) is shown in Table 5.10.

<table>
<thead>
<tr>
<th>( x_0 ) ( \backslash ) ( x_3 ), ( y )</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The column sums are 2, 6, 3, 3, respectively. Again they must add up to same total of 14. Dividing each element in the JFT by its column sum yields the extended Conditional Probability Table (CPT) shown in Table 5.11.

<table>
<thead>
<tr>
<th>( x_0 ) ( \backslash ) ( x_3 ), ( y )</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>2/3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/3</td>
<td>0</td>
<td>2/3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1/6</td>
<td>1/3</td>
<td>1/3</td>
</tr>
</tbody>
</table>
5.7.1 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.

```scala
def classify (z: VectoI): (Int, String, Double) =
{
  val prob = new VectorD (p_y)
  for (i <- 0 until k; j <- 0 until n if fset(j)) {
    prob(i) *= (if (parent(j) > -1) p_XyP(i, j, z(j), z(parent(j)))
               else p_XyP(i, j, z(j), 0))
  } // for
  val best = prob.argmax ()
  (best, cn(best), prob(best))
} // classify
```

5.7.2 Cross-Validation

Again there are two classes: TANBayes0 that uses conventional “additive” cross-validation and TANBayes that uses more 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

```scala
class TANBayes0 (x: MatriI, y: VectoI, fn: Array [String] = null,
                 k: Int = 2, cn: Array [String] = Array (“no”, “yes”),
                 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) =
```
The \texttt{TANBayes} class is similar, but uses a more efficient cross-validation method.

### 5.7.3 Exercises

1. Use the Integer-based \texttt{TANBayes} to build classifiers for (a) the \texttt{ExampleTennis} problem and (b) the Breast Cancer problem (data in \texttt{breast-cancer.arff} file). Compare its accuracy to that of \texttt{NullModel} and \texttt{NaiveBayes}.

2. Re-engineer \texttt{TANBayes} to use correlation instead of conditional mutual information. Compare the results with the current \texttt{TANBayes} implementation.
5.8 Bayesian Network

The \texttt{BayesNetwork} class implements a Bayesian Network (BN) Classifier suitable for discrete input data. It classifies an 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,

\[
y^* = \arg \max_{y \in \mathcal{D}_y} P(y) \prod_{j=0}^{n-1} P(x_j|p(x_j), y)
\]

where \( p(x_j) \) is the set 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 govern by the number of other variables it is dependent upon. If this number is large, the CPT may become prohibitively large.

Training is achieved by ...

---

**Example Problem:**

**Class Methods:**

@\param \texttt{dag} the directed acyclic graph specifying conditional dependencies
@\param \texttt{table} the array of tables recording conditional probabilities
@\param \texttt{k} the number of classes

\begin{verbatim}
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 =
\end{verbatim}
5.9 Decision Tree ID3

A Decision Tree classifier \([13, 12]\) will take an input vector \(x\) and classify it, i.e., give one of \(k\) class values to \(y\) by applying a set of decision rules configured into a tree. Abstractly, the decision rules may be viewed as a function \(f\).

\[
y = f(x) = f(x_0, x_1, \ldots x_{n-1}) \tag{5.10}
\]

The DecisionTreeID3 \([10]\) class implements a Decision Tree classifier using the Iterative Dichotomiser 3 (ID3) algorithm. The classifier is trained using an \(m\)-by-\(n\) data matrix \(X\) and a classification vector \(y\). Each data vector in the matrix is classified into one of \(k\) classes numbered 0, 1, \ldots, \(k \- 1\). Each column in the matrix represents a \(x\)-variable/feature (e.g., Humidity). The value count \(\text{vc}\) vector gives the number of distinct values per feature (e.g., 2 for Humidity).

5.9.1 Entropy

In decision trees, the goal is to reduce the disorder in decision making. Assume the decision is of the yes(1)/no(0) variety and consider the following decision/classification vectors: \(y = (1, 1, \ldots, 1, 1)\) or \(y' = (1, 0, \ldots, 1, 0)\). In the first case all the decisions are yes, while in the second, there are an equal number of yes and no decisions. One way to measure the level of disorder is Shannon entropy. To compute the Shannon entropy, first convert the \(m\)-dimensional decision/classification vector \(y\) into a \(k\)-dimensional probability vector \(p\).

\[
def \text{frequency (): VectorD =}
\{
    val p = new VectorD (k)
    for (i <- 0 until m) p(y(i)) += 1
    p / md
\} // frequency
\]

For the two cases, \(p = (1, 0)\) and \(p' = (.5, .5)\), so computing the Shannon entropy \(H(p)\),

\[
H(p) = - \sum_{i=0}^{k-1} p_i \log_2(p_i) \tag{5.11}
\]

we obtain \(H(p) = 0\) and \(H(p') = 1\), which indicate that there is no disorder in the first case and maximum disorder in the second case.

\[
def \text{entropy (p: VectorD): Double =}
\{
    var sum = 0.0
    for (pi <- p if pi > 0.0) sum -= pi * log2 (pi)
    sum // return entropy, a number in the interval [0, max]
\} // entropy
\]

Letting the dimensionality of the probability vector be \(k\), the maximum entropy is given by \(\log_2(1/k)\), which is 1 for \(k = 2\). The maximum base-\(k\) entropy is always 1.
\[ H(p) = - \sum_{i=0}^{k-1} p_i \log_k(p_i) \]

Entropy is used as measure of the impurity of a node (e.g., to what degree is it a mixture of ‘-’ and ‘+’). For a discussion of additional measures see [12]. For a deeper dive into entropy, relative entropy and mutual information see [3].

### 5.9.2 Example Problem

Let us consider the Tennis example from NullModel and NaiveBayes and compute the entropy level for the decision of whether to play tennis. There are 14 days worth of training data see Table 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>
<thead>
<tr>
<th>Day</th>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ H(p) = H(\frac{5}{14}, \frac{9}{14}) = -\frac{5}{14} \log_2(\frac{5}{14}) - \frac{9}{14} \log_2(\frac{9}{14}) = 0.9403 \]

Recall that the features are Outlook \(x_0\), Temp \(x_1\), Humidity \(x_2\), and Wind \(x_3\). To reduce entropy, find the feature/variable that has the greatest impact on reducing disorder. If feature/variable \(j\) is factored into the decision making, entropy is now calculated as follows:

\[
\sum_{v=0}^{vc_j-1} \frac{\nu(x_{-j} = v)}{m} H(p_{x_{-j}=v})
\]

where \(\nu(x_{-j} = v)\) is the frequency count of value \(v\) for column vector \(x_{-j}\) in matrix \(X\). The sum is the weighted average of the entropy over all possible \(vc_j\) 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.

$$\sum_{v=0}^{2} \frac{\nu(x_0 = v)}{m} H(p_{x_0 = v})$$

For $v = 0$, we have 2 no (0) cases and 3 yes (1) cases ($2 \cdot 3$, $3 \cdot 3$), for $v = 1$, we have ($0 \cdot 4$, $4 \cdot 4$) and for $v = 2$, we have ($3 \cdot 2$, $2 \cdot 2$).

$$\frac{\nu(x_0 = 0)}{14} H(p_{x_0 = 0}) + \frac{\nu(x_0 = 1)}{14} H(p_{x_0 = 1}) + \frac{\nu(x_0 = 2)}{14} H(p_{x_0 = 2})$$

We are left with computing three entropy values:

$$H(p_{x_0 = 0}) = H\left(\frac{2}{5}, \frac{3}{5}\right) = -\frac{2}{5} \log_2\left(\frac{2}{5}\right) - \frac{3}{5} \log_2\left(\frac{3}{5}\right) = 0.9710$$

$$H(p_{x_0 = 1}) = H\left(\frac{0}{1}, \frac{4}{1}\right) = -\frac{0}{1} \log_2\left(\frac{0}{1}\right) - \frac{4}{1} \log_2\left(\frac{4}{1}\right) = 0.0000$$

$$H(p_{x_0 = 2}) = H\left(\frac{3}{5}, \frac{2}{5}\right) = -\frac{3}{5} \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \log_2\left(\frac{2}{5}\right) = 0.9710$$

The weighted average is then $0.6936$, so that the drop in entropy (also called information gain) is $0.9403 - 0.6936 = 0.2467$. As shown in Table 5.13, the other entropy drops are 0.0292 for Temperature (1), 0.1518 for Humidity (2) and 0.0481 for Wind (3).

Table 5.13: Choices for Principal Feature

<table>
<thead>
<tr>
<th>$j$</th>
<th>Variable/Feature</th>
<th>Entropy</th>
<th>Entropy Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Outlook</td>
<td>0.6936</td>
<td>0.2467</td>
</tr>
<tr>
<td>1</td>
<td>Temperature</td>
<td>0.9111</td>
<td>0.0292</td>
</tr>
<tr>
<td>2</td>
<td>Humidity</td>
<td>0.7885</td>
<td>0.1518</td>
</tr>
<tr>
<td>3</td>
<td>Wind</td>
<td>0.8922</td>
<td>0.0481</td>
</tr>
</tbody>
</table>

Hence, Outlook ($j = 0$) should be chosen as the principal feature for decision making. As the entropy is too high, make a tree with Outlook (0) as the root and make a branch for each value of Outlook: Rain (0), Overcast (1), Sunny (2). Each branch defines a sub-problem.

**Sub-problem $x_0 = 0$**

The sub-problem for Outlook: Rain (0) see Table 5.14 is defined as follows: Take all five cases/rows in the data matrix $X$ for which $x_0 = 0$.

If we select Wind ($j = 3$) as the next variable, we obtain the following cases: For $v = 0$, we have ($0 \cdot 3$, $3 \cdot 3$), so the probability vector and entropy are

$$p_{x_3 = 0} = \left(\frac{0}{5}, \frac{3}{5}\right) \quad H(p_{x_3 = 0}) = 0$$

108
Table 5.14: Sub-problem for node $x_0$ and branch 0

<table>
<thead>
<tr>
<th>Day</th>
<th>$x_{-1}$</th>
<th>$x_{-2}$</th>
<th>$x_{-3}$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For $v = 1$, we have $(2-, 0+)$, so the probability vector and entropy are

$$p_{x_{-3}=1} = \left( \frac{2}{5}, \frac{0}{5} \right) \quad H(p_{x_{-3}=1}) = 0$$

If we stop expanding the tree at this point, we have the following rules.

\[
\text{if } x_0 = 0 \text{ then } \\
\quad \text{if } x_3 = 0 \text{ then yes} \\
\quad \text{if } x_3 = 1 \text{ then no} \\
\text{if } x_0 = 1 \text{ then yes} \\
\text{if } x_0 = 2 \text{ then no}
\]

The overall entropy can be calculated as the weighted average of all the leaf nodes.

\[
\frac{3}{14} \cdot 0 + \frac{2}{14} \cdot 0 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.9710 = .3468
\]

**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 5.15 is defined as follows: Take all five cases/rows in the data matrix $X$ for which $x_0 = 2$.

