Big Data Analytics using ScalaTion.
Pi Day version 3.14.15.9.26

John A. Miller
Department of Computer Science
University of Georgia

March 14, 2015
Contents

1 Introduction to Analytics 5
  1.1 Preliminaries ........................................................................................................... 6
    1.1.1 Probability .......................................................................................................... 6
    1.1.2 Vectors and Matrices ........................................................................................ 8
    1.1.3 Notational Convention ....................................................................................... 9

2 Univariate Models 11
  2.1 Prediction .................................................................................................................. 12
    2.1.1 Predictor ........................................................................................................... 13
    2.1.2 SimpleRegression .............................................................................................. 14
    2.1.3 Regression ......................................................................................................... 16
    2.1.4 TranRegression ................................................................................................ 21
    2.1.5 PolyRegression ................................................................................................ 22
    2.1.6 TrigRegression .................................................................................................. 23
    2.1.7 ResponseSurface ............................................................................................... 24
    2.1.8 ANOVA ............................................................................................................. 25
    2.1.9 ANCOVA ........................................................................................................... 26
    2.1.10 GLM .................................................................................................................. 27
    2.1.11 GZLM ............................................................................................................... 28
    2.1.12 ExpRegression ................................................................................................ 29
    2.1.13 NonLinRegression ............................................................................................ 31
    2.1.14 Perceptron ....................................................................................................... 33
    2.1.15 ARMA .............................................................................................................. 36
  2.2 Classification .............................................................................................................. 38
    2.2.1 Classifier ............................................................................................................ 38
    2.2.2 ClassifierInt ....................................................................................................... 38
    2.2.3 ClassifierReal .................................................................................................... 39
    2.2.4 NaiveBayes ....................................................................................................... 40
    2.2.5 NaiveBayesR ..................................................................................................... 43
    2.2.6 SelNaiveBayes .................................................................................................. 44
    2.2.7 AugNaiveBayes ............................................................................................... 45
    2.2.8 BayesNetwork .................................................................................................. 46
    2.2.9 LogisticRegression ............................................................................................ 47
    2.2.10 PoissonRegression ......................................................................................... 49
    2.2.11 DecisionTreeID3 ............................................................................................. 51
    2.2.12 DecisionTreeC5 .............................................................................................. 55
2.2.13 SupportVectorMachine ................................................. 56
2.2.14 KNN Classifier ......................................................... 57

3 Multivariate Models .......................................................... 59
  3.1 Multivariate Prediction .................................................. 60
  3.1.1 NeuralNet .............................................................. 61
  3.2 Multivariate Classification ............................................. 62

4 Clustering ........................................................................ 63
  4.1 Clusterer ................................................................. 63
  4.2 KMeansClustering ....................................................... 64
  4.3 HierClustering ........................................................... 65
  4.4 MarkovClustering ........................................................ 66

5 Reduction ....................................................................... 67
  5.1 Reducer ................................................................. 67
  5.2 PrincipalComponents .................................................. 68
Chapter 1

Introduction to Analytics

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).
1.1 Preliminaries

1.1.1 Probability

Probability is used to measure the likelihood of certain events occurring, such as flipping a coin and getting a head, rolling two dice and getting a sum of 7, or getting a full house in five card draw. Given an event \( A \), the probability of its occurrence is denoted by

\[
P(A) \in [0, 1]
\]

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

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

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.

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

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

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

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

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

Using the definition of a CDF, one can determine the expected value (or mean) for the random variable using a Riemann-Stieltjes integral.
\begin{align*}
\mathbb{E}[y] &= \int_{D_y} y \, dF_y(y) \\
\text{For continuous random variables, if the function } F_{\parallel RNy} \text{ is differentiable, a probability density function (pdf) may be defined.}\n&f_y(y) = \frac{dF_y(y)}{dy} \\
\text{In case the random variable is discrete, a probability mass function (pmf) may be defined.}\n&\quad p_y(y_i) = F_y(y_i) - F_y(y_{i-1}) \\
\text{In addition, one may be interested in the median or half quantile.}\n&\quad Q[y] = F_y^{-1}(\frac{1}{2}) \\
\text{Similarly, we may be interested in the mode, which is the average of the points of maximal probability mass.}\n&\quad M[y] = \arg\max_{y \in D_y} p_y(y) \\
\text{For continuous random variables, it is the average of points of maximal probability density.}\n&\quad M[y] = \arg\max_{y \in D_y} f_y(y) \\
\text{Conditional expectation is defined as follows:}\n&\quad \mathbb{E}[y|x] = \int_{D_y} y \, dF_{y|x}(y) \\
\text{Understanding of some of techniques discussed requires some background in conditional probability. Consider the probability of rolling a natural (i.e., 7 or 11) with two dice where the random variable } y \text{ is the sum of the dice.}\n&\quad P(y \in \{7,11\}) = 1/6 + 1/18 = 2/9 \\
\text{If you knew you rolled a natural, what is the conditional probability that you rolled a 5 or 7?}\n&\quad 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} = 3/4 \\
\text{This is the conditional probability of rolling a 5 or 7 given that you rolled a natural.}\n\text{More generally, the conditional probability that } y \in A \text{ given that } x \in B \text{ is the joint probability divided by the probability that } x \in B.\n&\quad P(y \in A \mid x \in B) = \frac{P(y \in A, x \in B)}{P(x \in B)} \\
\text{where}\n&\quad P(y \in A, x \in B) = P(x \in B \mid y \in A) P(y \in A) \\
\text{Therefore, the conditional probability of } y \text{ given } x \text{ is}\n\end{align*}
\[ P(y \in A \mid x \in B) = \frac{P(x \in B \mid y \in A) \cdot P(y \in A)}{P(x \in B)} \]

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

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

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. What is the probability that you selected the counterfeit coin? Obviously, the probability is 1/3. This is the prior probability. Now suppose you flip the coin three times and get all heads. Let \( x \) indicate whether the selected coin is fair (0) or counterfeit (1) and \( y \) equal the number of heads rolled. Using Bayes Theorem, we have,

\[
P(x = 1 \mid y = 3) = \frac{P(y = 3 \mid x = 1) \cdot P(x = 1)}{P(y = 3)} = \frac{1 \cdot (1/3)}{5/12} = \frac{4}{5}
\]

where \( P(y = 3) = (1/3)(1) + (2/3)(1/8) = 5/12 \). The 4/5 is the posterior probability.

Note, random number and random variate generators can be found in ScalaTion’s `random` package.

### 1.1.2 Vectors and Matrices

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

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

\[
\mathbf{x} = \begin{bmatrix} x_0, x_1, x_2 \end{bmatrix} = \begin{bmatrix} 0.577, 0.557, 0.577 \end{bmatrix} \\
\mathbf{y} = \begin{bmatrix} y_0, y_1, y_2 \end{bmatrix} = \begin{bmatrix} 1.0, 1.0, 0.0 \end{bmatrix}
\]

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. The dot product of two vectors is simply the sum of the products of their elements.

\[
\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n-1} x_i y_i = 1.154
\]

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

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

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,
\[
\mu(x) = \frac{1 \cdot x}{n}
\]
\[
\sigma^2(x) = \frac{(x - \mu(x)) \cdot (x - \mu(x))}{n} = \frac{(x \cdot x)}{n} - \mu(x)^2
\]
\[
\sigma(x, y) = \frac{(x - \mu(x)) \cdot (y - \mu(y))}{n} = \frac{(x \cdot y)}{n} - \mu(x) \mu(y)
\]
\[
\rho(x, y) = \frac{\sigma(x, y)}{\sigma(x) \sigma(y)}
\]
\[
cos(\theta) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{1.154}{1 \cdot \sqrt{2}} = 0.816
\]

so the angle \( \theta = 0.616 \) radians.

