IBM® Netezza® Analytics
Release 11.x

In-Database Analytics Developer's Guide

Revised: Oct. 05, 2017

Part Number 00X6331-03 Rev. 8

Note: Before using this information and the product that it supports, read the information in APPENDIX A on page 333.

© Copyright IBM Corporation 2011, 2017.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Preface

Introduction to IBM Netezza In-Database Analytics

2 Algorithmic Tasks

Working with Data Sets

Using Netezza Analytic Procedures

5 Metadata Management

6 Data Exploration

Tree-Shaped Bayesian Networks

Discretization

Standardization and Normalization

10 Data Imputation

11 Model Diagnostics

12 Random Sampling

13 Naive Bayes

14 Decision Trees

15 Nearest Neighbors

16 Linear Regression

17 Regression Trees

18 K-Means Clustering

19 Divisive Clustering

20 TwoStep Clustering

21 Association Rules

22 Sequential Patterns and Rules

23 Time Series Forecasting

24 Generalized Linear Models

25 Model Deployment

26 PMML Support

27 Utility Functions – Probability Distributions

28 Bulk Algorithms

APPENDIX A

Notices and Trademarks

List of Tables

Preface

Audience for This Guide

You should have an in-depth understanding of mathematics and statistics to use the IBM Netezza In-Database Analytics package. In addition, it is helpful to have an understanding of SQL and Netezza Stored Procedures and you should be familiar with the basic operation and concepts of the IBM Netezza appliance and the Netezza software.

Purpose of This Guide

This guide provides an introduction to the IBM Netezza In-Database Analytics package. It provides background information about the purpose and application of each algorithm, as well as guidelines for proper usage.

Symbols and Conventions

Note on Terminology: The terms *User-Defined Analytic Process (UDAP)* and *Analytic Executable (AE)* are synonymous.

The following conventions apply:

Italics for emphasis on terms and user-defined values, such as user input.

Upper case for SQL commands, for example, INSERT or DELETE.

Bold for command line input, for example, **`nzsystem stop`**.

Bold to denote parameter names, argument names, or other named references.

Angle brackets (< >) to indicate a placeholder (variable) that should be replaced with actual text, for example, `inza-<release_number>.zip`.

A single backslash (“\”) at the end of a line of code to denote a line continuation. Omit the backslash when using the code at the command line, in a SQL command, or in a file.

When referencing a sequence of menu and submenu selections, the “>” character denotes the different menu options, for example, **Menu Name > Submenu Name > Selection**.

If You Need Help

If you are having trouble using the IBM Netezza appliance, IBM Netezza Analytics or any of its components:

Retry the action, carefully following the instructions in the documentation.

Go to the IBM Support Portal at: http://www.ibm.com/support. Log in using your IBM ID and password. You can search the Support Portal for solutions. To submit a support request, click the **Service Requests & PMRs** tab.

If you have an active service contract maintenance agreement with IBM, you may contact customer support teams via telephone. For individual countries, please visit the Technical Support section of the IBM Directory of worldwide contacts (http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html#phone)

Comments on the Documentation

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Please send us an e-mail message at netezza-doc@wwpdl.vnet.ibm.com and include the following information:

The name and version of the manual that you are using

Any comments that you have about the manual

Your name, address, and phone number

We appreciate your comments.

CHAPTER 1

Introduction to IBM Netezza In-Database Analytics

Overview

The IBM Netezza In-Database Analytics package is for users and developers interested in leveraging the development and use of analytic algorithms to perform research or other business-related activities. The package brings data mining capabilities to the Netezza platform, enabling data mining tasks on large data sets using the computational power and parallelization mechanisms provided by the Netezza appliance.

Most currently available data mining tools suffer significant performance limitations when applied to large data sets. These limitations may be two-fold:

space: if system memory is used for storing data sets and auxiliary data structures to achieve high performance, the limited memory size and/or address space prevents applying data mining tools to large data sets.

time: if external storage is used for storing data sets or auxiliary data structures to overcome memory limitations, the resulting performance decline makes application of data mining tools to large data sets impractical.

Overcoming both these limitations, the parallel architecture of the Netezza database environment enables high-performance computation on large data sets, making it the ideal platform for large-scale data mining applications.

Mining large data sets might seem unnecessary, as good data mining models can often be created from data samples. However, the widespread practice of using small data samples when working with large data sets is typically a matter of necessity not choice. When highly reliable data mining results are required, no substantial data portions should be discarded. For complex data mining tasks, creating data samples of an appropriate size and structure may be a non-trivial task. The IBM Netezza In-Database Analytics package provides the tools necessary for mining the spectrum of data set sizes.

List of Algorithms

IBM Netezza In-Database Analytics is a data mining application that includes many of the key techniques and popular real-world algorithms used with data sets. Table 1 lists the data mining algorithms described in this guide, grouped into tasks.

Table 1: Algorithms described in this guide

Task	Algorithm
Data Exploration	Moments
	Quantiles
	Outlier Detection
	Frequency Table
	Histogram
	Pearson's Correlation
	Spearman's Correlation
	Covariance
	Mutual Information
	Chi-Square Test
	t -Test
	Mann-Whitney-Wilcoxon Test
	Wilcoxon Test
	Canonical Correlation
	One-Way ANOVA
	Multivariate Analysis of Variance (MANOVA)
	Principal Component Analysis
	Tree-Shaped Bayesian Networks
Data Transformation	Discretization

Task	Algorithm
	Standardization and Normalization
	Data Imputation
Model Diagnostics	Misclassification Error
	Confusion Matrix
	Mean Absolute Error
	Mean Square Error
	Relative Absolute Error
	Percentage Split
	Cross-Validation
Classification	Naive Bayes
	Decision Trees
	Nearest Neighbors
Regression	Linear Regression
	Regression Trees
	Generalized Linear Models
Clustering	K-Means Clustering
	Divisive Clustering
	TwoStep Clustering
Association Rule Mining	Association Rules (including FP-Growth)
Time Series	ARIMA
	Exponential Smoothing
	Seasonal Trend Decomposition
Bulk Algorithms	Bulk Matrix Operations
	Bulk Linear Regression

Task	Algorithm
	Bulk Principal Component Analysis

Terminology and Notation

When describing data mining tasks and algorithms, unless explicitly noted otherwise, assume that the analyzed data set describes a set of *instances* from a given *domain* X and each instance $x \in X$ is represented by a set of *attributes*. The domain is a set of real-world entities represented by the data, such as people, events, transactions, products, or devices. An attribute can be considered a function $a : X \mapsto A$ that assigns a value to each instance from the domain. All instances from the domain are described by a common set of attributes $a_1 : X \mapsto A_1$, $a_2 : X \mapsto A_2$, . . ., $a_n : X \mapsto A_n$. An instance $x \in X$ is represented by a vector of its attribute values $a_1\ x, a_2\ x, \ldots, a_n\ x$.

It is important to distinguish between *discrete* and *continuous* attributes. If a variable can take on any value between two specified values, it is called a continuous variable; otherwise, it is called a discrete variable. Continuous attributes take numerical values for which arithmetic operations can be meaningfully performed. Discrete attributes take a finite number of values that can be tested for equality, but cannot be reasonably used for any arithmetic calculations, even if they are represented numerically, which is common. For example, in counting attendees to an event, the attendance can be any integer between zero and the maximum capacity of the location. However, attendance cannot be any *number*—for example a fractional number—between those two limits, therefore, the attribute is discrete.

For some data mining tasks there is one specific *target attribute*. For example, this is the case for the classification task, where the target attribute is designated by $c : X \mapsto C$ and called the target concept. It also applies in the regression task, where the target attribute is designated by $f : X \mapsto \mathbb{R}$ and called the *target function*. In database terms, instances correspond to table rows and attributes correspond to table columns. When writing $a\ x$ to designate the value of attribute a for instance x, you refer to the value of the column representing a for the row representing x.

Using This Guide

This guide supplements the *IBM Netezza In-Database Analytics Reference Guide.* It describes the IBM Netezza In-Database Analytics call interface and provides some background information about the described algorithms and the data mining tasks they are used to solve. The algorithm description, which explains the algorithm purpose and principle of operation, is intended to help you choose the appropriate algorithms for a given task, run the selected algorithms, and interpret the results.

The *IBM Netezza In-Database Analytics Reference Guide* provides an overview and parameter description for each algorithm in the package. This information is also available as online help that can be requested by calling the stored procedures with the **'help'** argument.

This guide provides reproducible examples, with real data, that can be used as starting points for experiments. The examples demonstrate typical useful scenarios, which may include several different parameter combinations. They are described to an extent necessary to understand, modify, and repeat the examples, but may not fully cover the scope of possible parametrizations. The algorithms represent various complexity levels. They also differ significantly in their computational costs and the degree of possible parametrization.

The algorithms described in this guide are exposed to the user via different interfaces, all available from the Netezza software as stored procedures. No specific data mining knowledge or experience is required to follow the presented discussion and run the examples. The relatively brief descriptions of particular algorithms refer to the literature for more details where necessary.

CHAPTER 2
Algorithmic Tasks

Overview

There are a number of major data mining task categories leveraged by the IBM Netezza In-Database Analytics package:

- Data exploration
- Data transformation
- Model diagnostics
- Predictive modeling , including
  - Classification
  - Regression
  - Clustering
  - Association rule mining

The data exploration, data transformation, and model diagnostics categories are broad task families. They are comprised of several detailed analytical tasks, grouped together based on their common goals. While they are described in conjunction with details specific to the related algorithms, their concepts are covered only briefly here.

The predictive modeling category includes the classification, regression, clustering, and association rule mining tasks. As the name implies, the predictive modeling tasks are used to create models from the data. The models represent particular types of relationships that have a predictive utility and are used to make conclusions about new data from the same domain.

These tasks are defined by the model type and criteria used to evaluate model quality.

Data Exploration

The broad scope of the Data Exploration task family includes identifying both measures to characterize data distribution and relationships within the data. Data distribution is typically used with a single attribute to provide information about its most typical values, diversity, and possible outliers or unlikely values. Data relationships are used to detect dependencies within multiple attributes—usually two—where one or more attributes affect the distribution on another attribute.

Data exploration can be divided into two purposes:

distribution description

relationship identification

In both cases, different levels of detail are possible and different algorithms can be used, depending on attribute types and the level of detail required.

The main purpose of data exploration is to gain familiarity with the data, initially assess its predictive utility, detect possible data quality problems, and make observations that may be useful for subsequent analytical processing. While it is typically followed by predictive modeling, it may produce results that are useful on their own.

Data Transformation

The Data Transformation task family provides an intermediate step to transform the analyzed data set and make it more suitable for subsequent analytical processing. These transformations may modify values of selected attributes to satisfy requirements of classification, regression, clustering, or association rule mining algorithms used for predictive modeling.

The IBM Netezza In-Database Analytics package addresses three specific data transformations:

discretization—replacement of continuous attributes by discrete attributes, with values corresponding to original value intervals.

standardization and normalization—modification of continuous attributes to achieve desired distribution properties.

data imputation—assignment of values to fill missing attribute values.

Discretization

The discretization process assigns a discrete value to each interval of continuous attribute a to create a new discrete attribute a'. A discretization algorithm determines the interval boundaries that are likely to preserve as much useful information provided by the original attribute as possible. Data set discretization should preserve the relationship between the class and the discretized attributes if the data set is to be used for creation of a classification model.

Standardization and Normalization

Standardization and normalization transformations use the original continuous attribute a to generate a new continuous attribute a' that has a different range or distribution than the original attribute. Common transformations modify the range to fit the $[-1,1]$ interval (normalization) or modify the distribution to have a mean of 0 and a standard deviation of 1 (standardization).

Data Imputation

Many analytic algorithms require that the data set has no missing attribute values. However, real-world data sets frequently suffer from missing attribute values. Missing value imputation provides usable attribute values in place of the missing values, allowing the algorithms to run.

Model Diagnostics

The Model Diagnostics task family is used to reliably assess the quality of predictive models. To be useful, the predictive model must exhibit high accuracy both on the data set used to create it, but more importantly on new, unseen data from the same domain. Because the need for accuracy is high, the model diagnostic process requires appropriately selected model quality indicators, used to evaluate the model's prediction on a given data set. It also requires model evaluation procedures, which estimate quality indicator values on unseen data.

By providing model quality estimates, model diagnostics help determine if a model is acceptable for a given application. If several candidate models were generated, perhaps obtained using different algorithms, parameter setups, attribute sets or data transformations, model diagnostics make it possible to select an appropriate model for use.

Model Quality Indicators

Model quality indicators can be calculated on an arbitrary data set, including a training data set, which usually overestimates the model’s quality significantly. To avoid the optimistic bias, model evaluation procedures make a separate validation or a test set that is used only for evaluation redundant. This removes the optimistic bias, but can raise new issues of pessimistic bias and variance. Pessimistic bias can occur if the quality of the evaluated model is reduced by preventing a large portion of the available data from being used for model creation. Alternately, using a small test set increases the evaluation variance, resulting from a chance factor in data set selection, which makes the estimates unreliable. The variance can be kept low by aggregating the results of multiple independent evaluations, but at considerable computational expense.

Model Evaluation Procedures

The main challenge with model evaluation procedures is appropriately handling the tradeoff between the bias, variance, and computational cost.

The data set used for the evaluation process is referred to as the *validation* or *test* set, mostly depending on the purpose of evaluation. The distinction is based on the application context. For intermediate evaluation, used to guide model selection from several candidate models, it is more common to speak about the validation set. For the final evaluation of the model ultimately selected, it is more common to speak about the test set.

When creating models, it is acceptable to use all the available data. However, you should never use a subset of the *same data* to evaluate the model. Thus, it is reasonable to build a final model to be deployed for a particular application using the whole available data set, after another model had been built on a smaller training set using the same algorithm and parameter setup, and then evaluated on the remaining validation or test set. The quality indicators obtained from the evaluation process can serve as conservative, non-overestimating quality estimates for the final all-inclusive data model.

Predictive Modeling

Algorithmic Classification

Classification is a fundamental processes in business and in every day life. As the name implies, classification is the act of taking individual items and categorizing them into classes based on knowledge about the items. It is possible to classify items differently based on the criteria used. Classification is a common business activity, where it may be useful to organize customers, employees, transactions, stores, factories, products, devices, documents or any other types of instances into a set of re-defined meaningful classes.

In data mining, where analysis is being performed on a large amount of data, building classification models based on available data is a central task. Building a classification model requires that classes be defined from the data. Once defined, the data is then organized based on the defined classes. A challenge arises in data mining because typically the correct class labels are not known before the model is built.

Task Definition

The classification task consists of assigning instances from a given domain into a set of *classes*. The domain is described by a set of discrete- or continuous-valued attributes. Classes can be considered values of a selected discrete target attribute. The class represents a property of the classified instances that either becomes known later or, less often, a property that is difficult or costly to determine. It is for this reason that the correct class labels are generally unknown at the time of model creation, but they are provided for in a subset of the domain.

If unknown, the class is only available on a limited historical data set. This is why the application of a classification model is also referred to as *prediction*, and classification is considered one of the *predictive modeling* tasks.

The classification task can be used by a *classification algorithm* to create a classification model. The resulting model is a machine-friendly representation of the knowledge needed to classify any possible instance from the same domain that is described by the same set of attributes. The classification model representation may also be human-readable, but this is not always the case.

Mathematically, if:

- X denotes the domain

C is the set of classes

c is a particular class

the classification task is described as finding a model $h\colon X \mapsto C$ that approximates the target concept $c : X \mapsto C$, which is unknown except for a subset $D \subset X$. Usually, only a smaller subset $T \subset D$, referred to as the training set, is directly used by the classification algorithm for identifying a model. The remaining instances from D are held out for other purposes, usually model evaluation.

In database terms, in a data set where rows represent instances to be classified, one selected column contains class labels, and the remaining columns represent attributes. The task is to use the data set to create a model that can generate class labels for an arbitrary new data set that has the same attribute where correct class labels may not be known.

A classification model type that deserves special interest is the probabilistic classifier model, which is capable of estimating class probabilities $P\ d|x$ for arbitrary instances $x \in X$ and all classes $d \in C$. While such probabilities can be used to predict class labels using the maximum-probability rule, it is not always the best approach, such as when non-uniform misclassification costs must be incorporated into the classification process. Misclassification costs can be specified via a cost matrix

, with rows corresponding to predicted classes, columns corresponding to true classes, and entries $[d_{1,} d_2]$ containing the corresponding numeric cost of predicting class d_1 for an instance of true class d_2 . A simplified cost vector representation is often sufficient, with a single value $[d]$ representing the cost of predicting any class $d' \neq d$ for an instance of true class d . Such per-class misclassification costs are much easier to incorporate in classification algorithms.

Misclassification costs can be based on application-specific domain knowledge, if available, or subjectively adjusted to make the model more sensitive to some classes that are considered more interesting or harder to predict. Whenever misclassification costs are non-uniform, the maximum-probability rule for probabilistic classifiers should be replaced by the minimum-cost rule, which predicts the class associated with the least expected misclassification cost.

Consider the case of two-class classification tasks with $C=\{0,1\}$. For such tasks the expected misclassification cost of predicting class 1 for instance *x* can be expressed as $P\ 0|x\ \ [1,0]\ \ P\ 1|x\ \ [1\ ,1]$. Assuming a zero cost of correct predictions and writing $[1]$ instead of $[1,0]$, this can be further simplified to $P\ 0|x\ \ [1]$. Similarly, the expected cost of predicting class 0 for instance *x* is $P\ 1|x\ \ [0]$. Now the condition for class 1 to be the minimum-cost class for instance *x* is that its expected cost is no greater than that associated with class 0, which can be written as follows:

$$P\ 0|x\quad |1| \leq P\ 1|x\quad |0| \tag{1}$$

After substituting $1-P\,(1|x)$ for $P\ 0|x$ the inequality can be solved, yielding:

$$P\ 1|x \geq \frac{[1]}{[0]\ \ [1]} \tag{2}$$

which is the minimum-cost rule for two-class tasks. The rule determines an appropriate probability cutoff value that leads to the minimization of misclassification costs. The cutoff is equal to 0.5, which corresponds to the maximum-probability rule if both types of mistakes have the same cost.

Model Evaluation

A useful classification model should be accurate—regularly generating correct class labels—not only for the data set from which it was created, but more importantly for any previously unseen data from the same domain. Therefore, a model-building algorithm must detect relationships between class labels and attribute values in the available data set and *generalize* them appropriately so that they are likely applicable to new data. One basic way to assess a classification model's performance on a given data set is the misclassification error, which is one of model quality indicators presented in more detail in the Model Diagnostics section.

The misclassification error on a data set may serve as a reliable estimator of the true error, also called the generalization error, which is the probability of misclassifying an arbitrary instance from the domain, assuming the data set used for error calculation is independent of the training set. If D denotes the full available data set with known class labels and $T \subset D$ is the training set, then another subset $S \subset D - T$ should be used for reliable model evaluation. It is the responsibility of model evaluation procedures, discussed in the Model Diagnostics section, to keep the data used for model evaluation separate from the training data.

Sometimes the misclassification error is neither sufficient nor the most important performance measure. The error implicitly assumes that each wrong (or correct) prediction carries the same weight. This is not necessarily the case in several applications, where some model errors may be more severe than others.

Applications

The classification task is a very useful abstraction of many practical prediction tasks in a variety of application domains. Classification models are used to predict the future behavior of people, diagnose technical devices, monitor financial transactions, recommend actions or predict their outcomes. Examples of practical applications of the classification task include:

- customer classification based on sociodemographic profile and purchase history
  - target groups for different types of incentives
  - loyal or disloyal
  - likely or unlikely to react to an incentive
  - likely or unlikely to make a purchase of specific type or within a specific time-frame
  - likely or unlikely to switch to a competitor
  - high-performing and low-performing
  - interested or uninterested in a specific type of advertisement
- retail store classification based on location and sales history
  - high-performing and low-performing
  - appropriate and inappropriate for selling specific product types
- technical device classification based on operating logs and measurements

likely or unlikely to fail within a specific time frame

requiring different types of maintenance actions

credit card transaction classification based on the current transaction information, transaction history, and cardholder information likely and unlikely to be fraudulent

network intrusion detection based on network traffic logs

text document classification into a set of topic classes

Regression

The regression task definition is nearly the same as the classification task, the difference being that the target attribute is continuous. As a result, the regression task applications are similar. The target attribute to predict can represent financial indicators, sales or purchases, physical measurements, technical device parameters or performance measures, and many other quantities that need to be predicted based on historical data. The classification and regression tasks are sometimes referred to jointly as *supervised learning tasks*, which constitutes a major subclass of *inductive learning*.

Task Definition

The task consists of finding a regression model, based on a data set with known target attribute values, that captures and appropriately generalizes the relationship between the continuous target attribute and other available attributes, continuous or discrete. Such a model could generate predictions of target attribute values for arbitrary new data sets containing the same attributes. Consider an unknown target function $f : X \mapsto \mathbb{R}$ that assigns a real value to each instance from the domain X , and assume the availability of a subset $D \subset X$ for which the values of f are known. A regression algorithm should use a training set $T \subseteq D$ to find a regression model $h: X \mapsto \mathbb{R}$, approximating the target function on the entire domain. Instances are represented by means of their attribute values, as described above for the classification task, and the regression model is a computational representation of predictively useful relationships between the target function and the attributes.

The most common reason of the target function being generally unknown, except for an available data set, is that it represents some quantity related to the future, that is needed before it is known. A regression model created based on historical data is expected to predict the target function on new data. This makes regression another example of *predictive modeling*.

Model Evaluation

Similar to classification, successful regression models must generalize the relationships between the target attribute and the remaining attributes identified in the training set. This generalization is necessary to deliver satisfactory performance on previously unseen data. To assess a model's generalization capabilities it should be evaluated on a data set independent of the training set, which in practical terms means selecting a subset $S \subseteq D - T$ for model evaluation. This is the task of model evaluation procedures, discussed in the Model Diagnostics section.

Performance measures commonly used for regression models, such as the mean absolute error, the mean square error, and the relative absolute error are described in the Model Diagnostics section. These errors represent different ways of measuring how the predictions differ from the true values. A correlation coefficient is also a widely adopted regression performance measure since for some applications measuring the correlation can be more appropriate than measuring the difference.

Applications

The regression task is most often applied for numerical prediction in scenarios where a quantity needs to be approximately determined before it becomes available. This may be an amount of money spent or earned, an industrial process or technical parameter value, production or sales volume, or some more specific indicators of business decision outcomes. It is not uncommon for the regression model predictions to be used in an automated or manual optimization process as the objective function. This assumes the predicted quantity needs to be optimized by changing a number of factors possibly impacting it, and the model predicts the effects of such changes.

Some typical examples include:

predicting the value of customer purchases based on purchase history, applied incentives, and seasonal factors

predicting the audience of TV or radio advertisements based on the location and broadcast schedule information

predicting the resale price of off-lease vehicles based on vehicle VIN-based features, its condition, mileage, warranty, and geographical location

predicting the production yield of oil or gas wells based on the geological conditions or the applied well drilling and completion technology

predicting the time to failure of a technical device based on operating logs and measurements

demand forecasting for a product based on a recent sales track and market situation

Clustering

Clustering can be considered a form of classification in which there are no *a priori* given class labels and there is no predetermined target attribute. Instead, the set of possible classes is obtained as a set of clusters, created by analyzing the similarity patterns present in the data. The goal is to group instances similar with respect to their attribute values in the same cluster, and considerably different instances in separate clusters.

Task Definition

The clustering task can be seen as a superposition of two sub-tasks:

cluster formation—data partitioning into similarity-based clusters

cluster modeling—classification with classes corresponding to the clusters

While two distinct algorithms could be used to address the two sub-tasks, it is more common to use a single clustering algorithm for both cluster formation and modeling. Using a single algorithm is convenient since the same principle responsible for cluster formation based on available data can be also used to classify new data to existing clusters. A clustering model provides both a list of clusters identified based on a data set and a mechanism for assigning new instances, described by the same set of attributes, to these clusters. It can be applied to new data as in the case of a classification model.

Formally, the clustering task consists in finding a clustering model $h : X \mapsto C_h$, where C_h is a set of similarity-based clusters associated with *h*, based on a data set $T \subseteq D \subset X$. Like the classification and regression task, the training set T may be a subset of the available data set D, containing instances described by a set of attributes, but this time with no designated target attribute. The clustering model should maximize the intra-cluster similarity and minimize the inter-cluster similarity. The exact meaning of this general similarity principle is established by particular clustering algorithms. Several algorithms use an explicit distance function to measure the (dis)similarity between instances.

An interesting extension of the clustering task occurs when adding the requirement of cluster hierarchization. The resulting *hierarchical clustering* task requests that each cluster be further partitioned into sub-clusters, which have their own sub-clusters, and so on. This implies a tree representation of the clustering model, with nodes representing clusters, and their descendant nodes representing their sub-clusters. A hierarchical clustering model assigns a sequence of clusters to each instance from consecutive clustering tree layers, or, alternatively, a single cluster from a specified level.

Model Evaluation

Unlike the classification and regression tasks, there are no widely accepted “objective” performance measures for clustering models. While a variety of measures have been described in mathematical literature, in practice they are used as supplementary rather than primary criteria for model selection. The latter are mostly based on domain and application-specific requirements and preferences. The former can be roughly divided into:

distance-based cluster quality measures—using an explicit distance function, they measure how close instances are from the same cluster to one another and how distant instances are from different clusters from one another

probabilistic cluster quality measures—treating the clustering as a representation of a mix of probability distributions, they measure how likely the data set is to have been generated from this mix

external cluster quality measures—assuming the availability of a reference attribute on some data subset, not used for clustering, but representing the available domain knowledge about the desired way of partitioning the data, they measure the consistency of the clustering with the reference attribute

The third category of cluster quality measures is of little practical interest, since whenever the clustering is needed, an appropriate reference attribute is unlikely to exist. Such measures are usually applied in research for benchmarking and comparing clustering algorithms. When using distance-based clustering algorithms, it is typical to evaluate their effects using distance-based quality measures using the same distance function.

Applications

Some clustering applications can be summarized as follows:

clustering can provide useful insights about the similarity patterns present in data and a clustering model can be considered as knowledge *per se*

customer segmentation

point of sale segmentation

document catalog creation

clustering can be performed on a selected subset of *observable* attributes that are available for all instances, and used to predict *hidden* attributes that are impossible or difficult to determine for some instances based on cluster membership

customer clustering based on socio-demographic attributes to predict attributes describing purchase behavior

point of sale segmentation based on location, building, and local population features, used to predict attributes describing selling performance

clustering performed on a set of “normal” instances can be used for anomaly detection, by issuing alerts for new instances that do not fit an existing cluster

network traffic clustering, used for intrusion detection

credit card transaction clustering, used for fraud detection

sensor signal clustering, used for device fault detection

clustering can be used as a domain decomposition method for some further data mining tasks, which may be easier to apply within homogenous clusters

customer clustering and classification with respect to loyalty within clusters

customer clustering and predicting reaction to incentives within clusters

credit card account clustering and classification with respect to fraud likelihood within clusters

product clustering and demand forecasting within clusters

used vehicle clustering and price prediction within clusters

Association Rule Mining

The association rule mining task assumes that instances from the domain can be described by so called itemsets rather than attribute values. Unlike attributes, the number of items associated with particular instances can and usually does differ substantially. One common example is a retail purchase transaction, described by the list of purchased items. The goal of association rule mining is to identify sets of frequently co-occurring items.

Task Definition

Assume that for each instance $x \in X$ there is a set of associated items $I_x \subset \boldsymbol{I}$, where $\boldsymbol{I}$ denotes the set of all possible items. This is equivalent to an attribute representation with one attribute $a_i : X \mapsto \{0,1\}$ for each item $i \in \boldsymbol{I}$. However. the sets of items corresponding to particular instances are usually very small subsets of the set of all possible items $\boldsymbol{I}$, which makes such a representation inconvenient. Any subset of $\boldsymbol{I}$ is called an *itemset*, and an ordered pair of itemsets $A, B \subset \boldsymbol{I}$ written as $A \Rightarrow B$, is called an *association rule*. The task of association rule mining consists in finding highly reliable and useful association rules based on a provided data set $D \subset X$. For consistency with the other data mining tasks discussed previously, this document uses the term “model” to refer to the output of the association rule mining task, that is, a set of association rules.

It is not uncommon to adopt a simplified view of the association rule mining task, where *frequent itemsets* are to be found rather than association rules. The meaning of ‘frequent’ refers to a high occurrence rate in the analyzed data set and is explained when discussing model evaluation. The difference between the rule and itemset formulation is not substantial, since frequent itemsets can be used to generate association rules in a relatively simple and efficient way without accessing the data. This is how association rule mining algorithms work.

Model Evaluation

For the association rule mining task the model itself is not evaluated, but rather its individual components: single association rules or itemsets. The evaluation is based on counting instances from a data set $S \subset X$, which can either be the same data set from which the model was derived or a new data set on which it is evaluated—with appropriate itemsets. The two most commonly used performance measures for association rules are:

support:

$$\mathrm{supp}_S(A \Rightarrow B) = \frac{|S_{A \cup B}|}{|S|} \tag{3}$$

Confidence:

$$\mathrm{conf}_S\; A \Rightarrow B = \frac{|S_{A \cup B}|}{|S_A|} \tag{4}$$

Where $S_I = \{x \in S \mid I_x \subseteq I\}$ for any itemset $I \subset \boldsymbol{I}$ is the subset of S consisting of instances such that their associated sets of items are completely contained in I. The support is therefore the ratio of all instances from S containing all items from A and B, and the confidence is the ratio of such instances to instances containing all items from A only. Sometimes the common numerator of the fractions defining the support and confidence is referred to as the absolute support, as opposed to the relative support defined above.

The support is also defined for itemsets. The support of itemset I on data set S is calculated as:

$$\mathrm{supp}_S\; I = \frac{|S_I|}{|S|} \tag{5}$$

Again, the numerator of this fraction is referred to as the *absolute* support. Itemsets with support exceeding a certain user-specified threshold are called *frequent itemsets*. Association rules are considered reliable and useful if both their support and confidence exceed user-specified thresholds.

Applications

The most common example of market basket analysis used to explain association rules is also likely the most common application. Association rules discovered from customer purchase transactions are highly valued in retail marketing. High-support and high-confidence product associations can be used to make individualized product recommendations for direct marketing campaigns, design cross-selling promotion offers, choose attractive rewards for loyalty programs, prepare advertisements or booklets, and adjust product placement on store shelves. The scope of possible association rule mining applications is much wider than just the retail world, however. For example, it is not uncommon to look for associations in medical or biological data, in technical device logs, in car or plane crash reports, in crime or terrorist attack reports.