Table 5.15: Sub-problem for node $x_0$ and branch 2

<table>
<thead>
<tr>
<th>Day</th>
<th>$x_{-1}$</th>
<th>$x_{-2}$</th>
<th>$x_{-3}$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It should be obvious that $y = 1 - x_{-2}$. For $v = 0$, we have $(0-, 2+)$, so the probability vector and entropy are
\[ p_{x_2=0} = \left( \frac{3}{5}, \frac{2}{5} \right) \quad H(p_{x_2=0}) = 0 \]

For \( v = 1 \), we have \((3-, 0+)\), so the probability vector and entropy are

\[ p_{x_2=1} = \left( \frac{3}{5}, \frac{2}{5} \right) \quad H(p_{x_2=0}) = 0 \]

At this point, the overall entropy is zero and the decision tree is the following (shown as a pre-order traversal from ScalaTion).

**Decision Tree:**

```
[ 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.

### 5.9.3 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 to 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.

### 5.9.4 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:
class DecisionTreeID3 (x: MatrixI, y: VectorI, fn: Array[String], k: Int, cn: Array[String],
    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, VectorI, VectorD) =

def gain (f: Int, path: List[(Int, Int)]: (Double, VectorI) =

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 ()

5.9.5 Exercises

1. Show for $k = 2$ where $pp = [p, 1 - p]$, that $H(pp) = p \log_2(p) + (1 - p) \log_2(1 - p)$. Plot $H(pp)$ 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 decision 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.
Chapter 6

Classification: Continuous Variables

For the problems in this chapter, the response/classification variable is still discrete, but some/all of the feature variables are now continuous. Technically, classification problems fit in this category, if it is infeasible or nonproductive to compute frequency counts for all values of a variable (e.g., for \( x_j \), the value count \( vc_j = \infty \)). If a classification problem almost fits in the previous chapter, one may consider the use of binning to convert numerical variables into categorical variables (e.g., convert weight into weight classes). Care should be taken since binning represents hidden parameters in the model and arbitrary choices may influence results.
6.1 ClassifierReal

The ClassifierReal abstract class provides a common foundation for several classifiers that operate on continuous (or real-valued) data.

Class Methods:

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

abstract class ClassifierReal (x: MatriD, y: VectoI, fn: Array[String], k: Int, cn: Array[String]) extends Classifier with Error

def vc_default: Array[Int] = Array.fill (n)(2)
def size: Int = m
def classify (z: VectoI): (Int, String, Double) = classify (z.toDouble)
def test (itest: IndexedSeq[Int]): Double =
def test (xx: MatriD, yy: VectoI): Double =
def calcCorrelation: MatriD =
def calcCorrelation2 (zrg: Range, xrg: Range): MatriD =
def featureSelection (TOL: Double = 0.01)
6.2 Gaussian Naive Bayes

The \texttt{NaiveBayesR} class implements a Gaussian Naive Bayes Classifier, which is the most commonly used such classifier for continuous input data. The classifier is trained using a data matrix $X$ and a classification vector $y$. Each data vector in the matrix is classified into one of $k$ classes numbered $0, 1, \ldots, k - 1$.

Class probabilities are calculated based on the population of each class in the training-set. Relative probabilities are computed by multiplying these by values computed using conditional density functions based on the Normal (Gaussian) distribution. The classifier is naive, because it assumes feature independence and therefore simply multiplies the conditional densities.

Starting with main results from the section on Naive Bayes (equation 4.5),

$$ y^* = \arg\max_{y \in \{0, \ldots, k-1\}} P(y) \prod_{j=0}^{n-1} P(x_j | y) $$

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.

$$ y^* = \arg\max_{y \in \{0, \ldots, k-1\}} P(y) \prod_{j=0}^{n-1} f(x_j | y) \quad (6.1) $$

Although the formula assumes the conditional independence of $x_j$s, the technique can be applied as long as correlations are not too high.

Using the Gaussian assumption, the conditional density of $x_j$ given $y$, is approximated by estimating the two parameters of the \texttt{Normal} distribution,

$$ x_j | y \sim \text{Normal} (\mu_c, \sigma_c^2) $$

where class $c \in \{0, 1, \ldots, k-1\}$, $\mu_c = \mathbb{E} [x | y = c]$ and $\sigma_c^2 = \mathbb{V} [x | y = c]$. Thus, the conditional density function is

$$ f(x_j | y = c) = \frac{1}{\sqrt{2\pi}\sigma_c} e^{-\frac{(x-\mu_c)^2}{2\sigma_c^2}} $$

Class probabilities $P(y = c)$ may be estimated as $\frac{m_c}{m}$, where $m_c$ is the frequency count of the number of occurrences of $c$ in the class vector $y$. Conditional densities are needed for each of the $k$ class values, for each of the $n$ variables (each $x_j$) (i.e., $kn$ are needed). Corresponding means and variances may be estimated as follows:

$$ \hat{\mu}_{cj} = \frac{1}{m_c} \sum_{i=0}^{m-1} (x_{ij} | y_i = c) $$

$$ \hat{\sigma}^2_{cj} = \frac{1}{m_c - 1} \sum_{i=0}^{m-1} ((x_{ij} - \hat{\mu}_{cj})^2 | y_i = c) $$

Using conditional density (cd) functions estimated in the \texttt{train} function (see code for details), an input vector $z$ can be classified using the \texttt{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:

class NaiveBayesR (x: MatriD, y: VectoI, fn: Array [String], k: Int = 2)
    extends ClassifierReal (x, y, fn, k)

@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: Array [String], k: Int = 2,
    cn: Array [String] = Array ("yes", "no")
    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)
    //
    val xy = new MatrixD ((7, 3), 2.95, 6.63, 0,  // joint data matrix
                            2.53, 7.79, 0,
                            3.57, 5.65, 0,  // first sample
                            116, 0.14, 0,
                            3, 0.48, 0,
                            2.54, 6.77, 0,
                            3.53, 5.66, 0)
    //
val fn = Array ("curvature", "diameter")   // feature names
val cn = Array ("pass", "fail")           // class names
val cl = NaiveBayesR (xy, fn, 2, cn)      // create NaiveBayesR classifier
6.3 Simple Logistic Regression

The SimpleLogisticRegression class supports simple logistic regression. In this case, the predictor vector \( x \) is two-dimensional \([1, x_1]\). Again, the goal is to fit the parameter vector \( b \) in the regression equation

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

where \( \epsilon \) represents the residuals (the part not explained by the model). This looks like simple linear regression, with the difference being that the response variable \( y \) is binary \((y \in \{0, 1\})\). Since \( y \) is binary, minimizing the distance, as was done before, may not work well. First, instead of focusing on \( y \in \{0, 1\} \), we focus on the conditional probability of success \( p_y(x) \in [0, 1] \), i.e.,

\[
p_y(x) = P(y = 1|x)
\]

For example, the random variable \( y \) could be used to indicate whether a customer will pay back a loan (1) or not (0). The predictor variable \( x_1 \) could be the customer’s FICA score.

6.3.1 mtcars Example

Another example is from the Motor Trends Cars (mtcars) dataset (see https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html, gist.github.com/seankross/a412dfbd88b3db70b74b). Try using mpg to predict/classify the car’s engine as either V-shaped(0) or Straight(1) as in V-6 or S-4. Use SimpleRegression to predict \( p_y(x_1) \) where \( y \) is V/S and \( x_1 \) is mpg. Plot \( y \) versus \( x_1 \) and then add a vector to the plot for the predicted values for \( p_y \). Using simple linear regression to predict \( p_y(x_1) \) would correspond to the following equation.

\[
p_y(x) = b_0 + b_1 x_1
\]

6.3.2 Logistic Function

Still the linear relationship 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. This suggests that some “S-curve” function such as the logistic function may be more useful. The standard logistic function (sigmoid function) is

\[
\text{logistic}(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}
\]

Letting \( z = b_0 + b_1 x_1 \), we obtain

\[
p_y(x) = \text{logistic}(b_0 + b_1 x_1) = \frac{e^{b_0 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}}
\]

6.3.3 Logit Function

The goal now is to transform the right hand side into the usual linear form.

\[
p_y(x) = \frac{e^{b \cdot x}}{1 + e^{b \cdot x}}
\]

Solving for \( e^{b \cdot x} \) yields
\[ e^{b \cdot x} = \frac{p_y(x)}{1 - p_y(x)} \]

Taking the natural logarithm of both sides gives

\[ \ln \left( \frac{p_y(x)}{1 - p_y(x)} \right) = b \cdot x = b_0 + b_1 x \]

where the function on the left hand side is called the logit function. Putting the model in this form shows it is a special case of a Generalized Linear Model (see Chapter 7) and will be useful in the estimation procedure.

### 6.3.4 Maximum Likelihood Estimation

Imagine you wish to create a model that is able to generate data that looks like the observed data (i.e., the data in the dataset). The choice of values for the parameters \( b \) (treated as a random vector) will impact the quality of the model. Define a function of \( b \) that will be maximized when the parameters are ideally set to generate the observed data.

### 6.3.5 Likelihood Function

We can think of this function as the likelihood of \( b \) given the predictor vector \( x \) and the response variable \( y \).

\[ L(b|x, y) \]

In this case, \( y \in \{0, 1\} \), so if we estimate the likelihood for a single data instance (or row), we have

\[ L(b|x, y) = p_y(x)^y (1 - p_y(x))^{1-y} \]  

(6.5)

If \( y = 1 \), then \( L = p_y(x) \) and otherwise \( L = 1 - p_y(x) \). These are the probabilities for the two outcomes for a Bernoulli random variable (and equation 6.5 concisely captures both).

For each instance \( i \in \{0, \ldots, m - 1\} \), a similar factor is created. These are multiplied together for all the instances (in the dataset, or training or testing). The likelihood of \( b \) given the predictor matrix \( X \) and and the response vector \( y \) is then

\[ L(b|x, y) = \prod_{i=0}^{m-1} p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i} \]  

(6.6)

### 6.3.6 Log-likelihood Function

To reduce round-off errors, a log (e.g., natural log, ln) is taken

\[ l(b|x, y) = \sum_{i=0}^{m-1} y_i \ln(p_y(x_i)) + (1 - y_i)\ln(1 - p_y(x_i)) \]

This is referred as the log-likelihood function. Collecting \( y_i \) terms give

\[ l(b|x, y) = \sum_{i=0}^{m-1} y_i \ln \left( \frac{p_y(x_i)}{1 - p_y(x_i)} \right) + \ln(1 - p_y(x_i)) \]
Substituting $b \cdot x_i$ for $\logit$ and $\frac{b \cdot x_i}{1 + e^{b \cdot x_i}}$ for $p_y(x_i)$ gives

$$l(b|x,y) = \sum_{i=0}^{m-1} y_i (b \cdot x_i - \ln(1 + e^{b \cdot x_i}))$$

(6.7)

Multiplying the log-likelihood by -2 makes the distribution approximately Chi-square.

$$-2l = -2 \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})$$

Or since $b = [b_0, b_1]$,

$$-2l = -2 \sum_{i=0}^{m-1} y_i (b_0 + b_1 x_{i1}) - \ln(1 + e^{b_0 + b_1 x_{i1}})$$

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.

$$-2l = -2 \sum_{i=0}^{m-1} y_i \beta_i - \beta_i - \ln(e^{-\beta_i} + 1)$$

The computation of $-2l$ is carried out in SCALATION via the ll method.