Vector notation facilitates concise mathematical expressions, e.g.,
which are the population mean, variance, covariance and correlation, respectively. The size of the population
is \( n \), which corresponds to the length of the vector. A vector of all ones is denoted by \( \mathbf{1} \). Note, the sample
mean uses the same formula, while the sample variance and covariance divide by \( n - 1 \), rather than \( n \) (sample
indicates that only some fraction of population is used in the calculation).

A matrix may be viewed as a collection of vectors, one for each row in the matrix. ScalaTion supports
retrieval of row vectors, column vectors and matrix elements. Given an \( m \)-by-\( n \) matrix \( \text{val a} = \text{MatrixD}(m, n) \), the following operations are supported.

\[
A = \ a = \text{matrix}
\]
\[
a_{i-} = a(i) = \text{row vector } i
\]
\[
a_{-j} = a.\text{col}(j) = \text{column vector } j
\]
\[
a_{ij} = a(i, j) = \text{the element at row } i \text{ and column } j
\]

Common operations on matrices are supported as well, e.g., matrix multiplication \( \text{val c = a * b} \).

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

1.1.3 Notational Convention

<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>
Chapter 2

Univariate Models
2.1 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 an output/response scalar,
- $x$ is an input/predictor 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 $B$. This chapter focuses on univariate models (scalar responses), while the next chapter on multivariate models will treat the responses as vectors.

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 $x$.

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

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

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

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

The ‘Predictor’ trait provides a common framework for several predictors.

Trait Methods:

trait Predictor

def train ()
def predict (z: VectorD): Double
def predict (z: VectorI): Double = predict (z.toDouble)
def predict (z: MatrixD): VectorD
2.1.2 SimpleRegression

The ‘SimpleRegression’ class supports simple linear regression. In this case, the 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 a $m$-by-2 matrix $X$ and $m$ response values, stored in an $m$ dimensional vector $y$, solve the following optimization problem,

$$\min_{b} \|y - Xb\|$$

$$\min_{[b_0, b_1]} \|y - [1 \cdot x_1][b_0 \ b_1]\|$$

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

This is equivalent to minimizing the dot product

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

Taking the gradient $[\partial / \partial b_0, \partial / \partial b_1]$ using the derivative product rule and setting it equal to zero yields two equations. Setting $\partial / \partial b_0$ to zero yields,

$$-21 \cdot (y - (b_0 1 + b_1 x_1)) = 0$$
$$-21 \cdot y + 21 \cdot (b_0 1 + b_1 x_1) = 0$$

while setting $\partial / \partial b_1$ to zero yields,

$$-2x_1 \cdot (y - (b_0 1 + b_1 x_1)) = 0$$
$$-2x_1 \cdot y + 2x_1 \cdot (b_0 1 + b_1 x_1) = 0$$

From the first equation, we can solve for $b_0$ in terms of $b_1$.

$$b_0 (1 \cdot 1) + b_1 (1 \cdot x_1) = 1 \cdot y$$
$$b_0 = (1 \cdot y - b_1 (1 \cdot x_1)) / n$$

Using the definition for means $\mu(x_1)$ and $\mu(y)$, the expression shortens to

$$b_0 = \mu(y) - b_1 \mu(x_1)$$

From the second equation, we can solve for $b_1$.

$$b_0 (1 \cdot x_1) + b_1 (x_1 \cdot x_1) = x_1 \cdot y$$
$$b_1 = (x_1 \cdot y - (1 \cdot x_1)(1 \cdot y)/n) / (x_1 \cdot x_1 - (1 \cdot x_1)^2/n)$$
$$b_1 = ((x_1 \cdot y)/n - \mu(x_1) \mu(y)) / ((x_1 \cdot x_1)/n - \mu(x_1)^2)$$

Using the definitions for covariance $\sigma(x_1, y)$ and variance $\sigma^2(x_1)$, the expression shortens to

$$b_1 = \sigma(x_1, y) / \sigma^2(x_1)$$
The $b_0$ parameter gives the intercept, while the $b_1$ parameter gives the slope of the line that best fits the data points.

**Example Problem:** 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 through these points. What are the intercept and slope? Pass the $X$ matrix and $y$ vector as arguments to the `SimpleRegression` class to obtain the $b$ vector.

```scala
// 5 data points: constant x1
val x = new MatrixD((5, 2), 1.0, 0.0, // x 5-by-2 matrix
                    1.0, 1.0,
                    1.0, 2.0,
                    1.0, 3.0,
                    1.0, 4.0)
val y = VectorD(2.0, 3.0, 5.0, 4.0, 6.0) // y vector

val rg = new SimpleRegression(x, y)
rg.train()
println("fit = " + rg.fit) // parameters and quality of fit

val z = VectorD(1.0, 5.0) // predict y for one point
println("predict (" + z + ") = " + rg.predict(z))
```

**Class Methods:**

@param x the input/design matrix augmented with a first column of ones
@param y the response vector

class SimpleRegression(x: MatrixD, y: VectorD)
    extends Predictor with Error

def train ()
def fit: Tuple2[VectorD, Double] = (b, rSquared)
def predict (z: VectorD): Double = b dot z
def predict (z: MatrixD): VectorD = z * b
The ‘Regression’ class supports multiple linear regression. In this case, \( x \) is multi-dimensional \([1, x_1, ... x_k]\). The goal is to fit the parameter vector \( b \) in the regression equation

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

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 \( b \). Each sample pairs an \( x \) input vector with a \( y \) response value. The \( x \) vectors are placed into a data/design matrix \( X \) row-by-row with a column of ones as the first column in \( X \). The individual response values taken together form the response vector \( y \). The matrix-vector product \( Xb \) provides an estimate for the response vector, so the goal is to minimize the distance between \( y \) and its estimate.

\[
\text{min}_b \| y - Xb \|
\]

This is equivalent to minimizing the dot product

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

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

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

Dividing by 2 and moving the term involving \( b \) to the left side, results in the Normal equations.

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

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

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

**Inverse Technique**

The simplest technique is Matrix Inversion, which involves computing the inverse of \( X^t X \).

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

The expression involving the \( X \) matrix is referred to as the pseudo-inverse \( X^{-1} \).