CHAPTER 3
Working with Data Sets

Sample Data Sets

Most data sets used throughout this guide come from the *UCI Machine Learning Repository*, widely used by the data mining community for benchmarking algorithms. This section lists the data sets used in the guide, as well as data set configuration instructions.

CensusIncome

The *CensusIncome* data set comes from the 1994 and 1995 population surveys by the US Census Bureau. The UCI repository contains two different data sets extracted from these surveys. While the smaller *Adult* data set is sometimes also known as “Census Income,” this guide refers to the larger *Census-Income (KDD)* data set—used here to demonstrate data exploration, data transformation, classification, and clustering algorithms—as the “*CensusIncome* data set.”

Data Description

The data set contains 299,285 instances and 40 attributes, 33 discrete and 7 continuous. The attribute **income** represents the class for classification. The remaining attributes describe the demographic, social, professional, family, and financial situation of individuals to be classified into two income classes.

Data Set Configuration

Refer to the *IBM Netezza Analytics Administrator's Guide* for instructions for acquiring the data file and configuring the database tables and data, which must be performed before the examples in this guide based on the *CensusIncome* data set can be used.

WineQuality

The *WineQuality* data set describes the properties of the Portuguese *Vinho Verde* wine. The UCI repository contains one data set for white wines and one for red wines. This guide uses the larger white wine data set to demonstrate data exploration and regression algorithms.

Data Description

The data set contains nearly 4900 instances and 12 continuous attributes. One attribute, **quality**, represents the target function for regression. It is the quality evaluation provided by experts and expressed numerically in the $[0,10]$ range. The remaining attributes represent physicochemical wine properties that should be used to predict wine quality.

Data Set Configuration

Refer to the *IBM Netezza Analytics Administrator's Guide* for instructions for acquiring the data file and configuring the database tables and data, which must be performed before the examples in this guide based on the *WineQuality* data set can be used.

Retail

The *Retail* data set used in the Netezza database is based on the *Belgian Retail* data set, which is market basket data from an anonymous Belgian retail store, available from the *FIMI Dataset Repository* courtesy of Tom Brijs. It is used in this guide to demonstrate association rule mining.

Data Description

The data set covers over 88,000 store receipts from more than 5000 customers. Each receipt represents a purchase transaction and has a number of items associated with it. A more detailed description of the data set can be found in Tom Brijs' paper Retail Market Basket Data Set.

Data Set Configuration

Refer to the *IBM Netezza Analytics Administrator's Guide* for instructions for acquiring the data file and configuring the database tables and data, which must be performed before the examples in this guide based on the *Retail* data set can be used.

CHAPTER 4

Using Netezza Analytic Procedures

When enabling a database for use with analytics procedures, you must both enable the database and set individual user access permissions. There are four scripts used to ensure that databases are properly created and that access rights for various classes of user are properly set. See the *IBM Netezza Analytics Administrator's Guide* for a complete description of setup, as well as important usage notes.

Call Interface

The algorithms discussed in this manual are available through stored procedures that can be called directly from nzSQL. The call interface follows the following common pattern:

```
CALL proc('arg1=val1, arg2=val2, …, argn=valn');
```

Each procedure has a single character string (NVARCHAR) argument parsed as a parameter string, which contains a number of comma-separated argument specifiers. A single argument specifier of the form

```
arg=val
```

provides a value **val** for a named argument **arg**. Some procedures accept a list of semicolon-separated values for a single argument:

```
arg=val1; val2;...; valm
```

Algorithm parameter names can be supplied in any case (upper, lower, or mixed) and in any order. White space between parameter names, equal signs, and other separator characters are also allowed. Algorithm argument values fall in to the following categories:

numeric constants

boolean constants

string constants

table names or column names

When the parameter string is parsed for a particular algorithm, its arguments are recognized and cast to appropriate data types. Arguments that specify table or column names are also converted to the database's default letter case, unless bounded by double quotes. In the following example, if the stored procedure is called in a database defined to use upper-case names, the value of argument **intable** is converted to upper case and the value of argument **outtable** remains in lower case due to the double quotes around the argument value:

```
CALL proc('intable=tab1, outtable="tab2"');
```

will be implicitly converted to:

```
CALL PROC('INTABLE=TAB1, OUTTABLE="tab2"');
```

The same double-quoting mechanism is required when passing table or column names containing non-letter special characters that cannot be directly used in *nzSQL*. Table and column names can contain national characters.

For some procedures, a column name passed as an argument value must pass an additional value used to specify a column-specific operation or parameter for the algorithm. In such cases a colon is used to separate the actual name and the additional value assigned to it:

```
arg=colname:val
arg=colname1:val1; colname2:val2; …; colnamem:valm
```

The **'help'** parameter string can be used with any procedure to display the list of accepted arguments and a description of the algorithm's operation.

```
CALL proc('help');
```

It is best to run analytic stored procedures in user-created databases that have been properly initialized to work with models, not in the NZA database. Algorithms always write output tables or models to the current database. Subsequent upgrades to Netezza In-Database Analytic functions could result in deleting your output tables or models if they are stored in the NZA database. Input tables referenced in the 'intable' parameter may be qualified with a database name qualifier (for example, nza..CensusIncome). See Working With Models or Metadata Management for more information on initializing models.

Some common parameters are described in the table below, although not all listed parameters are used by every algorithm. Many algorithms accept or require these parameters; this varies from algorithm to algorithm. This is not a complete list of all possible parameters. Parameters that are specific to a particular algorithm or a small set of algorithms are discussed fully in the the detailed algorithm section.

Table 2: Partial list of common parameters

Parameter	Description
by	Identifies a column name for grouping, in algorithms that support this. For example: `'by=age'`
check	Sets the type of validation check the algorithm performs on

Parameter	Description

	the data.
	▶ A setting of NULL checks the data for a NULL value in any column.
	▶ A setting of ALL checks that each ID column value is unique and that there are no NULL values in the columns. (Note that this can be quite time consuming.)
	▶ A setting of NONE turns off the validation check. For example:
	`'check=all'`
	`'check=none'`
	`'check=nulls'`

coldefrole	Identifies the default role for the columns of the input table. See Column Properties for more information. For example: `'coldefrole=ignore'`
coldeftype	Identifies the default type for the columns of the input table. See Column Properties for more information. For example: `'coldeftype=cont'`
colPropertiesTable	Identifies the name of the table that contains information about the properties of the columns of the input table. These properties are used by the algorithm. See Column Properties for more information. For example: `'colPropertiesTable=colPropertiesCensus'`
help	Generates help output for the specific algorithm. When used, this must be the only parameter supplied on the call. For example: `call nza..hist('help')`
id	Identifies the unique identifier column name for the table being processed. If this parameter is required and not supplied, the default column name is assumed to be 'id'. For example: `'id=itemid'`
incolumn	Identifies a single column or set of columns as input to the algorithm. Columns can have 'properties' and 'roles' associated with them. See Column Properties for more information. For example:

Parameter	Description

	`'incolumn=mycolumn'`
	`'incolumn=col1; col2; col3'`

intable	Identifies the name of the input table that contains the data on which the algorithm will operate. The table name may contain a database qualifier. For example: `'intable=mydb..mytable'`
model	Identifies the name of the model to be created (for algorithms that generate a model) or the name of the model to be processed (for algorithms that act on existing models). When creating a model, the model is created in the database where the algorithm is run. The database must be initialized to work with metadata management. (See Metadata Management for more information.) The model name must be unique within the database and cannot already exist. For example: `'model=my_arule_model'`
outtable	Identifies the name of the output table to be created. The table is created in the database where the algorithm is run. The table cannot already exist in the database or an error is issued. For example: `'outtable=myouttable'`
target	Identifies the column name that splits the input data into different class groups. If this parameter is required and not supplied, the default column name is assumed to be 'class'. For example: `'target=sex'`

Column Properties

Column properties provide the user with control over designating the type of data in columns being processed without relying on the algorithm's default casting of data types. Supplying a property value allows the caller to identify nominal and numeric columns. In addition to types, the column property specification also allows the caller to identify the role a column plays in the computation being done by the algorithm. Column roles are specific to each individual algorithm. Column types have a defined set of values.

The following are valid columns types:

nominal [nom]

continuous [cont]

To supply a property to a column in the *incolumn* list, the property comes after the column name separated by a colon. For example:

```
CALL    nza..EWDISC('intable=nza..CensusIncome_train,
incolumn=age:5;    num_persons_worked_for_employer:2;
weeks_worked_in_year:5,
outtable=ci_ewd');
```

In the example above, additional data for the columns being referenced in the *incolumn* parameter is supplied. The meaning of the *role* or *descriptive* data that follows the colon (:) is specific to this algorithm. For more details on a specific algorithm, refer to the *IBM Netezza In-Database Analytics Reference Guide*.

In the next example, column *type* for the columns being referenced in the *incolumn* parameter is supplied:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train,
incolumn=DETAILED_INDUSTRY_RECODE:nom; AGE:cont, id=id, target=income,
model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

Columns having roles of **target** and/or **id** can be also specified with the *incolumn* parameter. Using this method, the example above would now look as follows:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train,
incolumn=DETAILED_INDUSTRY_RECODE:nom; AGE:cont; id:id; income:target,
model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

Properties that are related to more than one parameter are separated by a “|” vertical bar. For example:

```
CALL nza..STD_NORM('intable=nza..CensusIncome_train,incolumn=age:S;
wage_per_hour:N; capital_gains:L;capital_gains|capital_losses:C, id=id,
outtable=CensusIncome_train_std_num');
```

In addition to setting column properties for individual columns in the *incolumn* parameter, it is also possible to set a default type for all columns using the *coldeftype* parameter. If coldeftype is specified and a column property is specified for a column in the *incolumn* parameter, the property specified in the *incolumn* parameter takes precedence. For example:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train, coldeftype=nom,
incolumn=AGE:cont; capital_gains:cont; capital_losses:cont;
dividends_from_stocks:cont;
NUM_PERSONS_WORKED_FOR_EMPLOYER:ignore; WAGE_PER_HOUR:ignore;
WEEKS_WORKED_IN_YEAR:ignore; id:cont,
id=id, target=income, model=ci_tree1, eval=gini,
minimprove=0.005, minsplit=1000');
```

In this example, by default, columns are treated as nominal except those which are continuous due to being specifically designated as such in the *incolumn* parameter.

Similar to *coldeftype*, default role can be specified with *coldefrole,* as in the example below:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train, coldefrole=ignore,
incolumn=DETAILED_INDUSTRY_RECODE:nom; AGE: cont, id=id, target=income,
model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

In the example above all columns will be ignored except those that are defined by incolumn, id, and target.

Another method for supplying column properties for an input table is to create a column properties table associated with the specific input table. This provides a convenient way to specify this information for tables that are used frequently in algorithms and ensures consistency in how the data is handled with each algorithmic call. When naming a column properties table in an algorithm's call the caller does not need to specify column properties in the *incolumn* parameter or use the *coldeftype* or *coldefrole* parameters, but may do so to override the properties specified in the column properties table.

You can use the following procedures to create and manage a column properties table:

COLUMN_PROPERTIES

COLUMN_PROPERTIES_CHECK

SET_COLUMN_PROPERTIES

GET_COLUMN_LIST

Using a Column Properties Table

The following example returns a list of specified columns separated by semicolons. For example:

```
CALL nza..GET_COLUMN_LIST('colPropertiesTable=colPropertiesIris,
role=input;id;target;input, type=nom;cont, separator=;');
```

```
                   GET_COLUMN_LIST
     -----------------------------------------
      "ID";"SEPALLENGTH";"SEPALWIDTH";"CLASS"
```

Below is an example of calling an algorithm supplying a column properties table, which would have been created in a separate step.

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train,
colPropertiesTable=colPropertiesCensus,
incolumn=DETAILED_INDUSTRY_RECODE:nom; AGE:cont, id=id, target=income,
model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

COLUMN_PROPERTIES Procedure

This procedure creates a column properties table. All columns are set by default (that is, numeric types are set to continuous). For example:

```
call nza..COLUMN_PROPERTIES('intable=nza..CensusIncome,
outtable=colPropertiesCensus');
```

This table also defines a *role* for each column. The following are possible roles:

id – column is an identifier

target – column is a target value

ignore – attribute is ignored

objweight – contains weights of objects (if algorithm supports weighting of objects, if not, this column is ignored

input – for all input variables (default role)

The example below demonstrates how to create a column properties table, using *coldeftype* and *incolumn* parameters, with a combination of default values and overrides to default values.

```
call nza..COLUMN_PROPERTIES('intable=nza..Iris,
outtable=colPropertiesIris, coldeftype=cont,
incolumn=id:id; class:nom; petallength:ignore; petalwidth:ignore');
```

The procedure nza..COLUMN_PROPERTIES() creates a new table:

```
 IDCOL  |   COLNAME   |           COLDATATYPE           | COLTYPE | COLROLE |
 COLWEIGHT
---------------------------------------------------------------------------------
      1 | ID          | INTEGER                         | cont    | id      |     1
      2 | SEPALLENGTH | DOUBLE PRECISION                | cont    | input   |     1
      3 | SEPALWIDTH  | DOUBLE PRECISION                | cont    | input   |     1
      4 | PETALLENGTH | DOUBLE PRECISION                | cont    | ignore |      1
      5 | PETALWIDTH  | DOUBLE PRECISION                | cont    | ignore  |     1
      6 | CLASS       | NATIONAL CHARACTER VARYING(12)  | nom     | input   |     1
```

You can modify this table and pass it as column properties to procedures that support this. The default Coltype is set based on a column datatype (read from the table definition). Colweight is set to 1 by default.

COLUMN_PROPERTIES_CHECK Procedure

This procedure checks if the *colPropertiesTable* table is correct. Below is an example:

```
CALL nza..COLUMN_PROPERTIES_CHECK('intable=nza..iris,

colPropertiesTable=colPropertiesIris');
```

An exception is raised if:

type is incorrect or inconsistent with the table datatype

more than one id or objweight is defined

column name is incorrect (does not match input table columns)

type does not match the specification [cont, nom]

role and type do not match one of the allowed values

the datatype cannot be cast on an attribute (for example if the varchar column has been defined as continuous)

SET_COLUMN_PROPERTIES Procedure

This procedure sets and/or updates properties in the specified *colPropertiesTable* table.

```
CALL nza..SET_COLUMN_PROPERTIES('intable=nza..iris,
colPropertiesTable=colPropertiesIris, incolumn=petallength:input;
petalwidth:input');
```

GET_COLUMN_LIST Procedure

This procedure retrieves a list of specified columns from a *colPropertiesTable*.

```
CALL nza..GET_COLUMN_LIST('colPropertiesTable=colPropertiesIris,
role=input;id;target;input, type=nom;cont, separator=;');
```

The above example returns a list of specified columns separated by semicolons. For example:

```
              GET_COLUMN_LIST
-----------------------------------------
 "ID";"SEPALLENGTH";"SEPALWIDTH";"CLASS"
```

Below is an example of calling an algorithm supplying a column properties table, which would have been created in a separate step.

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train,
colPropertiesTable=colPropertiesCensus,
incolumn=DETAILED_INDUSTRY_RECODE:nom; AGE:cont, id=id, target=income,
model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

The table *colPropertiesTable* contains property data for the input table columns, but the *incolumn* parameter has higher priority and overrides the settings from the column properties table. For example, in the example above, *age* will be treated as *continuous* regardless of what is specified in the column properties table.

Algorithms Supporting Column Properties

Below is a list of algorithms that support column properties:

Naïve Bayes (NAIVEBAYES)

Decision tree (DECTREE)

Regression tree (REGTREE)

Divisive clustering (DIVCLUSTER)

K-means (KMEANS)

kNN (KNN)

GLM

Linear model

PCA

Tree-shaped Bayesian networks

Missing Value Support

Many real world databases suffer from missing values in tables. One solution to this problem is to preprocess these tables to either:

remove rows or columns with missing values using SQL queries replace missing values with some special value using SQL queries

impute the value by using the Netezza Analytics IMPUTE_DATA procedure

For easier processing and better model quality and predictions, Netezza Analytics provides an internal solution to deal with the missing values. Depending on the algorithm, functions will either intelligently handle or skip the missing table data.

The following selected algorithms are capable of building or applying models using tables with missing values, internally handling missing values in an appropriate manner (instead of just ignoring instances with missing values):

Decision Trees

Regression Trees

Naïve Bayes classifier

For other algorithms, if rows contain missing values, the rows are ignored, but the table is still used. Preprocessing is still possible, using the Netezza Analytics supplied IMPUTE_DATA procedure, but is not required. Note that preprocessing is not "automated." When receiving a data set with missing values, each algorithm either processes them in its own way (decision trees, regression trees, naive Bayes) or ignores instances (rows) with missing values. Preprocessing may be explicitly applied to get rid of missing values. For algorithm-specific details of missing value support, see the algorithm description in the *IBM Netezza In-Database Analytics Reference Guide*.

Working With Models

The Metadata Management feature provides an environment for managing the analytics models created by the Netezza Analytics software. The implementation of the Metadata Management component is done on top of the existing database system, using stored procedures and user-defined functions.

All analytics models created by the various Netezza Analytics functions (like DECTREE or KMEANS) are registered in a catalog. There are administrative views and functions that are provided for model management. The Metadata Management system provides the following capabilities:

Listing information about models

Performing basic operations on models (for example, delete, copy, rename, update)

Performing advanced operations on models (for example, print, PMML format and export)

Securing data (grant and revoke privileges on models and model operations)

Note: The Metadata Management feature is utilized by all algorithms that generate models. Any database used for models must be initialized for use by the Metadata Management feature. This is done by calling **nza..initialize().**

All model manipulation should be done using the Metadata Management provided functions. Model tables should not be altered, updated, dropped, etc.. using normal SQL/DDL statements.

For more information, see the section on Metadata Management.

CHAPTER 5

Metadata Management

Introduction

IBM Netezza Analytics provides a set of data mining algorithms that generate analytic models. An analytic model is considered to be any data mining model, such as a decision tree or a regression tree, generated by one of the data mining algorithms. Each algorithm stores a generated analytic model in one or more database tables or Netezza Matrices. The tables can be read by the user via normal SELECT statements; the matrices can be accessed using functions provided by the Netezza Matrix Engine.

In addition, a set of tools for Metadata Management for the analytic models are provided. Metadata management provides a means to track information about the tables created and used by analytic models. The tools used for this tracking are a set of database objects, such as:

tables, views, and stored procedures that provide a catalog for analytics models

functions to manage the models

a security environment for the models

Metadata Management provides provisions for exporting models using *Predicitive Model Markup Language* (PMML), an XML-based format used to store and exchange data mining models between different modeling platforms.

The Metadata Management component is implemented “on top” of the existing database system. This method allows analytics models to be managed in a manner similar to other database objects such as tables and views.

Structure

The Metadata Management interface consists of a set of database views and a set of stored procedures.

The catalog for the metadata of analytics models is made up of a number of tables. All analytics models created by the various data mining functions, such as DECTREE or KMEANS, are registered in the catalog. The internal catalog tables for metadata storage of analytics models are not directly accessible for the normal analytics user; data access is provided by special views and procedures.

Custom views and a set of public procedures are used to perform the necessary operations on the analytics models. The metadata for analytic models for a given database is stored in that database. As a result, the database views used for metadata are also located in the local database. These views are created during database initialization. The stored procedures used for Metadata Management, however, are centrally located in the NZA database, and are created during IBM Netezza Analytics installation. For more information on database initialization, refer to the Metadata Administration Objects section.

Privileges for operations on models can also be managed for users and groups.

Preparing the Database

Each database where analytics models are to be used must be initialized by the system administrator using the **create_inza_db.sh** script. In addition, each user that needs to use the analytics functions must be prepared by the system administrator using the **create_inza_db_user.sh** script. For more information on these scripts, refer to the *IBM Netezza Analytics Administrator’s Guide*.

Metadata Management Tools

A number of administrative views and other functions are offered that allow you to work with the metadata information for each model. In addition, there is a security layer for the analytics models that provides privileges for operations on models that can be granted to users and groups.

The metadata management system provides the following tools, accessible though the public interface of Stored Procedures and Database Views described in the following sections.

Model Property Objects

List all analytics models and their properties, for example name, owner, create date, etc.

List all tables/views/matrices belonging to an analytics model

List all parameters used to create an analytics model

List all column properties defined for the input data used to create a model

Filter the listed objects by their properties

Model Management Objects

Delete one or more analytics models, both underlying tables/view/matrices as well as metadata

Copy an analytics model

Rename an analytics model

Update the properties of an analytics model

Update the contents of an analytics model

Print an analytics model to the console

Convert analytics models to the PMML format and export them, for example to use for visualization

Model Security Objects

Set initial security privileges on new analytics models

Change the owner of an analytics model

Grant or revoke privileges on operations with analytics models to users and groups

Metadata Administration Objects

Enable a database for metadata management

Determine the version of the metadata objects

Determine if a database is initialized to utilize metadata

Remove metadata objects from a database

Usage Scenarios

The functionality offered by the metadata management system provides access to analytics models and their properties. The stored procedures and database views that are the public interface to the metadata management functionality are typically accessed through the command line or using SQL scripts executed on the command line. Access using ODBC/JDBC also allows external applications to perform management functions.

Some typical scenarios for using the management functions on analytic models are:

Using the list functions to get an overview of the models

Using the drop functions to remove models that are no longer needed

Using the alter function to categorize and describe the models

Using the copy function to exchange a model with other users on the same NPS

Using the PMML conversion function to create a PMML model, for example, for using a PMML visualizer

get a quick overview of a model using the print function

Model Naming Conventions

Model Name Length

Model names are limited to 64 characters since they are used to generate names for model components (tables/views/matrices). Since internal table names have a character limit, the model name in turn must also be limited. For more information see the Model Component Naming Conventions section.

Model Name Uniqueness

Model names must be unique throughout a database. As a result a user cannot create a model with the same name as a model that is already used in the database, regardless of the user that created the model.

System Case

By default, the Netezza system uses uppercase letters for table names and model components. This is known as the *system case*. The system case can be configured to use lowercase instead, which was the default in earlier Netezza releases; however, it is not recommended to convert an existing uppercase system to a lowercase system or vice versa. When entering table names or model component names in SQL commands on an uppercase system, they are converted to all uppercase unless they are enclosed in double quotes ("). When surrounded by double quotes, names are saved as entered.

Model Component Naming Conventions

Each algorithm stores its analytic models in one or more database components (tables/views/matrices). The case used for the naming conventions is dependent on the case setting of the NPS. By default, the Netezza system uses uppercase letters to display SQL output. The system case can be configured to use lowercase instead, which was the default in earlier Netezza releases.

The system automatically converts identifiers, such as database, table, and column names, to the default system case, which is Upper on new systems. To use mixed case and/or spaces, double quotes must be used around the identifier. For example:

```
CREATE TABLE "Emp Table" (emp_id integer, emp_name char(20));
SELECT emp_id FROM "Emp Table";
```

General Component Naming Conventions

Each component is assigned a *canonical name* using a special naming schema. The name is generated in the form:

<Prefix>_<Derived_model_name>_<Derived_component_type>_<Sequence_ID>

Where:

<Prefix> is always **NZA_META** for uppercase systems and **nza_meta** for lowercase systems..

<Derived_model_name> is the model name created using the conventions described in the Model Naming Conventions section. The maximum length is 64 characters.

<Derived_component_type> is the usage type in the case specified for the database, for example **MODEL**, **PMML**, or **INPUT** for uppercase databases and **model**, **pmml**, or **input** for lowercase

databases. Any instances of one or more consecutive spaces are converted to an underscore ("_"). Maximum length is 38 characters if a sequence ID is used; 42 characters otherwise. For more information on table types, refer to NZA..LIST_COMPONENTS Procedure section.

The sequence ID, if needed; otherwise omitted. Maximum length is 4 characters.

Model Naming Conventions

Due to certain table naming requirements on the IBM Netezza Appliance, the derived model name has some limitations.

White Space Conversion

One or more consecutive spaces in a model name are converted to a single underscore ("_"). In SQL, using spaces in table names force the table name to be quoted in SQL statements. However, readable model names often include spaces. Using an underscore eliminates the need for such quoting.

Note: Some similar model names, for example "Model 1" and "Model_1" cannot be used in the same database, as they both resolve to "Model_1".

Case Conversion

The *derived* model name is the only element of the name where the case may not be automatically converted based on the system case. Model names, like table names in the database, are case-sensitive, therefore converting them to the system case could cause conflicts.

As an example, consider a model called "Sample Model" that, due to its structure, uses two tables with usage type "Column Statistics." In this case, the table names are:

NZA_META_Sample_Model_COLUMN_STATISTICS_0

NZA_META_Sample_Model_COLUMN_STATISTICS_1

Case Sensitivity Examples

Since model names are treated in a similar manner to database objects in SQL statements, the following apply:

A model name is converted to the system case.

A model name in double quotes is not converted .

Model name literals in a where clause, for example in the list_model() procedure, are not converted.

For example, Table 3 illustrates some examples for a Netezza appliance whose system case is the default uppercase.

Table 3: Case sensitivity examples

Command	Result
CREATE TABLE foo (x integer)	Creates a table FOO

Command	Result
nza..kmeans('model=foo ... ')	Creates a model FOO
CREATE TABLE "foo" (x integer)	Creates a table foo
nza..kmeans('model="foo" ... ')	Creates a model foo
SELECT * FROM _V_TABLE WHERE TABLENAME = 'foo';	Returns the table information only if the table name is foo (literals in a WHERE clause are not converted).
nza..list_models('where=modelname=''foo'' ');	Lists the model only if the model name is foo (not FOO).

Model Property Objects

For accessing model properties, the following views and stored procedures are defined:

Views

- V_NZA_MODELS View
- V_NZA_COMPONENTS View
- V_NZA_PARAMS View
- V_NZA_COLPROPS View

Stored procedures

- NZA..LIST_MODELS Procedure
- NZA..LIST_COMPONENTS Procedure
- NZA..LIST_PARAMS Procedure
- NZA..LIST_COLPROPS Procedure

The views are required for programmatic access; however, a number of stored procedures are defined that output a subset of the view's information to the console. These procedures are intended to be used in a manner similar to the “\” commands in nzsql, that is, the procedures display only the basic information for the objects in a more readable format.

V_NZA_MODELS View

The V_NZA_MODELS view provides access to the core metadata of an analytic model. This view lists all analytic models and their properties. Each record contains the metadata for an individual model. Following are the view columns:

Table 4: Columns of the V_NZA_MODELS view

Column Name	Column Type	Description
MODELNAME (unique key column)	NVARCHAR(64)	The name of the analytics model.
OWNER	NVARCHAR(128)	The name of the owner of the model.
CREATOR	NVARCHAR(128)	The name of the creator of the model.
CREATED	TIMESTAMP	The date and time when the model was created.
MODIFIED	TIMESTAMP	The date and time when the analytics model was most recently modified.
STATE	VARCHAR(16)	The state of the model.
DESCRIPTION	NVARCHAR(8192)	A user-defined description of the model.
COPYRIGHT	NVARCHAR(128)	A copyright notice for the model.
MININGFUNCTION	VARCHAR(64)	The mining function of the model.
ALGORITHM	VARCHAR(64)	The name of the algorithm used to create the model.
COMPONENTFORMAT	VARCHAR(16)	A version identifier indicating the format of the model tables.
APPLICATIONNAME	VARCHAR(64)	The name of the application that created the model.
APPLICATIONVERSION	VARCHAR(16)	The version of the application that created the model.
USERCATEGORY	NVARCHAR(64)	A user-defined category name.

Detailed Column Descriptions

MODELNAME—The name of the analytics model. For more information, refer to the Model Naming Conventions section.

OWNER—The name of the regular database user that is the owner of the model. Initially, the owner is set to the same name as the creator.

Note: If the name of the database user is changed, the view shows the changed name correctly. However, if the database user is dropped from the database, the view shows the name of the dropped user until a new owner is assigned by an administrator.

CREATOR—The name of the regular database user who called the procedure to build the model.

Note: If the name of the database user is changed, the view shows the changed name correctly. However, if the database user is dropped from the database, there is no procedure to change the creator name.

CREATED—The date and time when the analytics model was created. This value cannot be changed.

MODIFIED—The date and time when the analytics model was most recently modified. This value is updated each time there is a change to the model metadata or the model contents via one of the provided procedures.

STATE—The current state of the model. Valid values are “Complete”, “Creating” and “Updating”.

Creating—The model is being calculated.

Updating—The model is being updated.

Complete—No creation or update process is running. Write access to the model metadata and model contents might be restricted when a model is not in the “Complete” state.

Note: For very large data sets, the model may remain in the **Creating** state for some time.

DESCRIPTION—A free text field intended to be used to provide a description of, and any other information about, the analytics model. The initial value is NULL. When model generation is complete, the user can update the column texts using the NZA..ALTER_MODEL stored procedure. For more information refer to the NZA..ALTER_MODEL Procedure section.

COPYRIGHT—A free text field intended to be used to provide a copyright notice. This is useful if , for example, the model is converted to a PMML model, which should have a copyright section. The initial value is NULL. The user can update the column once the model generation is completed. When model generation is complete, the user can update the column texts using the NZA..ALTER_MODEL stored procedure. For more information refer to the NZA..ALTER_MODEL Procedure section.

MININGFUNCTION—The mining function of the model. The values are supplied by the individual mining algorithms and, where applicable, are based on the PMML standard values.

ALGORITHM—The name of the algorithm used to create the analytics model. The values are supplied by the individual mining algorithms, and where applicable, are based on the PMML standard values.

COMPONENTFORMAT—This format of the model tables. The makeup of the model components, such as the number of tables, table layout, views, or matrices can change for an algorithm between software versions. Since it is possible for different databases with different formats of metadata or model components to exist in parallel, this value indicates the software version format for the component. Generally, the component format version is of the form <major_version>.<minor_version>. As an example, all component formats that did not change since release 1.x.x are shown as “1.0”.

Optionally, the value can also contain a second component format number which indicates the earliest version of IBM Netezza Analytics that is compatible with the format to the string. For example, a value of “3.5(2.0)” indicates that the component format is 3.5, but the components can be read by any application that expects component format 2.0 or above.

Finally, if the version in brackets is missing, there is no backward compatibility. That is, “3.5” is the same as “3.5(3.5)”.

APPLICATIONNAME—The name of the application used to create the model. Currently, the value is always “IBM Netezza Analytics”.

APPLICATIONVERSION—The version of the application that was used to create the model.

USERCATEGORY—A free text field intended to be used to provide a category for the model. For example, a project name.

V_NZA_COMPONENTS View

The V_NZA_COMPONENTS view provides access to the names and types of the *components*, that is tables/views/matrices, associated with an analytic model. Each record contains the metadata for a specific component of an individual model in a given database. Following are the view columns:

Table 5: Columns of the V_NZA_COMPONENTS view

Column Name	Column Type	Description
MODELNAME (unique key column)	NVARCHAR(64)	The name of the analytics model.
NAME (unique key column)	NVARCHAR(128)	The name of a component (table, view, or matrix).
SCHEMA (unique key column)	NVARCHAR(128)	The schema that contains the component.
DATABASE (unique key column)	NVARCHAR(128)	The database that contains the component.
TYPE	VARCHAR(16)	The component type.
MANAGEMENT	VARCHAR(16)	The management level of the component.
USAGETYPE	VARCHAR(64)	The usage type of the component.
SEQID	SMALLINT	An identifier used if more than one component of the same type exists.

Detailed Column Descriptions

MODELNAME—The name of the analytics model. For more information, refer to the Model Naming Conventions section.

NAME—The name of a component in either the current database or another database. If the name of the table/view is changed, the view shows the changed name correctly; there is no procedure to change matrix names.

SCHEMA—The schema that contains the component. If NPS multiple schema mode is enabled, the schema for managed components is INZA.

DATABASE—The database that contains the component. Managed components must be in the current database; therefore, the value must be the current database.

TYPE—The component type. Valid values are “Table”, “View”, and “Matrix”.

MANAGEMENT—There are two types of management used to distinguish components.

Managed—The life cycle of these objects is controlled by the provided procedures. They are dropped when the model is dropped, renamed when the model is renamed, and so on. These objects must reside in the current database. The components containing the model contents are managed by IBM Netezza Analytics. Note that matrices are public objects and can be manipulated by any user. Therefore, it is possible that changes in the matrix components of a model make the model inconsistent.

Referenced—The name of these objects are included in the metadata, but the objects are not managed, nor is referential integrity enforced. These objects can reside in a database other than the current database. Examples of referenced components are input components for an algorithm and PMML tables. If these objects are dropped, the reference remains.

USAGETYPE—The type of the component. For example, the core table of an analytic model is of type “Model”; PMML tables are of type “PMML”; an input table is of type “Input”.

SEQID—The optional sequence ID for the object. Typically, this value is NULL. However, if an algorithm stores more than one component of the same type, they are distinguished by the sequence ID. The ID is a small integer value greater than or equal to zero and selected by the algorithm.

V_NZA_PARAMS View

The V_NZA_PARAMS view provides access to the individual parameters passed to the model building procedure used to create or update a model, including the name of the stored procedure. Each record contains the metadata for a specific parameter used for an individual model.

Note: Some passed parameters, for example the “model” or the “id” parameter. are not available via this view, The view also shows parameters that were not passed to the procedure but have a default value.

Table 6: Columns of the V_NZA_ PARAMS view

Column Name	Column Type	Description
MODELNAME (unique key column)	NVARCHAR(64)	The name of the analytics model.
TASKSEQ (unique key column)	SMALLINT	Identifies a task used to create/update the model.
PARAMETERNAME (unique key column)	VARCHAR(64)	Name of the parameter.
PARAMETERTYPE	VARCHAR(64)	Data type of the parameter.
PARAMETERVALUE	NVARCHAR(8192)	Value of the parameter.

Detailed Column Descriptions

MODELNAME—The name of the analytics model. For more information, refer to the Model Naming Conventions section.

TASKSEQ—Every operation to create or update a model can be viewed as a task; each task for a given model is assigned a task sequence ID. When a model is created, the task sequence ID is set to 1. All parameters used in the creation of that model are stored with ID 1. When a model is updated, the task sequence ID is incremented, and the new ID is used to store the parameters of the update procedure. That way we can store the complete series of stored procedure calls that have been executed on a model. In this manner, the complete series of stored procedure calls that have been executed on a model can be recorded.

PARAMETERNAME—The name of a parameter that has been specified in the parameter string of the model building or model updating procedure. There is a special parameter named “procedure”. This parameter indicates the name of the stored procedure used to build the model.

Note: Parameters that represent column properties are not available in this view; however, they are available via the V_NZA_COLPROPS view. See the V_NZA_COLPROPS View section for more information.

PARAMETERTYPE—The data type of the parameter. This column contains SQL data type names.

PARAMETERVALUE—The value of the parameter as it is passed to the model building or updating stored procedure. Values are stored as strings. The information from the PARAMETERTYPE column can be used to convert the value to another type, if needed.

V_NZA_COLPROPS View

This view provides access to the column properties of the input table. Special properties of the input table columns can be defined, for example, if a column should be treated as continuous or nominal. These properties can be passed as parameters in the command line, or in a special table. In the event the same property is specified twice, the property in the command line overrides the property definition on the special table.

Each record contains the metadata for specific property of an individual column for a given model. **Table 7: Columns of the V_NZA_ COLPROPS view**

Column Name	Column Type	Description
MODELNAME (unique key column)	NVARCHAR(64)	The name of the analytic model.
COLUMNNAME (unique key column)	NVARCHAR(128)	The name of an input table Λ column.
PROPERTYNAME (unique key column)µ	VARCHAR(64)	The name of a column property.
PROPERTYTYPE	VARC	Data type of the property.

Column Name	Column Type	Description
	HAR(64)	
PROPERTYVALUE	NVARCHAR(64)	The value of the column property.

Detailed Column Descriptions

MODELNAME—The name of the analytics model. For more information, refer to the Model Naming Conventions section.

COLUMNNAME—The name of a column in the input table. If no column properties are specified for an input table column, it is not shown.

PROPERTYNAME—The name of the property. Valid property names are “idcol”, “coldatatype”, “coltype”, “colrole” and “colweight”.

coldatatype—The SQL data type of the input column

coltype—The mining type of the input column, either nominal or continuous

colrole—A special role of the input column, for example, target or id

colweight—A weight for the input column

idcol—The physical sequence number of a column.

An example of idcol is in the instance of a command such as CREATE TABLE NEWTABLE (A INT, B VARCHAR, C DOUBLE) then A has idcol 1, B has idcol 2 and C has idcol3

PROPERTYTYPE—The data type of the property. Typically the property values are special words so the type is VARCHAR; however, the weight property has a numerical type, for example INTEGER or DOUBLE.

PROPERTYVALUE—The value of the property.

NZA..LIST_MODELS Procedure

The NZA..LIST_MODELS procedure allows the user to output a list of models that match the criteria specified in the parameter string, provided the user has been granted the LIST privilege or any other object privilege on the model. The output is a list of the selected models with columns **MODELNAME**, **OWNER**, **CREATED**, **STATE**, **MININGFUNCTION**, **ALGORITHM**, and **USERCATEGORY**. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

Example