```scala
def ll (b: VectorD): Double =
{
    var sum = 0.0
    var bx = 0.0 // beta
    for (i <- y.range) {
        bx = b(0) + b(1) * x(i, 1)
        sum += y(i) * bx - bx - log (exp (-bx) + 1.0)
    } // for
    -2.0 * sum
} // ll
```

SimpleLogisticRegression Class

Class Methods:

@param x the input/design 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: Matrix, y: VectorI, fn: Array[String] = Array("one", "x1"),
```
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 coefficient: VectoD = b
def fit: VectoD =
def fitLabels: Seq [String] = Seq ("n_dev", "r_dev", "aic", "pseudo_rSq")
override def classify (z: VectoD): (Int, String, Double) =
override def classify (z: VectoI): (Int, String, Double) = classify (z.toDouble)
def reset () { /* Not Applicable */ }

6.3.7 Exercises

1. Plot the standard logistic function (sigmoid).

```scala
import scalation.analytics.ActivationFun.sigmoidV
val z = VectorD.range (0, 160) / 10.0 - 8.0
val fz = sigmoidV (z)
new Plot (z, fz)
```

6.4 Logistic Regression

The LogisticRegression class supports logistic regression. In this case, \( \mathbf{x} \) may be multi-dimensional \([1, x_1, \ldots, x_k]\). Again, the goal is to fit the parameter vector \( \mathbf{b} \) in the regression equation

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

where \( \epsilon \) represents the residuals (the part not explained by the model). This looks like multiple linear regression. The difference being that the response variable \( y \) is binary \((y \in \{0, 1\})\). Since \( y \) is binary, minimizing the distance, as was done before may not work well. First, instead of focusing on \( y \in \{0, 1\} \), we focus on the conditional probability of success \( p_y(\mathbf{x}) \in [0, 1] \), 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 \( g = \text{logit} \).

\[
\text{logit}(\mu(\mathbf{x})) = \ln \frac{p_y(\mathbf{x})}{1 - p_y(\mathbf{x})} = \mathbf{b} \cdot \mathbf{x}
\]

This is the logit regression equation. Second, instead of minimizing the sum of squares errors, we wish to maximize the likelihood of predicting correct outcomes. For the \( i^{th} \) 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.

\[
L(\mathbf{b}|\mathbf{x}, y) = \prod_{i=0}^{m-1} p_y(\mathbf{x}_i)^{y_i} (1 - p_y(\mathbf{x}_i))^{1 - y_i} \tag{6.8}
\]

Following the same derivation steps, will give the same log-likelihood that is in equation 6.7.

\[
l(\mathbf{b}|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i \mathbf{b} \cdot \mathbf{x}_i - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x}_i}) \tag{6.9}
\]

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 \(b\). Various optimization techniques may be used to search for optimal values for \(b\). Currently, SCALATION uses BFGS, a popular general-purpose QuasiNewton NLP solver. Other possible optimizers include LBFGS and IRWLS. For a more detailed derivation, see [http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf](http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf).

### 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

```scala
class LogisticRegression (x: Matrix, y: Vector, fn: Array[String] = null, cn: Array[String] = Array("no", "yes")) extends ClassifierReal (x, y, fn, 2, cn)
def ll (b: Vector): Double =
def ll_null (b: Vector): Double =
def train (itest: IndexedSeq[Int]): LogisticRegression =
def train_null ()
def coefficient: Vector = b
def fit: Vector =
def fitLabels: Seq[String] = Seq("n_dev", "r_dev", "aic", "pseudo_rSq")
override def classify (z: Vector): (Int, String, Double) =
override def classify (z: VectorI): (Int, String, Double) = classify (z.toDouble)
def forwardSel (cols: Set[Int]): (Int, Vector, Vector) = ???
def backwardElim (cols: Set[Int]): (Int, Vector, Vector) =
def vif: Vector =
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 [S] section 4.6.2.
2. Use Logistic Regression to classify whether a customer will purchase caravan insurance. The Caravan dataset is in the ISLR library, see [S] section 4.6.6.
6.5 Simple Linear Discriminant Analysis

The SimpleLDA class supports Linear Discriminant Analysis which is useful for multiway classification of continuously valued data. The response/classification variable can take on \( k \) possible values, \( y \in \{0, 1, \ldots, k-1\} \). The feature variable \( x \) is one dimensional for SimpleLDA, but can be multi-dimensional for LDA discussed in the next section. Given the data about an instance stored in variable \( x \), pick the best (most probable) classification \( y = c \).

As was done for Naive 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 Naive Bayes classifiers,

\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)} \tag{6.10}
\]

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 \( \sigma^2 \) is the pooled variance (weighted average of \( \mathbb{V}[x|y=c] \)). Thus, the conditional density function is

\[
f(x|y=c) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}
\]

Substituting into equation 6.10 gives

\[
P(y|x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y) \tag{6.11}
\]

where

\[
f(x) = \sum_{c=0}^{k-1} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y=c)
\]

Because of differing means, each conditional density will be shifted resulting in a mountain range appearance when plotted together. Given a data point \( x \), the question becomes, which mountain is it closest to in the sense of maximizing the conditional probability expressed in equation 6.11.

\[
P(y|x) \propto \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y)
\]

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).

\[
\ln(P(y|x)) \propto \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y))
\]  \hspace{1cm} (6.12)

The right hand side functions in 4.12 are linear in \(x\) and are called discriminant functions \(\delta_c(x)\).

Given training data vectors \(x\) and \(y\), define \(x_c\) (or \(x_c\) in the code) to be the vector of all \(x_i\) values where \(y_i = c\) and let its length be denoted by \(m_c\). Now the \(k\) means may be estimated as follows:

\[
\hat{\mu}_c = \frac{1}{m_c} \cdot x_c
\]

The common variance may be estimated using a pooled variance estimator.

\[
\hat{\sigma}^2 = \frac{1}{m-k} \sum_{c=0}^{k-1} ||x_c - \mu_c||^2
\]

Finally, \(\frac{m_c}{m}\) can be used to estimate \(P(y)\).

These can easily be translated into ScalaTion code. Most of the calculations are done in the \texttt{train} method. It estimates the class probability vector \(py\), the group means vector \(mu\) and the pooled variance. The vectors \texttt{term1} and \texttt{term2} capture the \(x\)-term \((\mu_c/\sigma^2)\) and the constant term \((\mu_c^2/2\sigma^2 - \ln(P(y)))\) in equation 6.12.

```scala
def train (itest: IndexedSeq[Int]): SimpleLDA =
{
  py = VectorD (xc.map (_.dim / md)) // probability y = c
  mu = VectorD (xc.map (_.mean))    // group means
  var sum = 0.0
  for (c <- 0 until k) sum += (xc(c) - mu(c)).normSq
  sig2 = sum / (m - k).toDouble    // pooled variance
  term1 = mu / sig2
  term2 = mu~^2 / (2.0 * sig2) - py.map (log (_))
  this
} // train
```

Given the two precomputed terms, the \texttt{classify} method simply multiplies the first by \(z(0)\) and subtracts the second. Then it finds the \texttt{argmax} of the \texttt{delta} vector to return the class with the maximum \texttt{delta}, which corresponds the most probable classification.

\[
y^* = \text{argmax}_c \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y))
\]  \hspace{1cm} (6.13)

```scala
override def classify (z: VectorD): (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: Array [String] = Array ("x1"), k: Int = 2,
                   cn: Array [String] = Array ("no", "yes"))
    extends ClassifierReal (MatrixD (Seq (x)), y, fn, k, cn)

def train (itest: IndexedSeq [Int]): SimpleLDA =
    override def classify (z: VectoD): (Int, String, Double) =
    def reset () { /* Not Applicable */ }

6.5.1 Exercises

1. Generate two samples using Normal (98.6, 1.0) and Normal (101.0, 1.0) with 100 in each sample. Put the data instances into a single x vector. Let the y vector be 0 for the first sample and 1 for the second. Use SimpleLDA to classify all 200 data points and determine the values for tp, tn, fn and fp. See scalation.analytics.classifier.SimpleLDATest2.
6.6 Linear Discriminant Analysis

Like SimpleLDA, the LDA class supports Linear Discriminant Analysis that is used for multiway classification of continuously valued data. Similarly, the response/classification variable can take on \(k\) possible values, \(y \in \{0, 1, \ldots, k-1\}\). Unlike SimpleLDA, this class is intended for cases where the feature vector \(x\) is multi-dimensional. The classification \(y = c\) is chosen to maximize the conditional probability of class \(y\) given the \(n\)-dimensional data/feature vector \(x\).

\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)} \quad (6.14)
\]

where

\[
f(x) = \sum_{c=0}^{k-1} f(x|y=c)P(y=c)
\]

In the multi-dimensional case, \(x|y\) has a multivariate Gaussian distribution, \(\text{Normal}(\mu_c, \Sigma)\), where \(\mu_c\) are the mean vectors \(E[x|y = c]\) and \(\Sigma\) is the common covariance matrix (weighted average of \(C[x|y = c]\). The conditional density function is given by

\[
f(x|y = c) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c)}
\]

Dropping factors independent of \(c\) and multiplying by \(P(y = c)\) gives

\[
f(x|y = c)P(y = c) \propto e^{-\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c)} P(y = c)
\]

Taking the natural logarithm

\[
\ln(P(y|x)) \propto -\frac{1}{2}(x-\mu_c)^t \Sigma^{-1} (x-\mu_c) + \ln(P(y = c))
\]

The discriminant functions are obtained by multiplying out and again dropping terms independent of \(c\).

\[
\delta_c(x) = x^t \Sigma^{-1} \mu_c - \frac{\mu_c^t \Sigma^{-1} \mu_c}{2} + \ln(P(y = c)) \quad (6.15)
\]

As in the last section, the means for each class \(c\) (\(\mu_c\)), the common covariance matrix (\(\Sigma\)), and the class probabilities (\(P(y)\)) must be estimated.