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

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

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

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

\[
x\_pinv = (x\_t * x).inv * x\_t
\]
**Cholesky Factorization Technique**

A faster and slightly more stable technique is to use Cholesky Factorization. Starting with the Normal Equations, define

\[ A = X^tX \]

A is a positive definite, symmetric matrix, so it may factored using Cholesky Factorization into

\[ A = LL^t \]

where L is a lower triangular matrix. Substituting, we obtain

\[ LL^t\mathbf{b} = X^t\mathbf{y} \]

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

\[ L\mathbf{w} = X^t\mathbf{y} \]

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

\[ L^t\mathbf{b} = \mathbf{w} \]

**QR Factorization Technique**

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

\[ X = QR \]

where \( Q \) is an orthogonal matrix and \( R \) matrix is a right upper triangular matrix. Starting again with the Normal Equations,

\[ X^tX\mathbf{b} = X^t\mathbf{y} \]

simply substitute \( QR \) for \( X \).

\[ (QR)^tQR\mathbf{b} = (QR)^t\mathbf{y} \]

Taking the transpose gives

\[ R^tQ^tQR\mathbf{b} = R^tQ^t\mathbf{y} \]

and using the fact that \( Q^tQ = I \), we obtain the following:

\[ R^tR\mathbf{b} = R^tQ^t\mathbf{y} \]

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

\[ R\mathbf{b} = Q^t\mathbf{y} \]

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

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

ScalaTion uses Gram-Schmidt orthogonalization to factor \( X \) into the product of \( Q \) and \( R \).
Singular Value Decomposition Technique

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

$$ y = Xb $$

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

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

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

$$ y = U\Sigma V^t b $$

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

$$ y = Ud $$

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

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

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

$$ d = U^ty $$

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

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

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

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 either Fac_Cholesky, SVD or Inverse. For more information see http://see.stanford.edu/materials/1soeldsee263/05-1s.pdf.

object RegTechnique extends Enumeration
{
    type RegTechnique = Value
    val Fac_QR, Fac_Cholesky, SVD, Inverse = Value
}

import RegTechnique._
The `train` function below computes `b` and related statistics on the quality of the fit, e.g., the coefficient of determination $R^2$. Both the `Inverse` and `Fac.QR` techniques compute the pseudo-inverse, while `Fac.Cholesky` uses the factor-solve paradigm and SVD uses its own approach.

```scala
def train ()
{
  b = if (x_pinv == null) fac.solve (y)
   else x_pinv * y // parameter vector [b0, b1, ... bk]
  val e = y - x * b // residual/error vector
  val sse = e dot e // residual/error sum of squares
  val sst = (y dot y) - y.sum^2.0 / m // total sum of squares
  val ssr = sst - sse // regression sum of squares
  val rSquared = ssr / sst // coefficient of determination (R-squared)
  rBarSq = 1.0 - (1.0-rSquared) * r_df // R-bar-squared (adjusted R-squared)
  fStat = ssr * (m-k-1.0) / (sse * k) // F statistic (msr / mse)
} // train
```

Note, ~^ is the exponentiation operator provided in ScalaTion, where the first character is ~ to give the operator higher precedence than multiplication (*).

**Exercise**: Explain what $R^2$ is and how it relates to Pearson correlation coefficient.

**Example Problem**: Solving a regression problem in ScalaTion simply involves, creating the data/design matrix $X$ and response vector $y$ and then creating a `Regression` object upon which `train` and `fit` are called. The Texas Temperature data-set below is from [http://www.stat.ufl.edu/~winner/cases/txtemp.ppt](http://www.stat.ufl.edu/~winner/cases/txtemp.ppt).

```scala
// 16 data points: Constant x1 x2 x3
// Lat Elev Long County
val x = new MatrixD ((16, 4), 1.0, 29.767, 41.0, 95.367, // Harris
                      1.0, 32.850, 440.0, 96.850, // Dallas
                      1.0, 26.933, 25.0, 97.800, // Kennedy
                      1.0, 31.950, 2851.0, 102.183, // Midland
                      1.0, 34.800, 3840.0, 102.467, // Deaf Smith
                      1.0, 33.450, 1461.0, 99.633, // Knox
                      1.0, 28.700, 815.0, 100.483, // Maverick
                      1.0, 32.450, 2380.0, 100.533, // Nolan
                      1.0, 31.800, 3918.0, 106.400, // El Paso
                      1.0, 34.850, 2040.0, 100.217, // Collington
                      1.0, 30.867, 3000.0, 102.900, // Pecos
                      1.0, 36.350, 3693.0, 102.083, // Sherman
                      1.0, 30.300, 597.0, 97.700, // Travis
                      1.0, 26.900, 315.0, 99.283, // Zapata
                      1.0, 28.450, 459.0, 99.217, // Lasalle
                      1.0, 25.900, 19.0, 97.433) // Cameron

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

val rg = new Regression (x, y)
```

19
rg.train ()
println ("full model: fit = " + rg.fit)
val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println ("predict (" + z + ") = " + rg.predict (z))
println ("reduced model: fit = " + rg.backElim ())

The source code for this example is at
http://www.cs.uga.edu/~jam/scalation_1.1/src/main/scala/apps/analytics/TempRegression.scala

Class Methods:

@param x the input/design m-by-n matrix augmented with a first column of ones
@param y the response vector
@param technique the technique used to solve for b in x.t*x*b = x.t*y

class Regression (x: MatrixD, y: VectorD, technique: RegTechnique = Fac_QR)
    extends Predictor with Error

def train ()
def train (yy: VectorD)
def fit: Tuple4 [VectorD, Double, Double, Double] = (b, rSquared, rBarSq, fStat)
def predict (z: VectorD): Double = b dot z
def predict (z: MatrixD): VectorD = z * b
def backElim (): Tuple4 [Int, VectorD, Double, Double] =
def vif: VectorD =
2.1.4 TranRegression

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

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

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

The transformation is done in the implementation of the ‘TranRegression’ class by creating a tranformed vector \( \text{yy} \) from \( y \) and passing it to the ‘Regression’ class (multiple linear regression).

```scala
val yy = y.map (transform)
val rg = new Regression (x, yy, technique)
```

Example Problem:

Class Methods:

```scala
class TranRegression (x: MatrixD, y: VectorD, transform: FunctionS2S = log,
                    technique: RegTechnique = Fac_QR)
    extends Predictor with Error

def train () { rg.train () }

def train (yy: VectorD) { rg.train (yy) }

def fit: Tuple4 [VectorD, Double, Double, Double] = rg.fit

def predict (z: VectorD): Double = rg.predict (z)

def predict (z: Matrix): VectorD = rg.predict (z)

def backElim (): Tuple4 [Int, VectorD, Double, Double] = rg.backElim ()

def vif: VectorD = rg.vif
```
2.1.5 PolyRegression

The 'PolyRegression' class supports polynomial regression. In this case, \( \mathbf{x} \) is formed from powers of a single parameter \( t \), \([1, t, t^2, \ldots, t^k]\). The goal is to fit the parameter vector \( \mathbf{b} \) in the regression equation

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

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

A training set now consists of two vectors, one for the \( m \)-vector \( \mathbf{t} \) and one for the \( m \)-vector \( \mathbf{y} \). An easy way to implement polynomial regression is to expand each \( t \) value into an \( \mathbf{x} \) vector to form a data/design matrix \( \mathbf{X} \) and pass it to the 'Regression' class (multiple linear regression).

```scala
val x = new MatrixD (t.dim, order + 1)
for (i <- 0 until t.dim) x(i) = expand (t(i))
val rg = new Regression (x, y, technique)
```

Example Problem:

Class Methods:

```scala
@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
@param technique the technique used to solve for \( \mathbf{b} \) in \( \mathbf{x} \cdot \mathbf{x} \cdot \mathbf{b} = \mathbf{x} \cdot \mathbf{y} \)

class PolyRegression (t: VectorD, y: VectorD, order: Int, technique: RegTechnique = Fac_QR)
    extends Predictor with Error

    def expand (t: Double): VectorD =
    def train () { rg.train }
    def train (yy: VectorD) { rg.train (yy) }
    def fit: Tuple4 [VectorD, Double, Double, Double] = rg.fit
    def predict (z: Double): Double = rg.predict (expand (z))
    def predict (z: VectorD): Double = rg.predict (z)
    def predict (z: Matrix): VectorD = rg.predict (z)
    def backElim (): Tuple4 [Int, VectorD, Double, Double] = rg.backElim ()
    def vif: VectorD = rg.vif
```
## 2.1.6 TrigRegression

The ‘TrigRegression’ class supports trigonometric regression. In this case, \(x\) is formed from trigonometric functions of a single parameter \(t\), \([1, \sin(\omega t), \cos(\omega t), \ldots, \sin(k \omega t), \cos(k \omega t)]\). 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 \(y\). 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).

```scala
val x = new MatrixD (t.dim, 1 + 2 * k)
for (i <- 0 until t.dim) x(i) = expand (t(i))
val rg = new Regression (x, y, technique)
```

### Example Problem:

**Class Methods:**

- `@param t` the input vector: \(t_i\) expands to \(x_i\)
- `@param y` the response vector
- `@param k` the maximum multiplier in the trig function (\(k \omega t\))
- `@param w` the base displacement angle in radians
- `@param technique` the technique used to solve for \(b\) in \(x^\top x \cdot b = x^\top y\)

```scala
class TrigRegression (t: VectorD, y: VectorD, k: Int, w: Double = Pi, technique: RegTechnique = Fac_QR)
  extends Predictor with Error
```

```scala
def expand (t: Double): VectorD =
def train () { rg.train () }
def train (yy: VectorD) { rg.train (yy) }
def fit: Tuple4 [VectorD, Double, Double, Double] = rg.fit
def predict (z: Double): Double = rg.predict (expand (z))
def predict (z: VectorD): Double = rg.predict (z)
def predict (z: Matrix): VectorD = rg.predict (z)
def backElim (): Tuple4 [Int, VectorD, Double, Double] = rg.backElim ()
def vif: VectorD = rg.vif
```

23
2.1.7 ResponseSurface

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), the quadratic regression equation is the following:

\[ y = b \cdot x' + \epsilon = b_0 + b_1x_0 + b_2x_0^2 + b_3x_0x_1 + b_4x_1 + b_5x_1^2 + \epsilon \]

The number of terms \((nt)\) in the model increases quadratically with the dimensionality of the space \((n)\) according to the formula for triangular numbers shifted by \((n \rightarrow n + 1)\).

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

For order three models, the goal is to fit a cubic surface. In two dimensions (2D), the cubic regression equation is the following:

\[ y = b \cdot x' + \epsilon = b_0 + b_1x_0 + b_2x_0^2 + b_3x_0^3 + b_4x_0x_1 + b_5x_0x_1^2 + b_6x_0x_1 + b_7x_1 + b_8x_1^2 + b_9x_1^3 + \epsilon \]

Naturally, the number of terms in the model increases cubically with the dimensionality of the space \((n)\) according to the formula for tetrahedral numbers shifted by \((n \rightarrow n + 1)\).

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

Example Problem:

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`

```scala
class ResponseSurface (x_: MatrixD, y: VectorD, cubic: Boolean = false, technique: RegTechnique = Fac_QR)
  extends Predictor with Error

  def allForms (): MatrixD =
  def qForms (p: VectorD): VectorD =
  def cForms (p: VectorD): VectorD =
  def train () { rsm.train () }
  def train (yy: VectorD) { rsm.train (yy) }
  def fit: Tuple4 [VectorD, Double, Double, Double] =
  def predict (z: VectorD): Double =
  def predict (z: Matrix): VectorD =
  def backElim (): Tuple4 [Int, VectorD, Double, Double] = rsm.backElim ()
  def vif: VectorD = rsm.vif
```

24
2.1.8 ANOVA

An ANalysis Of VAriance (ANOVA) model may be developed using the ‘ANOVA’ class. This type of model comes into play when input variables are binary or categorical. In the binary case, a single dummy zero-one variable 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 \(l\) dummy variables \(d_j\).

\[
y = b \cdot x + \epsilon = b_0 + b_1d_1 + \ldots b_{l}d_{l} + \epsilon
\]

The dummy variables are binary and are used to determine the level of a binary or 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 \(d_1, d_2, \text{and } d_3\) 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}; (0, 0, 0) \implies \text{High}.

In ScalaTion, ANOVA 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. Note, a more traditional implementation ‘Anova’ is provided in the \texttt{stat} package.

Example Problem:

Class Methods:

\[
\text{class ANOVA (t: VectorI, y: VectorD, levels: Int, technique: RegTechnique = Fac_QR)} \\
\text{extends Predictor with Error}
\]

\[
def \text{assignDummyVars ()}
\]
\[
def \text{train () \{ rg.train \}}
\]
\[
def \text{train (yy: VectorD) \{ rg.train (yy) \}}
\]
\[
def \text{fit: Tuple4 [VectorD, Double, Double, Double] = rg.fit}
\]
\[
def \text{predict (z: VectorD): VectorD = rg.predict (z)}
\]
\[
def \text{predict (z: Matrix): VectorD = rg.predict (z)}
\]
\[
def \text{backElim (): Tuple4 [Int, VectorD, Double, Double] = rg.backElim ()}
\]
\[
def \text{vif: VectorD = rg.vif}
\]
2.1.9 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 variable \( d \) 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 \( d_j \).

\[
y = b_0 + b_1 x_1 + \ldots + b_k x_k + b_{k+1} d_1 + \ldots + b_{k+l} d_l + \epsilon
\]

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 \( X \) 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*x*b = x.t*y

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

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

def assignVars ()
def assignDummyVars ()
def train () { rg.train }
def train (yy: VectorD) { rg.train (yy) }
def fit: Tuple4 [VectorD, Double, Double, Double] = rg.fit
def predict (z: VectorD): Double = rg.predict (z)
def predict (z: Matrix): VectorD = rg.predict (z)
def backElim (): Tuple4 [Int, VectorD, Double, Double] = rg.backElim ()
def vif: VectorD = rg.vif
```
2.1.10 GLM

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) transformed multiple linear regression, (4) polynomial regression, (5) trigonometric regression, (6) response surface regression, (7) ANOVA and (8) ANCOVA. It provides factory methods for creating specific types of general linear models, based on the signatures of the parameters.

Example Problem:

Class Methods:

```scala
trait GLM

def setAdd_1 (_add_1: Boolean = true) { add_1 = _add_1 }

def setTechnique (_technique: RegTechnique = Fac_QR) { technique = _technique }

def apply (x: VectorD, y: VectorD): SimpleRegression =

def apply (x: MatrixD, y: VectorD): Regression =

def apply (xy: MatrixD): Regression =

def apply (x: MatrixD, y: VectorD, transform: FunctionS2S): TranRegression =

def apply (xy: MatrixD, transform: FunctionS2S): TranRegression =

def apply (t: VectorD, y: VectorD, k: Int): PolyRegression =

def apply (ty: MatrixD, k: Int): PolyRegression =

def apply (t: VectorI, y: VectorD, levels: Int): ANOVA =

def apply (x_: MatrixD, t: VectorI, y: VectorD, levels: Int): ANCOVA =

object GLM extends GLM
```

27
2.1.11 GZLM

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,...,\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>


Since Logistic and Poisson Regression are on discrete domains, they have been placed under Classification (see the next section).

Example Problem:

Class Methods:

```scala
object GZLM extends GLM

def apply (x: MatrixD, y: VectorI, cn: Tuple2 [String, String]): LogisticRegression =
def apply (x: MatrixD, y: VectorI, cn: Array [String]): PoissonRegression =
def apply (x: MatrixD, nonneg: Boolean, y: VectorD): ExpRegression =
```
2.1.12 ExpRegression

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: MatrixD, nonneg: Boolean, y: VectorD)
    extends Predictor with Error
def ll (b: VectorD): Double = 0.0
def ll_null (b: VectorD): Double = 0.0
def train () { }
def train (yy: VectorD) { }
def fit: Tuple4 [VectorD, Double, Double, Double] = null
def predict (z: VectorD): Double = 0.0
def predict (z: Matrix): VectorD = null
def backElim (): Tuple4 [Int, VectorD, Double, Double] = null
def vif: VectorD = null
2.1.13 NonLinRegression

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.

\[
y = \frac{(b_0 + b_1 x_1)}{(b_2 + x_1)} + \epsilon
\]

A training set consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector \( b \). Given an input matrix \( X \) consisting of \( m \) input vectors and an output vector \( y \) consisting of \( m \) output values, minimize the distance between the target output vector \( y \) and the predicted output vector \( f(X; b) \).

\[
\min_b \|y - f(X; b)\|
\]

Again, it is convenient to minimize the dot product of the error with itself,

\[
p(b) = (y - f(X; b)) \cdot (y - f(X; b))
\]

For non-linear regression, a Least-Squares (minimizing the residuals) method can be used to fit the parameter vector \( b \). Unlike the linear case (where one simply sets the gradient to zero), since the formula is non-linear in \( b \), Non-linear Programming (NLP) is used to minimize the Sum of Squares Error (\( SS_E \)). A user defined function \( f: (\text{VectorD}, \text{VectorD}) \rightarrow \text{Double} \) is passed as a class parameter. This function is used to create a predicted output value \( z_i \) for each input vector \( x_i \). The \( \text{sseF} \) method applies this function to all \( m \) input vectors to compute predicted output values. These are then subtracted from the target output to create an error vector \( e \), which when dot producted with itself yield \( SS_E \).

```scala
def sseF (b: VectorD): Double =
{
    val z = new VectorD (m) // create vector z to hold predicted outputs
    for (i <- 0 until m) z(i) = f (x(i), b) // compute values for z
    val e = y - z // residual/error vector
    e dot e // residual/error sum of squares
} // sseF
```

ScalaTion’s \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 \( \text{sseF} \). 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
```


Example Problem:

Class Methods:
@param x the input/design matrix augmented with a first column of ones
@param y the response vector
@param f the non-linear function \( f(x, b) \) to fit
@param b_init the initial guess for the parameter vector \( b \)

class NonLinRegression (x: 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: Tuple2 [VectorD, Double] = (b, rSquared)
def predict (z: VectorD): Double = f(z, b)
def predict (z: MatrixD): VectorD =
2.1.14 Perceptron

The ‘Perceptron’ class supports single-valued 2-layer (input and output) Neural-Networks. Given several input vectors and output values (training data), optimize/fit the weights connecting the layers. The weights are analogous to the parameter vector $b$ in regression. After training, given an input vector $x$, the net can be used to predict the corresponding output value $y$, i.e.,

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

Note, $b_0$ is treated as the bias, so $x_0$ must be 1.

A training set consisting of $m$ input-output pairs is used to minimize the error in the prediction by adjusting the weight vector $b$. Given an input matrix $X$ consisting of $m$ input vectors and an output vector $y$ consisting of $m$ output values, minimize the distance between the target output vector $y$ and the predicted output vector $f(Xb)$.

$$\min_b ||y - f(Xb)||$$

As was the case with regression, it is convenient to minimize the dot product of the error with itself (in this case for convenience, we actually minimize half the dot product).

$$p(b) = \frac{1}{2}(y - f(Xb)) \cdot (y - f(Xb))$$

using gradient-descent. Letting the activation function $f = \text{sigmoid}$,

$$\text{sigmoid}(t) = 1/(1 + e^{-t})$$

simplifies the computation of the gradient, since the derivative of $\text{sigmoid}(t)$ is $\text{sigmoid}(t)(1 - \text{sigmoid}(t))$.

The $\text{sigmoid}$ 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. Therefore, the gradient becomes the following:

$$\nabla p(b) = -X^t((y - f(Xb)f(Xb)(1 - f(Xb)))$$

Replacing $y - f(Xb)$ with $\epsilon$ gives

$$\nabla p(b) = -X^t(\epsilon f(Xb)(1 - f(Xb)))$$

Gradient descent works by iteratively moving in the opposite direction as the gradient until the error drops 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 1. The main training loop is shown below.

```scala
breakable { for (k <- 0 until MAX_ITER) { // kth learning phase
  val z = sigmoid (x * b) // vector of predicted outputs
  val e = y - z // target outputs - predicted outputs
  b += x.t * (e * z * (_1 - z)) * eta // adjust the weights
  println ("weights for " + k + "th phase: b = " + b + ", error e = " + e)
  if ((e dot e) < 2.0 * EPSILON) break // break when error is small enough
} // for
```
A perceptron can be considered to be a special type of non-linear regression.

The sigmoid activation functions, one for scalars and one for vectors, are defined in the Perceptron companion object.

Example Problem: The Texas Temperature regression problem can also be analyzed using a perceptron.

```scala
// 16 data points: Constant x1 x2 x3 County
// Lat Elev Long
val x = new MatrixD((16, 4), 1.0, 29.767, 41.0, 95.367, // Harris
                    1.0, 32.850, 440.0, 96.850, // Dallas
                    1.0, 26.933, 25.0, 97.800, // Kennedy
                    1.0, 31.950, 2851.0, 102.183, // Midland
                    1.0, 34.800, 3840.0, 102.467, // Deaf Smith
                    1.0, 33.450, 1461.0, 99.633, // Knox
                    1.0, 28.700, 815.0, 100.483, // Maverick
                    1.0, 32.450, 2380.0, 100.533, // Nolan
                    1.0, 31.800, 3918.0, 106.400, // El Paso
                    1.0, 34.850, 2040.0, 100.217, // Collington
                    1.0, 30.867, 3000.0, 100.217, // Pecos
                    1.0, 36.350, 3693.0, 102.083, // Sherman
                    1.0, 30.300, 597.0, 97.700, // Travis
                    1.0, 26.900, 315.0, 99.283, // Zapata
                    1.0, 28.450, 459.0, 99.217, // Lasalle
                    1.0, 25.900, 19.0, 97.433) // Cameron
val y = VectorD(56.0, 48.0, 60.0, 46.0, 38.0, 46.0, 53.0, 46.0,
                 44.0, 41.0, 47.0, 36.0, 52.0, 60.0, 56.0, 62.0)
val ann = new Perceptron(x, y)
time { ann.train() }
println("full model: fit = "+ ann.fit)
val z = VectorD(1.0, 30.0, 1000.0, 100.0)
println("predict ("+ z + ") = "+ ann.predict(z))

The source code for this example is at

Class Methods:

@param x the 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

class Perceptron (x: MatrixD, y: VectorD, eta: Double = 1.0)
  extends Predictor with Error

def setWeights (b0: VectorD) { b = b0 }
```