```
CALL NZA..LIST_MODELS('where=OWNER=''JOE''
AND ALGORITHM=''classification''');
```

NZA..LIST_COMPONENTS Procedure

The NZA..LIST_COMPONENTS procedure allows the user to output a list of components that match the criteria specified in the parameter string, provided the user has been granted the LIST privilege or any other object privilege on the model. The output is a list of the selected components with columns **MODELNAME**, **NAME**, **SCHEMA**, **DATABASE**, **TYPE**, **MANAGEMENT**, **USAGETYPE**, and

SEQID. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

Example

```
CALL NZA..LIST_COMPONENTS('where=OWNER=''BOB'' AND USAGETYPE=''Model''');
```

NZA..LIST_PARAMS Procedure

The NZA..LIST_PARAMS procedure allows the user to output a list of parameters used to create models using criteria specified in the parameter string, provided the user has been granted the LIST privilege or any other object privilege on the model. The output is a list of parameters with columns **MODELNAME**, **TASKSEQ**, **PARAMETERNAME**, **PARAMETERTYPE**, and **PARAMETERVALUE**. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

Example

```
CALL NZA..LIST_PARAMS('where=MODELNAME=''My_Model''');
```

NZA..LIST_COLPROPS Procedure

The NZA..LIST_COLPROPS procedure allows the user to output a list of column properties that match the criteria specified in the parameter string, provided the user has been granted the LIST privilege or any other object privilege on the model. The output is a list of column properties with columns **MODELNAME**, **COLUMNNAME**, **PROPERTYNAME**, **PROPERTYTYPE**, **PROPERTYVALUE**. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

Example

```
CALL NZA..LIST_COLPROPS('where=PROPERTYNAME=''COLDATATYPE''');
```

Model Management Objects

For performing Model Management, the following stored procedures are defined:

Stored Procedures

- NZA..MODEL_EXISTS Procedure
- NZA..DROP_MODEL Procedure
- NZA..DROP_ALL_MODELS Procedure
- NZA..ALTER_MODEL Procedure
- NZA..COPY_MODEL Procedure
- NZA..PRINT_MODEL Procedure
- NZA..PMML_MODEL Procedure
- NZA..EXPORT_PMML Procedure

NZA..MODEL_EXISTS Procedure

The NZA..MODEL_EXISTS procedure determines if the specified model exists. When run, the system returns TRUE if the model exists; otherwise it returns FALSE.

NZA..DROP_MODEL Procedure

The NZA..DROP_MODEL procedure drops the model with the specified name, provided the user has been granted the privilege to drop the model. All managed model components (tables/views/matrices) and all metadata regarding the model are removed. If the procedure executes successfully, the system returns TRUE and outputs the model name. If the model metadata could be removed but one of the model components could not be dropped, the system returns FALSE and outputs a warning. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

NZA..DROP_ALL_MODELS Procedure

The NZA..DROP_ALL_MODELS procedure drops the selected models from the database, provided the user had been granted the DROP privilege for all models. Once run, the procedure outputs the names of all dropped models. The procedure could also be used by the administrator to drop all models before cleaning up metadata. If the procedure executes successfully, the system returns TRUE. However, since the procedure calls DROP_MODEL for each individual model, the procedure returns TRUE only if all DROP MODEL calls were successful. For more information on granting privileges, refer to the NZA..GRANT_MODEL Procedure section.

NZA..ALTER_MODEL Procedure

The NZA..ALTER_MODEL procedure alters properties of the specified model, provided the user has been granted the ALTER privilege for the model. At least one of the optional parameters must be specified. Multiple properties can be modified with a single call by specifying all of the applicable parameters.

Notes:

The model name and owner can be changed only if the state of the model is “Complete”.

When a model name is changed, the names of the model's managed components are changed accordingly.

Models that have been built using the model building procedures on the local machine get the application name ”IBM Netezza Analytics” and the current application version which are read-only and cannot be changed.

NZA..COPY_MODEL Procedure

The NZA..COPY_MODEL procedure copies the specified model, including all managed components and metadata, to the current database, provided the user has been granted the SELECT privilege for the source model. Models can be copied either from the current database or from another database. The model is copied into the current database, with the new model name specified by the value provided by the copy parameter. The active user is set as the owner of the copied model.

Note: Only models whose state is “Complete” can be copied.

NZA..PRINT_MODEL Procedure

The NZA..PRINT_MODEL procedure generates a formatted presentation of the specified model.

Notes:

Only models whose state is "Complete" can be printed.

The print function is not available for all algorithms. In the event a model cannot be printed, a warning message is returned.

NZA..PMML_MODEL Procedure

The NZA..PMML_MODEL procedure creates a PMML representation of the analytics model and stores it in a database table, provided the user has been granted the SELECT privilege for the model. The resulting PMML string is split into several fragments, each stored in a separate row. Each row is assigned a sequence ID to define the order of the rows.

Notes:

Only models whose state is "Complete" can be converted to PMML.

The PMML function is not available for all algorithms. In the event a model cannot be converted to PMML, a warning message is returned.

Details

The table specified by outtable can be an existing table or a new table.

If the specified outtable does not exist, it is created with the columns MODELNAME, PMML_SEQ_ID, PMML.

If the specified outtable exists, the first column with datatype NVARCHAR(n <= 128) is used to store the model name, the first column with datatype INTEGER/SMALLINT/BIGINT is used to store the sequence ID, and the first column with datatype NVARCHAR(n >= 1024) is used to store the PMML fragments.

The model name column is optional for existing tables. However, a table that has no model name column can store only one PMML model. If an existing table with a model name column already contains a model with the same name, an exception is thrown. If an existing table without a model name column is not empty, an exception is also thrown.

If the user has been granted the ALTER privilege as well as the SELECT privilege on the model, the PMML table is added to the metadata of the analytics model. If the metadata already contains a reference to a PMML table for this model, it is overwritten.

Important: PMML tables are not created/dropped automatically when a model is created/dropped. In addition, the contents of regular components and PMML tables are not synchronized. However, since only one PMML table reference for each model is maintained, the possibility of a large amount of inconsistent data is minimized.

NZA..EXPORT_PMML Procedure

The NZA..EXPORT_PMML procedure exports an analytic model to a file. This can be done by either:

exporting the analytics model as a PMML model and then saving it to a file

exporting the analytics model as a PMML table and then saving it to a file

PMML Model to File

When exporting from a model, if there is already a PMML table for the model registered in the metadata, the model is taken from this table and written to the file name specified. There is no need to specify the optional **outtable** parameter. If there is not a registered PMML table but an outtable was specified, the procedure first calls the PMML_MODEL procedure to create the PMML table and then exports the model from the resulting table. If an outtable was not specified, the model is converted and exported without using an intermediate PMML table. This method uses the following form of the procedure:

```
EXPORT_PMML('<model>,<file>,[<outtable>]')
```

The following privileges are required:

Action	Privilege
To export a model to a file	LIST privilege on the model
A PMML table reference exists in the metadata	SELECT privilege on this PMML table
A PMML table reference does not exist in the metadata	SELECT privilege on the model.
PMML_MODEL is called to create the specified outtable/ outtable is registered in the metadata as PMML table for this model	ALTER privilege on the model

PMML Table to File

When exporting from a table, the PMML table is specified by the **intable** parameter. If the PMML table contains more than one model, use the **model** parameter to identify the appropriate model. This method uses the following form of the procedure:

```
EXPORT_PMML('<intable>,<file>,[<model>]')
```

In this scenario, the only privilege needed is SELECT privilege on the specified PMML table.

Security and Administration

The Metadata Management functionality provides a security layer that can be used to manage access to models and the related metadata by users. In addition, Metadata Management provides procedures for enabling and disabling access to metadata objects for a given database, as well as the ability to get metadata-related information about a database.

Security

Metadata Management allows security to be applied to models. The security layer limits a user's access rights to models and metadata. For example, a user cannot drop a model simply by dropping its model table; instead users must use the metadata interface to manage their models.

The security mechanism for analytics models:

Allows the definition of privileges for users or groups on models and their metadata

Prevents a scenario where all models in the database are publicly available or are able to be modified by any user

Ensures that the model data is consistent with the model metadata

Privileges

Security is governed by a system of *privileges* that can be granted to users. A user must be granted privileges on analytic models in order to perform operations on them. Privileges can be granted or revoked similar to database privileges. However, the existing SQL commands GRANT and REVOKE cannot be used for analytics models, therefore Metadata Management uses a custom set of privileges.

There are two types of privileges. The CREATE privilege is an *admin privilege*, that is, it is not bound to a specific model, since, by definition, the model does not yet exist. The remaining privileges—LIST, SELECT, DROP, ALTER, and UPDATE—are *object privileges*. The object privileges are bound to a specific model and can be granted to users and groups, including users that are not in the inza_users group. They can also be granted to the special group “public”. The table below defines the privileges available for analytic models:

Table 8: Model privileges

Privilege	Type	Description
CREATE	Administrator	Allows the user/group to create an analytics model.
LIST	Object	Allows the user/group to display a model and its properties.
SELECT	Object	Allows the user/group to read the model components.

Privilege	Type	Description
DROP	Object	Allows the user/group to drop a model.
ALTER	Object	Allows the user/group to change the properties of a model.
UPDATE	Object	Allows the user/group to change the contents of a model.

The CREATE privilege is granted to all IBM Netezza Analytics users, that is, users that are members of the group “inza_users”[1].

When a model is created, there are no object privileges associated with it. Instead, the user who created the model becomes the model's owner. Initially, only the model's creator (who gets all privileges), the database owner, and the user ADMIN can view and manipulate the model. For other users to gain access to the model, the owner, the database owner, or the user ADMIN must grant privileges to it.

A user who has one of the privileges SELECT, DROP, ALTER, UPDATE can list models, even if this privilege is not granted explicitly.

The SELECT privilege for a model is granted on/revoked from all managed components, since the SELECT privilege allows to read these objects.

The UPDATE privilege is not needed directly for metadata management, but rather is is needed for stored procedures in the algorithm components that update the model contents, such as PRUNE_DECTREE.

When a privilege is given to a user/group, it can be given “with grant option.” When the grant option is given, the user can in turn give the privilege to other users.

Important: The security model ensures that the regular IBM Netezza Analytics user does not bring the metadata into an inconsistent state. However, in the database, the database owner as well as the administrator (user ADMIN) have extensive privileges that supersede the privileges granted by the security model. As a result, it is possible for these users to change data in the tables and to make changes in the metadata that are not allowed by the official procedures available to standard users. Changes by these users could result in inconsistent data.

Matrices and the Security Model

Managed matrices are not included in the security model. While the security model can be thought of as relating generally to all model components, certain privileges have no effect on matrices. Matrices are public objects, that is, all matrix access procedures can be called by everyone on every matrix in the database. This public access prevents the security model from being applied to them.

Important: Managed matrices have public read/write access privileges, therefore it is possible that they can be illegally dropped or modified by any user.

[1] System Administrators can add users to this group using the **create_inza_db_user.sh** script, typically run when configuring a database for use with Netezza Analytics.

Administration

Metadata management provides a set of procedures that can be used to enable metadata functionality for a database. The database owner and the ADMIN user are the only privileged users who can execute these functions. However, it is strongly recommended that while working with other IBM Netezza Analytics functions, these highly privileged IDs not be used; regular user IDs should be used instead.

Administrator Functions

Using the provided functions, the database owner and ADMIN user can enable metadata for a database, disable metadata and remove metadata objects from a database, and request information about the database, including whether the database is enabled for metadata use and, if so, its metadata version. For more detailed information, refer to the Metadata Administration Objects section.

In addition, these users have the ability to grant and revoke privileges for other users. While standard users can be granted these privileges, only the database owner and ADMIN user can be guaranteed to have this ability. For more detailed information, refer to the Model Security Objects section.

Dropped Users

In the event that the user who is recorded as the owner of a model is dropped from the database, the V_NZA_MODELS view shows the name of the dropped user as the owner until a new owner is assigned. In that event, a new owner must be set using a valid, active user name. Refer to the NZA..ALTER_MODEL Procedure section.

Database Cleanup

Over the course of time, delete and update commands on the metadata tables produce unused records. While the number of such records is likely to be low, the administrator can reclaim this space using the GROOM command, if desired.

Migration

When new releases of IBM Netezza Analytics become available, updated or new functionality, either in Metadata Management or in the Model Components, may affect the metadata tables. In this event, *migration* from one metadata version to another is required.

Note: Typically, metadata procedures check for the correct initialization of metadata before executing. Therefore, if metadata migration is required, the Metadata Management module issues a warning message.

Migration is initiated by calling the NZA..INITIALIZE procedure, which migrates the metadata tables. To perform migration, the user must have the ALTER model privilege. For more information, refer to the NZA..INITIALIZE Procedure section.

Since different databases with different formats of metadata or model components can exist in parallel, migration is performed independently per database.

WARNING: If the old data is needed, the user must copy the model before migration occurs. Otherwise the data will be lost.

While the typical scenario is to migrate from the release that immediately preceded the current release, it is possible that migration may be necessary from older releases. For example, if a user has release 2.0 and now installs release 4.0, the process is to perform a migration from the 2.0 format to the 3.0 format, the another migration from the 3.0 format to the 4.0 format.

Concurrency

The Netezza database supports only the isolation level SERIALIZABLE. This isolation level limits the concurrency of transactions. If two transactions read and write to the same database table, it is possible that the younger transaction aborts with the error:

```
ERROR: <your database>.ADMIN.<a table> : Could not serialize -
transaction aborted
```

Since the algorithm and metadata management procedures use the same set of metadata tables, this error can also happen when IBM Netezza Analytics procedures are called.

Note that the error can only happen if two IBM Netezza Analytics procedures work with models in the same database. If different databases are used (each database has its own set of metadata tables), this error cannot occur.

If you see this error message, you can:

Run your procedure at a later time.

Switch off serialization checking for the current session. To do so, enter the following on the command line (this command should be run outside of a transaction):

```
set serializable = false;
```

NZA..MIGRATE_MODEL Procedure

The NZA..MIGRATE_MODEL procedure migrates the components of a model that was created in a previous version of Netezza Analytics to the format of the current version. Model components are, for example, tables, views, and matrices. If the migration is not needed, the procedure does not migrate the components and shows an appropriate message.

If a procedure that uses a model detects that the model must be migrated before it can be processed, you get the following message:

```
The model <model-name> is stored in an older format that is not
supported (Format Version = 1.0). Call MIGRATE_MODEL to convert this
model to the latest format.
```

The MIGRATE_MODEL procedure does not migrate the metadata of models but only the model components. The migration of the metadata for all databases and models is done by the script update_inza_dbs.sh. Typically, this script is run during the upgrade of Netezza Analytic. You can, however, also call the script manually.

NZA..REGISTER_MODEL Procedure

With the NZA..REGISTER_MODEL procedure, you can register a model that was created by using IBM

Netezza Analytics 1.x so that you can use it with Netezza Analytics 2.x Metadata Management.

Take this procedure before you work with IBM Netezza Analytics 3.x because a direct registration to IBM Netezza Analytics 3.x is not supported.

To register a model, you must have been granted the SELECT privilege on the previous model tables.

Before registration is done, the appropriate database must be initialized. Once initialized, the procedure must be called separately for each model. In addition, it is important that the procedure uses the same parameters used to create the model originally. The procedure assumes that the model was created using the algorithm specified by the procedure parameter and checks only for correct parameters and the availability of the underlying tables.

If the value of a parameter is not known, a question mark (?) can be specified if the default should not be used.

The original tables and matrices are not modified. Once under model management control, the caller of the function is added as the creator and owner of the model.

Note: FP-GROWTH models cannot be migrated to ARULE models and therefore cannot be registered.

Example

```
call nza..register_model('procedure=DIVCLUSTER, model=divcluster_adult,
intable=nza..adult, outtable=divcluster_adult_out,id=id, target=income,
distance=euclidean, maxiter=5, minsplit=30, maxdepth=3,randseed=12345');
```

The parameter values provided are not validated against the model. Using the example above, if distance was set to Manhattan when the model was created (instead of Euclidean), registration will be incorrect.

Model Security Objects

For setting model Security, the following stored procedures are defined:

Stored Procedures

NZA..GRANT_MODEL Procedure

The NZA..GRANT_MODEL procedure grants one or more privileges on a model to one or more users or groups. At least one user or group must be specified.

NZA..REVOKE_MODEL Procedure

The NZA..REVOKE_MODEL procedure revokes a privilege on a model from users or groups. For a list of valid values to use with the **privilege** parameter, see Table 8 in the Privileges section.

NZA..LIST_PRIVILEGES Procedure

The NZA..LIST_PRIVILEGES procedure lists either the object privileges or grant permissions for the specified user. If no user is specified, the procedure lists privileges for all users, provided the requesting user has permissions to view the privilege list. The **grant** parameter of the procedure controls whether object (the default) or grant permissions are displayed. To display grant privileges:

```
NZA..LIST_PRIVILEGES('grant=true')
```

The output columns are **USERNAME**, **MODELNAME**, and **PERMISSIONS**.

Notes:

Only the effective privileges are listed; individual user or group privileges are not shown.

To view privileges, the user must have select access on the system view _V_SYS_PRIV.

Privileges of the ADMIN user are not shown, since this user always has all privileges.

Metadata Administration Objects

For performing model administration, the following stored procedures are defined:

NZA..INITIALIZE Procedure

METADATA_VERSION Procedure

NZA..IS_INITIALIZED Procedure

NZA..CLEANUP() Procedure

NZA..INITIALIZE Procedure

The NZA..INITIALIZE() procedure is called to enable a database for the analytic functions and creates all database objects needed to manage the metadata for the analytic models. It is called when a new database for IBM Netezza Analytics is set up during the execution of the **create_inza_db.sh** script. In addition, calling the NZA..INITIALIZE procedure manually can initiate a migration from one metadata version to another, if needed. For more information, refer to the Migration section.

METADATA_VERSION Procedure

The METADATA_VERSION() procedure is used to determine the version of the metadata objects used in the current database. It is generated in the current database when the INITIALIZE() procedure is applied on it. If the metadata has not been initialized, this procedure does not exist.

NZA..IS_INITIALIZED Procedure

The NZA..IS_INITIALIZED() procedure determines if the metadata objects have been created in the specified database. The procedure also check whether the version of the metadata objects is as expected.

NZA..CLEANUP() Procedure

The NZA..CLEANUP() procedure drops all metadata objects in the current database. To successfully clean up the metadata objects, all metadata tables must be empty. If the tables are not empty, NZA..DROP_ALL_MODELS() must be called to delete all models. For more information, refer to the NZA..DROP_ALL_MODELS Procedure section.

CHAPTER 6

Data Exploration

Background

Data exploration algorithms constitute a collection of useful computational techniques used in virtually all data mining projects to become familiar with the data. This section reviews those available in the IBM Netezza In-Database Analytics package.

The data exploration algorithms can be divided into the following categories:

distribution description

relationship identification

The distribution description category consists of algorithms used to describe the empirical distribution of single attributes or the joint distribution of multiple—usually two—attributes. Algorithms in the relationship identification category detect and quantify relationships between attributes. These algorithms are standard and widely known. The adopted definitions of all calculated quantities are provided in the subsections below. Implementation-specific details are covered, if necessary, when presenting the available functions and usage examples.

Moments

Moments are quantities used to describe certain aspects of continuous attribute distributions. Of particular interest are the *central moments* or *moments around the mean*. The *kth* central moment is the mean of differences between attribute values and the attribute mean raised to the power of k, which for an attribute a and data set D can be written as:

$$D_{a,k} = \frac{1}{|D|} \sum_{x \in D} a\, x - m_a\, D^{k} \qquad (6)$$

Where:

$$m_a\ D = \frac{1}{|D|}\sum_{x\in D} a\ x \qquad (7)$$

is the mean value of attribute a in data set D , the basic location measure. Notice that the 0th central moment is 1 and the 1st central moment is 0, which means that only $k \geq 2$ central moments actually say something about the data. In practice, $k \in \{2,3, 4\}$ is used.

The 2nd central moment is the *variance*, which measures the dispersion of the distribution. The 3rd and 4th central moments are usually used in a *standardized* form, that is, divided by the corresponding power (3 or 4, respectively) of the standard deviation. The 3rd standardized central moment is called the *skewness* and serves as a common distribution asymmetry measure:

$$\text{skew}_a\ D = \frac{{}_{a,\ 3}\ D}{s_a^3\ D} \qquad (8)$$

where $s_a\ D$ is the standard deviation of attribute a on data set D . It takes a value of 0 for symmetrical distributions, negative values for a longer left tail, and positive values for a longer right tail.

The 4th standardized central moment is the *kurtosis* and serves as a measure of distribution peakedness. Typically, a constant of 3 is subtracted from the 4th standardized central moment in a kurtosis calculation, with the result sometimes referred to as *excess kurtosis*:

$$\text{kurt}_a\ D = \frac{{}_{a\ ,\ 4}\ D}{s_a^4\ D} - 3 \qquad (9)$$

This correction makes the kurtosis of a normal distribution equal to 0. It is negative for distributions flatter than normal and positive for distributions more peaked than normal.

Quantiles

Quantiles constitute a convenient and intuitive description of continuous attribute distribution that allow observation of location, dispersion, and asymmetry. Quantiles of a continuous attribute are values from its range taken at regular intervals of its cumulative distribution. The quantile of order $p \in [0, 1]$ of attribute a on data set D can be defined as a value q_p satisfying the following conditions:

$$\frac{|D_{a\ \ p}|}{}_{\leq p} \qquad (10)$$

$$\frac{|D_{a\leq\ p}|}{|D|} \geq p \tag{11}$$

where $D_{a\ \ p}$ and $D_{a\ \leq p}$ denote the subsets of D where the inequality in the subscript is satisfied by the values of attribute a . Informally, the order p quantile of an attribute is a value that cuts off the lower $p \cdot 100$ % of instances in the data set with respect to the attribute's values. The order 0.5 quantile is also known as the median, which partitions the data set into halves, and the 0.25, 0.5, and 0.75 order quantiles are called the quartiles, which partition the data set into quarters, with the first, second, and third designated as q_1 , q_2 ,q_3 , respectively. Other common special cases include:

- deciles—quantiles of order 0.1, 0.2,..., 0.9
- percentiles—quantiles of order 0.01, 0.02,..., 0.99

When using quartiles, the second quartile q_2 , also known as the median, serves as a location measure; the difference between the third and the first quartile $q_3 - q_1$ called the *inter-quartile* range serves as a dispersion measure; and the following ratio, called the *quartile skewness*, can be used as an asymmetry measure:

$$\frac{q_3-q_2 \ - \ q_2-q_1}{q_3-q_1} \tag{12}$$

Outlier Detection

It is common to apply the following quartile-based outlier detection criteria:

$$a\ x \quad q_1 - \ q_3-q_1 \tag{13}$$

$$a\ x \quad q_3 q_3-q_1 \tag{14}$$

These report the value of attribute a for instance x as outlying if it is below the first quartile or above the third quartile by more than the inter-quartile range multiplied by a coefficient a , which controls the aggressiveness of outlier detection. A commonly-used value for this parameter is 1.5.

Frequency Table

A univariate frequency table describes the distribution of a discrete attribute by providing the occurrence count for each unique value. A bivariate frequency table similarly describes the joint probability distribution of two discrete attributes, by providing the occurrence count for each distinct combination of their values.

Histogram

A histogram is a frequency table counterpart for continuous attributes. Although usually presented visually as a graph, it can be considered a table providing occurrence counts for a series of disjoint intervals covering the range of the attribute. The intervals can be of equal or inequal width. The number of intervals and their boundaries can be specified manually to ensure the histogram is most meaningful and readable, or adjusted automatically to the distribution.

Pearson’s Correlation

Pearson’s correlation coefficient, also known as the linear correlation coefficient, measures the degree of linear relationship between two continuous attributes. For attributes a_1 and a_2 it is calculated on data set D as follows:

$$\mathrm{corP}_{a_1,a_2} \; D = \frac{\sum_{x\in D} \; a_1 \; x - m_{a_1} \; D \; a_2 \; x - m_{a_2} \; D}{\sum_{x\in D} \; a_1 \; x - m_{a_1} \; D^{\;2} \sum_{x\in D} \; a_2 \; x - m_{a_2} \; D^{\;2}} \qquad (15)$$

where $m_{a_1} \; D$ and $m_{a_2} \; D$ denote the mean values of attributes a_1 and a_2 on data set D, respectively. In practical terms, it measures how close the data points are to a straight line in the two-dimensional space of a_1 and a_2.

The linear correlation coefficient falls in the $[-1,1]$ interval, with absolute values approaching 1 denoting nearly linear relationships, absolute values above 0.7 usually considered to indicate strong relationships, and absolute values below 0.3 usually considered to indicate no relationship. The sign of the correlation coefficient indicates whether the relationship is increasing (positive correlation: when one attribute increases the other tends to increase) or decreasing (negative correlation: when one attribute increases the other tends to decrease).

Spearman’s Correlation

When searching for relationships between attributes, there may be an interest in nonlinear relationships that appear weak according to Pearson’s correlation, but may actually be strong. One way to detect them is to use Spearman’s correlation coefficient, also known as the rank correlation coefficient, which measures the degree of monotonic relationship between two continuous attributes. For attributes a_1 and a_2 it is calculated on data set D as follows:

$$\mathrm{corS}_{a_1,a_2} \; D = 1 - \frac{6\sum_{x\in D} \; r_{a_1} \; x - r_{a_2} \; x^{\;2}}{|D| \; |D|^{2} - 1} \qquad (16)$$

where $r_{a_1} \; x$ and $r_{a_2} \; x$ denote the ordinal number rank of instance x in relation to attributes a_1 and a_2, respectively.

Unlike linear correlation it is ordering and not the attribute values that impact rank correlation. The rank correlation coefficient falls in the $[-1,1]$ interval, with absolute values approaching 1 denoting nearly linear relationships, absolute values above 0.7 usually considered to indicate strong relationships, and absolute values below 0.3 usually considered to indicate no relationship. The sign of the correlation coefficient indicates whether the relationship is increasing (positive correlation: when one attribute increases the other tends to increase) or decreasing (negative correlation: when one attribute increases the other tends to decrease).

Covariance

The covariance for a pair of continuous attributes provides a dependency measure that is similar and closely related to the linear correlation. The formula for calculating the covariance of attributes a_1 and a_2 on data set D is:

$$\operatorname{cov}_{a_1,a_2}(D) = \frac{1}{|D|-1}\sum_{x\in D}\left(a_1(x)-m_{a_1}(D)\right)\left(a_2(x)-m_{a_2}(D)\right) \tag{17}$$

which can be seen to relate to the corresponding Pearson’s correlation coefficient defined above as follows:

$$\operatorname{cov}_{a_1,a_2}(D) = \operatorname{corS}_{a_1,a_2}(D)\, s_{a_1}(D)\, s_{a_2}(D) \tag{18}$$

Where $s_{a_1}(D)$ and $s_{a_2}(D)$ are the standard deviations of attributes a_1 and a_2, respectively, on data set D.

It is common to calculate and use a *covariance matrix* for a set of n continuous attributes $a_1, a_2, \ldots, a_n$, which is an $n\times n$ matrix containing the covariance for each pair of attributes from this set. The covariance matrix is a convenient way to detect and describe linear relationships within a set of attributes. Consider the covariance matrix for two sets of attributes, $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$ and $a_{j_1}, a_{j_2}, \ldots, a_{j_n}$. It is an $m\times n$ matrix containing the covariance for each pair of (a_{i_k}, a_{j_l}), attributes where $k=1,2,\ldots,m; l=1,2,\ldots,n$.

Mutual Information

The mutual information is an information theory-based dependence measure intended for discrete attributes. For attributes a_1 and a_2, it is calculated on data set D as follows:

$$\begin{aligned} I_{a_1,a_2}(D) &= \sum_{v_1\in A_1}\sum_{v_2\in A_2} P\left(a_1(x)=v_1, a_2(x)=v_2\right) \\ &\quad\cdot\log\frac{P\left(a_1(x)=v_1, a_2(x)=v_2\right)}{P\left(a_1(x)=v_1\right)P\left(a_2(x)=v_2\right)} \end{aligned} \tag{19}$$

where the base of the logarithm can be arbitrary, but is usually taken as 2, and the probabilities are estimated from data set D based on value occurrence counts:

$$P\,a_1\quad x=v_1 \quad = \frac{|D_{a_1=v_1}|}{|D|} \tag{20}$$

$$P\,a_1\quad x=v_1 \quad = \frac{|D_{a_2=v_2}|}{|D|} \tag{21}$$

$$a_1 x=v_1 , a_2 x=v_2 = \frac{|D_{a_1=v_1} \cap D_{a_2=v_2}|}{|D|} \tag{22}$$

By putting a condition in a subscript to a data set here and thereafter we refer to the subset satisfying the condition. The quantity measures the information about one attribute that is contained in the other. It is non-negative, equal to 0 for independent attributes, and symmetric.

Conditional Entropy

The conditional entropy is an information theory-based unidirectional dependence measure intended for discrete attributes. For attributes a_1 given a_2, it is calculated on data set D as follows:

$$H_{a_1 \vee a_2}(D) = \sum_{v_1 \in A_1} \sum_{v_2 \in A_2} P\left(a_1(x)=v_1, a_2(x)=v_2\right) \frac{P\left(a_2(x)=v_2\right)}{¿ log\, P\left(a_1(x)=v_1, a_2(x)=v_2\right)} \tag{23}$$

The entropy can be considered as a measure of variation of an attribute in the data set. The conditional entropy is then a measure of how the first variable varies if we keep the second fixed.

Chi-Square Test

The chi-square (or “ $_2$”) test is used to detect statistically significant relationships between discrete attributes. The formula for calculating the $_2$ statistic for attributes a_1 and a_2 on data set be D can written as:

$$a^2_{1,a_2} D = \sum_{v_1 \in A_1} \sum_{v_2 \in A_2} \frac{\left|D_{a_1=v_1, a_2=v_2}\right| - E_{a_1=v_1, a_2=v_2}}{E_{a_1=v_1, a_2=v_2}}^{2} \tag{24}$$

Where $D_{a_1=v_1,\, a_2=v_2}$ is the subset of D with the value of a_1 equal v_1 and the value of a_2 equal v_2 , and $E_{a_1=v_1,\, a_2=v_2}$ is the expected number of such instances under the null hypothesis of no relationship between the attributes. This expected number is calculated as:

$$E_{a_1=v_1,\, a_2=v_2} = \frac{|D_{a_1=v_1}| \cdot |D_{a_2=v_2}|}{|D|} \tag{25}$$

where $D_{a_1=v_1}$ and $D_{a_2=v_2}$ denote the subsets of D with the conditions from the subscripts satisfied. The statistic accumulates the discrepancies between the actual and expected numbers of attribute value associations, standardized by dividing by the expected value. It is large whenever the actual numbers differ from the expected ones substantially, which is unlikely under the null hypothesis of no relationship.

The $_2$ statistic is a random variable approximately following the $_2$ distribution. It can be used to determine the p -value (“right tail”), which is the probability of obtaining a $_2$ statistic value at least as large as actually calculated assuming the null hypothesis. If sufficiently low, typically, 0.05 or 0.01, it is considered justification for rejecting the null hypothesis. Alternatively, the corresponding value of the cumulative distribution function can be determined and used to reject the null hypothesis if sufficiently close to 1.

t-Test

t -test is any statistical hypothesis test in which the test statistic follows a Student's t distribution, if the null hypothesis is supported.

When a test statistics follows the normal distribution and we know apriori the standard deviation (scaling term) of this statistic, then the hypothesis would be tested using the normal distribution.

However, what is most frequently encountered, we have to estimate the standard deviation of the test statistics based on the sample, then the test statistic (under certain conditions) follows a student's t distribution. Student’s t -test comes in a number of variations, depending on the intended application:

a single mean

unpaired (for two means)

paired

for a regression line slope

These tests are applicable to normally distributed attributes only and using them for attributes with other distributions may lead to unreliable conclusions.

The t -test for a single mean tests the null hypothesis of the mean attribute value in the whole domain being equal to a specified theoretical value , based on the observed mean and variance in the data set. The underlying statistic is calculated as:

$$t_{a\,,}\quad D = \frac{m_a\ D -}{\frac{s_a\ D}{|D|}} \tag{26}$$

where $m_a\ D$ and $s_a\ D$ are the mean and standard deviation of attribute a on data set D.

The unpaired t -test is used to test the hypothesis of the mean attribute value in two sub-domains being equal based on their observed means and variance in the two corresponding subsets of the data sets. Both the sub-domains and the corresponding data subsets are also called groups. Typically, the partitioning into two groups is represented by two values of another discrete attribute. The statistic is calculated according to the following formula:

$$t_a\ D_1\,, D_2 = \frac{m_a\ D_1\ - m_a\ D_2}{\frac{s_a^2\ D_1}{|D_1|}\quad \frac{s_a^2\ D_2}{|D_2|}} \tag{27}$$

where $m_a\,D_1$ and $s_a\,D_1$ are the mean and standard deviation of attribute a on subset D_1 , and $m_a\,D_2$ and $s\ _a\,D_2$ are the mean and standard deviation of attribute a on subset $D\ _2$, respectively.

The paired t -test considers two attributes and a single data set to test the null hypothesis of the mean difference between the two attributes across the whole domain being 0. This is equivalent to the t -test for a single mean applied to a new attribute defined as the difference of the original two attributes:

$$t_{a_1\,,a_2}\ D = \frac{m_{a_1 - a\ 2}\ D}{\frac{S_{a_1\ -a_2}\ D}{|D|}} \tag{28}$$

where $m_{a\ _1 - a_2}\ D$ and $s_{a_1 - a_2}\ D$ denote the mean and standard deviation of differences between $_1$ and a_2 on data set D .

The t -test for a regression line slope is used to test the null hypothesis that two continuous attributes are linearly related with a given slope coefficient . For the application of this test, a regression line $a_2\ x = ba_1\ x\ \ v$ for two attributes a_1 and a_2 is using data set D , which yields an estimated slope coefficient $b_{a_1\,,\,a_2}\ D$ and intercept $v_{a\ _1\,,\,a_2}\ D$ calculated as:

$$b_{a_1\,,\,a_2}\ D = \frac{\sum_{x\in D}\ a_1\ \ x\ -\ m_{a\ _1}\ \ D\ \ a_2\ \ x\ -\ m_{a\ _2}\ \ D}{\sum_{x\in D}\ a_1\ \ x\ - m_{a_1}\ \ D\ ^2} \tag{29}$$

$$v_{a_1, a_2} D = m_{a_2} D - b_{a_1, a_2} D \, m_{a_1} D \qquad (30)$$

Then the estimated slope coefficient is compared to the theoretical one, , to achieve the following test statistic:

$$t_{a_1, a_2,} D = \frac{b_{a_1, a_2} D -}{s_{b_{a_1, a_2}} D} \qquad (31)$$

where $m_{a_1} D$ and $m_{a_2} D$ are the means values of attributes a_1 and a_2 on D, and:

$$s_{b_{a_1, a_2}} D = \frac{\frac{1}{|D|-2} \sum_{x \in d} a_2 \; x - b_{a_1, a_2} \; D \, a_1 x \;\; v_{a_1, a_2} \; D}{\sum_{x \in D} a_1 \; x \; \overline{} \; m_{a_1} D} \qquad (32)$$

is the standard deviation of the estimated slope coefficient.

All these t statistics can be considered random variables with different realizations for different data sets from the same domain following Student’s distribution. These variables can be used to determine the p -values or cumulative distribution function values on which to base the decision to accept or reject the null hypotheses. A sufficiently low p -value—typically, 0.05 or 0.01—justifies rejecting the null hypothesis. Likewise, cumulative distribution function values sufficiently close to 0 or 1 justify rejecting the null hypothesis. This has the additional advantage of specifying whether the obtained test statistic appeared unlikely low or unlikely high according to the distribution assuming the null hypothesis.

Mann-Whitney-Wilcoxon Test

The Mann-Whitney-Wilcoxon test is a non-parametric counterpart of the unpaired t -test. It is used to verify the hypothesis that the locations of a continuous attribute differ significantly in two sub-domains based on the observed differences of their location in the corresponding two data subsets. As with the unpaired t -test, the partitioning into two groups is usually represented by two values of a discrete attribute.

The hypothesis is tested without making any assumptions on their distributions. It is possible by defining a test statistic that does not directly depend on attribute values, but rather on their ordering, which is similar in spirit to Spearman’s rank correlation. It is calculated for attribute a based on data subsets D_1 and D_2 as follows:

$$u_a \; D_1, D_2 = \sum_{x_1 \in D_1} \left| \{x_2 \in D_2 \mid a \, x_2 \quad a \, x_1 \} \right| \frac{1}{2} \left| \{x_2 \in D_2 \mid a \, x_2 = a \, x_1 \} \right| \qquad (33)$$

For each instance from the first subset, the test counts the number of instances in the second subset for which the value of attribute a is less than or equal to the value of attribute a of the first subset. Cases where the value is equal, are weighted with ½. If the statistic approaches its minimum or maximum values—0 and $|D_1| \cdot |D_2|$, respectively—it indicates a strong location difference that is li to occur not only for the observed data sets, but also for the whole corresponding sub-domains The distribution of the Mann-Whitney-Wilcoxon statistic, necessary to determine the p -values or cumulative distribution function values used for rejecting or accepting the null hypothesis of no location difference, is tabulated for small data sets and approximated by a normal distribution for larger data sets.

Wilcoxon Test

The Wilcoxon test is related to the paired t -test in the same way as the Mann-Whitney-Wilcoxon test is related to the unpaired t -test. It is a non-parametric approach to testing the hypothesis that two continuous attributes do not differ significantly, that is their median difference in the whole domain is 0. The underlying Wilcoxon statistic. also known as the signed rank statistic, is calculated for attributes a_1 and a_2 on data set D as follows:

$$w_{a_1, a_2} D = \sum_{x \in \{x' \in D | a_2 \; x' \quad a_1 \; x'\}} r_{a_1 - a_2} \; x \tag{34}$$

where $r_{a_1 - a_2} \; x$ is the rank of instance x with respect to the absolute difference between a_1 and a_2 , that is, $|a_1 \; x \; - a_2 \; x \;|$. The formula sums the ranks for all instances where the value of the second attribute is less than the value of the first attribute. The distribution of the statistic is tabulated for small data sets and approximated by a normal distribution for larger data sets.

Canonical Correlation

Just as Pearson’s correlation measures the degree of linear relationship between two attributes, the canonical correlation measures the degree of linear correlation between two sets of attributes, that is, it looks for many-to-many rather than one-to-one dependencies. Consider two sets of attributes, $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$ and $a_{j_1}, a_{j_2}, \ldots, a_{j_n}$. So called *canonical variates* are identified for these sets, which are new attributes formed as linear combinations of the original attributes:

$$u \; x = \sum_{k=1}^{m} {}_{i_k} a_{i_k} \quad x \tag{35}$$

$$\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_m}$$

$$v \; x = \sum_{k=1}^{n} {}_{j_k} a_{j_k} \quad x \tag{36}$$

where the coefficients of these linear combinations, and $\alpha_{j_1}, \alpha_{j_2}, \ldots, \alpha_{j_n}$, are called *canonical coefficients*. They are chosen so as to maximize the linear correlation between the

canonical variates on the data set D , $\mathrm{corS}_{u,v}\, D$. The maximum achieved linear variates analyzed correlation between is the canonical correlation between the two attribute the canonical sets:

$$\mathrm{corC}_{a_{i_1},\dots,a_{i_m};\, a_{j_1},\dots,a_{j_n}}(D) = \max_{a_{i_1},\dots,a_{i_m};\, a_{j_1},\dots,a_{j_n}} \mathrm{corS}_{u,v}(D) \tag{37}$$

One-Way ANOVA

The one-way ANOVA (*ANalysis Of VAriance*) can be considered an extension of the unpaired t -test for two means that can compare the means in more than two groups. The underlying statistical test, called the F -test, is equivalent to the t -test in the case of two groups with the F statistic equal to the square of the corresponding t statistic. In practice, the one-way ANOVA is applied whenever there are at least three groups to consider. As with the basic t -test, a single continuous attribute is considered, for which location differences in subgroups are to be detected.

Apart from the extension to more than two groups, the one-way ANOVA can take into account *experimental design*. This term refers to one common application of the one-way ANOVA to experimentally analyze the effects of some *treatments* to a set of *experimental units* or *subjects*. Treatments determine the grouping, as each possible treatment corresponds to one group. Experimental units are represented by instances in the data set, with the analyzed continuous attribute representing the effects of particular treatments observed for particular units. In this setting, the experimental design determines how treatments are assigned to particular units, that is, how the grouping is performed. In other words, the experimental design represents information about the procedure used to collect the analyzed data that is taken into account in the analysis.

Two common experimental designs are the *completely randomized design* (CRD) and the *random block design* (RBD). The CRD is the simplest design assuming that treatments are assigned to experimental units at random. Groups do not need to be of equal size, but there can be no factors in the experiment that affect the likelihood of a particular treatment being assigned to a particular unit. In the more complex random block design, it is assumed that experimental units are arranged into a number of blocks of the same size, equal to the number of possible treatments, according to some factor that is suspected to impact the treatment effects. Units from each block are randomly assigned to different treatments. This approach “filters out” the possible impact of the factor used for blocking, by making sure that treatments are randomly assigned within the blocks. Discrete attributes in the analyzed dataset are represented by the grouping or treatment in either case, and by the blocking for the RBD design.

Multivariate Analysis of Variance (MANOVA)

MANOVA is a statistical procedure to compare multivariate variance models for different groups of input data. It is a generalization of One-Way ANOVA , whereas not only a single variable but multiple dependent variables are supported.

Similar to ANOVA, MANOVA is often used to measure the effects of treatments for a group of subjects or experimental units. A typical example is to show different versions of a website to different user groups to measure different quantities for each user. A quantity is, for example, the time that the user spent on the website or the number of purchases. The results are then analyzed to find out whether these quantities significantly differ for each of the user groups.

MANOVA is most suitable for several dependent variables that are moderately correlated. If the dependent variables are too highly correlated, that is, they are measuring the same effect, it would be better to use a single ANOVA model. If the dependent variables are uncorrelated, you could apply a set of pair-wise ANOVAs to measure each effect independently.

In general, you can get the following information by using the MANOVA procedure:

If there are interactions among the dependent variables

If there are interactions among the independent variables

If changes in the independent variables significantly affect the dependent variables

The MANOVA algorithm is based on the test for equality of location parameters in the group of *k* samples. The test is done under the following assumption:

$${}_i{\sim}N_p(\mu_i, \Sigma)i.i.d.\,,$$

$\Sigma>0$ where:

Each sample *i* is independently drawn from the *p* dimensional multivariate normal distribution with the sample-specific mean vector μ_i

► Each sample *i* consists of n_i multivariate observations (column vectors) that are

denoted by $y_{i,j}$, where $1\ j\ n_i$

All groups share the common covariance matrix Σ

Matrix Σ is non-singular

Requirements and assumptions for the computation

The following conditions apply to the computation:

As the test is based on the comparison of covariance estimators, the computational procedure requires the following formula to start the computation:

$${}_{i=1}^{k}\; n_i \quad p$$

For the description of the statistical tests and test procedures, the following notations are used:

$$N=\sum_{i=1}^{k} n_i$$

▲ $\hat{\mu}_i=\frac{1}{\ }\sum^{n_k} y_{i,j}$

▲ $\hat{\mu}=\frac{1}{N}\sum_{i=1}^{k}\sum_{j=1}^{n_k} y_{i,j}$

▲ $E=\sum_{i=1}^{k}\sum_{j=1}^{n_k}(y_{i,j}-\hat{\mu}_i)(y_{i,j}-\hat{\mu}_i)^T$

- $H = \sum_{i=1}^{k} n_i (\hat{\mu} - \hat{\mu}_i)(\hat{\mu} - \hat{\mu}_i)^T$

The k-sample tests for equality of location parameters test the following hypothesis:

$$\begin{cases} H_0 : \mu_1 = \dots = \mu_k \\ H_A : \exists_{i \neq j}\ \mu_i \neq \mu_j \end{cases}$$

According to the assumptions, matrix E and matrix H are independent and distributed through the Wishart distribution in the following way:

- Independent from H_0 : $E \sim W_p(\Sigma, \nu_E)$
- Under H_0 : $H \sim W_p(\Sigma, \nu_H)$

  With the following degrees of freedom:

  $_E = N - k$,

  $_H = k - 1$

- Under H_A , matrix H is distributed through the non-central Wishart distribution.

The tests for the H_0 hypothesis and the H_A hypothesis are:

Wilks' test

Roy's test

Pillai's test

Hotelling's test

These tests are based on matrix H and matrix E. The test statistics can be expressed in terms of the eigenvalues and eigenvectors of matrix A.

Matrix A is defined as follows:

$$A = E^{-1} H$$

To ensure numerical stability, the eigenproblem of matrix A

$$(A - \lambda_i I) q_i = 0$$

is reformulated to a generalized eigenproblem of matrix H and matrix E

$$E^{-1} H - \lambda_i I) q_i = 0 \Leftrightarrow (H - \lambda_i E) q_i = 0$$

The solution of the generalized eigenproblem is represented in the following way:

$$_1 \dots \lambda_p$$

Wilks' test

Wilks' lambda distribution Λ is defined as follows:

$$\Lambda = \frac{|E|}{|H + E|} = \prod_{i=1}^{p} \frac{1}{1 + \lambda_i}$$

Under the null hypothesis (H_0), the test statistics follow Wilks' lambda distribution:

$$\Lambda \sim \Lambda(p, \nu_E, \nu_H)$$

You can compute Wilks' lambda distribution by using the following schemes:

Bartlett's approximation

Rao's F approximation

► Exact F-distribution when constraints concerning p , ν_E , ν_H are met.

For the description of approximation schemes, the following notation is used:

$$=\nu_E-\frac{p-\nu_H+1}{2}$$

For the different approximation schemes, the following notations are used:

Bartlett's approximation

$$-f\log(\Lambda(p,\nu_E,\nu_H))\sim\chi^2[p\,\nu_H]$$

The degree of accuracy of three decimal places for the approximation is reached when

$$p^2+\nu_H{}^2 \quad \frac{1}{3}f$$

Tip: Use Bartlett's approximation instead of Rao's F approximation if this degree of accuracy is reached, given that neither p nor ν_H are equal 1 or 2.

Rao's F approximation

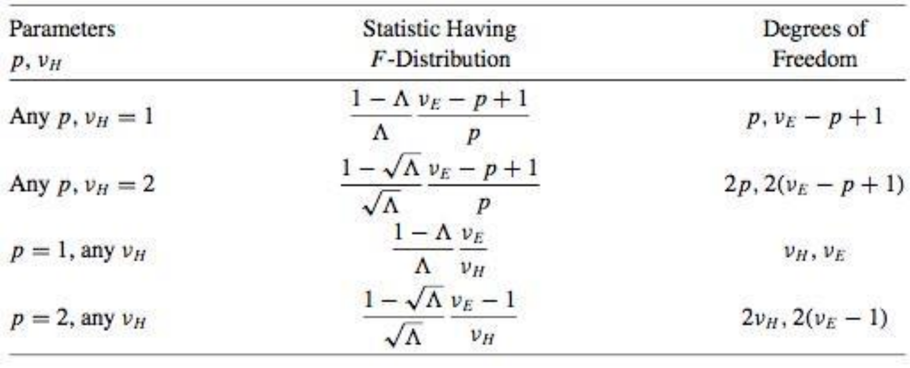
$$\sqrt{\quad}\left\{\begin{array}{ll}(p\,\nu_H)^2-4 & if\ p^2+\nu_H{}^2-5\neq0\\ 1 & otherwise\end{array}\right\}$$

$$\lambda=\frac{p\,\nu_H-2}{4}$$

The asymptotic distribution of $\Lambda(p,\nu_E,\nu_H)$ can be expressed as follows:

$$\left(\frac{1-\sqrt[d]{\Lambda(p,\nu_E,\nu_H)}}{\sqrt[d]{\Lambda(p,\nu_E,\nu_H)}}\right)\left(\frac{fd-2\lambda}{p\,\nu_H}\right)\sim F(p\,\nu_H,fd-2\lambda)$$

When $p^2+\nu_H{}^2-5=0 \Leftrightarrow p\,\nu_H=2$ or $\nu_H=p=1$, the expression is reduced to the exact distribution. If p = 1, or if p = 2, or if ν_H = 1, or if ν_H = 2, the following numerical simplifications are made:



Parameters p, ν_H	Statistic Having F-Distribution	Degrees of Freedom
Any $p, \nu_H = 1$	$\frac{1-\Lambda}{\Lambda}\frac{\nu_E-p+1}{p}$	p, ν_E-p+1
Any $p, \nu_H = 2$	$\frac{1-\sqrt{\Lambda}}{\sqrt{\Lambda}}\frac{\nu_E-p+1}{p}$	$2p, 2(\nu_E-p+1)$
$p=1$, any ν_H	$\frac{1-\Lambda}{\Lambda}\frac{\nu_E}{\nu_H}$	ν_H, ν_E
$p=2$, any ν_H	$\frac{1-\sqrt{\Lambda}}{\sqrt{\Lambda}}\frac{\nu_E-1}{\nu_H}$	$2\nu_H, 2(\nu_E-1)$

Roy's test

The test statistics is defined as follows:

$$\theta=\frac{\lambda_1}{1+\lambda_1}$$

However, because the approximations of the θ distriution are not satisfactory, the upper bound on the distribution of λ_1 is considered instead.

Let $d=max\,(p\,,\nu_H)$ under H_0 be $F_{upper}=\frac{(\nu_E-d-1)\lambda_1}{d}\sim F\,[d\,,\nu_E-d-1]$, where F_{upper} is a conservative approximation. That means, if H_0 is accepted by using F_{upper}, the exact test that is based on θ, would also accept H_0.

Pillai's test

The following notation is used:

$$s=min(\nu_H\,,p)$$

$$m=\frac{1}{2}\,(|\nu_H-p|-1)$$

$$=\frac{1}{2}\,(\nu_E-p-1)$$

The test statistics is defined as follows:

$$V=trace[(E+H)^{-1}H]=\sum_{i=1}^{s}\frac{\lambda_i}{1+\lambda_i}$$

The approximate distribution under H_0 is as follows:

$$\frac{(2M+s+1)V}{(2m+s+1)(s-V)}\sim F\,[s\,(2\,m+s+1)\,,s\,(2\,M+s+1)]$$

Hotelling's test

The following notation is used:

$$s=min(\nu_H\,,p\,)$$

$$a=p\,\nu_H$$

$$B=\frac{(\nu_E+\nu_H-p-1)(\nu_E-1)}{(\nu_E-p-3)(\nu_E-p)}$$

$$b=4+\frac{a+2}{B-1}$$

$$c=\frac{a\,(b-2)}{b\,(\nu_E-p-1)}$$

The test statistics is defined as follows:

$$=trace\,[\,E^{-1}\,H\,]=\sum_{i=1}^{s}\lambda_i$$

The approximate distribution under H_0 is as follows:

$$\frac{U}{c} \sim F\,[a\,,\,b]$$

When $\nu_H = 1$ (two sample comparison), the test statistic distribution is reduced to the exact Hotelling distribution. You can transform the Hotelling distribution to the F distribution by using the following algorithm:

$$\frac{\nu_E - p+1}{p}\, U \sim F\,[\,p\,,\,\nu_E - p+1]$$

Principal Component Analysis

Principal component analysis (PCA) is an analytical procedure based on an orthogonal linear data transformation to a new representation space. It can be thought of as replacing the original attributes by new attributes, called principal components, which correspond to the directions in the original attribute space exhibiting the greatest variance. The number of principal components used is at most equal to the number of original attributes, but it is often considerably less, since one of common motivations behind PCA is dimensionality reduction.

Principle component analysis, being strongly based on linear algebra calculations, can be directly applied to numerical data only. Data sets containing discrete attributes need to undergo preprocessing that numerically encodes their discrete values. With all attributes being continuous, the data set D can represented by a matrix A , with *rows corresponding to* n *attributes* and *columns corresponding to* $|D|$ *instances* (this is a typical convention adopted for PCA, although it is not consistent with the view of data sets as tables with rows representing instances and column representing attributes). Principal component analysis involves the following major operations performed on such a data matrix:

- calculating row means $u[i]$ for $i=1,2, ,n$ that is, attribute value means for each attribute, based on its values for all instances)
- calculating the deviations of data matrix elements from the row means, $B[i\,,j\,]=A[i\,,j\,]-u[i\,]$
- calculating the covariance matrix $C=\frac{1}{|D|}\,B\cdot B^T$
- calculating the eigenvectors and eigenvalues of the unbiased covariance matrix estimator such that $V^{-1}\cdot C\cdot V=E$, where V is the matrix of eigenvectors and E is the diagonal matrix of eigenvalues of C
- sorting the columns of the V and E matrices in the decreasing order of eigenvalues
- (optionally) trimming the V and E matrices to a selected number of highest eigenvalues and the corresponding eigenvectors, to be used as basis vectors for a new representation space
- finding projection to the selected eigenvectors

The obtained eigenvectors – which are the principal components that were looked for – can be considered transformed attribute value vectors in a new representation space (also called feature vectors). If not all eigenvectors are selected, this results in dimensionality reduction.

Corresponding eigenvectors and eigenvalues could also be computed using Singular Value

Decomposition algorithm, which provides a more stable but significantly slower and much more memory-consuming alternative to the procedure presented above.

PCA can be considered an advanced data exploration algorithm used to find patterns in the data and identify a transformed data representation that highlights this patterns. Whereas simple data exploration algorithms may be sufficient for exploring single attributes or pairs of attributes, the utility of PCA most clearly manifests itself for multidimensional data (with several attributes), where these simple algorithms are not sufficient. If the analyzed data set does indeed exhibit strong patterns, the new representation obtained using PCA can be considerably compressed by dimensionality reduction, without a significant information loss. This makes it possible to also consider PCA a powerful data transformation technique that can be applied prior to further analytical work. Areas where this technique has proven particularly useful are those where high-dimensional data sets are encountered, including text mining, image analysis, biological data analysis, customer preference and taste analysis (collaborative filtering), etc.

Available Functionality

The following distribution description algorithms are available:

moments— the MOMENTS stored procedure is used for calculating distribution moments and related summary statistics for continuous attributes, such as mean, variance, standard deviation, skewness, excess kurtosis, minimum, and maximum.

quantiles— the QUANTILE, QUARTILE, and MEDIAN stored procedures are used for calculating distribution quantiles and related summary statistics for continuous attributes.

outlier detection—the OUTLIERS stored procedure is used for detecting outlying values of continuous attributes based on quartiles.

frequency table—the UNITABLE and BITABLE stored procedures are used for calculating univariate (for single attributes) and bivariate (for attribute pairs) frequency tables, showing distinct attribute value occurrence counts).

histogram—the HISTOGRAM stored procedure is used for calculating histograms for continuous attributes (occurrence counts per intervals).

The following relationship identification algorithms are available:

Pearson’s correlation—the CORR stored procedure is used for calculating Pearson’s linear correlation for a pair of continuous attributes.

Spearman’s correlation—the SPEARMAN_CORR stored procedure is used for calculating Spearman’s rank correlation for a pair of continuous attributes.

covariance—the COV and COVARIANCEMATRIX stored procedures are used for calculating the covariance of a pair of continuous attributes and the covariance matrix for two sets of continuous attributes.

mutual information—the MUTUALINFO stored procedure is used for calculating the mutual information for a pair of discrete attributes. Note that the mutual information is a concept related to the entropy – see the stored procedures for ENTROPY, JOINTENTROPY, and CONDENTROPY.

chi-square test—the CHISQ_TEST stored procedure is used for performing Pearson's $_2$ test for independence for a pair of discrete attributes.

t **-test**—the T_ME_TEST, T_UMD_TEST, T_PMD_TEST, and T_LS_TEST stored procedures are used for performing Student's *t* -test for a single continuous attribute (a single mean), for a single continuous attribute in two subgroups (two means, unpaired), for two continuous attributes (two means, paired), and for a linear regression slope.

Mann-Whitney-Wilcoxon test—the MWW_TEST stored procedure is used for performing the Mann-Whitney-Wilcoxon non-parametric test for a continuous attribute in two subgroups (unpaired).

Wilcoxon test—the WILCOXON_TEST stored procedure is used for performing the Wilcoxon non-parametric test for two continuous attributes (paired).

canonical correlation—the CANONICAL_CORR stored procedure is used for calculating the canonical correlation for two sets of continuous attributes.

one-way ANOVA—the ANOVA_CRD_TEST and ANOVA_RBD_TEST stored procedures are used for performing the analysis of variance test for a continuous attribute in multiple groups, using either:

the completely randomized design, with two specified attributes—one discrete attribute representing the treatment or grouping, and one continuous attribute representing the effect of treatments.

the randomized block design with an additional specified discrete attribute representing the blocks.

MANOVA—the MANOVA_ONE_WAY_TEST and MANOVA_TWO_WAY_TEST stored procedures test the multivariate analysis of variance for different groups of input data with two or more dependent variables.
These stored procedures use one of the following designs:

Completely randomized design, with two specified attributes—one discrete attribute that represents the treatment or grouping, and one continuous attribute that represents the effect of treatments.

Randomized block design with an additional specified discrete attribute that represents the blocks.

All procedures that perform statistical tests adopt the convention of presenting the cumulative distribution value of the test statistic on output instead of the *p* -value. This approach has the advantage of indicating whether the obtained test statistic appeared unlikely low or unlikely high according to the distribution assuming the null hypothesis, whenever rejecting the null hypothesis is recommended. The calculated cumulative distribution value for the test statistic is also referred to as the *percentage point*.

Examples of Algorithm Functionality

In this section, the functionality of each algorithm is illustrated with examples. Note that the Netezza implementations have been optimized to work with large data sets, which in some cases requires using approximation techniques that may produce slightly different results for medium or small data sets.

Moments Example

The following stored procedure call demonstrates how to calculate moments for the continuous age attribute in the *CensusIncome* data set:

```
CALL nza..MOMENTS('intable=nza..CensusIncome,
incolumn=age, outtable=ci_m_age');
```

The specified output table **ci_m_age** contains a single row with the following fields::

columnname—the attribute name

countt—the number of instances on which the moments calculation is based

average—the mean value of the specified attribute

variance—the variance of the specified attribute

skewness—the skewness of the specified attribute

kurtosis—the kurtosis excess of the specified attribute

stdev—the standard deviation of the specified attribute

minimum—the minimum value of the specified attribute

maximum—the maximum value of the specified attribute

Examining the output from

```
SELECT * FROM ci_m_age;
```

reveals that the **age** attribute ranges from 0 to 90 years, with a mean value of about 34.5 and standard deviation of about 22. The distribution has a noticeably longer right tail with a skewness of about 0.37 and is considerably less peaked than normal with a kurtosis of about −0.73.

The same calculation can be repeated in subgroups determined by another attribute. The following example uses the **income** attribute to split the moments calculation into two groups:

```
CALL nza..MOMENTS('intable=nza..CensusIncome, incolumn=age,
by=income, outtable=ci_m_age_income');
```

The output table has the same structure as the previous example with an additional column named as the grouping attribute, and contains one row for each distinct value. The remaining fields contain the same statistics as described above, calculated separately within groups. Examining its contents using

```
SELECT * FROM ci_m_age_income;
```

reveals considerable differences of age distribution in the high-income and low-income groups. The high-income group contains people that are older on the average, but with less age standard deviation and a more peaky (concentrated around the mean) age distribution.

When analyzing data sets with many attributes, it may be inconvenient to issue a separate call and inspect a separate output table to get a distribution description for each attribute. In these cases, the **SUMMARY1000** procedure is useful. It is capable of generating the same descriptive statistics for up to 1000 attributes in one call, as demonstrated below:

```
CALL nza..SUMMARY1000('intable=nza..CensusIncome,
                       Vaincolumn=age;
                       wage_per_hour;
                       capital_gains;
                       capital_losses;
                       dividends_from_stocks;
                       num_persons_worked_for_employer;
                       weeks_worked_in_year,
                       outtable=ci_m_continuous');
```

The above call requests distribution description statistics be calculated for all continuous attributes in the *CensusIncome* data set by specifying them via the incolumn argument (if not specified, all attributes will be considered). The resulting output table contains one row for each input attribute and the following columns:

columnname—the attribute name

columnid—the ordinal attribute identification number

countt—the number of instances on which the calculation is based

average—the mean value of the corresponding attribute

variance—the variance of the corresponding attribute

stdev—the standard deviation of the corresponding attribute,

skewness—the skewness of the corresponding attribute

kurtosis—the kurtosis excess of the corresponding attribute

minimum—the minimum value of the corresponding attribute

maximum—the maximum value of the corresponding attribute

nonmissingcases – the number of instances on which the calculation is based

missing – the number of instances for which the value of the attribute is missing

distinctvalues – in case of nominal attributes – the number of distinct values

mostfrequentvalue - in case of nominal attributes – the value that was most frequent (or among the most frequent)

mostfrequentcases- in case of nominal attributes – the count of instances for which the above value is the attribute value.

You can omit the incolumn argument, in which case all the columns will be summarized (provided they are fewer than 1000). For example:

```
CALL nza..SUMMARY1000('intable=nza..CensusIncome, outtable=ci_m_all');
```

As for moments, you can also call the procedure with the **by** parameter.

```
CALL nza..SUMMARY1000('intable=nza..CensusIncome,

outtable=ci_m_allby, by=income');
```

Using **Select * from ci_m_allby** will show you the result of this procedure.

Quantiles Example

To calculate a single quantile value of specified order use the following call:

```
CALL nza..QUANTILE('intable=CensusIncome, incolumn=age, quantiles=0.25')
```

It returns the 0.25 order quantile, also referred to as the 1st quartile, of the **age** attribute. Multiple quantiles for the same attribute can be calculated in one call by specifying more than one value in a semicolon-separated list:

```
CALL nza..QUANTILE('intable=nza..CensusIncome, incolumn=age, quantiles=0.0;
0.25; 0.5; 0.75; 1.0, outtable=ci_age_q');
```

This calculates the 0, 0.25, 0.5, 0.75, and 1 order quantiles or the minimum, the 1st quartile, the 2nd quartile (median), the 3rd quartile, and the maximum, and stores the values in the specified output table **ci_age_q**. The table contains one row per quantile and two columns:

p—the quantile order,

value—the quantile value.

Examining the received contents using

```
SELECT * FROM ci_age_q;
```

reveals that the **age** attribute is within the range from 0 to 90, which corresponds to the moments example, with the 1st quartile equal to 15, the median equal to 33, and the 3rd quartile equal to 50.

While the **QUANTILE** procedure demonstrated above calculates quantiles of arbitrary orders in the 0,1] interval, there are separate procedures to handle the most common special cases. To calculate the 1st, 2nd, or 3rd quartile use the **QUARTILE** procedure, as in the following example:

```
CALL nza..QUARTILE('intable=nza..CensusIncome, incolumn=age, quartile=3');
```

This call returns the requested 3rd quartile of the **age** attribute and is equivalent to:

```
CALL nza..QUANTILE('intable=nza..CensusIncome,
incolumn=age, quantiles=0.75');
```

Similarly, the **MEDIAN** procedure returns the median, as demonstrated below:

```
CALL nza..MEDIAN('intable=nza..CensusIncome, incolumn=age');
```

which is equivalent to the following calls of **QUARTILE** or **QUANTILE**:

```
CALL nza..QUARTILE('intable=nza..CensusIncome, incolumn=age, quartile=2');

CALL nza..QUANTILE('intable=nza..CensusIncome, incolumn=age, quantiles=0.5');
```

with respect to the calculated values, but uses a different specialized algorithm optimized for large data sets. The **MEDIAN** procedure is not recommended for input tables with less than millions of rows, for which the general quantile/quartile algorithm may perform better.

Note that if a single quantile is to be calculated, the value is returned and no output table is created.

```
CALL nza..QUANTILE('intable=nza..CensusIncome, incolumn=age,
                     quantiles=0.25, outtable=test');

     QUANTILE
     ----------
            15
     (1 row)
```

To calculate several quantiles at once, separate the quantile values with semicolons. The quantile values are returned in the output table and the number returned by QUANTILE function indicates a number of quantiles computed.

```
CALL nza..QUANTILE('intable=nza..CensusIncome, incolumn=age,
                     quantiles=0.25;0.75, outtable=test');

     QUANTILE
     ----------
             2
     (1 row)

SELECT * FROM test;

            | VALUE
     ------+-------
       0.75 |    50
       0.25 |    15
      (2 rows)
```

Outlier Detection Example

The following stored procedure call performs outlier detection for the age attribute of the *CensusIncome* data set:

```
CALL nza..OUTLIERS('intable=nza..CensusIncome,
incolumn=age, outtable=ci_age_out');
```

The output table contains a single column containing the values of the input attribute that fall outside the non-outlying range, which is below the 1st quartile or above the 3rd quartile by more than the inter-quartile range multiplied by a coefficient that defaults to 1.5. The multiplier value can be set to a different value via the **multiplier** argument to change the aggressiveness of outlier detection. In the sample data, the above call found no outliers, so using the following modified call demonstrates a more aggressive attempt by using **multiplier=1**:

```
CALL nza..OUTLIERS('intable=nza..CensusIncome, incolumn=age, multiplier=1,
outtable=ci_age_out1');
```

Examining the contents of the output table:

```
SELECT * FROM ci_age_out1 ORDER BY 1;
```

reveals that age values above 85 are identified as outlying.

Frequency Table Example

The following call demonstrates how to create a frequency table for a single discrete attribute, using the example of the education attribute in the *CensusIncome* data set:

```
CALL nza..UNITABLE('intable=nza..CensusIncome,
incolumn=education, outtable=ci_education_ft');
```

The output table contains the following columns:

education—distinct values of the education attribute specified as the input attribute; note that in normal usage, the column is named after the input attribute

count—the number of occurrences or absolute frequency of each value

freq—the relative frequency percentage of each value

cum—the cumulative percentage of each value, if ordered lexicographically

Examining the output for the **education** attribute using

```
SELECT * FROM ci_education_ft ORDER BY 1;
```

reveals that 3.3% of the population have Masters degrees, 0.6% have Ph.D. degrees, and 23.8% are children. Further, those with 10th grade, 11th grade, and 12th grade education with no diploma account for 8.3%.

To examine the distribution of education within subgroups corresponding to different income classes, a bivariate frequency table can be created as follows:

```
CALL nza..BITABLE('intable=nza..CensusIncome, incolumn=income:x; education:y,
outtable=ci_education_income_ft');
```

The output table contains columns corresponding to the two specified input attributes, containing all distinct combinations of their values, and a **count** column with the number of occurrences of each combination. It can be inspected as demonstrated below:

```
SELECT * FROM ci_education_income_ft order by 1, 2;
```

The same output can be produced by:

```
SELECT income, education, count(*)
FROM CensusIncome
GROUP BY income, education
ORDER BY income, education;
```

Histogram Example

Previous examples revealed a wealth of information about the distribution of the age attribute in the *CensusIncome* data set, but a histogram can give a much more detailed picture. The following example shows how it can be created:

```
CALL nza..HIST('intable=nza..CensusIncome,
incolumn=age, outtable=ci_age_hist1');
```

The output table contains the following columns:

idx—the interval number

bleft—the left interval bound

bright—the right interval bound

counts—the number of instances with the value of the input attribute in the corresponding interval

By default, intervals are right-closed, but this can be changed by specifying the **right=FALSE** argument. To examine the histogram data issue the following query:

```
SELECT * FROM ci_age_hist1 ORDER BY 1;
```

The above call runs the **HIST** procedure in automatic mode, so the number of equal-width intervals are determined automatically. For certain instances, however, it might be useful to explicitly specify the number of intervals:

```
CALL nza..HIST('intable=nza..CensusIncome, incolumn=age, nbreaks=5,
outtable=ci_age_hist2');
```

or even specify breaks to make the resulting intervals more meaningful:

```
CREATE TABLE CensusIncome_age_breaks (break
integer); INSERT INTO CensusIncome_age_breaks VALUES
(0); INSERT INTO CensusIncome_age_breaks VALUES (5);
INSERT INTO CensusIncome_age_breaks VALUES (12);
INSERT INTO CensusIncome_age_breaks VALUES (18);
INSERT INTO CensusIncome_age_breaks VALUES (25);
INSERT INTO CensusIncome_age_breaks VALUES (35);
INSERT INTO CensusIncome_age_breaks VALUES (50);
INSERT INTO CensusIncome_age_breaks VALUES (65);
INSERT INTO CensusIncome_age_breaks VALUES (80);
INSERT INTO CensusIncome_age_breaks VALUES (90);
```

```
CALL nza..HIST('intable=nza..CensusIncome, incolumn=age,
btable=CensusIncome_age_breaks, bcolumn=break, outtable=ci_age_hist3');
```

When explicitly specifying histogram breaks, ensure they span the entire range of the attribute. Examining the output from:

```
SELECT * FROM ci_age_hist3 ORDER BY 1;
```

reveals a sufficiently detailed, but easily comprehended description of the distribution of the **age** attribute.

Pearson’s Correlation Example

The example call below illustrates how to measure the relationship between the alcohol and quality attributes in the *WineQuality* data set using Pearson’s correlation coefficient:

```
CALL nza..CORR('intable=nza..WineQuality, incolumn=alcohol; quality');
```

The resulting value of about 0.44 indicates there is a linear relationship—albeit weak—between the two attributes, with more alcohol tending to give better quality.

The correlation can be also calculated in subgroups determined by the values of a selected grouping attribute. Use the income attribute for grouping in the *CensusIncome* data set, and find the correlation between the age and **wage_per_hour** attributes:

```
CALL nza..CORR('intable=nza..CensusIncome, incolumn=age; wage_per_hour,
by=income, outtable=ci_corP_age_wage_per_hour_income');
```

This time an output table must be created, containing one column with correlation values, one column with distinct values of the selected grouping attribute, and one row for each of them. Inspecting the contents using

```
SELECT * FROM ci_corP_age_wage_per_hour_income;
```

verifies that there is no linear relationship between the **age** and **wage_per_hour** attributes both in high-income and low-income groups.

You can compute a correlation matrix for two sets of attributes (computing in groups via the **by** parameter is also possible):

```
CALL nza..CORRELATION1000MATRIX('intable=nza..WineQuality,
incolumn=fixed_acidity| volatile_acidity| citric_acid| residualsugar:X;
density| pH| alcohol:Y, outtable=wq_cormat');

SELECT * from wq_cormat;
```

You can compute a list of correlations for a list of pairs (computing in groups via the **by** parameter is also possible):

```
CALL nza..CORRELATION500PAIRS('intable=nza..WineQuality,
incolumn=fixed_acidity:volatile_acidity; citric_acid: residualsugar; density:
pH; alcohol: fixed_acidity:, outtable=wq_corlst');

    SELECT * from wq_corlst;
```

Spearman’s Correlation Example

The following call requests that Spearman’s rank correlation be calculated for the same pair of attributes, alcohol and quality, from the *WineQuality* data set:

```
CALL nza..SPEARMAN_CORR('intable=nza..WineQuality, incolumn=alcohol;
quality');
```

The returned value is marginally greater than for Pearson’s correlation.

```
CALL nza..SPEARMAN_CORR_S('intable=nza..WineQuality,
incolumn=alcohol; quality');
```

It shows more information (percentage point, t-statistics and degrees of freedom).

Covariance Example

To calculate the covariance between the alcohol and quality attributes on the *WineQuality* data set one can issue the following call:

```
CALL nza..COV('intable=nza..WineQuality, incolumn=alcohol; quality');
```

As with linear correlation, the covariance can be calculated in subgroups determined by the values of a specified grouping attribute. You can demonstrate this capability on the *CensusIncome* data set, using the income attribute for grouping and looking for the covariance of the **age** and **wage_per_hour** attributes:

```
CALL nza..COV('intable=nza..CensusIncome, incolumn=age;
wage_per_hour, by=income, outtable=ci_cov_age_wage_per_hour_income');

SELECT * FROM ci_cov_age_wage_per_hour_income;
```

The output table contains only one column, **COVARIANCE**, containing the covariance value for each value of the specified grouping attribute.

The covariance matrix for two sets of continuous attributes can be calculated as demonstrated by the following call:

```
CALL nza..COVARIANCE1000MATRIX('intable=nza..WineQuality,
incolumn=fixed_acidity| volatile_acidity| citric_acid| residualsugar:X;
density| pH| alcohol:Y, outtable=wq_covmat');

SELECT * from wq_covmat;
```

This calculates the covariances for attribute pairs from the two specified attribute sets on the WineQuality data set.

Note that computing in groups via the **by** parameter is also possible.

You may compute a list of covariances for a list of pairs (the **by** parameter is also possible here).

```
CALL nza..COVARIANCE500PAIRS('intable=nza..WineQuality,
incolumn=fixed_acidity:volatile_acidity; citric_acid: residualsugar; density:
pH; alcohol: fixed_acidity:, outtable=wq_covlst');


SELECT * from wq_covlst;
```

Mutual Information Example

Mutual information can be used to measure the degree of relationship between the education and income discrete attributes on the *CensusIncome* data set as follows:

```
CALL nza..MUTUALINFO('intable=nza..CensusIncome, incolumn=education;
income');
```

It can also be used for the **sex** and **income** attributes:

```
CALL nza..MUTUALINFO('intable=nza..CensusIncome, incolumn=sex; income');
```

The much lower result in the second example indicates that income is more strongly related to education than to gender. You can also examine the relationship between the **education** and **sex** attributes in subgroups corresponding to income levels:

```
CALL nza..MUTUALINFO('intable=nza..CensusIncome, incolumn=education;
sex, by=income, outtable=ci_mi_education_sex_income');

Select * FROM ci_mi_education_sex_income;
```

The output table contains one row for each distinct value of the grouping value, containing the mutual information value within the corresponding group. In this instance, relationship appears to be weak in both groups.

Conditional Entropy Example

A call computing the entropy of sex attributed:

```
CALL nza..entropy('intable=nza..CensusIncome, incolumn=sex');
```

being close to 1 reveals that both men and women are equally represented in the data.

The joint entropy of sex and income:

```
CALL nza..joint_entropy('intable=nza..CensusIncome, incolumn=sex;income');
```

is quite close to the sum of entropies for sex and for income so that it is no wonder that conditional entropy of income on sex

```
CALL nza..cond_entropy('intable=nza..CensusIncome, incolumn=sex:X;income:Y');
```

is nearly identical with that for income, meaning sex has no significant impact of income.

Chi-Square Test Example

The following call demonstrates how to verify the significance of the relationship between the **education** and **sex** attributes on the *CensusIncome* data set using the

χ^2 test:

```
CALL nza..CHISQ_TEST('intable=nza..CensusIncome, incolumn=education; sex,
outtable=ci_chi2_education_sex');

SELECT * FROM ci_chi2_education_sex;
```

The output table contains a single row with the value of the ^{2} statistic, the number of its degrees of freedom, and the corresponding value of the ^{2} cumulative distribution function value shown in the **percentage** column. If close to 1, this value justifies rejecting the null hypothesis. The obtained value of nearly 1 indicates the relationship between the attributes is statistically significant. The same calculation can be also repeated separately in the two different income groups:

```
CALL nza..CHISQ_TEST('intable=nza..CensusIncome, incolumn=education;
sex, by=income, outtable=ci_chi2_education_sex_income');

SELECT * FROM ci_chi2_education_sex_income;
```

The results show that, while statistically significant in both groups, the relationship between education and gender is more prominent in the low-income group.

Note that computing in groups via the **by** parameter is also possible here.

t-Test Example

The following call demonstrates how to apply the *t* -test for a single mean to verify the null hypothesis that the mean value of the *WineQuality* data set **quality** attribute is 5:

```
CALL nza..T_ME_TEST('intable=nza..WineQuality, incolumn=quality, mean=5,
outtable=wq_t_quality_5');

SELECT * FROM wq_t_quality_5;
```

The output table contains the test’s cumulative distribution function value (in the **percentage** column), the *t* statistic value, and the number of degrees of freedom. If the cumulative distribution function value is close to 0, for example, values of 0.05 or less, it justifies rejecting the null hypothesis of the true population mean being equal the specified value and recommends an alternative hypothesis of the true population mean being less than the specified value. If it is close to 1, for example, 0.95 or greater, it justifies rejecting the null hypothesis of the true population mean being equal to the specified value and recommends an alternative hypothesis of the true population mean being greater than the specified value. In this example, the returned value of nearly 1 suggests the true mean wine quality exceeds 0.5.

The same test can be applied to verify the null hypothesis of the mean value of the **weeks_worked_in_year** in the population being equal 24 based on the *CensusIncome* data set:

```
CALL nza..T_ME_TEST('intable=nza..CensusIncome,
incolumn=weeks_worked_in_year, mean=24,
outtable=ci_t_weeks_worked_in_year_24');

SELECT * FROM ci_t_weeks_worked_in_year_24;
```

The **percentage** column in the output table contains a cumulative distribution value of near 0 for the test statistic, which suggests that the true mean differs significantly from and is less than 24. The same hypothesis can be also verified in different income groups by specifying the **by** argument appropriately:

```
CALL nza..T_ME_TEST('intable=nza..CensusIncome,
incolumn=weeks_worked_in_year, mean=24, by=income,
outtable=ci_t_weeks_worked_in_year_24_income');

SELECT * FROM ci_t_weeks_worked_in_year_24_income;
```

As you can see, whereas the true mean is below 24 in the low-income group, it exceeds 24 in the high-income group. To further explore the dependence between the **weeks_worked_in_year** and **income** attributes, you can apply the unpaired *t* -test for two means to verify the null hypothesis that the true means of the former differ in the subpopulations determined by the latter as follows:

```
CALL nza..T_UMD_TEST('intable=nza..CensusIncome,
incolumn=weeks_worked_in_year, class=income, class1="50000+.", class2="-
50000.", outtable=ci_t_weeks_worked_in_year_income');

SELECT * FROM ci_t_weeks_worked_in_year_income;
```

The **percentage** value of 1 indicates a statistically significant difference, with the mean value in the high-income group greater than in the low-income group. The same can be repeated for the **wage_per_hour** attribute:

```
CALL nza..T_UMD_TEST('intable=nza..CensusIncome, incolumn=wage_per_hour,
class=income, class1="50000+.", class2="-50000.",
outtable=ci_t_wage_per_hour_income');

SELECT * FROM ci_t_wage_per_hour_income;
```

Here, too, the mean value in the high-income group is confirmed to be significantly greater than in the low-income group.

The attribute used to split the data set into two groups does not have to be binary as long as you only consider instances with two different values. The following example demonstrates how the same unpaired *t* -test can be used to examine whether the mean value of the **wage_per_hour** attribute differs significantly between never married and divorced individuals, which are represented by 2 out of 7 possible values of the **marital_status** attribute:

```
CALL nza..T_UMD_TEST('intable=nza..CensusIncome, incolumn=wage_per_hour,
class=marital_status, class1="Never married", class2=”Divorced”,
outtable=ci_t_wage_per_hour_marital_status');

SELECT * FROM ci_t_wage_per_hour_marital_status;
```

The percentage value near 0 indicates that the mean wage per hour value in the never married group is below that of the divorced group.

The following call demonstrates the regression line slope version of the *t* -test, applied to verify the null hypothesis that the **alcohol** and **quality** attributes are linearly related with a regression line slope of 5.

```
CALL nza..T_LS_TEST('intable=nza..WineQuality, incolumn=alcohol:X; quality:Y,
slope=5, outtable=wq_t_ls_alcohol_quality');

SELECT * FROM wq_t_ls_alcohol_quality;
```

The obtained **percentage** value is well below 0.01, providing sufficient justification to reject the null hypothesis and indicates that the true regression line slope is less than the one specified via the **slope** parameter.

Note that computing in groups via the by parameter is also possible here.

Mann-Whitney-Wilcoxon Test Example

To demonstrate the Mann-Whitney-Wilcoxon test, follow the same scenario as for the unpaired *t* - test, which is its non-parametric counterpart. Start by verifying whether the **weeks_worked_in_year** attribute differs significantly between the two income groups:

```
CALL nza..MWW_TEST('intable=nza..CensusIncome, incolumn=weeks_worked_in_year,
class=income');
```

The procedure returns a string (VARCHAR object) containing:

the values of the test statistic, tagged **uStat**

the parameters of the normal approximation of its distribution

the corresponding cumulative distribution function value, a percentage, tagged pp

The corresponding cumulative distribution function value is nearly 1, indicating a statistically significant difference, with the values in the high income group usually greater than in the low-income group. The same can be repeated for the **wage_per_hour** attribute:

```
CALL nza..MWW_TEST('intable=nza..CensusIncome,
incolumn=wage_per_hour, class=income');
```

Again, the mean value in the high-income group is confirmed to be significantly greater than in the low-income group.

Canonical Correlation Example

The following call demonstrates the usage of canonical correlation to find relationships between two sets of attributes of the *WineQuality* data set, one containing the fixed_acidity, volatile_acidity, citric_acid, and residualsugar attributes and the other containing the density, pH, and alcohol attributes:

```
CALL nza..CANONICAL_CORR('intable=nza..WineQuality, incolumn=fixed_acidity:X;
volatile_acidity:X; citric_acid:X; residualsugar:X; density:Y; pH:Y;
alcohol:Y, outtable=wq_cancor');
```

```
SELECT * FROM wq_cancor;
```

The resulting output table contains a single row and a single column. The calculated results are presented in text form, and contain the highest achieved correlation value between canonical variates along with the corresponding canonical coefficient for the two specified sets of input attributes. The correlation of about 0.95 obtained in this example indicates the two sets of attributes in the *WineQuality* data set are closely related.

One-Way ANOVA Example

To illustrate the one-way ANOVA algorithm, use a modified version of the *WineQuality* data set. Assume the alcohol attribute represents the treatment applied to wines, and the quality attribute represents the observed effect. To make it possible, the alcohol attribute is discretized using the equal-frequency algorithm into four intervals as follows:

```
CALL nza..EFDISC('intable=nza..WineQuality, outtable=wq_efd, binprec=1,

incolumn=alcohol:4');

CALL nza..APPLY_DISC('intable=nza..WineQuality, btable=wq_efd,

outtable=WineQuality4, replace=TRUE');
```

To use the discretized alcohol attribute as the treatment, each of the values must have the same occurrence count, which is a theoretical application condition for the completely randomized design ANOVA. This is not strictly achieved by equal-frequency discretization. In this instance, it is enforced as follows:

```
CREATE TABLE WineQuality4s AS
(SELECT * FROM WineQuality4 WHERE alcohol=1 LIMIT 1100)
UNION
(SELECT * FROM WineQuality4 WHERE alcohol=2 LIMIT 1100)
UNION
(SELECT * FROM WineQuality4 WHERE alcohol=3 LIMIT 1100)
UNION
(SELECT * FROM WineQuality4 WHERE alcohol=4 LIMIT 1100);
```

Using the resulting data set, the completely randomized design of the one-way ANOVA test can be used:

```
CALL nza..ANOVA_CRD_TEST('intable=WineQuality4s,

incolumn=quality, treatment=alcohol');
```

The procedure returns a text string containing the full set of calculated quantities, including the value of the F -test statistic and the corresponding cumulative distribution function value, which is nearly 1 in this example, indicating a significant relationship between the alcohol contents and the quality.

You may want to look at several attributes, not just the quality attribute, at the same time.

```
CALL nza..ANOVA_CRD_TEST('intable=WineQuality4s,
incolumn=quality;volatile_acidity ; citric_acid ,
treatment=alcohol,outtable = wqan');
```

As there will be more output rows now, you need to have an output table. By inspecting it, we see that the alcohol differentiates not only the quality, but also other attributes like citric_acid or volatile_acidity.

Manova Example

The MANOVA_ONE_WAY_TEST and MANOVA_TWO_WAY_TEST stored procedures have the following syntax:

```
CALL nza..MANOVA_ONE_WAY_TEST(NVARCHAR(ANY) paramString)
CALL nza..MANOVA_TWO_WAY_TEST(NVARCHAR(ANY) paramString)
```

Parameters are:

paramString—input parameters
specification Type: NVARCHAR(ANY)

intable— the input table name
Type: NVARCHAR(ANY)

outtable— the output table name
Type: NVARCHAR(ANY)

factor1—the input table column that identifies a first
factor Type: NVARCHAR(ANY)

factor2—the input table column that identifies a second
factor For MANOVA_TWO_WAY_TEST only
Type: NVARCHAR(ANY)

incolumn—the input table observation columns, separated by a semicolon (;)
Type: NVARCHAR(ANY)

by—the input table column that splits the table into subtables, separated by a semicolon (;)
Type: NVARCHAR(ANY)

type—columns (traditional) or trcv (task/row/column/value, where row is the default value)
Type: NVARCHAR(ANY)

id—the input table column that uniquely identifies records
Type: NVARCHAR(ANY)

A string that confirms the execution of the process is returned. The output is contained in the output table of type TEXT.

To illustrate the MANOVA algorithm, the *WineQuality* data set is used. This data set examines how the pH value, the chlorides, and the density of the wine influence the wine quality.

In the example, the input table is converted to row-column-value (RCV) format, and then a MANOVA one-way stored procedure is called. The result is printed by calling the PRINT_MANOVA_ONE_WAY_TEST stored procedure. The resulting table shows the pH-values and the significance for the Wilks' test, the Roy's test, the Pillai's test, and the Hotelling's test.