Class Methods:

@param x the real-valued training/test data vectors stored as rows of a matrix
@param y the training/test classification vector, where \(y_i\) = class for row \(i\) of the matrix \(x\)
@param fn the names for all features/variables
@param k the number of classes (\(k\) in \(\{0, 1, \ldots, k-1\}\))
@param cn the names for all classes

class LDA (x: MatrixD, y: VectorI, fn: Array [String], k: Int = 2, cn: Array [String] = Array ("no", "yes")) extends ClassifierReal (x, y, fn, k, cn)

127
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 KNNClassifier class is used to classify a new vector \( z \) into one of \( k \) classes \( y \in \{0, 1, \ldots, k-1\} \). It works by finding its \( k \)-nearest neighbors to the point \( z \). These neighbors essentially vote according to their classification. The class with the most votes is selected as the classification of vector \( z \). Using a distance metric, the \( k \) vectors nearest to \( z \) are found in the training data, which are stored row-wise in data matrix \( X \). The corresponding classifications are given in vector \( y \), such that the classification for vector \( x_i \) is given by \( y_i \).

In ScalaTion to avoid the overhead of calling \( \text{sqrt} \), the square of the Euclidean distance is used (although other metrics can easily be swapped in). The squared distance from vector \( x \) to vector \( z \) is then

\[
d(x) = d(x, z) = ||x - z||^2
\]

The distance metric is used to collect the \( k \) nearest vectors into set \( \text{top}_k(z) \), such that there does not exists any vector \( x_j \notin \text{top}_k(z) \) that is closer to \( z \).

\[
\text{top}_k(z) = \{ x_i | i \in \{0, \ldots, k-1\} \text{ and } \nexists (x_j \notin \text{top}_k(z) \text{ and } d(x_j) < d(x_i) ) \}
\]

In case of ties for the most distant point to include in \( \text{top}_k(z) \) one could pick the first point encountered or the last point. A less biased approach would be to randomly break the tie.

Now \( y(\text{top}_k(z)) \) can be defined to be the vector of votes from the members of the set, e.g., \( y(\text{top}_4(z)) = [1,0,1] \). The ultimate classification is then simply the mode (most frequent value) of this vector (e.g., 1 in this case).

\[
y^* = \text{mode} \ y(\text{top}_k(z))
\]

6.7.1 Lazy Learning

Training in the KNNClassifier class is lazy, i.e., the work is done in the classify method, rather than the train method.

```scala
override def classify (z: VectorD): (Int, String, Double) =
{
    kNearest (z) // set topK to kappa nearest
    for (i <- 0 until kappa) count(y(topK(i)._1)) += 1 // tally votes per class
    val best = count.argmax () // class with maximal count
    reset () // reset topK and counters
    (best, cn(best), count(best)) // return best class, its name and votes
} // classify
```

The kNearest method finds the \( k \times \) vectors closest to the given vector \( z \). This method updates topK by replacing the most distant \( x \) vector in topK with a new one if it is closer. Each element in the topK array is a tuple \( (j, d(j)) \) indicating which vector and its distance from \( z \). Each of these selected vectors will have their vote taken, voting for the class for which it is labelled. These votes are tallied in the count vector. The class with the highest count will be selected as the best class.
KNN Classifier Class

Class Methods:

@param x the vectors/points of classified data stored as rows of a matrix
@param y the classification of each vector in x
@param fn the names of the features/variables
@param k the number of classes
@param cn the names for all classes
@param kappa the number of nearest neighbors to consider

class KNN_Classifier (x: Matrix, y: Vector, fn: Array[String], k: Int, cn: Array[String],
kappa: Int = 3)
    extends ClassifierReal (x, y, fn, k, cn)

def distance (x: Vector, z: Vector): Double = (x - z).normSq

def kNearest (z: Vector):

def train (itest: IndexedSeq[Int]): KNN_Classifier =

override def classify (z: Vector): (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 \( tp, tn, fn, fp \).

//
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 adapted to work for prediction problems?
6.8 Decision Tree C45

The DecisionTreeC45 class implements a Decision Tree classifier using the C4.5 algorithm. The classifier is trained using a data matrix 'x' and a classification vector 'y'. Each data vector in the matrix is classified into one of 'k' classes numbered '0, ..., k-1'. Each column in the matrix represents a feature (e.g., Humidity). The 'vc' array gives the number of distinct values per feature (e.g., 2 for Humidity).

Example Problem:

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 of the features/variables
@param isCont Boolean value to indicate whether according feature is continuous
@param vc the value count array indicating number of distinct values per feature
@param k the number of classes

class DecisionTreeC45 (val x: MatrixI, val y: VectorI, fn: Array[String],
                        isCont: Array[Boolean], vc: VectorI = null, k: Int = 2)
    extends ClassifierInt (x, y, fn, vc, k)

    def frequency (fCol: VectorI, value: Int, cont: Boolean = false, thres: Double = 0):
    def entropy (prob: VectorD): Double =
    def gain (f: Int): Double =
    def calThreshold (f: Int)
    def nextXY (f: Int, value: Int): (MatrixI, Array[Int]) =
    def train ()
    def buildTree (opt: (Int, Double))
    def printTree
    override def classify (z: VectorI): Int =
    def classify (z: VectorD): Int =
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: Array[String], val cn: Array[String])
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.

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 k the number of classes

```scala
class SupportVectorMachine (x: Matrix, y: Vector, fn: Array[String], k: Int = 2)
  extends ClassifierReal (x, y, fn, k)

  def l_D (a: Vector): Double =
  def g (a: Vector): Double = a dot y
  def find_w ()
  def find_b ()
  def train ()
  def fit: (Vector, Double) = (w, b)
  def classify (z: Vector): Int = (signum (w dot z + b)).toInt
```

133
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 \( \mu(x) = \mathbb{E}[y|x] \) and an error term (or multiplying in the case of multiplicative errors).

\[
y = \mu(x) + \epsilon
\]

In the second step, the mean function is related to a linear combination of the predictor variables, i.e., \( b \cdot x \)

\[
g(\mu(x)) = b \cdot x
\]

where \( g \) is a function that links \( y \)'s mean to a linear combination of the predictor variables. When \( g \) is the identity function and residuals/errors are Normally distributed, we have a General Linear Model (GLM).

Several additional combinations of link functions and residual distributions are commonly used as shown in the table below.

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Response Type (y)</th>
<th>Link Function</th>
<th>Residual Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>binary {0,1}</td>
<td>logit</td>
<td>Bernoulli Distribution</td>
</tr>
<tr>
<td>Poisson Regression</td>
<td>integer {0,\ldots,\infty}</td>
<td>ln</td>
<td>Poisson Distribution</td>
</tr>
<tr>
<td>Exponential Regression</td>
<td>continuous [0,\infty)</td>
<td>ln or reciprocal</td>
<td>Exponential Distribution</td>
</tr>
<tr>
<td>General Linear Model (GLM)</td>
<td>continuous ((-\infty,\infty))</td>
<td>identity</td>
<td>Normal Distribution</td>
</tr>
</tbody>
</table>

Table 7.1: Types of Generalized Linear Models


Since the response variable for Logistic Regression is defined on finite domains, it has been placed under Classification (see the next chapter).

Example Problem:

Class Methods:
object GZLM extends GLM

def apply (x: MatriD, y: VectoI, cn: Array [String]): LogisticRegression =
def apply (x: MatriD, y: VectoI, fn: Array [String], poisson: Boolean): PoissonRegression =
def apply (x: MatriD, nonneg: Boolean, y: VectoD): ExpRegression =

7.0.1 Further Reading

1. Generalized Linear Models (GLM) [16]
7.1 Exponential Regression

The ExpRegression class can be used for developing Exponential Regression models. The response variable \( y \) is estimated by the product of a mean function and exponentially distributed residuals/errors \( \epsilon \).

\[
y = \mu(x) \epsilon
\]

The probability density function (pdf) for the Exponential distribution may be defined as follows:

\[
f(t; \lambda) = \lambda e^{-\lambda t}
\]

The link function \( g \) for Exponential Regression is the ln function (alternatively the reciprocal function).

\[
g(\mu(x)) = \ln(\mu(x)) = b \cdot x
\]

Expanding the dot product and using the inverse link function yields the following:

\[
\mu(x) = e^{b \cdot x} = e^{b_0 + b_1 x_1 + \ldots + b_k x_k}
\]

The residuals \( \epsilon_i = y_i / \mu(x_i) \) are distributed Exponential(1), so

\[
f(y_i / \mu(x_i)) = \frac{1}{\mu(x_i)} e^{-y_i / \mu(x_i)}
\]

Therefore, the likelihood function for Exponential Regression is as follows:

\[
L = \prod_{i=0}^{m-1} \frac{1}{\mu(x_i)} e^{-y_i / \mu(x_i)}
\]

Substituting for \( \mu(x_i) \) gives

\[
L = \prod_{i=0}^{m-1} e^{-b \cdot x_i} e^{-y_i / e^{b \cdot x_i}}
\]

Taking the natural logarithm gives the log-likelihood function.

\[
LL = \sum_{i=0}^{m-1} -b \cdot x_i - \frac{y_i}{e^{b \cdot x_i}}
\]

See [http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf](http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf) for more details.

**Example Problem:**

Class Methods:

- @param x the data/design matrix
- @param nonneg whether to check that responses are nonnegative
- @param y the response vector

class ExpRegression (x: MatriD, nonneg: Boolean, y: VectoD)
extends Predictor 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 ()
override def diagnose (yy: VectoD)
override def fit: VectoD = super.fit.asInstanceOf [VectorD] ++ VectorD (rBarSq, fStat, aic, bic)
override def fitLabels: Seq [String] = super.fitLabels ++ Seq ("rBarSq", "fStat", "aic", "bic")
def predict (z: VectoD): Double = exp (b dot z)
def test (n: Int = 10000, k: Int = 5): (Int, Int, VectoD, VectoD, Double) =
7.2 Poisson Regression

The PoissonRegression class can be used for developing Poisson Regression models. In this case, a response $y$ may be thought of as a count that may take on a nonnegative integer value. The probability density function (pdf) for the Poisson distribution with mean $\lambda$ may be defined as follows:

$$ f(y; \lambda) = \frac{\lambda^y}{y!} e^{-\lambda} $$

Again, treating this as a GZLM problem,

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

$$ g(\mu(x)) = b \cdot x $$

The link function $g$ for Poisson Regression is the ln (natural logarithm) function.

$$ \ln(\mu(x)) = b \cdot x $$

The residuals $\epsilon_i$ are distributed according to the Poisson distribution.

$$ \frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)} $$

Therefore, the likelihood function for Poisson Regression is as follows:

$$ L = \prod_{i=0}^{m-1} \frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)} $$

Taking the natural logarithm gives the log-likelihood function.

$$ LL = \sum_{i=0}^{m-1} y_i \ln(\mu(x_i)) - \mu(x_i) - \ln(y_i!) $$

Substituting $\mu(x_i) = e^{b \cdot x_i}$ yields the following:

$$ LL = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i} - \ln(y_i!) $$

Since the last term is independent of the parameters, removing it will not affect the optimization.

$$ LL_2 = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i} $$

See [http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf](http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf) for more details.