34
def setWeights (i: Int = 0)
def train () { if (b == null) setWeights (); minimizeError () }
def minimizeError ()
def fit: VectorD = b
def predict (zi: VectorD): Double = sigmoid (b dot zi)
def predict (zi: MatrixD): VectorD = sigmoid (zi * b)

object Perceptron

def sigmoid (t: Double): Double = 1.0 / (1.0 + exp (-t))
def sigmoid (t: VectorD): VectorD =
2.1.15 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 \(\epsilon_t\).

\[
y_t = \mu + \phi_1 y_{t-1} + ... + \phi_p y_{t-p} + \epsilon_t
\]

The code in ScalaTion 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.

\[
y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q}
\]
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: VectorD, t: VectorD)
    extends Predictor with Error

    def est_ar (p: Int = 1): VectorD =
    def durbinLevinson: MatrixD =
    def ar (phi: VectorD): VectorD =
    def est_ma (q: Int = 1): VectorD =
    def ma (theta: VectorD): VectorD =
    def train ()
    def predict (y: VectorD): Double =
    def predict (z: MatrixD): VectorD =
    def plotFunc (fVec: VectorD, name: String)
    def smooth (l: Int): VectorD =
2.2 Classification

When the output/response $y$ is defined on small domains, e.g., $\mathbb{B}$ or $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$, then the problem shifts from prediction to classification.

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

As with Regression in continuous domains, several of the modeling techniques in this section will focus on estimating the conditional expectation of $y$ given $x$.

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

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

$$y = \mathbb{M}[y|x]$$

Rather than find a real number that is a best predictor, one of a set of distinct given values (e.g., 0 (false), 1 (true); negative (-1), positive (1); or low (0), medium (1), high (2)) is chosen. Abstractly, we can label the classes $C_0, C_1, \ldots, C_{k-1}$. In the case of classification, the `train` function is still used, but the `classify` and `classifyAll` functions replace the prediction functions.

2.2.1 Classifier

The ‘Classifier‘ trait provides a common framework for several classifiers.

Trait Methods:

trait Classifier

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>train ()</td>
<td></td>
</tr>
<tr>
<td>classify (z: VectorI): Tuple2 [Int, String]</td>
<td></td>
</tr>
<tr>
<td>classify (z: VectorD): Tuple2 [Int, String]</td>
<td></td>
</tr>
</tbody>
</table>

2.2.2 ClassifierInt

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

Class Methods:

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the integer-valued training data vectors stored as rows of a matrix</td>
</tr>
<tr>
<td>y</td>
<td>the training classification vector, where $y_i =$ class for row $i$ of the matrix $x$</td>
</tr>
<tr>
<td>fn</td>
<td>the names of the features/variables</td>
</tr>
<tr>
<td>k</td>
<td>the number of classes</td>
</tr>
<tr>
<td>cn</td>
<td>the names for all classes</td>
</tr>
</tbody>
</table>

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

def vc_default: VectorI =
def classify (z: VectorD): Tuple2 [Int, String] =
def test (xx: MatrixI, yy: VectorI): Double =

38
2.2.3 ClassifierReal

The ‘ClassifierReal’ abstract class provides a common foundation for several classifiers that operate on 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 \( y_i = \text{class for row } i \) 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: MatrixD, y: VectorI, fn: Array[String], k: Int, 
    cn: Array[String]) 
    extends Classifier with Error

def classify (z: VectorI): Tuple2[Int, String] =
def test (xx: MatrixD, yy: VectorI): Double =
2.2.4 NaiveBayes

The ‘NaiveBayes’ class implements a Naive Bayes (NB) Classifier suitable for discrete input data. Given an input vector \( \mathbf{x} \), a Bayesian classifier attempts to find the class that maximizes the conditional probability that the correct class is \( y \). It may be 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 \( y \). Using Bayes Theorem, this needs to be weighted by the prior probability of \( y \), i.e., \( P(y) \).

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

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

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

The classifier is said to be naive, when we assume that the \( x_i \)'s are sufficiently uncorrelated to take the product of their conditional probabilities (independence rule).

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

The solution is the class \( y \) that is the argmax of the product of the prior probability and all the conditional probabilities.

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

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 \( 0, 1, \ldots, k - 1 \). Prior probabilities are calculated based on the population (frequency) of each class in the training-set. The prior probability that \( y \) equals \( l \) can be estimated by the number of elements in the vector where \( y_i \) equals \( l \) divided by the total number of elements.

\[
P(y = l) = \frac{\#\{i | y_i = l\}}{m}
\]

where \( \# \) counts the elements in the set. Conditional probabilities are calculated as a ratio.

\[
P(x_j = h | y = l) = \frac{\#\{i | x_{ij} = h, y_i = l\}}{\#\{i | y_i = l\}}
\]

If m-estimates are used, the conditional probabilities are adjusted slightly as follows:

\[
P(x_j = h | y = l) = \frac{(\#\{i | x_{ij} = h, y_i = l\} + me/vc_j)}{(\#\{i | y_i = l\} + me)}
\]

where \( me \) (e.g., 3) is the parameter used for the m-estimate (note, these will prevent dividing by zero).

**Example Problem:** An Integer-based Naive Bayes 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 from [http://suanpalm3.kmutnb.ac.th/teacher/FileDL/choochart82255418560.pdf](http://suanpalm3.kmutnb.ac.th/teacher/FileDL/choochart82255418560.pdf)
val x = new MatrixI ((14, 4), 2, 2, 1, 1, // day 2
  1, 2, 1, 0, // day 3
  0, 1, 1, 0, // day 4
  0, 0, 0, 0, // day 5
  0, 0, 0, 1, // day 6
  1, 0, 0, 1, // day 7
  2, 1, 1, 0, // day 8
  2, 0, 0, 0, // day 9
  0, 1, 0, 0, // day 10
  2, 1, 0, 1, // day 11
  1, 1, 1, 1, // day 12
  1, 2, 0, 0, // day 13
  0, 1, 1, 1) // day 14

val y = VectorI (0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0) // classification: 0(No), 1(Yes))

val fn = Array ("Outlook", "Temp", "Humidity", "Wind") // feature names
val vc = VectorI (3, 3, 2, 2) // value count for each feature

val nb = new NaiveBayes (x, y, fn, vc, 2, 0) // create a classifier

nb.train ()
val z = VectorI (2, 2, 1, 1) // new vector to classify
println ("classify (" + z + ") = " + nb.classify (z))
def frequencies ()
def train ()
def classify (z: VectorI): Tuple2 [Int, String] =
2.2.5 NaiveBayesR

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

Prior probabilities are calculated based on the population of each class in the training-set. Relative posterior 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.