```
CALL nza..COL2TRCV_MANOVA_ONE_WAY_TEST('intable=nza..winequality,
incolumn=ph;chlorides;density,id=id, factor1=quality, outtable=trcv1');
CALL nza..MANOVA_ONE_WAY_TEST('intable=trcv1, outtable=res_1');
CALL nza..PRINT_MANOVA_ONE_WAY_TEST('intable=res_1');
```

Principal Component Analysis Example

To demonstrate the creation and application of Principal Component Analysis model this example uses the *WineQuality* data set:

```
CALL nza..PCA('intable=nza..WineQuality, model=wq_pca, id=id, scaleData=TRUE,
centerData=TRUE, forceEigensolve=FALSE');
```

To apply the model (project the data into the space spanned by the selected number of principal components of the corresponding dimensionality) the following call should be used:

```
CALL nza..PROJECT_PCA('intable=nza..WineQuality, model=wq_pca,
id=id, outtable=wq_proj4, pcNumber=4');
```

CHAPTER 7
Tree-Shaped Bayesian Networks

Tree-shaped Bayesian networks formally belong to the data exploration category. However, this algorithm is considerably more complex than other data exploration algorithms and not as widely known, warranting detailed description.

A Bayesian network can be considered a graphical representation of probabilistically described relationships within a set of attributes, allowing probabilistic inference to be performed. The representation is created by extracting the structural properties of the distribution from the data.

Creating and using general Bayesian networks are algorithmically and computationally complex. Tree-shaped Bayesian networks, however, constitute a simplified subclass of Bayesian networks with restrictions imposed on the type of attribute relationships that can be discovered and represented. The restrictions permit simpler and more efficient algorithms as well as more straightforward interpretation. Tree-shaped Bayesian networks may be not sufficient for highly-accurate prediction, but provide an excellent qualitative description of the relationship structure observed in the data.

Background

Bayesian networks can be used to represent a joint probability distribution of multiple discrete attributes. A Bayesian network consists of two essential parts:

structure—a directed acyclic graph, where nodes represent attributes and directed edges represent “direct influence”

conditional probability distributions—the conditional probability of a node given its parents. or the marginal distribution for the node, if there are no parent nodes

A Bayesian network can be used in a formal probabilistic inference process, but even in informal interpretations, can be a valuable source of the following insights into the data:

determining primary and secondary relationships between attributes

which attributes are independent or relatively independent given other attributes

strength of the primary and secondary relationships

An analogous representation for continuous attributes is possible. Under the assumption of normality, dependencies between continuous attributes can be expressed in a simplified form of partial correlations. This is based on the following equation that holds for normal random variables and Y:

$$E\ Y|X\ = EY\ \ r\ _{Y} \quad \frac{X - EX}{_{X}} \tag{38}$$

where $_X$ is the standard deviation of X, $_Y$ is the standard deviation of Y, E is the expected value symbol, and r is the correlation coefficient of X and Y. The root cause of correlations is of interest when looking at Bayesian networks for continuous attributes.

Apart from providing valuable insights about the relationship structure in the data, Bayesian networks can be considered predictive models and applied to new data. Unlike ordinary classification and regression models, the target attribute for Bayesian network prediction is not specified at the model building phase, but at the model application phase. In the most basic case, the network can be applied to predict any attribute of those for which it was created, based on the known values of the remaining attributes. When considering only continuous attributes and using correlations to describe their relationships, as discussed above, the Bayesian network implicitly represents a collection of regression models for each attribute. Note that the simplicity of this representation does not guarantee high prediction quality, particularly if there no sufficiently strong linear correlations in the data, but it can be still useful as a quick model prototyping approach.

If the Bayesian network has the structure of a tree, spanning the specified set of attributes, then the inference process is simplified, because if Z is on the path from X to Y, then the correlation between X and Y can be decomposed as follows:

$$r_{XY} = r_{XZ} \cdot r_{ZY} \tag{39}$$

Another important property of a Bayesian network tree is that there exists a Chow-Liu algorithm approximating any joint probability distribution with a tree in an optimal way.

In tree-shaped Bayesian networks, edge orientations become immaterial and therefore inference about which attribute is caused by which becomes impossible. As such, the edges in a tree-shaped Bayesian network should be considered undirected. All other structural insights, however, are possible. In addition, the overview of major attribute relationships becomes more comprehensive, which is important when the number of attributes numbers in the hundreds or more.

Applications

Bayesian network tree discovery algorithms can be applied whenever a Bayesian network is needed for:

determining the major dependencies among attributes

Creating a simplistic model to derive correlations between attributes from a small set of pre-computed ones

identifying subsets of attributes that do not appear to be related to the same topic, by splitting the tree by the weakest links

identifying attributes of central importance, that is, those that are central in the network, with many links

Available Functionality

The IBM Netezza In-Database Analytics package implements the tree version of Bayesian networks for continuous attributes. It provides the stored procedures listed below.

TBNET1G—A spanning tree is constructed linking all specified attributes based on the strongest correlations, providing the user an overview of the most significant interrelations governing the whole set of attributes.

TBNET2G—A spanning tree is constructed linking all specified attributes based on the strongest correlations. The distinctive feature compared to **TBNET1G** and standard Bayesian network construction algorithms is that one specifies two subsets of attributes and the resulting tree is bipartite — nodes within each set are not directly connected. This feature is particularly useful when the two subsets of attributes characterize distinct objects or distinct parts of an object, and only “external” links between different objects/parts are of interest and not “internal” ones within.

BTBNET_GROW— has the functionality of TBNET1G with the additional possibility to include binary variable(s) by specifying either an identity relation (an attribute equals a value or not, for discrete attributes mainly) or a-greater-than-relation (for discretizing continuous one). These new variables will be treated like “continuous” with “truth” being represented by 1 and “falsehood” by 0.

TBNET_GROW—A spanning tree is constructed linking all specified attributes based on the strongest correlations. The generated model is suitable for predicting values of any attribute with the **TBNET_APPLY** procedure.

TBNET_APPLY—The Bayesian network generated by is used for value prediction **TBNET_GROW** in a continuous table based on the formula:

$$E\ Y|X\ = EY \qquad r \quad \frac{X - EX}{y \qquad X} \tag{40}$$

TANET_GROW— (tree-augmented network) A spanning tree is constructed linking all specified attributes based on the strongest correlations. Subsequently the coefficients are trained separately in classes identified by a class attribute. The generated model is suitable for predicting values of any attribute with the TANET_APPLY procedure.

TANET_APPLY—The Bayesian network generated by TANET_GROW is used for value prediction in a continuous table in a way similar to TBN_APPLY, but with coefficients specific for particular classes.

MTBNET_GROW— for each level of a class attribute, a separate spanning tree is constructed linking all specified attributes based on the strongest correlations. The generated model is suitable for predicting values of any attribute with the TANET_APPLY procedure.

MTBNET_DIFF— lists the differences between trees at the different levels of the class attribute.

Examples

The functionality of the tree-shaped Bayesian networks implementation is illustrated by the examples using the *WineQuality* data set.

The **TBNET1G** procedure, can be applied to all attributes of the data set in the following manner:

```
CALL nza..TBNET1G('intable=nza..WineQuality,
                   incolumn=fixed_acidity;
                   volatile_acidity;
                   citric_acid;
                   residualsugar;
                   chlorides;
                   free_sulfur_dioxide;
                   total_sulfur_dioxide;
                   density;
                   pH;
                   sulphates;
                   alcohol;
                   quality,
                   model=wq_tbn1');
```

The resulting model, specified via the model argument, consists of a table containing the network structure with each row representing a network edge and the following columns:

BNID—the internally assigned Bayesian network ID

VARXID—the internally assigned ID of the first attribute of the edge

VARXNAME—the name of the first attribute of the edge

VARYID—the internally assigned ID of the second attribute of the edge

VARYNAME—the name of the second attribute of the edge

CORR—the linear correlation between the attributes of the edge

Note that, since the network is undirected, the representation of an attribute being shown as the first or second on an edge carries no special meaning.

You can inspect the network created for the *WineQuality* data set as follows:

```
SELECT * FROM nza_meta_wq_tbn1_model ORDER BY ABS(CORR) DESC;
```

Notice that **residualsugar** and **density** are the two most strongly correlated attributes. The **density** attribute is also highly correlated—negatively—to the **alcohol** attribute. The correlation values can be verified to match those returned by the **CORR** procedure:

```
CALL nza..CORR('intable=nza..WineQuality, incolumn=residualsugar; density');
```

```
CALL nza..CORR('intable=nza..WineQuality, incolumn=density; alcohol');
```

If you are particularly interested in attributes most related to the **quality** attribute, you can filter the corresponding edges by adding an appropriate **WHERE** clause to the above query:

```
SELECT * FROM nza_meta_wq_tbn1_model
WHERE VARXNAME='QUALITY' OR VARYNAME='QUALITY'
ORDER BY ABS(CORR) DESC;
```

Other noteworthy optional parameters not illustrated in the above example include:

> **baseidx**—the numeric ID to be assigned to the first attribute, for easier internal management
>
> **samplesize**—the size of the sample to take if the number of instances/attributes is too large, where processing time may be too long for the user

While the **samplesize** parameter is not required for the *WineQuality* data set (due to its small size) the following call demonstrates the effect of the **baseidx** parameter being set to 1:

```
CALL nza..TBNET1G('intable=nza..WineQuality,
                   incolumn=fixed_acidity;
                   volatile_acidity;
                   citric_acid;
                   residualsugar;
                   chlorides;
                   free_sulfur_dioxide;
                   total_sulfur_dioxide;
                   density;
                   pH;
                   sulphates;
                   alcohol;
                   quality,
                   baseidx=1,
                   model=wq_tbn1_1');

SELECT * FROM nza_meta_wq_tbn1_1_model ORDER BY ABS(CORR) DESC;
```

The node numbering starts from the specified value. This argument may be useful for configuring disjoint node number spaces when creating and analyzing several networks for the same data set using different attribute sets.

With the **TBNET2G** procedure, the **incolumn** argument allows you to specify two disjoint attribute subsets, Each attribute name is followed by a colon (:) and either X or Y to distinguish the two sets. The algorithm considers only inter-subset links and not intra-subset links when creating the network structure. This is demonstrated in the example below, where the set of all attributes in the *WineQuality* data set is split into two subsets:

```
CALL nza..TBNET2G('intable=nza..WineQuality,
                   incolumn=fixed_acidity:X;
                   volatile_acidity:X;
                   citric_acid:X;
                   residualsugar:X;
                   chlorides:X;
                   free_sulfur_dioxide:X;
                   total_sulfur_dioxide:X;
                   density:Y;
                   pH:Y;
                   sulphates:Y;
```

```
                    alcohol:Y;
                    quality:Y,
                    model=wq_tbn2');
```

You can examine the created structure using:

```
SELECT * FROM nza_meta_wq_tbn2_model ORDER BY ABS(CORR) DESC;
```

All the created edges link attributes from one set to attributes from the other set. In the second subset, it is the **density** attribute that has the most frequent and strongest relationships with the attributes from the first subset.

The **TBNET_GROW** procedure is called in the same way as the **TBNET1G** procedure.

```
CALL nza..TBNET_GROW('intable=nza..WineQuality_train,
                      incolumn=fixed_acidity;
                      volatile_acidity;
                      citric_acid;
                      residualsugar;
                      chlorides;
                      free_sulfur_dioxide;
                      total_sulfur_dioxide;
                      density;
                      pH;
                      sulphates;
                      alcohol;
                      quality,
                      model=wq_tbnm');
```

This call uses the training subset of the *WineQuality* data to allow subsequent evaluation of the predictions on the test subset.

Examining the resulting output table:

```
SELECT * FROM nza_meta_wq_tbnm_model ORDER BY ABS(CORR) DESC;
```

reveals the generated contents to be richer. For each network edge the means and standard deviations of the two nodes being linked are calculated and stored, making it possible to apply the network for prediction. This is demonstrated by the following two calls, one predicting the **quality** attribute and the other predicting the **density** attribute on the test set:

```
CALL nza..TBNET_APPLY('intable=nza..WineQuality_test, id=id,

target=quality, outtable=WineQuality_quality_tbnm, model=wq_tbnm');

CALL nza..TBNET_APPLY('intable=nza..WineQuality_test, id=id,

target=density, outtable=WineQuality_density_tbnm, model=wq_tbnm');
```

The quality of the predictions can be evaluated using the mean square error:

```
CALL nza..MSE ('pred_table=WineQuality_quality_tbnm,
pred_column=quality_pred, pred_id=id, true_table=nza..WineQuality_test,

true_column=quality, true_id=id');


CALL nza..MSE ('pred_table=WineQuality_density_tbnm,
pred_column=density_pred, pred_id=id, true_table=nza..WineQuality_test,
true_column=density, true_id=id');
```

Use the following auxiliary view to put the true and predicted values next to each other:

```
CREATE VIEW WineQuality_tbnm_pred AS
SELECT PQ.quality_pred, T.quality as quality, PD.density_pred, T.density as
density
FROM WineQuality_quality_tbnm PQ, WineQuality_density_tbnm PD,
nza..WineQuality_test T
WHERE PQ.id=T.id AND PD.id=T.id;
```

You can also calculate their test set correlations:

```
CALL nza..CORR('intable=WineQuality_tbnm_pred,
incolumn=quality_pred; quality');
CALL nza..CORR('intable=WineQuality_tbnm_pred,

incolumn=density_pred; density');
```

In this example, the density attribute turns out to be a much better predictor than the quality attribute. The results obtained for the quality attribute, however, are comparable to those possible with the *k*NN or regression tree algorithms. For more information on *k*NN, see Nearest Neighbors. For more information on regression tree algorithms, see Regression Trees.

The **TBNET_APPLY** procedure accepts an optional **type** argument that can be used to select one of the following variations of the prediction algorithm:

best—using the most correlated neighbor node, which is the default value

neighbors—using weighted prediction of all neighbor nodes

nn-neighbors—the same as above, but with NULL-value nodes, that is, those corresponding to attributes that have missing values for the instance for which the prediction is calculated, skipped

The previous two calls to **TBNET_APPLY** can be repeated with the prediction type set to neighbors:

```
CALL nza..TBNET_APPLY('intable=nza..WineQuality_test, id=id, target=quality,

type=neighbors, outtable=WineQuality_quality_tbnm_n, model=wq_tbnm');

CALL nza..TBNET_APPLY('intable=nza..WineQuality_test, id=id, target=density,

type=neighbors, outtable=WineQuality_density_tbnm_n, model=wq_tbnm');
```

You can evaluate the prediction quality:

```
CALL nza..MSE ('pred_table=WineQuality_quality_tbnm_n,
pred_column=quality_pred, pred_id=id, true_table=nza..WineQuality_test,

true_column=quality, true_id=id');

CALL nza..MSE ('pred_table=WineQuality_density_tbnm_n,
pred_column=density_pred, pred_id=id, true_table=nza..WineQuality_test,

true_column=density, true_id=id');
```

The achieved mean square error values are higher than with the default setting (**type=best**), which is desirable in most cases.

CHAPTER 8
Discretization

Discretization algorithms can be divided into two main categories:

unsupervised—the target concept (class attribute) is not used in the criteria for setting interval bounds,

supervised—the target concept (class attribute) is taken into account when seeking the most appropriate interval bounds.

Unsupervised algorithms are typically applied when discretization is performed without the intention of subsequently using the data for classification, or when a variety of classification tasks can be considered for the same data set, with different attributes used as the target concept. While both categories include algorithms of various levels of refinement and complexity, supervised algorithms tend to be more complex conceptually and computationally.

Background

IBM Netezza In-Database Analytics contains implementations of the following discretization algorithms:

equal-width discretization—an unsupervised discretization algorithm using the equal width criterion for interval bound setting

equal-frequency discretization—an unsupervised discretization algorithm using the equal frequency, that is, equal data count, criterion for interval bound setting

minimum-entropy discretization—a supervised discretization algorithm that identifies the most appropriate interval bounds by minimizing class distribution impurity

The equal-width and equal-frequency algorithms are less complex, and therefore more computationally efficient. They can handle large numbers of attributes and the quality of the discretization intervals they produce are sufficient for several applications.

The equal-width algorithm identifies the range of the discretized attribute and divides it evenly into a specified number of intervals. However. this approach is not robust with respect to outliers, which may throw off the algorithm by extending the attribute’s range, resulting in bad interval data. Use the equal-width algorithm only after checking the discretized attributes for outliers and removing them, if necessary.

The equal-frequency algorithm identifies interval bounds in a more robust way. It adapts to the actual data distribution by seeking intervals containing the values of the discretized attribute corresponding to the same number of instances. However, exact results may be impossible when the size of the data set does not divide evenly by the required number of intervals or when there are several instances where the discretized attribute takes the same values. For this reason, the algorithm allows the user to specify a parameter to control its disposition to modify the required number of intervals to achieve more uniform interval frequencies.

The minimum-entropy discretization is a top-down method that starts from a single interval covering the whole attribute range and divides it into smaller intervals. It seeks interval bounds that minimize the class *impurity* of subsets of instances corresponding to particular intervals, measured by the entropy. For interval I, it is calculated as:

$$E\ I = \sum_{d \in C} -P\ d|I \log_2 P\ d|I \qquad (41)$$

where d iterates over all classes, $P\ d|I = \frac{|T_I^d|}{|T_I|}$ is the probability of class d in interval I, estimated based on the subset of instances T_I where the attribute a being discretized takes value in interval I:

$$T_I = \{\ x \in T | a\ x \in I\ \} \qquad (42)$$

and d in the superscript denotes selection of instances of class d only. Dividing intervals continues until an automatic stop criterion based on the minimum description length (MDL) principle is satisfied.

Applications

Discretization is one of the most commonly applied data transformations for data mining. Typically, it is performed as a preprocessing step before classification, when the classification algorithm to be applied requires discrete attributes or is more efficient using them.

While the most common reason for discretizing continuous attributes is a modeling algorithm’s inability to use continuous attributes, there may be more reasons to consider it a useful data transformation even if the algorithm to be used subsequently supports continuous attributes directly:

it is likely to considerably reduce the computational effort of modeling

it usually results in a simpler and more readable model

it may help prevent overfitting by eliminating some opportunities to overfit

Available Functionality

The following discretization functionality is available in IBM Netezza In-Database Analytics:

the equal-width, equal-frequency, and minimum-entropy algorithms — the **EWDISC**, **EFDISC**, and **EMDISC** stored procedures,

user-specified number of intervals as stop criteria for the equal-width and equal-frequency algorithms

an automatic MDL-based stop criterion for the minimum-entropy algorithm

output tables containing interval bounds on output

output views containing the original data set with a discretized attribute attached

The actual number of intervals is guaranteed to be the same as specified on input only for the equal-width algorithm. The equal-frequency algorithm can produce fewer intervals if necessary to achieve a sufficiently uniform interval frequency. This is controlled by a precision parameter, **binprec**, that determines how large a discrepancy between the theoretically required and actual interval frequency is acceptable. The theoretically required interval frequency is the data set size divided by the number of requested intervals.

Examples

The example illustrates using each of the discretization algorithms in the IBM Netezza In-Database Analytics package to discretize all continuous attributes in the *CensusIncome* data set. The following three calls request that all the continuous attributes present in the data set be discretized using the equal-width, equal-frequency, and minimum-entropy algorithms.

```
 CALL nza..EWDISC('intable=nza..CensusIncome_train,
                         incolumn=age:5;
                               wage_per_hour:4;
                               capital_gains:3;
                               capital_losses:3;
                               dividends_from_stocks:3;
                               num_persons_worked_for_employer:2;
                               weeks_worked_in_year:5,
                         outtable=ci_ewd');
```

```
CALL nza..EFDISC('intable=nza..CensusIncome_train, incolumn=age:5;

                               wage_per_hour:4;
                               capital_gains:3;
                               capital_losses:3;
                               dividends_from_stocks:3;
                               num_persons_worked_for_employer:2;
                               weeks_worked_in_year:5,
                         binprec=1, outtable=ci_efd');
```

```
CALL nza..EMDISC('intable=nza..CensusIncome_train,
                         target=income,
                         incolumn=age;
```

```
          wage_per_hour;
          capital_gains;
          capital_losses;
          dividends_from_stocks;
          num_persons_worked_for_employer;
          weeks_worked_in_year,
      outtable=ci_emd');
```

The equal-width and equal-frequency algorithms are instructed to create specific numbers of intervals for each attribute after the colons following the attribute names. The above calls requested five (5) intervals for the age and **weeks_worked_in_year** attributes, four (4) intervals for the **wage_per_hour** attribute, three (3) intervals for the **capital_gains**, **capital_losses**, and **dividends_from_stocks** attributes, and two (2) intervals for the **num_persons_worked_for_employer** attribute.

The equal-frequency algorithm takes an additional **binprec** argument that specifies the tolerance for the frequency equality condition, which may be impossible to satisfy exactly. The actual number of instances in an interval can differ from the "theoretically" required number of instances per interval, calculated as the data set size divided by the number of requested intervals, by no more than **binprec** · 100%. The value of 1 used in the example is the maximum, which instructs the algorithm to accept even substantially unequal interval frequencies. However, it may still divide the discretized attribute range to less intervals than requested. For the minimum-entropy algorithm the class column with respect to which the supervised discretization process is performed is specified via the **target** argument.

All the discretization procedures generate output tables containing interval bounds for each discretized attribute. These intervals can be applied to replace the original continuous attributes with their discretized versions both in the data set on which the intervals were identified, as well as on a new data set containing the same attributes. This capability is crucial whenever discretization is performed as part of a modeling process where one has to apply the discretization intervals identified on the training set to another data set for which the model is used to generate predictions.

Regardless of the algorithm used to create discretization intervals, they can be applied to a data set in the same way, as demonstrated below for the training test:

```
CALL nza..APPLY_DISC('intable=nza..CensusIncome_train,

btable=ci_ewd, outtable=CensusIncome_train_ewd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_train,

btable=ci_efd, outtable=CensusIncome_train_efd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_train,

btable=ci_emd, outtable=CensusIncome_train_emd, replace=TRUE');
```

The **replace=TRUE** argument requests that the original continuous attributes are not preserved in the output tables.

You can then apply the same discretization intervals to the validation and test sets:

```
CALL nza..APPLY_DISC('intable=nza..CensusIncome_val, btable=ci_ewd,

outtable=CensusIncome_val_ewd, replace=TRUE');
```

```
CALL nza..APPLY_DISC('intable=nza..CensusIncome_val, btable=ci_efd,
outtable=CensusIncome_val_efd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_val, btable=ci_emd,
outtable=CensusIncome_val_emd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_test, btable=ci_ewd,
outtable=CensusIncome_test_ewd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_test, btable=ci_efd,
outtable=CensusIncome_test_efd, replace=TRUE');

CALL nza..APPLY_DISC('intable=nza..CensusIncome_test, btable=ci_emd,
outtable=CensusIncome_test_emd, replace=TRUE');
```

The continuous attributes in the validation and test sets have been discretized using the discretization intervals identified by each of the three discretization methods on the training set. This process should always be performed when using discretization as a data transformation on data that is to be used for predictive modeling.

CHAPTER 9

Standardization and Normalization

Standardization and normalization are arithmetic transformations applied to continuous attributes to modify their values to achieve a desired change of the range or distribution.

Background

The two basic standardization and normalization operations are specified as follows:

standardization—subtract the mean and divide the difference by the standard deviation, so that the resulting modified attribute has mean 0 and standard deviation 1:

$$a'x = \frac{a x - m_a D}{s_a D} \tag{43}$$

where $m_a D$ and $s_a D$ denote the mean and standard deviation of attribute a on data set D being transformed,

normalization—divide by the maximum absolute value, so that the resulting modified attribute has values in the $[-1, 1]$ interval:

$$a'x = \frac{a x}{\max_{x \in D} |a x|} \tag{44}$$

More refined transformations, which use the length (Euclidean norm) of attribute value vectors, are sometimes useful. Specifically, let $// a //_D$ be the norm of the vector of values of attribute a on data set D being transformed:

$$\|a\|_D = \sqrt{\sum_{x \in D} a^2 x} \tag{45}$$

and let $\| a_{\{i_1, i_2, \ldots, i_m\}} x \|$ be the norm of the m-element vector of values of selected attributes $i_1, i_2, \ldots, i_m$ for instance (row) x:

$$\| a_{\{i_1, i_2, \ldots, i_m\}} x \| = \sqrt{\sum_{k=1}^{m} a_{i_k}^2 x} \tag{46}$$

The former can be called the *column norm* of attribute *a* and the latter called the row norm of attributes $a_{i_1, i_2, \ldots, i_m}$. Then the following additional types of normalization can be considered:

unit normalization—divide by the column norm, so that the resulting modified attribute has a column norm of 1:

$$a' x = \frac{a x}{\|a\|} \tag{47}$$

row normalization—for each attribute from a specified subset, divide by the corresponding row norm, so that the modified attributes have a row norm of 1 for each instance:

$$a'_{i_k} x = \frac{a_{i_k} x}{\| a_{\{i_1, i_2, \ldots, i_m\}} x \|} \tag{48}$$

vector normalization—for each attribute from a specified subset, divide by the maximum corresponding row norm, so that the modified attributes have row norms between 0 and 1:

$$a'_{i_k}(x) = \frac{a_{i_k}(x)}{\max_{x \in D} \| a_{\{i_1, i_2, \ldots, i_m\}}(x) \|} \tag{49}$$

Applications

The primary reason to apply standardization, normalization, and related transformation is to make the data better suited for some algorithms to be applied subsequently. This is typical for memory-based classification and regression, as well as distance-based clustering algorithms. The quality of predictions generated by such algorithms strongly depends on the employed distance measures. An appropriate transformation may be necessary to ensure an attribute does not have too much or too little impact on the calculated distance.

Available Functionality

The IBM Netezza In-Database Analytics package offers the following standardization and normalization functionality via the **STD_NORM** stored procedure:

standardization of one or more specified attributes

normalization of one or more specified attributes

unit normalization of one ore more specified attributes

row normalization of a specified set of attributes

vector normalization of a specified set of attributes

multiple attributes transformed, possibly with different transformations, in one call

Different standardization or normalization operations can be requested for different attributes in a single call to the procedure that implements this functionality. These operations are specified using the **transform** parameter that may contain several semicolon-separated transformation specifiers of the following form:

attr:L—leave attribute **attr** unchanged

attr:S—standardize attribute attr

attr:N—normalize attribute **attr**

attr:U—apply unit normalization to attribute **attr**

attr1/attr2/attr3:C—apply row normalization to attributes **attr1**, **attr2**, **attr3**, where one or more attributes must be specified

attr1/attr2/attr3:V—apply vector normalization to attributes **attr1**, **attr2**, **attr3**, where one or more attributes must be specified

All attributes being transformed must be numeric data types. An output table is created that contains appropriately modified attributes.

Examples

To illustrate the application of all the various types standardization and normalization, they are applied to transform continuous attributes in the *CensusIncome* data set as follows:

the **age** attribute is standardized

the **wage_per_hour** attribute is normalized

the **dividends_from_stocks** is unit-normalized

the **capital_gains** and **capital_losses** attributes are row-normalized

the **num_persons_worked_for_employer** and **weeks_worked_in_year** attributes are vector-normalized

These operations are performed by the following call:

```
CALL nza..STD_NORM('intable=nza..CensusIncome, incolumn=age:S;

                                     wage_per_hour:N;
```

```
                      dividends_from_stocks:U;
                      capital_gains/capital_losses:C;
                      num_persons_worked_for_employer/weeks_worked_in_year:V,
                    id=id,
                    outtable=ci_sn1');
```

The output table contains one column for each attribute being transformed, with a prefix attached to the name indicating the type of transformation applied, **std_**, **nrm_**, **nru_** or **nrc_**. For vector-normalized attributes the output column name is created by concatenating the names of the input columns used, separated by an underscore and prefixed by **nrm_**.

For a more practical example, consider the following calls requesting that all continuous attributes in the *CensusIncome* data set be standardized. The operation is performed separately for the training, validation, and test subsets, and their resulting standardized versions can be used for creating and evaluating classification or clustering models.