Example Problem:

Class Methods:
@param x the input/data matrix augmented with a first column of ones
@param y the integer response vector, $y_i \in \{0, 1, \ldots \}$
@param fn the names of the features/variable

class PoissonRegression (x: Matrd, 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

Non-Linear Models
8.1 Non-Linear Regression

The NonLinRegression class supports non-linear regression. In this case, \( x \) can be multi-dimensional \([1, x_1, \ldots, x_k]\) and the function \( f \) is non-linear in the parameters \( b \). As before, the goal is to fit the parameter vector \( b \) in the regression equation

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

where \( \epsilon \) represents the residuals (the part not explained by the model). Note that \( y = b_0 + b_1 x_1 + b_2 x_1^2 + \epsilon \) is still linear in the parameters. The example below is not, as there is no transformation that will make the formula linear in the parameters.

\[
y = (b_0 + b_1 x_1)/(b_2 + x_1) + \epsilon
\]

A training set consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector \( b \). Given an input matrix \( X \) consisting of \( m \) input vectors and an output vector \( y \) consisting of \( m \) output values, minimize the distance between the target output vector \( y \) and the predicted output vector \( f(X; b) \).

\[
\min_b \|y - f(X; b)\|
\]

Again, it is convenient to minimize the dot product of the error with itself,

\[
p(b) = (y - f(X; b)) \cdot (y - f(X; b))
\]

For non-linear regression, a Least-Squares (minimizing the residuals) method can be used to fit the parameter vector \( b \). Unlike the linear case (where one simply sets the gradient to zero), since the formula is non-linear in \( b \), Non-Linear Programming (NLP) is used to minimize the Sum of Squares Error (\( SS_E \)). A user defined function \( f: (\text{VectoD}, \text{VectoD}) \Rightarrow \text{Double} \) is passed as a class parameter. This function is used to create a predicted output value \( z_i \) for each input vector \( x_i \). The \texttt{sseF} method applies this function to all \( m \) input vectors to compute predicted output values. These are then subtracted from the target output to create an error vector \( e \), which when dot producted with itself yield \( SS_E \).

```scala
def sseF (b: 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 \texttt{minima} and \texttt{maxima} packages provide several solvers for linear, quadratic, integer and non-linear programming. Currently, the \texttt{QuasiNewton} class is used for finding an optimal \( b \) by minimizing \texttt{sseF}. The \texttt{QuasiNewton} optimizer requires an initial guess for the parameter vector \( b \).

```scala
val bfgs = new QuasiNewton (sseF) // minimize sse using NLP
b = bfgs.solve (b_init) // estimate for b from optimizer
```

142
Example Problem:

**Class Methods:**

@param x the input/data matrix augmented with a first column of ones
@param y the response vector
@param f the non-linear function $f(x, b)$ to fit
@param b_init the initial guess for the parameter vector $b$

```scala
class NonLinRegression (x: MatrixD, y: VectorD,
                          f: (VectorD, VectorD) => Double,
                          b_init: VectorD)
  extends Predictor with Error

def sseF (b: VectorD): Double =
def train ()
def train (yy: VectorD)
def fit: (VectorD, Double) = (b, rSquared)
def predict (z: VectorD): Double = f(z, b)
def predict (z: MatrixD): VectorD =
```
8.2 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).

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 the 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 $f$.

$$ y = f(\mathbf{b} \cdot \mathbf{x}) + \epsilon = f(\sum_{j=0}^{n-1} b_j x_j) + \epsilon $$

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 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 target output vector $\mathbf{y}$ and the predicted output vector $\mathbf{yp} = f(X\mathbf{b})$.

$$ \min_{\mathbf{b}} \| \mathbf{y} - f(X\mathbf{b}) \| $$

As was the case with regression, it is convenient to minimize the dot product of the error with itself. In particular, we aim to minimize half of this value, half $sse$ (hsse).

$$ hsse(\mathbf{b}) = \frac{1}{2} (\mathbf{y} - f(X\mathbf{b})) \cdot (\mathbf{y} - f(X\mathbf{b})) \quad (8.1) $$

8.2.1 Optimization

Optimization for Perceptrons and Neural Networks is typically done using an iterative optimization algorithm that utilizes gradients. Taking the partial derivative with respect to the $j^{th}$ weight, $b_j$, is a bit complicated since we need to use the chain rule and the product rule. First, letting $\mathbf{u} = X\mathbf{b}$ allows equation 7.1 to be simplified to

$$ hsse(\mathbf{b}) = \frac{1}{2} (\mathbf{y} - f(\mathbf{u})) \cdot (\mathbf{y} - f(\mathbf{u})) $$

The chain rule from vector calculus to be applied is
These two partial derivatives are
\[
\frac{\partial hsse}{\partial b_j} = \frac{\partial hsse}{\partial u} \cdot \frac{\partial u}{\partial b_j}
\]
where \( x_{-j} \) is the \( j^{th} \) column of matrix \( X \). The dot product of the two partial derivatives gives
\[
\frac{\partial hsse}{\partial b_j} = -x_{-j} \cdot f'(Xb)(y - f(Xb))
\]
Since the error vector \( \epsilon = y - f(Xb) \), we may simplify the expression.
\[
\frac{\partial hsse}{\partial b_j} = -x_{-j} \cdot f'(Xb) \epsilon
\]
It is helpful especially for multi-layer neural networks to define the delta vector \( \delta \) as follows:
\[
\delta = \frac{\partial hsse}{\partial u} = -f'(Xb) \epsilon
\]
The partial derivative of \( hsse \) with respect to \( b_j \) now simplifies to
\[
\frac{\partial hsse}{\partial b_j} = x_{-j} \cdot \delta
\]
Note, if we consider a single instance \( (x_i, y_i) \), equation 7.2 becomes
\[
\frac{\partial hsse}{\partial b_j} = -x_{ij} f'(x_i \cdot b) \epsilon_i = x_{ij} \delta_j
\]
Combining the partial derivatives in equation 7.1 into an \( n \)-dimensional vector (i.e., the gradient) yields
\[
\frac{\partial hsse}{\partial b} = -X^t[f'(Xb) \epsilon] = X^t \delta
\]
Since many optimizers such as gradient-descent, move in the direction opposite to the gradient by a distance governed by the learning rate \( \eta \) (alternatively step size), the following term should be added to the weight/parameter vector \( b \).
\[
X^t [f'(Xb) \epsilon] \eta = -X^t \delta \eta
\]
The right hand side is an \( n \)-by-\( m \) matrix, \( m \) vector product yielding an \( n \) vector result. The factor in brackets, \( [f'(Xb) \epsilon] \), is the elementwise vector product.

8.2.2 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 is often limited. The simplest activation function is the identity function where the signal is passed through unmodified. In this case, Perceptron is in alignment with Regression. This activation function is usually not intended for neural nets with more layers, since theoretically they can be reduced to a two layer network (although it may be applied in the last layer). More generally useful activation functions include sigmoid and tanh. For these activation functions
<table>
<thead>
<tr>
<th>Name</th>
<th>Function $u = f(t)$</th>
<th>Domain</th>
<th>Range</th>
<th>Derivative $f'(t)$</th>
<th>Inverse $t = f^{-1}(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity</td>
<td>$t$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>1</td>
<td>$u$</td>
</tr>
<tr>
<td>sigmoid</td>
<td>$[1 + e^{-t}]^{-1}$</td>
<td>$\mathbb{R}$</td>
<td>$(0, 1)$</td>
<td>$f(t)[1 - f(t)]$</td>
<td>$-\ln(\frac{1}{u} - 1)$</td>
</tr>
<tr>
<td>tanh</td>
<td>$\tanh(t)$</td>
<td>$\mathbb{R}$</td>
<td>$(-1, 1)$</td>
<td>$1 - f(t)^2$</td>
<td>$.5 \ln(\frac{1+u}{1-u})$</td>
</tr>
</tbody>
</table>

Table 8.1: Activation Functions

the outputs, the $y$ vector, needs to be transformed into the range specified for the activation function, see Table 7.1. It may be also useful to transform/standardize the inputs.

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 7.4 becomes

$$\frac{\partial hsse}{\partial b} = -X^t[f(Xb)[1 - f(Xb)]\epsilon]$$

A simple form of gradient-descent iteratively moves in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate $\eta$. Therefore, the parameter/weight vector $b$ is adjusted as follows:

$$b = b + X^t[f(Xb)[1 - f(Xb)]\epsilon] \eta$$

The iterative process is typically terminated when the drop in $hsse$ is small or a maximum number of iterations is exceeded. The parameters $\eta$ and $\text{MAX\_ITER}$ need careful adjustment to obtain nearly (locally) optimal values for $hsse$. Gradient-descent works by iteratively moving in the opposite direction as the gradient until the error changes fall below a threshold. The rate of convergence can be adjusted using the learning rate $\eta$ which multiplies the gradient. Setting it too low, slows convergence, while setting it too high can cause oscillation. In ScalaTion, $\eta$ defaults to 0.1. The train method contains the main training loop and is shown below.