If one can estimate the conditional density of \( x_j \) given \( y \), then the NaiveBayesR class can used. The best prediction for class \( y \) is the value that maximizes the product of the conditional densities multiplied by the prior probability, as shown below.

\[
y = \arg\max_{y \in D_y} P(y) \prod_{j=0}^{n-1} f(x_j | y)
\]

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 conditional density (cd) functions estimated in the train function (see code for details), an input vector \( z \) can be classified using the classify function.

```scala
def classify (z: VectorD): Int =
{
    for (c <- 0 until k; j <- 0 until n) prob(c) *= cd(c)(j)(z(j))
    prob.argmax () // class c with highest posterior probability
} // classify
```

Example Problem:

Class Methods:

```scala
@param x the real-valued data vectors stored as rows of a matrix
@param y the class vector, where \( y_i = \text{class for row} \ i \ \text{of the matrix} \ x \\
@param fn the names of the features/variables
@param k the number of classes

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

def checkCorrelation
def calcStats ()
def calcHistogram (x_j: VectorD, intervals: Int): VectorD =
def train ()
def classify (z: VectorD): Int =
```
2.2.6 SelNaiveBayes

The ‘SelNaiveBayes’ class implements a Selective Naive Bayes (SNB) Classifier suitable for discrete input data. Rather than taking the product of all \( n \) features/variables, a proper subset \( F \subset \{0,1,\ldots n-1\} \) of the features is selected.

\[
y = \arg\max_{y \in D_y} P(y) \prod_{j \in F} P(x_j | y)
\]

Various algorithms can be used to search for an optimal feature set \( F \). ScalaTion currently used a simple greedy algorithm that add the next best feature/variable until the improvement in accuracy drops below a threshold.
2.2.7 AugNaiveBayes

The ‘AugNaiveBayes’ 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 the root node of a Directed Acyclic Graph (DAG), where each node/variable has at most two parents.

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

Various algorithms can be used to select the best \( x_{p(j)} \) for each \( x_j \).
2.2.8 BayesNetwork

The ‘BayesNetwork’ class implements a Bayesian Network (BN) Classifier suitable for discrete input data. It classifies an input data vector \( x \) by determining which of \( k \) classes has the highest Joint Probability of \( x \) and the response/outcome \( y \) (i.e., one of the \( k \) classes) of occurring.

\[
P(y|x_0, x_1, \ldots, x_{n-1})
\]

Using the Chain Rule of Probability, the Joint Probability calculation can factored into multiple calculations of conditional probabilities as well as the prior probability of the response. For example, given three variables, the joint probability may be factored as follows:

\[
P(x_0, x_1, x_2) = P(x_0)P(x_1|x_0)P(x_2|x_0, x_1)
\]

Conditional dependencies are specified using a Directed Acyclic Graph (DAG). A feature/variable represented by a node in the network is conditionally dependent on its parents only,

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

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.

Training is achieved by ...

---

**Example Problem:**

**Class Methods:**

- `@param dag` the directed acyclic graph specifying conditional dependencies
- `@param table` the array of tables recording conditional probabilities
- `@param k` the number of classes

```scala
class BayesNetwork (dag: DAG, table: Array [Map [Int, Double]], k: Int)
  extends Classifier with Error

  def jp (x: VectorI): Double =
  def cp (i: Int, key: VectorI): Double =
  def train ()
  override def classify (z: VectorI): Int =
  def classify (z: VectorD): Int =
```

---

46
2.2.9 LogisticRegression

The ‘LogisticRegression’ class supports logistic regression. In this case, \( x \) may be multi-dimensional \([1, x_1, \ldots, x_k]\).

Again, the goal is to fit the parameter vector \( b \) in the regression equation

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

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 \( p_y(x) \in [0, 1] \), i.e.,

\[
p_y(x) = P(y = 1 | x)
\]

Still, \( p_y(x) \) is bounded, while \( b \cdot x \) is not. We therefore, need a transformation, such as the logit transformation, and fit \( b \cdot x \) to this function. Treating this as a GZLM problem,

\[
y = \mu(x) + \epsilon
\]

\[
g(\mu(x)) = b \cdot x
\]

we let the link function \( g = \text{logit} \).

\[
\text{logit}(\mu(x)) = \ln\frac{p_y(x)}{1-p_y(x)} = b \cdot x
\]

This is the logit regression equation. Second, instead of minimizing the sum of squares errors, we wish to maximize the likelihood of predicting correct outcomes. For the \( i^{th} \) training case \( x_i \) with outcome \( y_i \), the likelihood function is based on the Bernoulli distribution.

\[
p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}
\]

The overall likelihood function is the product over all \( m \) cases.

\[
L = \prod_{i=0}^{m-1} p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}
\]

To reduce round-off errors, a log (e.g., natural log, ln) is taken

\[
LL = \sum_{i=0}^{m-1} y_i \ln(p_y(x_i)) + (1 - y_i)\ln(1 - p_y(x_i))
\]

Collecting \( y_i \) terms give

\[
LL = \sum_{i=0}^{m-1} y_i \ln\frac{p_y(x_i)}{1-p_y(x_i)} + \ln(1 - p_y(x_i))
\]

Substituting \( b \cdot x_i \) for logit and \( \frac{e^{b \cdot x_i}}{1+e^{b \cdot x_i}} \) for \( p_y(x_i) \) gives

\[
LL = \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\]
This is called the log-likelihood function. Multiplying it by -2 makes the distribution approximately Chi-square.

\[-2LL = -2 \sum_{i=0}^{m-1} y_i b \cdot x_i - \ln(1 + e^{b \cdot x_i})\]

The likelihood can be maximized by minimizing \(-2LL\), 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. 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).

Example Problem:

Class Methods:

```scala
@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 cn the names for all classes

class LogisticRegression (x: MatrixD, y: VectorI, cn: Array[String] = Array("no", "yes")) extends Classifier with Error

def ll (b: VectorD): Double =
def ll_null (b: VectorD): Double =
def train ()
def train_null ()
def fit: Tuple5[VectorD, Double, Double, Double, Double] =
def classify (z: VectorD): Tuple2[Int, String] =
def classify (z: VectorI): Tuple2[Int, String] = classify (z.toDouble)
def backElim (): Tuple4[Int, VectorD, Double, Double] =
def vif: VectorD =
```

48
2.2.10 PoissonRegression

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/design matrix augmented with a first column of ones
@param y the integer response vector, $y_i$ in \{0, 1, ... \}
@param cn the optional names for all categories/classes

class PoissonRegression (x: MatrixD, y: VectorI, cn: Array[String] = null)
    extends Classifier with Error

    def ll (b: VectorD): Double =
    def ll_null (b: VectorD): Double =
    def train ()
    def train_null ()
    def fit: Tuple5 [VectorD, Double, Double, Double, Double] =
    def classify (z: VectorD): Tuple2 [Int, String] =
    def classify (z: VectorI): Tuple2 [Int, String] = classify (z.toDouble)
    def backElim (): Tuple4 [Int, VectorD, Double, Double] =
    def vif: VectorD =
2.2.11 DecisionTreeID3

The ‘DecisionTreeID3’ class implements a Decision Tree classifier using the 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 feature (e.g.,
Humidity). The \( v_c \) vector gives the number of distinct values per feature (e.g., 2 for Humidity).