```
CALL nza..STD_NORM('intable=nza..CensusIncome_train,
                         incolumn=age:S;
                               wage_per_hour:S;
                               capital_gains:S;
                               capital_losses:S;
                               dividends_from_stocks:S;
                               num_persons_worked_for_employer:S;
                   weeks_worked_in_year:S,
                   id=id, outtable=CensusIncome_train_std_num');


CREATE TABLE CensusIncome_train_std AS
SELECT N.*,
      class_of_worker,
      detailed_industry_recode,
      detailed_occupation_recode,
      education, enroll_in_edu_inst_last_wk,
      marital_status,
      major_industry_code,
      major_occupation_code,
      race,
      hispanic_origin,
      sex,
      member_of_a_labor_union,
      reason_for_unemployment,
      full_or_part_time_employment_stat,
      tax_filer_stat,
      region_of_previous_residence,
      state_of_previous_residence,   detailed_household_and_family_stat,
      detailed_household_summary_in_household,
      migration_code_change_in_msa,  migration_code_change_in_reg,
      migration_code_move_within_reg,
      live_in_this_house_1_year_ago,
      migration_prev_res_in_sunbelt,
      family_members_under_18,
      country_of_birth_father,
      country_of_birth_mother,
      country_of_birth_self,
      citizenship,
      own_business_or_self_employed,
      fill_inc_questionnaire_for_veterans_admin,
      veterans_benefits,
      year,
      income
FROM CensusIncome_train_std_num N, nza..CensusIncome_train
A WHERE N.id=A.id;

CALL nza..STD_NORM('intable=nza..CensusIncome_val, incolumn=age:S;

                               wage_per_hour:S;
                               capital_gains:S;
                               capital_losses:S;
                               dividends_from_stocks:S;
                               num_persons_worked_for_employer:S;
                         weeks_worked_in_year:S,
                   id=id, outtable=CensusIncome_val_std_num');


CREATE TABLE CensusIncome_val_std AS
SELECT N.*,
      class_of_worker,
      detailed_industry_recode,
      detailed_occupation_recode,
      education,
      enroll_in_edu_inst_last_wk,
      marital_status,
      major_industry_code,
      major_occupation_code,
      race,
      hispanic_origin,
      sex,
      member_of_a_labor_union,
      reason_for_unemployment,
      full_or_part_time_employment_stat,
      tax_filer_stat,
      region_of_previous_residence,
      state_of_previous_residence,   detailed_household_and_family_stat,
      detailed_household_summary_in_household,
      migration_code_change_in_msa,  migration_code_change_in_reg,
      migration_code_move_within_reg, live_in_this_house_1_year_ago,
       migration_prev_res_in_sunbelt,
      family_members_under_18,
      country_of_birth_father,
      country_of_birth_mother,
      country_of_birth_self,
      citizenship,
      own_business_or_self_employed,
      fill_inc_questionnaire_for_veterans_admin,
      veterans_benefits,
      year,
      income
FROM CensusIncome_val_std_num N, nza..CensusIncome_val
A WHERE N.id=A.id;

CALL nza..STD_NORM('intable=nza..CensusIncome_test,
                         incolumn=age:S;
                               wage_per_hour:S;
```

```
                                capital_gains:S;
                                capital_losses:S;
                                dividends_from_stocks:S;
                                num_persons_worked_for_employer:S;
                           weeks_worked_in_year:S,
                   id=id, outtable=CensusIncome_test_std_num');


CREATE TABLE CensusIncome_test_std AS
SELECT N.*,
      class_of_worker,
      detailed_industry_recode,
      detailed_occupation_recode,
      education,
      enroll_in_edu_inst_last_wk,
      marital_status,
      major_industry_code,
      major_occupation_code,
      race,
      hispanic_origin,
      sex,
      member_of_a_labor_union,
      reason_for_unemployment,
      full_or_part_time_employment_stat,
      tax_filer_stat,
      region_of_previous_residence,
      state_of_previous_residence,   detailed_household_and_family_stat,
      detailed_household_summary_in_household,
      migration_code_change_in_msa,  migration_code_change_in_reg,
      migration_code_move_within_reg, live_in_this_house_1_year_ago,
      migration_prev_res_in_sunbelt,
      family_members_under_18,
      country_of_birth_father,
      country_of_birth_mother,
      country_of_birth_self,
      citizenship,
      own_business_or_self_employed,
      fill_inc_questionnaire_for_veterans_admin,
      veterans_benefits,
      year,
      income
FROM CensusIncome_test_std_num N, nza..CensusIncome_test A
WHERE N.id=A.id;
```

CHAPTER 10

Data Imputation

Background

Data imputation is a practical transformation that provides values for missing attribute fields so that it is possible for algorithms that cannot process data sets with missing values to be used on the data set.

Imputation is one of the most popular approaches to missing value handling in data mining. Other possible approaches, each with their own limitations, include:

ignoring—skipping instances with missing values of one or more attributes

codomain extension—considering “missing” as an additional value added to an attributes codomain

internal processing—algorithm-specific techniques for internal missing value handling

The ignoring approach is not practical when instances with missing values constitute a large fraction of the data set. Even if missing values are minimal, it may have negative impact on the quality of analysis results.

The codomain extension is only applicable to discrete attributes, for which ”missing,” or any other term used, can be considered an actual value. This approach can give good results with predictive modeling algorithms as long as the distribution of missing values is the same in the training set used to create a model as in the data to which the model is subsequently applied.

The third approach, possible with some data mining algorithms, is likely to yield superior results, but typically adds considerable computational complexity.

Compared to the above mentioned alternatives, the data imputation approach can be considered a useful general purpose solution that represents a good compromise between quality and complexity. The idea is to provide missing attribute values with reasonable “guessed” values, which are usually:

modes (the most frequent values) for discrete attributes

means or medians for continuous attributes

Applications

Missing attribute values constitute the most widely encountered data quality issue that must be faced when analyzing real-world data sets. Many data mining algorithms cannot work directly with missing attribute values or require considerable additional computational effort to handle them internally. Filling them by imputation is a one-time transformation that enables subsequent analysis of the data set. It is worthwhile to consider for predictive modeling tasks—classification, regression, and clustering—where the trivial approach of ignoring missing values is usually inappropriate and refined internal missing value handling techniques are not available or are deemed too time-consuming for large data sets.

Available Functionality

The IBM Netezza In-Database Analytics package offers the following imputation functionality via the **IMPUTE_DATA** stored procedure:

discrete attribute value imputation with modes (the most frequent values)

continuous attribute value imputation with modes, means, or

medians. The imputation method is specified via the **method** argument.

Examples

To illustrate the application of data imputation, the following queries create an artificially corrupted copy of the *CensusIncome* data set, specifically from the training, validation, and test subsets.), with 5% of the values of two selected attributes, one discrete and one continuous, replaced by NULL:

```
CREATE TABLE CensusIncome_train_miss AS SELECT *
FROM nza..CensusIncome_train;
CREATE TABLE CensusIncome_test_miss AS SELECT * FROM nza..CensusIncome_test;

CREATE TABLE CensusIncome_val_miss AS SELECT * FROM nza..CensusIncome_val;

UPDATE CensusIncome_train_miss SET capital_gains=NULL WHERE RANDOM()<0.05;
UPDATE CensusIncome_test_miss SET capital_gains=NULL WHERE RANDOM()<0.05;
UPDATE CensusIncome_val_miss SET capital_gains=NULL WHERE RANDOM()<0.05;

UPDATE CensusIncome_train_miss SET sex=NULL WHERE RANDOM()<0.05;
UPDATE CensusIncome_test_miss SET sex=NULL WHERE RANDOM()<0.05;
UPDATE CensusIncome_val_miss SET sex=NULL WHERE RANDOM()<0.05;
```

Now request that the imputation with medians is applied to the corrupted data:

```
CALL nza..IMPUTE_DATA('intable=CensusIncome_train_miss,

outtable=CensusIncome_train_imp, method=median');

CALL nza..IMPUTE_DATA('intable=CensusIncome_test_miss,

outtable=CensusIncome_test_imp, method=median');
```

```
CALL nza..IMPUTE_DATA('intable=CensusIncome_val_miss,
outtable=CensusIncome_val_imp, method=median');
```

This performs the imputation operation using the selected method only for attributes to which it is applicable, that is, continuous attributes in the case of the median method. Attributes without missing values are left unchanged, which results in modification of only the corrupted **capital_gains** attribute. Now a second call can be used to perform imputation for the remaining corrupted discrete attribute **sex**:

```
CALL nza..IMPUTE_DATA('intable=CensusIncome_train_imp, method=freq');

CALL nza..IMPUTE_DATA('intable=CensusIncome_test_imp, method=freq');

CALL nza..IMPUTE_DATA('intable=CensusIncome_val_imp, method=freq');
```

This replaces any remaining missing attribute values with the most frequent values, leaving any other attributes unchanged. Note that no output table has been specified, which instructs the procedure to modify the table passed as the input table produced by the previous call rather than create a new copy. This results in training and test subsets of the *CensusIncome* data with values of the **capital_gains** and **sex** attributes imputed with medians and modes, respectively.

CHAPTER 11

Model Diagnostics

Model diagnostics algorithms are computational techniques used to provide reliable estimates of the predictive utility of data mining models. Such estimates are necessary to select one out of several models that can be generated for the same task using different algorithms or parameter setups, and to decide whether the quality of a final model is sufficient for a particular application.

Background

Algorithms used for model quality assessment include:

model quality indicators

model evaluation procedures

Model quality indicators are measures calculated based on the predictions generated by a given model on a given data set. They are task-specific, and different for different predictive modeling tasks. Model evaluation procedures are responsible for making sure the calculated indicators can serve as reliable model quality estimates on new, unseen data. They are task-independent and can be used for both the classification and regression tasks.

Misclassification Error

The misclassification error is the most commonly-used quality indicator for classification models. It is calculated for model h on data set S with respect to a target concept c (the true class labels of which are known on S) as follows:

$$e_S^c\ h = \frac{\left|\{\ x \in S | h\ x\ \neq c\ x\ \}\right|}{|S|} \qquad (50)$$

This is the ratio of the number of instances from S misclassified by model h to the total number of instances in S . You may also consider $1- e^c_S\ h$ as the *accuracy* of a classification model.

Confusion Matrix

The misclassification error does not always adequately represent the quality of classification models. In some applications, particularly with unbalanced classes or non-uniform misclassification costs, it may be necessary to look deeper into predictions generated for instances of different classes. This is possible using the confusion matrix, which for any pair of classes $d_1, d_2 \in C$ provides the number of instances (from data set S used for the evaluation) of true class d_2 classified to class d_1 :

$$M^c_S\ h[d_1, d_2] = \left| \{ x \in S \mid h\ x = d_1, c\ x = d_2 \} \right| \tag{51}$$

The distribution of misclassification mistakes represented by the confusion matrix can be described by a number of indicators, defined below. Assuming a two-class classification task with $C = \{0,\ 1\}$ where 1 denotes the “positive” class and 0 denotes the “negative” class, the notation can be described by the following table:

		Predicted h(x)	
		Negative (0)	Positive (1)
Actual C(x)	Negative (0)	TN	FP
	Positive (1)	FN	TP

The following table describes the terms used to refer to confusion matrix entries:

Table 9: Confusion matrix entries defined

Term	Description
TN	The number of *true negatives*, $M_S^c\ (h)[0, 0]$
FN	The number of *false negatives*, $M_S^c\ (h)[0, 1]$
FP	The number of *false positives*, $M_S^c\ (h)[1, 0]$
TP	The number of *true positives*, $M_S^c\ (h)[1,1\ ]$

Table 10: Confusion matrix-based quality indicators

Term	Description
True positive rate	The ratio of instances correctly classified as positive to all positive instances: $\frac{TP}{TP+FN}$
False positive rate	The ratio of instances incorrectly classified as positive to all negative instances: $\frac{FP}{FP+TN}$
Positive predictive value	The ratio of instances correctly classified as positive to all instances classified as positive: $\frac{TP}{TP+FP}$
True negative rate	The ratio of instances correctly classified as negative to all negative instances: $\frac{TN}{TN+FN}$
False negative rate	The ratio of instances incorrectly classified as negative to all positive instances: $\frac{FP}{FP+TP}$
Recall/ sensitivity	The same as the *true positive* rate.
Precision	The same as the *positive predictive* value.
Specificity	The same as 1 - *false positive* rate, which can also be stated as: $\frac{TN}{TN+FP}$
Accuracy	The ratio of the total number of correctly classified instances: $\frac{TN+TP}{TN+FN+FN+TP}$

►

These indicators describe the level at which the evaluated classifier succeeds or fails to correctly detect the positive class. Using all indicators is unnecessary as it results in redundant information; however, no single indicator can be considered sufficient, either. The following performance measures are usually used in complementary pairs:

true positive rate and false positive rate

precision and recall

sensitivity and specificity

The true positive rate, also called recall or sensitivity, is a member of all these pairs, under those names. It represents the share of positive instances correctly detected by the classifier. It should be maximized, but a value of 1 results in a classifier that always predicts the positive class. Thus, it must be accompanied by a complementary indicator. It could be the false positive rate, representing the share of negative instances that are incorrectly reported as positive or “false alarms.” This, too, should be minimized, and a trivial classifier achieving the perfect 0 false positive rate is the one issuing no alarms at all.

Precision measures the share of all instances predicted as positive that are truly positive, which should be maximized. Specificity is the same as the 1's complement of the false positive rate.

For classification tasks with more than two classes these indicators can be defined on a per-class basis, with instances of one class considered positive and instances of all other classes considered negative.

Mean Absolute Error

The mean absolute error (MAE) is used to evaluate the quality of regression models. For it model can be calculated on data set S with respect to target function f as follows:

$$\frac{}{|S|}\sum_{x\in S}|\ f\ x - h\ x\ | \tag{52}$$

Mean Square Error

A popular alternative quality measure for regression models is the mean square error (MSE):

$$\frac{1}{|S|}\sum_{x\in S} f\ x - h\ x^{2} \tag{53}$$

It increases the impact of large differences between predicted and true values and has favorable analytical properties, due to its differentiability. It is also not uncommon to calculate the root mean square error (RMSE) as:

$$\sqrt{\frac{1}{|S|}\sum_{x\in S} f\ x - h\ x^{\ 2}} \tag{54}$$

which is sometimes more convenient because the error is expressed in the same units defined for the target function.

Relative Absolute Error

For some applications a relative error measure for regression models is desired, which relates the difference between predicted values and true target function values to the spread of the latter. This is achieved by the relative absolute error (RAE), calculated as:

$$\frac{\sum_{x\in S} \left| f\ x - h\ x \right|}{\sum_{x\in S} \left| f\ x - m_f\ S \right|} \tag{55}$$

where $m_f(S)$ is the mean target function value in S.

Relative Square Error

The relative square error (RSE), a squared analog of the relative absolute error, is defined similarly as follows:

$$\frac{\sum_{x\in S} (f(x)-h(x))^2}{\sum_{x\in S} (f(x)-m_f(S))^2} \tag{56}$$

It relates the mean square error to the variance of the target function values in the data set used for the evaluation. It is also common to use the difference between 1 and this quantity, referred to as the coefficient of determination.

Percentage Split

The *percentage split* evaluation procedure, also referred to as *holdout* or *split sample*, is the most straightforward approach to model evaluation as it keeps the validation or test data separate from the training data. The idea is to randomly split the available data set into two subsets, one is used for model creation and the other used for model evaluation. It is common to use a 2:1 proportion for this partitioning.

Although computationally inexpensive, the percentage split procedure is not always a sufficiently reliable evaluation procedure. If too many instances are held out for evaluation, the reduced training set size is likely to degrade the quality of the model to be evaluated, resulting in a pessimistic evaluation *bias*. On the other hand, when the validation or test set becomes too small, the evaluation *variance* increases. In both cases the reliability of the evaluation suffers, and it may be difficult to balance the bias and variance tradeoff.

Despite these problems, the percentage split procedure remains the technique of choice whenever the size of the data set and the corresponding computational cost of model creation or prediction— whichever dominates for a particular algorithm—prevents the application of more refined procedures. Note that for sufficiently large data sets the harmful effects of bias and variance are less likely to be significant.

Cross-Validation

The k-fold cross-validation procedure makes it possible to reduce the evaluation bias while keeping the evaluation variance acceptably low. It randomly splits the available data set D into k disjoint subsets of roughly the same size $D_1, D_2, \ldots, D_k$, and then iterates over these subsets. On the i th iteration a model is built using $T_i = \cup_{j\neq i} D_j$ as the training set, and applied to generate predictions on D_i . Once all k iterations are completed, a predicted class label or target function value is generated for each instance in the data set, using the model built without this instance in the training set. The resulting vector of predictions can be compared to true class labels or target function values using one or more selected quality indicators.

A single iteration of k -fold cross-validation is equivalent to the percentage split procedure, with $\frac{k-1}{k}$ of data selected for training and $\frac{1}{k}$ of data selected for evaluation. For sufficiently large k this does not reduce the training set size to an extent that is be likely to impact model quality, since the validation set is small, which accounts for reduced bias. Still, in k iterations all available instances are used for model evaluation, which keeps the variance reasonably low unless k is set too high. In practice, k values of 10, 5, and 20, in that order of popularity, are most commonly used, which provide a good balance between bias, variance, and computational expense of model evaluation.

Cross-validation virtualizes the training and validation/test set, by permitting all available instances to be used both for model creation and model evaluation—although not simultaneously. As long as the cost of creating and applying multiple models is not prohibitive, it is the recommended evaluation procedure.

Available Functionality

The IBM Netezza In-Database Analytics package provides implementations of the quality indicators and evaluation procedures described in this section. The stored procedures to run to perform the various calculations are:

CERROR—misclassification error calculation

CONFUSION_MATRIX—confusion matrix generation

CMATRIX_STATS—derived indicators calculation for true positive rate, false positive rate, and precision

MSE—mean square error calculation

MAE—mean absolute error calculation

RAE—relative absolute error calculation

RSE—relative square error calculation

PERCENTAGE_SPLIT—the percentage split procedure for classification algorithms

CROSS_VALIDATION—k-fold cross-validation for classification algorithms

The quality indicators can be applied to tables containing predicted and true class labels or target function values. The evaluation procedures can be applied to arbitrary classification or regression algorithms available in IBM Netezza In-Database Analytics and permit passing arbitrary parameters to the selected modeling algorithms.

Examples

Examples for model quality indicators are demonstrated with the algorithms used to create classification and regression models. The examples shown in this section are limited to the evaluation procedures and are applied to evaluate decision tree models for the *CensusIncome* data set and regression tree models for the *WineQuality* data set. These procedures generate tables with model predictions on output, which can be used to calculate arbitrary quality measures. See the Decision Trees and Regression Trees sections for more detailed information about the modeling algorithms.

The classification and regression algorithm examples in this guide typically partition data into training and test sets, building models on the former and applying them to generate predictions on the latter with two separate stored procedure calls. The model building task can be automated using the **TRAIN_TEST** procedures. The procedure takes the training and test sets, as well as the algorithm name, and any additional parameters as input. It then generates the resulting model and test set predictions on output. The accuracy of these predictions is calculated and passed as the return value. The following call demonstrates how it can be used with decision trees and the *CensusIncome* data set:

```
CALL nza..TRAIN_TEST('modelType=dectree, traintable=nza..CensusIncome_train,
testtable=nza..CensusIncome_test, model=ci_tree1, id=id, target=income,
eval=gini, minsplit=1000, outtable=CensusIncome_income1');
```

The **eval** and **minsplit** arguments are passed to the **DECTREE** procedure, which is invoked by TRAIN_TEST. The test set predictions stored in the created output table can now be evaluated using any applicable quality measures, such as the misclassification error or confusion matrix-based indicators. This is demonstrated below:

```
CALL nza..CERROR('pred_table=CensusIncome_income0,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');
```

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income0, id=id, target=income,
matrixTable=ci_income0_cm');
```

```
CALL nza..CMATRIX_STATS('matrixTable=ci_income0_cm');
```

The same procedure can be applied to evaluate a regression model, as demonstrated below for regression trees and the *WineQuality* data set:

```
CALL nza..TRAIN_TEST('modelType=regtree, traintable=nza..WineQuality_train,
testtable=nza..WineQuality_test, model=wq_regtree1, id=id, target=quality,
minsplit=100, minimprove=0.05, outtable=WineQuality_quality1');
```

```
CALL nza..MSE('pred_table=WineQuality_quality0,
true_table=nza..WineQuality_test, pred_id=id, true_id=id,
pred_column=class, true_column=quality');
```

The test set accuracy, which is the return value of the **TRAIN_TEST** procedure, is not useful for regression, but the mean square error or another appropriate quality measure can be calculated externally, as in the example above.

To use random rather than predefined data partitioning in the training and test subsets, applying the percentage-split evaluation procedure, you can issue a similar call:

```
CALL nza..PERCENTAGE_SPLIT('intable=nza..CensusIncome, modelType=dectree,
fraction=0.7, model=ci_tree_ps, id=id, target=income, eval=gini,
minsplit=1000, outtable=CensusIncome_income_ps');
```

Here the **CensusIncome** table is internally split at random into a 70% training set and a 30% test set. The resulting model is stored under the name specified by the model argument, and the predictions on the internally selected test subset are stored in the table specified by the **outtable** argument. Their accuracy is passed as a return value.

This can be repeated for the *WineQuality* regression as follows:

```
CALL nza..PERCENTAGE_SPLIT(' intable=nza..WineQuality, modelType=regtree,
fraction=0.7, model=wq_regtree_ps, id=id, target=quality, minsplit=100,
minimprove=0.05, outtable=WineQuality_quality_ps');

CALL nza..MSE('pred_table=WineQuality_quality_ps,
true_table=nza..WineQuality, pred_id=id, true_id=id, pred_column=class,
true_column=quality');
```

Again, the test set accuracy returned by the procedure is not useful for regression, but any other more appropriate quality indicator can be calculated for the generated predictions table.

The cross-validation procedure can be used in a similar manner, as demonstrated by this example:

```
CALL nza..CROSS_VALIDATION('modelType=dectree, intable=nza..CensusIncome,
folds=5, model=ci_tree_cv, id=id, target=income, eval=gini, minsplit=1000,
outtable=CensusIncome_income_cv');
```

Again, the **eval** and **minsplit** arguments are passed to the **DECTREE** procedure, which is invoked by CROSS_VALIDATION. The complete *CensusIncome* data set is specified with the **intable** argument, to be internally split into 5 subsets (as specified by the **folds** argument). The procedure creates 5 models, each using 4/5 of the data for training and the remaining 1/5 for testing. Notice that, since for k-fold cross-validation each instance from the provided data set is used both as a training instance (k-1 times) and as a test instance (once), predictions are generated for the complete data set. They are stored in the output table and their accuracy is passed as the return value from the procedure. Additionally, a model is created using the full data set and stored under a name specified by the **model** argument. This is convenient for typical usage scenarios, where cross-validation is performed to estimate the quality of the full-data model that is subsequently deployed.

The predictions stored in the created output table can now be evaluated by comparing them to correct class labels for the full *CensusIncome* data set:

```
CALL nza..CERROR('pred_table=CensusIncome_income_cv,
true_table=nza..CensusIncome, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome,
resulttable=CensusIncome_income0, id=id, target=income,
matrixTable=ci_income0_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_cv_cm');
```

The final example shows how to perform 10-fold cross-validation for regression trees on the

WineQuality data set:

```
CALL nza..CROSS_VALIDATION('modelType=regtree,
intable=nza..WineQuality_train, folds=10, model=wq_regtree_cv,
id=id, target=quality, minsplit=100, minimprove=0.05,
outtable=WineQuality_quality_cv');
```

```
CALL nza..MSE('pred_table=WineQuality_quality1, true_table=nza..WineQuality,
pred_id=id, true_id=id, pred_column=class, true_column=quality');
```

CHAPTER 12

Random Sampling

Random sampling procedures are a vital component of many analytical systems. They can be used to select a test sample and a training sample for a model building process (machine learning). They can also be used to get a smaller sample of the training set, which you may do because of learning algorithm complexity considerations. In both cases, you would sample without replacement.

Another application of sampling is the learning methods based on bootstrapping. This requires many independent samples from the same data, which are preferentially applied if the available data sets are small or for other reasons where the sample independence is vital. Samples with replacement are usually drawn in this case.

In application, sampling is used for promotion campaigns, for example when you want only a representative set of customers to be subjects of an action.

In all cases, whether for use with scientific methods or business practices, uniform sampling is important.

The random sampling procedure described in this section creates a random sample of the rows of a table. Using it, you can obtain an exact number of rows in the sample or you can let the system sample each row with a fixed probability. Sampling can be performed with or without replacement.

Available Functionality

The **Random_Sample** stored procedure, covers the following functionality:

Creates random samples with a specified number of rows from a given table with replacement.

Creates random samples with a specified number of rows from a given table without replacement.

Creates stratified samples by indicating the column of which the values distribution is kept in the samples.

Allows you to specify the probability of each row to be in the sample.

Allows you to specify the input table columns to keep in the sample.

Allows you to specify the name of the output table (and to overwrite an existing table, if applicable).

Example

Consider four execution examples of the RANDOM_SAMPLE procedure on the *Adult* data set. The first example illustrates the creation of a 1000-row random sample, with columns id and income, and with replacement. Replacement means that some rows from the source data set can be duplicated)

```
CALL nza..RANDOM_SAMPLE('intable=nza..adult, size=1000, outtable=adult_size,

outsignature=id;income, outclear=true, replace=true, randseed=11213');
```

The second example shows the creation of a random sample of the same size and the same output columns, but without replacement (rows can not be duplicated).

```
CALL nza..RANDOM_SAMPLE('intable=nza..adult, num=1000, outtable=adult_num,

outsignature=id;income, outclear=true, replace=false, randseed=11213');
```

The third example shows the creation of a stratified sample of the same size and without replacement. The values distribution of the input column MARITAL_STATUS is kept in the sample.

```
CALL nza..RANDOM_SAMPLE('intable=nza..adult, num=1000, by=marital_status,
outtable=adult_strat, outsignature=id;income;marital_status, outclear=true,
replace=false,
randseed=11213');
```

The last example shows the creation of a random sample of a fixed faction equal to 0.3, without replacement. This means that (approximately) 30 percent of the source data set will be copied to the adult_frac table. Because the parameter outsignature is missing, all columns from the source data set will be copied to the output table.

```
CALL nza..RANDOM_SAMPLE('intable=nza..adult, fraction=0.3,

outtable=adult_frac, outclear=true, replace=false, randseed=11213');
```

CHAPTER 13

Naive Bayes

The naive Bayes classifier is a simpler classification algorithm than most, which makes it quick and easy to apply. While it does not compete with more sophisticated algorithms with respect to classification accuracy, in some cases it may be able to deliver similar results in a fraction of the computation time.

Background

The naive Bayes classifier algorithm uses the Bayes theorem to calculate the conditional class probability of the given attribute values, called the *posterior class probability*:

$$\begin{aligned} &P\;c\;x = d \mid a_1\;\;x = v_1\;,\;a_2 x = v_2\;, \ldots ,\;a_n\;\;x = v_n \\ &= \frac{P\;c\;x = d\;\cdot P\;a_1\;\;x = v_1\;, a_2\;\;x = v\,_2\;, \ldots ,\;a_n\;\;x = v_n = v_n \mid c\;x = d}{P\;a_1\;\;x = v_1\;, a_2\;\;x = v_2\;, \ldots ,\;a_n\;\;x = v_n = v_n} \end{aligned} \tag{57}$$

The numerator and denominator is referred to as the Bayes numerator and denominator, respectively. The calculation is based on the *prior class probabilities* $P\;c\;x\;\;= d$, which can be directly estimated from the data, and on the *joint inverse conditional attribute values probability* given the class:

$$P\;a_1\;\;x = v_1\;, a_2\;\;x = v_2\;, \ldots ,a_n\;\;x = v\;_n = v_n \mid c\;x = d \tag{58}$$

The joint inverse conditional attribute can be efficiently calculated as the product of per-attribute *conditional attribute value probabilities* $P\;a_i\;x = v\;_i \mid c\;x = d$, directly estimated from the data, assuming the conditional independence of attributes given the class. This assumption may be untrue in practice, which is why the algorithm is called “naive”. While violations of the independence assumption result in probability calculations yielding incorrect results, they may not directly affect the accuracy of predictions, which may be correct even when using incorrect probabilities. The naive Bayes classifier has been found to predict well in several domains where the independence assumption is not satisfied.

A result of this approach is that whenever the estimated probability of one attribute value within a class is 0, the posterior probability of this class is also calculated as 0. If this happens for all classes, no prediction is possible. One way to prevent this problem is to always replace zero probabilities with sufficiently small positive numbers. If there is no instance with the value of attribute a_i equal v_i in class d , the probability $P\ a_i\ x = v_i | c\ x = d$ can be set to half of the probability estimated when there is one such instance. A more refined approach is to use a modified probability estimation technique, known as the m -estimation, where the per-class attribute value frequencies observed for training instances are augmented by the *a priori* assumed frequencies for a small number of “hypothetical” instances. Without specific domain knowledge, these prior frequencies are assumed to be equal for all attribute values within particular classes, and the number of included hypothetical instances is usually equal to the number of possible attribute values.

Missing Value Support

Since Naive Bayes uses values of each attribute only for conditional probability calculations, assuming the conditional independence of attributes given the class, it handles missing values during model creation and prediction by ignoring them on a per-instance and per-attribute basis. More precisely, when estimating a conditional attribute-value given the class probabilities of the following form $P\ a_i\ x = v_i | c\ x = d$ for each attribute a_i , value v_i , and class d , only instances with non-missing values of a_i are taken into account. Similarly, when the product of such probabilities needs to be calculated during prediction for an instance x_* :

$$\prod_{i=1}^{n} P(a_i\ (x\ ) = a_i\ (\ x_*\ ) | c(\ x) = d\ ) \tag{59}$$

factors corresponding to missing $a_i(\ x_*\ )$ are skipped.

For data sets with no missing values the behavior remains unchanged. Data sets with missing values can be used for both model creation and prediction.

Applications

While classification accuracy is not the strongest point of the naive Bayes classifier, its other advantages make it well-suited to domains where classification models must be created quickly, without a significant computational effort, without the need for parameter tuning, and without any overfitting-prevention overhead. Similarly, it is well-suited to applications that require frequently creating numerous classification models, either because they are quickly outdated or because new classification tasks appear dynamically. There are some areas, however, where the naive Bayes classifier may compete with much more refined algorithms not only in efficiency, but also in accuracy. This may be the case for classification tasks with a large number of attributes, each of which may have some marginal impact on the class, and where there are no strong relationship patterns permitting prediction based on the values of a small number of the most influential attributes. One such area where the naive Bayes classifier is among the most successful algorithms is text classification.

Available Functionality

The Netezza implementation of the naive Bayes classifier, exposed as the **NAIVEBAYES** stored procedure, covers the following functionality:

- naive Bayes model building by estimating probabilities
- support for discrete and continuous attributes
- all continuous attributes are automatically discretized using one of three algorithms:
    - equal-width discretization—(the default) an unsupervised discretization algorithm using the equal width criterion for interval bound setting (disc=ew)
    - equal-frequency discretization—an unsupervised discretization algorithm using the equal frequency, that is, equal data count, criterion for interval bound setting (disc=ef)
    - minimum-entropy discretization—a supervised discretization algorithm that identifies the most appropriate interval bounds by minimizing class distribution impurity (disc=em)
- support column properties definition mechanism
- two methods of conditional attribute-value probability estimation
    - ordinary frequency-based estimation with replacing zero probabilities with small numbers corresponding to ½ of an instance
    - m -estimation with uniform priors and m set to the number of attribute values for each attribute
- predicting most likely class labels and class probabilities for all classes

Examples

Consider the application of the naive Bayes classifier to the *CensusIncome* data set. Input data can contain a mix of continuous and nominal attributes types. All types can be either detected automatically or user-specified. To define types manually, use the incolumn and coldeftype parameters (see Column Properties for more information). The naive Bayes classifier also allows you to define a discretization algorithm, as well as the number of bins:

```
CALL nza..NAIVEBAYES('intable=CensusIncome_train, disc=ew, bins=20, id=id,
target=income, model=ci_nb_ewd');

CALL nza..NAIVEBAYES('intable=CensusIncome_train, disc=ef, bins=20, id=id,
target=income, model=ci_nb_efd');

CALL nza..NAIVEBAYES('intable=CensusIncome_train, disc=em, id=id,
target=income, model=ci_nb_emd');
```

The created model is represented by a table containing one row for each attribute-value-class combination and the following columns:

attribute—the attribute

val—the value

class—the class

classvalcount—the number of occurrences of the corresponding value of the corresponding attribute within the given class in the training set

classcount—the number of occurrences of the given class in the training set

attrclasscount—the number of not null occurrences of the given class in the training set

totalcount—the number of all instances in the training set

These are sufficient for calculating prior class probabilities and conditional attribute value probabilities used for naive Bayes prediction.