```scala
def train (yy: VectoD = y): Perceptron =
{
  if (b == null) setWeights ()
  var sse0 = Double.MaxValue
  for (it <- 0 until MAX\_ITER) {
    val yp = f1V (x * b)
    val e = y - yp
    // b += x.t * (e * yp * (_1 - yp)) * eta // sigmoid case
    b += x.t * (e * faDV (yp)) * eta
    val sse = e dot e
    if (DEBUG) println (s"weights for $it th phase: b = $b, sse = $sse")
    if (sse0 - sse < EPSILON) return this
    sse0 = sse
  } // for
  this
} // train
```
A perceptron can be considered to be a special type of non-linear or transformed regression.

This class defaults to the sigmoid activation functions, one for the function and one for the derivative, which are defined in the ActivationFun object. Several other types of activation functions are available in the ActivationFun object.

Class Methods:

@param x the m-by-n input matrix (training data consisting of m input vectors)
@param y the output vector (training data consisting of m output values)
@param eta the learning/convergence rate (requires adjustment)
@param f1 the activation function (mapping scalar => scalar)
@param f1DV the derivative of the vector activation function

class Perceptron (x: MatriD, y: VectoD, private var eta: Double = 0.1,
    f1: FunctionS2S = sigmoid _,
    f1DV: FunctionV_2V = sigmoidDV _)
    extends PredictorMat (x, y)

def setWeights (w0: VectoD) { b = w0 }
def setWeights (ymin: Double = 0.0, ymax: Double = 1.0, stream: Int = 0)
def reset (eta_: Double) { eta = eta_ }
def train (yy: VectoD = y): Perceptron =
    override def eval ()
    override def predict (z: VectoD): Double = f1 (b dot z)
    override def predict (z: MatriD): VectoD = f1V (z * b)
def crossVal (k: Int = 10)

8.2.3 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
    // 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
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 ann = new Perceptron (x, y)
ann.train ().eval ()
println ("coefficient = " + ann.coefficient)
println ("full mod fit = " + ann.fitMap)

val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println ("predict (" + z + ") = " + ann.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. Show that Perceptron with activation function $f$ is the same as TranRegression with tranform $f^{-1}$. Explain any differences in the parameter/weight vector $b$ and the sum of squared errors $sse$. 
8.3 Neural Networks

The \texttt{NeuralNet} abstract class provides the basic structure and API for a variety of Neural Networks.

Class Methods:

\begin{verbatim}
@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 eta the learning/convergence rate (typically less than 1.0)

abstract class NeuralNet (x: MatriD, y: MatriD, private var eta: Double = 0.1)
    extends Predictor with Error

    def weights: Array [MatriD]
    def setWeights (stream: Int = 0, limit: Double = 1.0 / sqrt (nx))
    def reset (eta_: Double) { eta = eta_ }
    def train (): NeuralNet
    def train (yy: VectoD): NeuralNet = train ()
    def eval ()
    def eval (xx: MatriD, yy: MatriD)
    def fitLabel: Seq [String] = fitA(0).fitLabel
    def fitMap ()
    def predict (z: VectoD): Double = predictV (z)(0)
    def predictV (z: VectoD): VectoD
    def predict (z: MatriD): MatriD
    def crossValidate (algor: (MatriD, MatriD) => NeuralNet, k: Int = 10): Array [Statistic] =
    def crossVal (k: Int = 10)
\end{verbatim}
8.4 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.

The weights on the edges are analogous to the parameter vector \( \mathbf{b} \) in regression. Each output \( y_k \), has its own parameter vector \( \mathbf{b}_k \). These are collected as column vectors into a parameter matrix \( \mathbf{B} \), 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 net can be used to predict the corresponding output vector \( \mathbf{y} \). The network predicts an output value \( y_j \) by taking the weighted sum of it inputs and passing this sum through an activation function \( f \).

\[
y_k = f\left(\sum_{j=0}^{n_x-1} b_{jk} x_j\right) + \epsilon_k
\]

This can written in vector form as follows:

\[
\mathbf{y} = f(\mathbf{B} \cdot \mathbf{x}) + \mathbf{\epsilon} = f(\mathbf{B}^\top \mathbf{x}) + \mathbf{\epsilon}
\]

Given several input vectors and output vectors in a training dataset \((i = 0, \ldots, m-1)\), the goal is to optimize/fit the parameters/weights \( \mathbf{B} \). The training dataset consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight matrix \( \mathbf{B} \). Given an input matrix \( \mathbf{X} \) consisting of \( m \) input vectors and an output matrix \( \mathbf{Y} \) consisting of \( m \) output vectors, minimize the distance between the target output matrix \( \mathbf{Y} \) and the predicted output vector \( f(X\mathbf{B}) \)

\[
\min_\mathbf{B} \| \mathbf{Y} - f(X\mathbf{B}) \|_F
\]

where \( \| \cdot \|_F \) is the Frobenius norm, \( \mathbf{X} \) is \( m \)-by-\( n_x \), \( \mathbf{Y} \) is \( m \)-by-\( n_y \), and \( \mathbf{B} \) is \( n_x \)-by-\( n_y \). Other norms may be used as well, but the square of the Frobenius norm will give the overall sum of squared errors \( \text{sse} \).

8.4.1 Optimization

As was the case with regression, it is convenient to minimize the dot product of the error with itself. We do this for each of the columns of the \( \mathbf{Y} \) matrix to get the \( \text{sse} \) for each \( y_k \) and sum them up. The goal then is to simply minimize the objective function \( \text{sse}(\mathbf{B}) \). As in the last section, we work with half of the sum of squared errors \( \text{sse} \) (or \( \text{hsse} \)).

\[
\text{hsse}(\mathbf{B}) = \frac{1}{2} \sum_{k=0}^{n_y-1} (\mathbf{y}_k - f(X\mathbf{b}_k)) \cdot (\mathbf{y}_k - f(X\mathbf{b}_k))
\]

This nonlinear optimization problem may be solved by a variety of optimization techniques, including Gradient-Descent, Stochastic Gradient Descent or Quasi-Newton optimizers.

Most optimizers require a derivative and ideally these should be provided in functional form (otherwise the optimizer will need to numerically approximate them). For the sigmoid activation function,
sigmoid(t) = \frac{1}{1 + e^{-t}}

the derivative is

sigmoid(t)[1 - sigmoid(t)]

To minimize the objective function given in equation 7.6, we decompose it into ny functions.

\[ h\text{sse}_j(b_k) = \frac{1}{2} (y_k - f(Xb_k)) \cdot (y_k - f(Xb_k)) \]

Notice that this is the same equation as 7.1, just with subscripts on y and b.

In Regression, we took the gradient and set it equal to zero. Here, gradients will need to be computed by the optimizer. The gradient will be the same as given in equation 7.4, just with subscripts added.

\[ \frac{\partial h\text{sse}}{\partial b_k} = - X^t [f'(Xb_k) \epsilon_k] = X^t \delta_k \]  

(8.7)

For the sigmoid function, 

\[ f'(Xb_k) = f(Xb_k)[1 - f(Xb_k)] \]

so

\[ \frac{\partial h\text{sse}}{\partial b_k} = - X^t [f(Xb_k)[1 - f(Xb_k)] \epsilon_k] \]

Again, moving in the direction opposite to the gradient by a distance governed by the learning rate \( \eta \) the following term should be added to the weight/parameter vector \( b_k \).

\[ X^t [f'(Xb_k) \epsilon_k] \eta = - X^t \delta_k \eta \]  

(8.8)

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 eta the learning/convergence rate (typically less than 1.0)
@param f1 the activation function (mapping scalar => scalar)
@param f1D the derivative of the vector activation function

class NeuralNet_2L (x: Matrix, y: Matrix, eta: Double = 0.1,
                 f1: FunctionS2S = sigmoid _,
                 f1D: FunctionV_2V = sigmoidDV _)
    extends NeuralNet (x, y, eta)

    def weights: Array [Matrix] = Array (bb)
    def setWeights (stream: Int = 0, limit: Double = 1.0 / sqrt (nx))
    def train (): NeuralNet_2L =
    def predictV (z: Vector): Vector = f1V (bb dot z)
    def predict (z: Matrix): Matrix = f1M (z * bb)
    def crossVal (k: Int = 10)
8.5 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 $x$, while the outputs are given by the output vector $y$. Between these two layers is a single hidden layer, whose intermediate values will be denoted by the vector $z$. Each input $x_j$ is associated with an input node in the network, while each output $y_k$ is associated with an output node in the network. The input layer consists of $n_x$ input nodes, the hidden layer consists of $n_z$ hidden nodes, and the output layer consists of $n_y$ output nodes. There are two sets of edges. Edges in the first set connect each input node with each hidden node, i.e., there are $n_x n_z$ such edges in the network. 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. 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. Similarly, the hidden layer will also need such a special node.

Training involves an interactive procedure (e.g., gradient descent) that adjust weights to minimize an objective function such as $sse$ or in this section half $sse$ (or $hsse$). Before the main loop, random weights need to be assigned to the first set of edges $A = [a_{jh}]_{n_x \times n_z}$ and the second set of edges $A = [a_{hk}]_{n_z \times n_y}$. Roughly as outlined in section 3 of [11], the `minimizeError` method can be broken into four steps:

1. Compute predicted values for output $yp$ and compare with actual values $y$ to determine the error $y - yp$.
2. Back propagate the error to determine the amount of correction needed at the output layer. Record this as vector $\delta^y$.
3. Back propagate the correction to the hidden layer and determine the amount of correction needed at the hidden layer. Record this as vector $\delta^z$.
4. Use the delta vectors, $\delta^y$ and $\delta^z$, to makes updates to weight matrices $A$ and $B$.

8.5.1 Optimization

In this subsection, the basic elements of the backpropagation algorithm are presented. In particular, we now go over the four steps outlined above in more detail.

1. Compute predicted values: Based on the randomly assigned weights to the $A$ and $B$ matrices, predicted outputs $yp$ are calculated. First values for the hidden layer $z$ are calculated, where the values for hidden node $h$, $z_h$, is given by

$$z_h = f_1(a_{-h} \cdot x) \quad \text{for } h = 0, \ldots, n_z - 1$$

where $f_1$ is the first activation function (e.g., sigmoid), $a_{-h}$ is column-$h$ of the $A$ weight matrix, and $x$ is the input vector for a training sample (row in the data matrix). (One may utilize multiple samples in each step or iterate each step over the samples.) Next, the values computed at the hidden layer are used to produce predicted outputs $yp$, where the value for output node $k$, $yp_k$, is given by

$$yp_k = f_2(b_{-k} \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1$$
where the second activation function $f_2$ may be the same as (or different from) the one used in the hidden layer and $b_{-k}$ is column-$k$ of the $B$ weight matrix. Now the difference between the actual and predicted output can be calculated by simply subtracting the two vectors, or elementwise, the error for the $k^{th}$ output, $\epsilon_k$, is given by

$$\epsilon_k = y_k - y_{pk} \quad \text{for} \quad k = 0, \ldots, n_y - 1$$

Obviously, for subsequent iterations, the updated/corrected weights rather than the initial random weights are used.

2. Back propagate to output layer: Given the computed error vector $\epsilon$, the delta/correction vector $\delta^y$ for the output layer may be calculated, where for output node $k$, $\delta_k^y$ is given by

$$\delta_k^y = -f'_2(b_{-k} \cdot z) \epsilon_k \quad \text{for} \quad k = 0, \ldots, n_y - 1 \quad (8.9)$$

where $f'_2$ is the derivative of the activation function (e.g., for sigmoid, $f'(t) = f(t)[1 - f(t)]$). The partial derivative of $hsse$ with respect to the weight connecting hidden node $h$ with output node $k$, $b_{hk}$, is given by

$$\frac{\partial hsse}{\partial b_{hk}} = z_h \delta_k^y \quad (8.10)$$

3. Back propagate to hidden layer: Given the delta/correction vector $\delta^y$ from the output layer, the delta vector for the hidden layer $\delta^z$ may be calculated, where for hidden node $h$, $\delta^z_h$ is given by

$$\delta^z_h = f'_1(a_{-h} \cdot x)[b_h \cdot \delta^y] \quad \text{for} \quad h = 0, \ldots, n_z - 1 \quad (8.11)$$

This equation is parallel to the one given for $\delta^y_k$ in that an error-like factor multiplies the derivative of the activation function. In this case, the error-like factor is the weighted average of the $\delta^y_k$ for output nodes connected to hidden node $h$ times row-$h$ of weight matrix $B$. The weighted average is computed using the dot product.

$$b_h \cdot \delta^y = \sum_{k=0}^{n_y-1} b_{hk} \delta_k^y$$

The partial derivative of $hsse$ with respect to the weight connecting input node $j$ with hidden node $h$, $a_{jh}$, is given by

$$\frac{\partial hsse}{\partial a_{jh}} = x_j \delta^z_h \quad (8.12)$$

4. Update weights: The weight matrices $A$ and $B$, connecting input to hidden and hidden to output layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate $\eta$ which moderates the adjustments to the weights.
\[ b_{hk} = b_{hk} - z_h \delta_k^y \eta \quad (8.13) \]

\[ a_{jh} = a_{jh} - x_j \delta_h^z \eta \quad (8.14) \]

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 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 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 \( hsse \) has been sufficiently small for \( q \) iterations.

### 8.5.2 Matrix Version

Given a training dataset consisting of an \( m \)-by-\( n_x \) input data matrix \( X \) and an \( m \)-by-\( n_y \) output data matrix \( Y \), the optimization equations may be re-written as shown below.

The optimizer 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 inner loop, divides the training dataset into \( n_{Bat} \) \( batches \). A batch is a randomly selected group/batch of rows. Each batch \((ib)\) is passed to the \texttt{updateWeights} \((x(ib), y(ib))\) method that updates the \( A \) and \( B \) parameter/weight matrices.

The \texttt{updateWeights} method computes predicted outputs taking their difference with actual/target values to compute a negative 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 activation function \( f_1 \) to the matrix product \( XA \).

\[ Z = f_1(XA) \]

2. The predicted output values \( Y_p \) are similarly computed by applying the second activation function \( f_2 \) to the matrix product \( ZB \).

\[ Y_p = f_2(ZB) \]

3. An error matrix \( E_n \), the opposite of the usually calculated error, is just the difference between the predicted and actual values.

\[ E_n = Y_p - Y \]

4. This information is sufficient to calculate delta matrices \( \Delta^y \) and \( \Delta^z \).

\[ \Delta^y = f'_2(ZB)E_n \]

154
\[ \Delta z = f'_1(XA) \Delta y B^t \]

5. As mentioned, the delta matrices form the basis (a matrix transpose $\times$ delta $\times$ the learning rate $\eta$) for updating the parameter/weight matrices, $A$ and $B$.

\[ B = B - Z^t \Delta y \eta \]
\[ A = A - X^t \Delta z \eta \]

The corresponding SCALATION code for the `updateWeights` method is show below. It returns the $\text{sse}$ for the given batch. Note: $f1M$ is the matrix version of the first activation function, while $f1D$ is its derivative.