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

```scala
def 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 \( p' = (0.5, 0.5) \), so computing the Shannon entropy \( H(p) \),

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

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

```scala
def entropy (p: 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.

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

Let us consider the Tennis example from ‘NaiveBayes’ and compute the entropy level for the decision
of whether to play tennis. There are 14 days worth of training data, which indicate that for 9 of the
days the decision was yes (play tennis) and for 5 it was no (do not play). Therefore, the entropy (if no
features/variables are considered) is

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

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:
This is the weighted average of the entropy over all possible \( v_{c_j} \) values for variable \( j \).

To see how this works, let us compute new entropy values assuming each feature/variable is used, in turn, as the principal feature for decision making. Starting with feature \( j = 0 \) (Outlook) with values of Rain (0), Overcast (1) and Sunny (2), compute the probability vector and entropy for each value and weight them by how often that value occurs.

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

\[
\frac{\sum_{v=0}^{v_{c_j}-1} \frac{\#\{l \mid x_{lj} = v\} \cdot H(p_{x_j=v})}{m}}{14} H(p_{x_0=0}) + \frac{\#\{l \mid x_{l0} = 1\} \cdot H(p_{x_0=1})}{14} + \frac{\#\{l \mid x_{l0} = 2\} \cdot H(p_{x_0=2})}{14}
\]

We are left with computing three entropy values:

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

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

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

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

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

<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>

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

The sub-problem for Outlook: Rain (0) 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−, 3+), so the probability vector and entropy are

$$p_{x_3=0} = \left( \frac{3}{7}, \frac{4}{7} \right) \quad H(p_{x_3=0}) = 0$$

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

$$p_{x_3=1} = \left( \frac{2}{5}, \frac{3}{5} \right) \quad H(p_{x_3=1}) = 0$$

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

```java
if x0 = 0 then
  if x3 = 0 then yes
  if x3 = 1 then no
if x0 = 1 then yes
if x0 = 2 then no
```

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

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

**Example Problem:** The Tennis example (see ‘NaiveBayes’) can also be analyzed using decisions trees.

```scala
val id3 = new DecisionTreeID3 (x, y, fn, vc) // create the classifier
id3.train ()

val z = VectorI (2, 2, 1, 1) // new vector to classify
println ("classify (" + z + ") = " + id3.classify (z))
```
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 feature/variable names
@param vc the value count array indicating number of distinct values per feature
@param k the number of classes

class DecisionTreeID3 (x: MatrixI, y: VectorI, fn: Array [String], vc: VectorI = null, k: Int = 2)
    extends ClassifierInt (x, y, fn, vc, k)

def frequency (dset: Array[Tuple2[Int,Int]], value: Int): Tuple2 [Double, VectorD] =
def entropy (prob: VectorD): Double =
def dataset (f: Int, path: Path): Array [Tuple2 [Int, Int]] =
def mode (a: Array[Int]): Int =
def gain (f: Int, path: Path): Double =
def train ()
def buildTree (path: Path): Node =
def classify (z: VectorI): Int =
2.2.12 DecisionTreeC45

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:

```scala
@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): Tuple2[MatrixI, Array[Int]] =
def train ()
def buildTree (opt: Tuple2[Int, Double])
def printTree
override def classify (z: VectorI): Int =
def classify (z: VectorD): Int =
```
2.2.13 SupportVectorMachine

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

class SupportVectorMachine (x: MatrixD, y: VectorI, fn: Array [String], k: Int = 2) extends ClassifierReal (x, y, fn, k)

def l_D (a: VectorD): Double =
def g (a: VectorD): Double = a dot y
def find_w ()
def find_b ()
def train ()
def fit: Tuple2 [VectorD, Double] = (w, b)
def classify (z: VectorD): Int = (signum (w dot z + b)).toInt
2.2.14  KNN_Classifier

The 'KNN_Classifier' class is used to classify a new vector \( z \) into one of \( k \) classes. It works by finding its \( knn \) nearest neighbors. These neighbors essentially vote according to their classification. The class with most votes is selected as the classification of \( z \). Using a distance metric, the \( knn \) vectors nearest to \( z \) are found in the training data, which is stored row-wise in the data matrix \( X \). The corresponding classifications are given in the vector \( y \), such that the classification for vector \( x_i \) is given by \( y_i \).

Example Problem:

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 knn  the number of nearest neighbors to consider

class KNN_Classifier (x: MatrixD, y: VectorI, fn: Array [String], k: Int,
   cn: Array [String], knn: Int = 3)
   extends ClassifierReal (x, y, fn, k, cn)

def distance (u: VectorD, v: VectorD): Double =
def kNearest (z: VectorD)
def train ()
def classify (z: VectorD): Tuple2 [Int, String] =

57
Chapter 3

Multivariate Models
3.1 Multivariate Prediction
3.1.1 NeuralNet

The `NeuralNet` class supports basic 3-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.,

$$z^i \rightarrow z^h = f(W^t z^i) \rightarrow z^o = g(V^t z^h)$$

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.

$$z^o = \text{sigmoid}(V^t \text{sigmoid}(W^t z^i))$$

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 back propagation (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: Tuple2 [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 =
def predict (zi: MatrixD): VectorD = predictAll (zi)(0)
```

61
3.2 Multivariate Classification
Chapter 4

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.

4.1 Clusterer

The ‘Clusterer’ trait provides a common framework for several clustering algorithms.

**Trait Methods:**

```scala
trait Clusterer

def cluster (): Array[Int]
def classify (y: VectorD): Int
def name_ (n: Array[String])
def getName (i: Int): String =
```
4.2 KMeansClustering

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: MatrixD, k: Int, s: Int = 0, primary: Boolean = true)
  extends Clusterer with Error

  def distance (u: VectorD, v: VectorD): Double =
  def assign ()
  def reassign (): Boolean =
  def pickCentroids ()
  def calcCentroids ()
  def cluster (): Array [Int] =
  def classify (y: VectorD): Int =
```

64
4.3 HierClustering

Cluster several vectors/points using hierarchical clustering. Start with each point forming its own cluster and merge clusters until there are only \( k \).

Example Problem:

Class Methods:

@param x the vectors/points to be clustered stored as rows of a matrix
@param k stop when the number of clusters equals k

class HierClustering (x: 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 =
4.4 MarkovClustering

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: MatrixD, 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 (): MatrixD =
    def cluster (): Array [Int] =
    def classify (y: VectorD): Int =

66
Chapter 5

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.

5.1 Reducer

The 'Reducer' trait provides a common framework for several data reduction algorithms.

Trait Methods:

trait Reducer

def reduce (): MatrixD
def recover (): MatrixD
5.2 PrincipalComponents

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: MatrixD)

def meanCenter (): VectorD =
def computeCov (): MatrixD =
def computeEigenVectors (eVal: VectorD): MatrixD =
def findPCs (k: Int): MatrixD =
def reduceData (): MatrixD =
def recover (): MatrixD = reducedMat * featureMat.t + mu
def solve (i: Int): Tuple2 [VectorD, VectorD] =