```scala
private def updateWeights (x: MatriD, y: MatriD): Double = {
  val z = f1M (x * aa); /* z.setCol (0, _1) */
  val yp = f2M (z * bb)
  val en = yp - y
  val dy = f2D (yp) ** en
  val dz = f1D (z) ** (dy * bb.t)

  bb -= z.t * dy * eta
  aa -= x.t * dz * eta
  en.normF ~^ 2
}
```

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 `ExampleConcrete` 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 $\text{id}$. If this is not done, response columns need to be re-scaled based on the training dataset. Since the testing dataset may have values outside this range, this approach may not be ideal.

For classification problems, it is common to have an output node for each response value for the categorical variable, e.g., “no”, “yes” would have $y_0$ and $y_1$, while “red”, “green”, “blue” would have $y_0$, $y_1$ and $y_2$. The softmax activation function is a common choice to the last layer for classification problems.

\[ f_i(t) = \frac{e^{t_i}}{1 + e^t} \text{ for } i = 0, \ldots n - 1 \]

**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 eta the learning/convergence rate (typically less than 1.0)
@param f1 the input-hidden layer activation function (mapping scalar => scalar)
@param f1D the derivative of the vector activation function

class NeuralNet_3L (x: MatriD, y: MatriD, private var nz: Int = -1,
    var eta: Double = 0.1, bsize: Int = 5,
    f1: FunctionS2S = sigmoid _,
    f1D: FunctionM_2M = sigmoidDM,
    f2: FunctionS2S = sigmoid _,
    f2D: FunctionM_2M = sigmoidDM)
    extends NeuralNet (x, y, eta)

def weights: Array [MatriD] = Array (aa, bb)
def setWeights (stream: Int = 0, limit: Double = 1.0 / sqrt (nx))
def train (): NeuralNet_3L =
def predictV (v: VectoD): VectoD = f2V (bb * f1V (aa * v))
def predict (x: MatriD): MatriD = f2M (f1M (x * aa) * bb)
def crossVal (k: Int = 10)
8.6 Multi-Hidden Layer Neural Networks

The `NeuralNetXL` class supports basic x-layer (input, \{hidden\} and output) Neural Networks. Given several input and output vectors (training data), fit the weights connecting the layers, so that for a new input vector \( z^i \), the net can predict the output vector \( z^o \) (\( z^h \) is the intermediate value at the hidden layer), i.e.,

\[
z^i \rightarrow z^h = f(W^i z^i) \rightarrow z^o = g(V^i z^h)
\]

A 3-layer Neural Net consists of several Perceptrons arranged into the hidden and output layers (the first layer is just for input - no Perceptrons are needed). As mentioned, a Perceptron can be thought of a special type of Nonlinear Regression. Neural Nets, however, have additional modeling power. Using sigmoid as the activation function, the predicted output \( z^o \) is computed using two matrix-vector multiplications and two applications of the activation function.

\[
z^o = sigmoid(V^i sigmoid(W^i z^i))
\]

With a three layer network (input, hidden and output layers), the intermediate vector \( z^h \) is calculated as the product of a weight matrix \( W \) transposed and the input vector \( z^i \) passed through the activation function. The response/output \( z^o \) is computed similarly using a second weight matrix \( V \). The \( W = [w_{jk}] \) matrix indicates the strength of the weight between input \( z^i_j \) and hidden \( z^h_k \). The first column in the weight matrix corresponds to the bias (just like the constant term in regression). Therefore, \( z^i_0 \) must be set to 1.0. The \( V \) matrix plays the same role between the hidden and output layers.

One common approach to optimize the \( W \) and \( V \) weights is to use backpropagation (see the ScalaTion code for details).

---

Example Problem:

Class Methods:

```scala
@param x the input matrix (training data consisting of m input vectors)
@param y the output matrix (training data consisting of m output vectors)
@param h the number of neurons in the hidden layer
@param eta the learning/convergence rate
class NeuralNet (x: MatrixD, y: MatrixD, h: Int, eta: Double = 1.0)
  extends Predictor with Error

  def setWeights (w0: MatrixD, v0: MatrixD) { w = w0; v = v0 }
  def setWeights (i: Int = 0)
  def train () { if (w == null) setWeights (); backProp () }
  def backProp ()
  def fit: (MatrixD, MatrixD) = (w, v)
  def predictAll (zi: VectorD): VectorD = sigmoid (v.t * sigmoid (w.t * zi))
  def predict (zi: VectorD): Double = predictAll (zi)(0)
  def predictAll (zi: MatrixD): MatrixD =
```

---

157
def predict (zi: MatriD): VectoD = predictAll (zi)(0)
Chapter 9

Temporal Models
9.1 State Space Models

The next state vector $x_{t+1}$ is dependent on the current state $x_t$ and a noise vector $v_t$.

$$x_{t+1} = Fx_t + Gv_t$$

At time $t$, the observation vector $y_t$ is dependent on the current state $x_t$ and a second noise vector $w_t$.

$$y_t = Hx_t + w_t$$

The transition/coefficient matrices, F, G and H indicate
9.2 Exponential Smoothing
9.3 Auto-Regressive (AR) Models
9.4 Moving-Average (MA) Models
9.5 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 the last \(p\) values each weighted by its own coefficient/parameter \(\phi_j\). The error/noise is represented by \(\epsilon_t\).

\[
y_t = \mu + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t
\]

The code in SCALA\(T\)ION works with zero mean data \(y_t\) where the mean \(\mu\) has been subtracted from the original data. The coefficients \(\phi\) (phi) are estimated using the Durbin-Levinson algorithm and extracted from the last row of the \(\psi\) (psi) matrix.

```scala
def durbinLevinson: MatrixD =
{
  val psi = new MatrixD (m+1, m+1)
  val r   = new VectorD (m+1); r(0) = c(0)
  for (t <- 1 to m) {
    var sum = 0.
    for (j <- 1 until t) sum += psi(t-1, j) * c(t-j)
    val a = (c(t) - sum) / r(t-1)
    psi(t, t) = a
    for (j <- 1 until t) psi(t, j) = psi(t-1, j) - a * psi(t-1, t-j)
    r(t) = r(t-1) * (1. - a * a)
  } // for
  pacf = psi.getDiag.slice (1, m+1) // PACF is the diagonal
  psi   // return the psi matrix
} // durbinLevinson
```

After these coefficients are estimated, the AR\((p)\) model can be used for forecasting.

```scala
def ar (phi: VectorD): VectorD =
{
  val p = phi.dim // order p for AR(p) model
  val f = new VectorD (n) // forecasts for x
  for (t <- p until n) { // start at t = p (enough data)
    var sum = 0.
    for (j <- 0 until p) sum += phi(j) * x(t-1-j)
    f(t) = sum
  } // for
  f // return the vector of forecasts
} // ar
```

A \(q\)th-order Moving Average MA\((q)\) model predicts the next value \(y_t\) from the combined effects of prior noise/disturbances.
\[ t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q} \]

There are multiple ways to combine multiple regression with time series analysis. One common technique called Time Series Regression is to use multiple linear regression and model its residuals using ARMA models.

---

**Example Problem:**

**Class Methods:**

@param y the input vector (time series data)
@param t the time vector

class ARMA (y: VectoD, t: VectoD) extends Predictor with Error

def est_ar (p: Int = 1): VectoD =
def durbinLevinson: MatriD =
def ar (phi: VectoD): VectoD =
def est_ma (q: Int = 1): VectoD =
def ma (theta: VectoD): VectoD =
def train ()
def predict (y: VectoD): Double =
def predict (z: MatriD): VectoD =
def plotFunc (fVec: VectoD, name: String)
def smooth (l: Int): VectoD =

---
9.7 SARIMA
9.8 Recurrent Neural Networks (RNN)

9.8.1 Long Short Term Memory (LSTM) Networks
9.9 Temporal Convolutional Networks (TCN)
9.10 ODE Parameter Estimation

\[ y = x(t) + \epsilon \]
\[ \frac{dx(t)}{dt} = f(x(t); b) \]

9.10.1 Non-Linear Least Squares (NLS)

9.10.2 Least Squares Approximation (LSA)
Chapter 10

Spatial Models
10.1 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 distance metric (e.g., Euclidean or Manhattan distance). Points in a cluster are according to some metric closer to each other than to points not in their cluster.
11.1 Clusterer

The Clusterer trait provides a common framework for several clustering algorithms.

Trait Methods:

```scala
trait Clusterer {
  def cluster (): Array[Int]
  def classify (y: VectoD): Int
  def name_ (n: Array[String])
  def getName (i: Int): String =
}
```
11.2 K-Means Clustering

The `KMeansClustering` class clusters several vectors/points using k-means clustering. Either (1) randomly assign points to $k$ clusters or (2) randomly pick $k$ points as initial centroids (technique (1) to work better and is the primary technique). Iteratively, reassign each point to the cluster containing the closest centroid. Stop when there are no changes to the clusters.

Example Problem:

Class Methods:

- `@param x` the vectors/points to be clustered stored as rows of a matrix
- `@param k` the number of clusters to make
- `@param s` the random number stream (to vary the clusters made)
- `@param primary` true indicates use the primary technique for initiating the clustering

```scala
class KMeansClustering (x: Matrix, k: Int, s: Int = 0, primary: Boolean = true)
  extends Clusterer with Error

  def distance (u: Vector, v: Vector): Double =
  def assign ()
  def reassign (): Boolean =
  def pickCentroids ()
  def calcCentroids ()
  def cluster (): Array[Int] =
  def classify (y: Vector): Int =
```
11.3 Hierarchical Clustering

Cluster several vectors/points using hierarchical clustering. Start with each point forming its own cluster and merge clusters until there are only \( k \).

---

Example Problem:

---

Class Methods:

@param \( x \) the vectors/points to be clustered stored as rows of a matrix
@param \( k \) stop when the number of clusters equals \( k \)

```java
class HierClustering (x: MatrixD, k: Int = 2)
    extends Clusterer with Error

    def distance (u: VectorD, v: VectorD): Double =
    def clustDist (setA: Set [VectorD], setB: Set [VectorD]): Double =
    def initClusters ()
    def cluster (): Array [Int] =
    def finalClusters ()
    def calcCentroids ()
    def classify (y: VectorD): Int =
```
11.4 Markov Clustering

The **MarkovClustering** class implements a Markov Clustering Algorithm (MCL) and is used to cluster nodes in a graph. The graph is represented as an edge-weighted adjacency matrix (a non-zero cell indicates nodes i and j are connected).

The primary constructor takes either a graph (adjacency matrix) or a Markov transition matrix as input. If a graph is passed in, the normalize method must be called to convert it into a Markov transition matrix. Before normalizing, it may be helpful to add self loops to the graph. The matrix (graph or transition) may be either dense or sparse. See the MarkovClusteringTest object at the bottom of the file for examples.

---

**Example Problem:**

---

**Class Methods:**

@param t either an adjacency matrix of a graph or a Markov transition matrix
@param k the strength of expansion
@param r the strength of inflation

class MarkovClustering (t: Matrix, k: Int = 2, r: Double = 2.0)
    extends Clusterer with Error

    def addSelfLoops (weight: Double = 1.0)
    def normalize ()
    def expand ()
    def inflate (): Boolean =
    def processMatrix (): Matrix =
    def cluster (): Array [Int] =
    def classify (y: Vector): Int =

---
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 (): Matrix

def recover (): Matrix
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 Analysis
13.4 Functional Regression
Chapter 14

Optimization
14.1 Ensemble Learning
Chapter 15

Optimization

Many optimization problems may be formulated as restricted forms of the following,

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad g(x) \leq 0 \\
& \quad h(x) = 0
\end{align*}
\]

where \( f(x) \) is the objective function, \( g(x) \leq 0 \) are the inequality constraints, and \( h(x) = 0 \) are the equality constraints. Consider the example below.

\[
\begin{align*}
\text{minimize} & \quad f(x) = (x_1 - 4)^2 + (x_2 - 2)^2 \\
\text{subject to} & \quad g(x) = [x_1 - 3, x_2 - 1] \leq 0 \\
& \quad h(x) = x_1 - x_2 = 0
\end{align*}
\]

If we ignore all the constraints, the optimal solution is \( x = [4, 2] \) where \( f(x) = 0 \), while enforcing the inequality constraints makes this solution infeasible. The new optimal solution is \( x = [3, 1] \) where \( f(x) = 2 \). Finally, the optimal solution when all constraints are enforced is \( x = [1, 1] \) where \( f(x) = 10 \). Note, for this example there is just one equality constraint that forces \( x_1 = x_2 \).
15.1 Method of Lagrange Multipliers

The Method of Lagrange Multipliers provides a means for solving constrained optimizations problems. For optimization problems involving only one equality constraint, one may introduce a Lagrange multiplier $\lambda$. At optimality, the gradient of $f$ should be orthogonal to the surface defined by the constraint $h(x) = 0$, otherwise, moving along the surface in the opposite direction to the gradient ($-\nabla f(x)$ for minimization) would improve the solution. Since the gradient of $h$, $\nabla h(x)$, is orthogonal to the surface as well, this implies that the two gradients should only differ by a constant multiplier $\lambda$.

$$-\nabla f(x) = \lambda \nabla h(x)$$

Computation of the gradients for the above example yields the first two equations,

$$-2(x_1 - 4) = \lambda$$
$$-2(x_2 - 2) = -\lambda$$
$$x_1 - x_2 = 0$$

while the third equation is simply the constraint itself $h(x) = 0$, which may be rewritten,

$$2x_1 + \lambda = 8$$
$$2x_2 - \lambda = 4$$
$$x_1 - x_2 = 0$$

This is a linear system of equations with 3 variables $[x_1, x_2, \lambda]$ and 3 equations that may be solved, for example, by LU Factorization. In this case, the last equation gives $x_1 = x_2$, so adding equations 1 and 2 yields $4x_1 = 12$. Therefore, the optimal value is $x = [3, 3]$ with $\lambda = 2$ where $f(x) = 2$.

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 parallel gradients and one for each of the two constraints. In general, the Lagrange multipliers may be collected into a vector $\lambda$.

$$- \nabla f(x) = \lambda \cdot \nabla h(x) \quad (15.1)$$

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.
15.2 Karush-Kuhn-Tucker Conditions

Introducing inequality constraints makes the situation a little more complicated. A generalization of the Method of Lagrange Multipliers based on the Karush-Kuhn-Tucker (KKT) conditions is needed. For minimization, the KKT conditions are as follows:

\[-\nabla f(x) = \alpha \cdot \nabla g(x) + \lambda \cdot \nabla h(x)\] (15.2)

The original constraints must also hold.

\[g(x) \leq 0 \quad \text{and} \quad h(x) = 0\]

Furthermore, the Lagrange multipliers for the inequality constraints \(\alpha\) are themselves constrained to be nonnegative.

\[\alpha \geq 0\]

When the objective function is at most quadratic and the constraints are linear, the problem of finding an optimal value for \(x\) is referred to a Quadratic Programming. Many estimation/learning problems in data science are of this form. Beyond Quadratic Programming lies problems in Nonlinear Programming. Linear Programming (linear objective function and linear constraints) typically finds less use (e.g., Quantile Regression) in estimation/learning, so it will not be covered in this Appendix, although it is provided by ScalaTion.

15.2.1 Active and Inactive Constraints
15.3 Quadratic Programming

The QuadraticSimplex class solves Quadratic Programming (QP) problems using the Quadratic Simplex Algorithm. Given a constraint matrix $A$, constant vector $b$, cost matrix $Q$ and cost vector $c$, find values for the solution/decision vector $x$ that minimize the objective function $f(x)$, while satisfying all of the constraints, i.e.,

$$
\begin{align*}
\text{minimize } & \quad f(x) = \frac{1}{2}x \cdot Qx + c \cdot x \\
\text{subject to } & \quad g(x) = Ax - b \leq 0
\end{align*}
$$

Before considering the type of optimization algorithm to use, we may simplify the problem by applying the KKT conditions.

$$
-\nabla f(x) = Qx + c = \alpha \cdot \nabla g(x) = \alpha \cdot A
$$

Adding the constraints gives the following $n$ equations and $2m$ constraints:

$$
Qx + c = \alpha \cdot A \\
Ax - b \leq 0 \\
\alpha \geq 0
$$

These equations have two unknown vectors, $x$ of dimension $n$ and $\alpha$ of dimension $m$.

The algorithm creates an simplex tableau. This implementation is restricted to linear constraints $Ax \leq b$ and $Q$ being a positive semi-definite matrix. Pivoting must now also handle non-linear complementary slackness.

Class Methods:

* @param a the M-by-N constraint matrix
* @param b the M-length constant/limit vector
* @param q the N-by-N cost/revenue matrix (second order component)
* @param c the N-length cost/revenue vector (first order component)
* @param x_B the initial basis (set of indices where $x_i$ is in the basis)

```scala
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 (x: 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 ()
15.4 Coordinate Descent
15.5 Gradient Descent
15.6 Stochastic Gradient Descent
15.7 Conjugate Gradient
15.8 Quasi-Newton Method
15.9 Alternating Direction Method of Multipliers
15.10 Nelder-Mead Simplex
Chapter 16

Parallel Computing
16.1 MIMD - Multithreading
16.2 SIMD - Vector Instructions
Chapter 17

Distributed Computing
17.1 Message Passing
17.2 Distributed Shared Memory
17.3 Distributed Functional Programming
Bibliography