The three resulting models can be used to generate test set predictions as follows:

```
CALL nza..PREDICT_NAIVEBAYES('model=ci_nb_ewd,
intable=CensusIncome_test, outtable=CensusIncome_income_nb_ewd,
outtableprob = CensusIncome_income_nb_ewd_prob');
```

```
CALL nza..PREDICT_NAIVEBAYES('model=ci_nb_efd,
intable=CensusIncome_test, outtable=CensusIncome_income_nb_efd,
outtableprob = CensusIncome_income_nb_efd_prob');
```

```
CALL nza..PREDICT_NAIVEBAYES('model=ci_nb_emd, intable=CensusIncome_test,
outtable=CensusIncome_income_nb_emd,outtableprob =
CensusIncome_income_nb_emd_prob');
```

Because the Netezza Analytics ID for the Predict procedure is optional, it is not necessary to define it if its name in test data is the same as the name in training data.

Recall from the Discretization section that the test sets used for each model application have been discretized using the same intervals that were determined on the corresponding training sets. The specified output table contains the generated predictions and has two columns, **id** and **class**. Another output table is also created, with the **_prob** suffix appended to the name, containing one row for each instance and possible class, and the following columns:

id—the instance identifier

class—the class

prob—the Bayesian numerator of the class for the instance, which is the product of the class probability and the conditional instance attribute value probabilities

lnprob—the natural logarithm of **prob**

The probabilities can be used to provide better insight into model predictions or modify model operation. The Bayesian numerators can be converted to regular probabilities by normalization: dividing by the sum of numerators for the same instance and all classes. This is demonstrated by the following query that adds the class probability to each instance and its predicted class label for the third model, obtained using the minimum-entropy discretization:

```
SELECT S.id, S.class, S.prob/S.sump as prob
FROM (SELECT id, class, prob, sum(prob) OVER (PARTITION BY id) AS sump FROM
CensusIncome_income_nb_emd_prob) S, CensusIncome_income_nb_emd P
WHERE S.id=P.id AND S.class=P.class;
```

The quality of the obtained predictions can be evaluated by calculating the misclassification error as follows:

```
CALL nza..CERROR('pred_table=CensusIncome_income_nb_ewd,
true_table=CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CERROR('pred_table=CensusIncome_income_nb_efd,
true_table=CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CERROR('pred_table=CensusIncome_income_nb_emd,
true_table=CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');
```

The three models perform similarly; however, the first, which is based on equal-frequency discretization, is slightly worse than the other two. With the misclassification error levels above 0.23 all three are considerably inferior to the models obtained in the examples presented in the Decision Trees section. The naive Bayes classifier does not appear to be particularly well-suited to the *CensusIncome* data set.

To enable the *m* -estimation technique for avoiding 0 probabilities during prediction, you can specify the **mestimation=TRUE** argument for the **PREDICT_NAIVEBAYES** procedure, as demonstrated below:

```
CALL nza..PREDICT_NAIVEBAYES('model=ci_nb_emd, intable=CensusIncome_test,
mestimation=TRUE, outtable=CensusIncome_income_nb_emd_mest');

CALL nza..CERROR('pred_table=CensusIncome_income_nb_emd_mest,
true_table=CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');
```

After analysis, it can be determines that the technique has little impact on the quality of predictions generated for the *CensusIncome* data.

More in-depth prediction quality analysis can be performed using the confusion matrix and the derived quality indicators, as demonstrated previously in the examples presented in the Decision Trees section. For the third model, obtained using the minimum-entropy discretization, proceed as follows:

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income_nb_emd, id=id, target=income,
matrixTable=ci_income_nb_emd_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_nb_emd_cm');
```

The result sheds new light on the quality of naive Bayes predictions for the *CensusIncome* data. The large misclassification error translates to a high false positive rate of nearly 0.25, but also a surprisingly high true positive rate of more than 0.9, with the high-income class considered positive. While the false positive rate is poor, the true positive rate is significantly better than observed in the Decision Trees section.

With Netezza Analytics 2.0 and later, all columns can be defined by types and roles.

```
CALL nza..NAIVEBAYES('model=NB_iris, intable=nza..iris, id=id,
target=class');
```

is equal to:

```
CALL nza..NAIVEBAYES('model=NB_iris, intable=nza..iris, incolumn= id:id;
class:target');
```

To define nominal and continuous attributes manually, you can use the nom and cont types:

```
CALL nza..NAIVEBAYES('model=NB_iris, intable=nza..iris,
incolumn=SEPALLENGTH:nom;SEPALWIDTH:cont;PETALLENGTH:cont;PETALWIDTH:cont;cla
ss:target,id=id');
```

It is also possible to create one-column properties definitions and pass them to the model multiple times:

```
call nza..COLUMN_PROPERTIES('intable=nza..iris, outtable=iris_columns,
coldeftype=cont,
incolumn=PETALWIDTH:nom;PETALLENGTH:cont:ignore;class:nom:target;id:id');

select * from iris_columns;

CALL nza..NAIVEBAYES('model=NB_iris, intable=nza..iris,
colPropertiesTable=iris_columns');
```

CHAPTER 14

Decision Trees

Background

In many classification applications it may be required or desirable not only to accurately classify instances, but also to inspect the model. The inspection makes it possible to explain its decisions, modify it, or combine with some existing background knowledge. In such applications, where both the high classification accuracy and human-readability of the model are required, the method of choice is typically going to be decision trees.

A decision tree is a hierarchical structure that represents a classification model using a “divide and conquer” approach. Internal tree nodes represent splits applied to decompose the data set into subsets, and terminal nodes, also referred to as leaves, assign class labels to sufficiently small or uniform subsets. Splits are specified by logical conditions based on selected single attributes, with a separate outgoing branch corresponding to each possible outcome.

The concept of decision tree construction is to select splits that decrease the impurity of class distribution in the resulting subsets of instances, and increase the domination of one or more classes over the others. The goal is to find a subset containing only or mostly instances of one class after a small number of splits, so that a leaf with that class label is created. This approach promotes simple trees, which typically generalize better.

Creating and using decision tree models involves three major algorithmic sub-tasks:

decision tree growing

decision tree pruning

decision tree prediction

Growing

Decision tree growing consists of creating a decision tree from a given data set by appropriately selecting splits and assigning class labels to leaves when no further splits are required or possible.

This is performed in a top-down fashion, starting from a single root node. A table representing a training data set is used as input. This table contains a number of columns representing attributes as well as a single column designated as the class attribute. The expected output is an appropriate representation of a decision tree built based on the provided training set.

Decision tree growing starts from a single node, corresponding to the complete training set. When a split is applied, each of the created descendant nodes corresponds to the appropriate subset of the training set, determined by the split outcome. Further splits can be applied to these nodes, resulting in new nodes corresponding to smaller subsets of instances, and so on. Nodes without further splits remain leaves.

The decision tree growing process can be viewed as the repeated application of the following key operations:

stop criteria

class label assignment

split selection

The stop criteria determine whether a split is applied in a node, or if it remains a leaf. The decision is based on the subset of training instances corresponding to the node. No split is applied when:

all instances in the corresponding subset are of the same class

the number of instances in the corresponding subset is less than a specified minimum

the level of the current node is greater than a specified maximum, with the level of the root node being 1, the level of its descendants being 2, and so on

the improvement of class impurity due to the best available split is less than a specified minimum

Class label assignment, although strictly necessary only for leaves, takes place for all nodes. The majority class label in the corresponding set of training instances is always assigned. The class label assignment is useful if the tree is subsequently pruned, which turns some nodes into leaves, and facilitates the human inspection of the tree structure.

Split selection assigns minimum-impurity splits to nodes for which the stop criteria were not satisfied. The set of candidate splits includes binary equality-based splits for all discrete attributes and binary inequality-based splits for all continuous attributes. An impurity measure is applied to subsets corresponding to split outcomes to evaluate candidate splits. One commonly used impurity measure is the entropy, defined for node $\boldsymbol{n}$ as follows:

$$E(\boldsymbol{n}) = \sum_{d \in C} -P(d|\boldsymbol{n})\log_2 P(d|\boldsymbol{n}) \tag{60}$$

where d iterates over classes, and $P(d|\boldsymbol{n}) = \frac{|T^d_{n}|}{|T_{\boldsymbol{n}}|}$ is the probability of class d based on $T_{\boldsymbol{n}}$, the subset of training instances corresponding to node $\boldsymbol{n}$, designated by $T_{\boldsymbol{n}}$, and the class in the superscript denotes selecting instances of this class only. Another popular impurity measure is the Gini index, calculated as:

$$\boxed{G(\boldsymbol{n})=1-\sum_{d\in C}P(d\,|\,\boldsymbol{n})^2} \qquad (61)$$

If a split is considered for node $\boldsymbol{n}$ that yields two descendant nodes, $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$, the selected impurity measure is applied to these descendant nodes and then the weighted average of their impurities is calculated to achieve the evaluation of the split, with the corresponding instance counts used as weights. Using the entropy this can be written as follows:

$$E(\boldsymbol{n}_1,\boldsymbol{n}_2) = E(\boldsymbol{n}_1)\frac{|T_{\boldsymbol{n}_1}|}{|T_{\boldsymbol{n}}|} + E(\boldsymbol{n}_2)\frac{|T_{\boldsymbol{n}_2}|}{|T_{\boldsymbol{n}}|} \qquad (62)$$

The split that yields minimum impurity is selected.

In all phases of the tree growing process it is possible to use instance or class weights to make the resulting decision tree model more sensitive to some instances or classes. Specifying a vector of numerical weights w_x for each instance x instructs the algorithm to weight the training instances accordingly when creating the decision tree. Specifying per-class rather than per-instance weights is a convenient way to assign the same weight to all instances of the same class. This does not actually add any complexity to the algorithm. Note that all operations described above use training data only to count instances in particular nodes satisfying some conditions. The effect of using weights is replacing these counts by the corresponding sums of weights. For integer weights, this is equivalent to replicating instance x from the training set w_x times.

The most important application of class weights is to make the decision tree growing algorithm cost-sensitive by assigning each class $d\in C$ a weight $w_d=\rho[d]$, where $[d]$ is the misclassification cost of predicting any class $d'\neq d$ for instances of true class d.

Pruning

The goal of decision tree pruning is to reduce the risk of overfitting by removing overgrown subtrees that do not improve the expected accuracy on new data. While growing such subtrees may be prevented by appropriately selected more aggressive stop criteria, a separate pruning process is typically a more reliable approach. Decision tree pruning receives a decision tree on input and is intended to produce a pruned version, ideally with less risk of overfitting.

There are two major categories of pruning algorithms:

using a separate pruning data set to estimate the expected accuracy on new data

using the training set for this estimate

A separate pruning data set is considered more reliable, although it has the disadvantage of removing a large subset of data from the training set, which in turn may reduce the quality of the grown tree. The result is a good pruning method applied to a mediocre tree. Using the training set has the advantage of using the data economically, but must resort to heuristics to work around the inability to estimate the true new data performance.

Reduced error pruning (REP) is the standard algorithm of the first category. It iterates over the decision tree nodes in a bottom-up order and considers replacing each node and attached subtree with a single leaf, with the majority training set class. If the accuracy of the replacement leaf on the pruning set is more than that of the original subtree, the pruning operation is applied. The accuracy estimation is based on the instances from the pruning set that have reached the point where the node under consideration is located when propagated down the tree. The weighted accuracy may be optionally used instead of the ordinary accuracy to better handle unbalanced classes, when high accuracy values do not necessarily imply high predictive utility. It is calculated by averaging per-class accuracies.

As with growing, instance or class weights can be taken into account in the pruning process, which affects only the calculation of the accuracy of nodes and leaves. .

Prediction

Decision tree prediction consists of using a previously grown, and possibly pruned, decision tree to generate class predictions for a data set. It is performed by applying the splits from the tree nodes to propagate instances from the data set down to the corresponding leaves. Decision tree prediction takes a decision tree and a data set on input and uses the tree to predict class labels for the data set. It can be also used to predict class probabilities based on the class distribution of training instances in leaves.

Missing Value Support

IBM Netezza In-Database Analytics supports the creation of decision trees for data sets that contain missing values. Many real-world data sets suffer from missing attribute value, yet decision tree growing is severely affected by missing values. In particular, the following operations cannot be performed in the usual manner if the values of one or more attributes are missing for some instances:

class distribution calculation (class counts for each node-attribute-value)

split evaluation (class impurity for each node-attribute-value)

split application (splitting data based on equality- or inequality-based conditions)

which rely on the availability of attribute values. The fractional instance technique, applied to overcome the problem, splits an instance with a missing value of some attribute used for a split into appropriately weighted fractional instances, assuming that each split outcome is possible with a corresponding probability.

The basic idea of the fractional instance technique is to virtually replace an instance by *fractional instances* whenever a split is to be applied on an unknown attribute value. Each fractional instance corresponds to one possible split outcome. In the Netezza Analytics implementation, instances with an unknown attribute value are replaced by two fractional instances, corresponding to the left and right outgoing tree branches. Each instance is assigned its copy count, initialized either to 1 or its weight, if specified. Then, if the instance is fractionated it is multiplied by the corresponding split outcome probabilities, determined based on those instances for which the values of the split attribute are known.

This technique is applied during all phases of decision tree processing: growing, pruning, and prediction. Whenever the number of instances in a subset is needed for stop criteria or split selection calculation during growing, the sum of the corresponding weights (copy counts) is used instead of ordinary subset counts. Similarly instance weights are taken into account for pruning criteria calculation. During prediction, whenever multiple fractions of the same instance arrive at multiple leaves, the predicted class label is determined by probability-based voting.

Missing value handling via the fractional instance technique increases the computational complexity of decision tree creation and application, but keeps the impact of missing values on model and prediction quality as low as possible. If the additional computational expense is undesirable, missing values may be imputed, as discussed in the Data Imputation section.

Applications

Decision trees are considered among the most accurate classifiers. Unlike some algorithms, they can handle both discrete and numerical attributes by using them in symbolically-represented split conditions. The human readability of decision tree models is an additional advantage that makes them particularly well-suited to applications that require both high accuracy and interpretability of the classification model. The latter makes it possible for a human expert to verify the model or to combine it with some background knowledge. Application areas where this is likely to be needed include customer classification, fraud detection, and diagnostics.

Available Functionality

The Netezza implementation of decision trees provided by the **DECTREE**, **GROW_DECTREE**, **PRUNE_DECTREE**, **PREDICT_DECTREE**, and **PRINT_MODEL** stored procedures covers the following functionality:

- top-down decision tree growing
- support for discrete and continuous attributes
- support for instance or class weights
- binary equality-based splits for discrete attributes (attribute = value)
- binary inequality-based splits for continuous attributes (attribute ≤ value)
- class labels and probabilities assigned to leaves and internal nodes
- split selection based on class impurity measures such as entropy or Gini index, with more definable via UDAs
- several stop criteria: uniform class
  - no candidate splits left
  - not enough instances, that is, less than a specified minimum required for a split
  - not enough improvement of class impurity (less than a specified minimum required for a split)
  - reaching the maximum allowed tree depth

reduced error pruning using accuracy or weighted accuracy pruning criteria

class label and probability prediction

decision tree structure printing

If the input table contains NULL attribute values (i.e., instances with missing attribute values), the fractional instance technique is automatically applied to handle them. The additional computational expense depends on the particular data set and the number of missing values, but typically there is a 5-30% time increase compared to the same data sets with missing values removed or imputed. If this computational expense is undesirable, instances with missing values should be removed or missing values imputed. The Data Imputation section describes and demonstrates how the latter can be accomplished.

The number of nodes that must be created, which is controlled by the stop criteria, determines the computational effort required to grow (and subsequently prune or apply) the decision tree. Using appropriate parameter settings for the minimum number of instances required for a split, the minimum impurity improvement required for a split, or the maximum tree depth may save computation time. Default settings result in moderately complex trees and should be usually reasonable to start with, but may need adjusting for particular data sets.

Examples

Consider creating a decision tree model for the *CensusIncome* data set. The following call requests growing a tree based on the CensusIncome_train table, using the id column as a unique instance identifier, and the income column as the class:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train, id=id,
target=income,model=ci_tree1, eval=gini, minimprove=0.005, minsplit=1000');
```

The resulting decision tree is stored as the **ci_tree1** model. The Gini index is used as a class impurity measure for split evaluation, and no further splits are attempted when there are less than 1000 instances left or if the maximum available split does not improve the impurity by at least 0.005.

Although the tree was grown with stop criteria settings of **minsplit=1000** and **minimprove=0.005**, which may partially prevent overfitting, it is still likely to benefit from pruning. This is performed by the following code, using the **CensusIncome_val** table as the pruning set:

```
CALL nza..PRUNE_DECTREE('model=ci_tree1, valtable=nza..CensusIncome_val');
```

If the **id** and **target** arguments are not specified in the call of the pruning procedure, as above, they will default to those previously specified for tree growing. The pruning operation modifies the specified decision tree in-place. The modified tree can be inspected using the same SQL call. Note the substantial reduction in tree size, from several hundred nodes to just a few dozen. The tree can be inspected as demonstrated before, by looking at the contents of the **ci_tree1** table, but a more readable printout is produced by calling the **PRINT_DECTREE** function:

```
CALL nza..PRINT_MODEL('model=ci_tree1');
```

The resulting output is presented below:

```
 -- decision tree model: "CI_TREE1" --
CAPITAL_GAINS <= 7298
| DIVIDENDS_FROM_STOCKS <= 0
 | if true then class -> - 50000.
 | WEEKS_WORKED_IN_YEAR <= 44
 | | if true then class -> - 50000.
 | | SEX = Female
 | | | DIVIDENDS_FROM_STOCKS <= 3750
 | | | | EDUCATION = Masters degree(MA MS MEng MEd MSW MBA)
 | | | | | if true then class -> - 50000.
 | | | | | EDUCATION = Doctorate degree(PhD EdD)
 | | | | | | if true then class -> 50000+.
 | | | | | | EDUCATION = Prof school degree (MD DDS DVM LLB JD)
 | | | | | | | if true then class -> 50000+.
 | | | | | | | if false then class -> - 50000.
 | | | | if false then class -> - 50000.
 | | | CAPITAL_LOSSES <= 1887
 | | | | DIVIDENDS_FROM_STOCKS <= 359
 | | | | | if true then class -> - 50000.
 | | | | | MAJOR_OCCUPATION_CODE = Professional specialty
 | | | | | | if true then class -> 50000+.
 | | | | | | MAJOR_OCCUPATION_CODE = Executive admin and managerial
 | | | | | | | if true then class -> 50000+.
 | | | | | | | if false then class -> - 50000.
 | | | | if false then class -> 50000+.
 CAPITAL_GAINS <= 9562
 | if true then class -> - 50000.
 | if false then class -> 50000+.
```

Each line in this textual tree representation corresponds to a node or a leaf, and the indentation reflects the tree level. For a node, the split condition is printed; for each leaf, the assigned class label is printed. Each node is followed by its left and right subtrees. For example, according to the printout presented above, the root node splits on the **capital_gains** attribute by comparing it with a threshold value of 7298. The left branch (**capital_gains ≤ 7298**) leads to a descendant node that splits on the **dividends_from_stocks** attribute, and the right branch (**capital_gains > 7298**) to a node that uses the **capital_gains** attribute again, comparing it with 9562. This node has two descendant leaves, labeled with low (**- 50000.**) and high (**50000+.**) income class labels..

For convenience, the growing and pruning phases can be performed by a single call:

```
CALL nza..DECTREE('intable=nza..CensusIncome_train,
valtable=nza..CensusIncome_val, id=id, target=income,

model=ci_tree2, eval=gini, minimprove=0.005, minsplit=1000');
```

This is equivalent to the separate grow and prune calls presented above. If the **DECTREE** procedure is called without specifying the pruning set via the **valtable** argument, no pruning is performed.

The accuracy quality measure is used for the pruning criterion by default, but the alternative weighted accuracy measure may be requested by specifying the **qmeasure=wacc** argument for **PRUNE_DECTREE** or **DECTREE**, as demonstrated below:

```
CALL nza..DECTREE('intable=nza..CensusIncome_train,
valtable=nza..CensusIncome_val, id=id, target=income, model=ci_tree3,

eval=gini, minimprove=0.005, minsplit=1000, qmeasure=wacc');
```

It may be actually more appropriate for unbalanced classes, being equally sensitive to the accuracy obtained for all classes.

The next two calls demonstrate the effect of other stop criteria that can be specified for decision tree growing:

```
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train, id=id,
target=income, model=ci_tree4, eval=gini, minimprove=0.0, maxdepth=5');
CALL nza..GROW_DECTREE('intable=nza..CensusIncome_train, id=id,
target=income, model=ci_tree5, eval=gini, minimprove=0.01, minsplit=1000');
```

The first limits the tree depth to 5 (the default maximum depth is 62) while setting the impurity improvement needed for split to 0, and the other requests that a split must improve the class impurity by at least 0.01 for at least 1000 instances to be accepted. The trees grown by these calls are considerably smaller than those created before.

To evaluate the performance of the created decision tree on the *CensusIncome* test set, proceed as follows:

```
CALL nza..PREDICT_DECTREE('model=ci_tree1, intable=nza..CensusIncome_test,
outtable=CensusIncome_income1, prob=TRUE');
```

This applies the **ci_tree1** decision tree to generate predictions on the test set, which are stored in the **CensusIncome_income1** table. Both predicted class labels and their probabilities are generated. The column used as a unique instance identifier does not need to be passed using the **id** argument if it is the same as previously during tree creation, but otherwise it may be explicitly specified.

You can then calculate the misclassification error achieved by the tree on the test set:

```
CALL nza..CERROR('pred_table=CensusIncome_income1,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');
```

The obtained error level may appear low, but subsequent inspection via the confusion matrix, using the following call to the **CONFUSION_MATRIX** procedure:

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income1, id=id, target=income,
matrixTable=ci_income1_cm');

SELECT * FROM ci_income1_cm ORDER BY 1, 2;
```

This reveals that the tree’s performance may not be be very good. While it is successful at detecting the low-income class, it usually fails to correctly predict the high-income class. This may not be surprising given the sparser representation of the high income in the data. If you consider the high-income class as positive and the low-income class as negative, you can describe the performance as yielding a satisfactorily low false positive rate with a very low true positive rate. These indicators can be calculated as follows, with the **class** argument used to specify which class is considered positive:

```
CALL nza..FPR('matrixTable=ci_income1_cm, class=50000+.');
```

Alternatively, a more extensive and comprehensible description of the model’s performance based on the confusion matrix can be printed using the **CMATRIX_STATS** procedure:

```
CALL nza..CMATRIX_STATS('matrixTable=ci_income1_cm');
```

The output includes, in particular, the true positive rate and the false positive rate with each class considered positive.

Similarly, examining the performance of the **ci_tree3** model created with weighted accuracy-based pruning, expected to handle unbalanced classes, better:

```
CALL nza..PREDICT_DECTREE('model=ci_tree3, intable=nza..CensusIncome_test,
outtable=CensusIncome_income3, prob=TRUE');
```

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income3, id=id, target=income,
matrixTable=ci_income3_cm');
```

```
CALL nza..CMATRIX_STATS('matrixTable=ci_income3_cm');
```

we may find some increase of the true positive rate for the high-income class.

Another way to look for improvement is to exploit the probabilistic prediction capability of decision trees. When dealing with a two-class task, as in this example, you can predict class labels by explicitly comparing estimated class probabilities against a cutoff value that is different from the default implicit 0.5 cutoff corresponding to predicting the most probable class. This possibility is demonstrated below using a cutoff value of 0.8 for the low-income class, which favors the high-income class.

```
CREATE VIEW CensusIncome_income3p AS SELECT id, CASE WHEN class='- 50000.'
AND prob>=0.8 THEN '- 50000.' ELSE '50000+. ' END AS class FROM
CensusIncome_income3;
```

The modified predictions can be evaluated using the confusion matrix:

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income3p, id=id, target=income,
matrixTable=ci_income3p_cm');
```

```
CALL nza..CMATRIX_STATS('matrixTable=ci_income3p_cm');
```

This evaluation reveals a further increase of the true positive rate, at the cost of some increase of the false positive rate. These are more useful predictions, despite a slightly greater overall error. While the default maximum-probability classification minimizes the misclassification error, by shifting the probability cutoff value you effectively incorporate misclassification costs into the prediction process and minimize the mean misclassification cost. The 0.8 cutoff for the low income class corresponds to a 4:1 misclassification cost matrix—in which failing to detect the high-income class is 4 times more costly than failing to detect the low-income class: $0.8=4 /\ 4\ 5$.

Another approach to incorporating misclassification costs in decision tree classification is to appropriately weight training instances during tree growing (and pruning instances during tree pruning, if the latter is performed). This allows instances of the more important/harder to predict/less frequent class, for which making mistakes is more costly, to receive more weight. An appropriate class weight table, representing the same 4:1 cost matrix assumed above, can be created and used for creating a decision tree model:

```
CREATE TABLE CensusIncome_weights AS SELECT DISTINCT income as class, CASE
WHEN income='50000+.' THEN 4 ELSE 1 END AS weight FROM
nza..CensusIncome_train;

CALL nza..DECTREE('intable=nza..CensusIncome_train,
valtable=nza..CensusIncome_val, id=id, target=income, model=ci_tree1w,
weights=CensusIncome_weights, valweights=CensusIncome_weights,
eval=gini, minimprove=0.005, minsplit=1000');
```

The **weights** argument is used to specify the table containing class or instance weights for tree growing, and the **valweights** argument specifies the weights table for pruning. For class weights these are usually the same, but for instance weights they would be different if the training and pruning sets were different (as they normally should). Class weights are specified using a table that includes the **class** and **weight** columns, as demonstrated above, with the values of the former matching the possible classes. To specify instance weights, a table including the **id** and **weight** columns are required, with the values of the former matching instance identifiers from the training or pruning set. This could be useful to make the model more sensitive to some particular instances, but class weights are sufficient and more convenient to incorporate per-class misclassification costs.

The resulting decision tree – likely to be larger, as more splits yielding a sufficient impurity improvemement may be found with more weight put to the less frequent class – can be evaluated in the same way as before:

```
CALL nza..PREDICT_DECTREE('model=ci_tree1w, intable=nza..CensusIncome_test,
outtable=CensusIncome_income1w, prob=TRUE');

CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income1w, id=id, target=income,
matrixTable=ci_income1w_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income1w_cm');
```

and observed to exhibit indeed a substantially higher improvement of the true positive rate than obtained previously by altering the prediction probability cutoff. In this case, the specified misclassification costs were incorporated directly into the tree structure by the growing and pruning process and not only into the prediction process, as with the probabilistic classification using the minimum-cost rule.

The decision tree algorithm can be applied to data sets with missing attribute values either directly, using the internal missing value handling capability, or after processing them with the data imputation algorithm, as described in the Data Imputation section. There, corrupted versions of the *CensusIncome* training, validation, and test set were created and then repaired by imputation. The two attributes used for that demonstration, **capital_gains** and **sex**, are used by the trees generated in this section and appear highly in the tree structure (the **capital_gains** attribute is used for the root node split), as seen in the printout of **ci_tree1** presented above. The following SQL code recreates and evaluates the tree using the corrupted version of the training, pruning, and test data sets after imputation:

```
CALL nza..DECTREE('intable=CensusIncome_train_imp,
valtable=CensusIncome_val_imp, id=id, target=income, model=ci_tree1imp,
eval=gini, minimprove=0.005, minsplit=1000');

CALL nza..PREDICT_DECTREE('model=ci_tree1imp, intable=CensusIncome_test_imp,

outtable=CensusIncome_income1imp, prob=TRUE');

CALL nza..CERROR('pred_table=CensusIncome_income1imp,
true_table=CensusIncome_test_imp, pred_id=id, true_id=id,

pred_column=class, true_column=income');

CALL nza..CONFUSION_MATRIX('intable=CensusIncome_test_imp,
resulttable=CensusIncome_income1imp, id=id, target=income,
matrixTable=ci_income1imp_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income1imp_cm');
```

The following sequence of calls repeats the same process directly using the corrupted data sets without imputation:

```
CALL nza..DECTREE('intable=CensusIncome_train_miss,
valtable=CensusIncome_val_miss, id=id, target=income,

model=ci_tree1miss, eval=gini, minimprove=0.005, minsplit=1000');

CALL nza..PREDICT_DECTREE('model=ci_tree1miss,
intable=CensusIncome_test_miss,
outtable=CensusIncome_income1miss, prob=TRUE');

CALL nza..CERROR('pred_table=CensusIncome_income1miss,
true_table=CensusIncome_test_miss, pred_id=id, true_id=id, pred_column=class,

true_column=income');

CALL    nza..CONFUSION_MATRIX('intable=CensusIncome_test_miss,
resulttable=CensusIncome_income1miss,   id=id,   target=income,
matrixTable=ci_income1miss_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income1miss_cm');
```

The quality of predictions clearly depends on the particular random distribution of missing values, which were generated at random, but in general internal missing value handling takes somewhat more time, but is likely to deliver better results in some situations.

Output Table Data Formats

The following tables are generated when you build decision trees:

NZA_META_<model_name>_MODEL

NZA_META_<model_name>_NODES

NZA_META_<model_name>_PREDICATES

NZA_META_<model_name>_COLUMNS

NZA_META_<model_name>_COLUMN_STATISTICS

NZA_META_<model_name>_DISCRETE_STATISTICS

NZA_META_<model_name>_NUMERIC_STATISTICS

NZA_META_<model_name>_MODEL

The NZA_META_<model_name>_MODEL table contains information about the entire tree model. The table contains only one line and has the following layout:

Table 11: Columns of the NZA_META_<model name>_MODEL table

Column	Data Type	Purpose
MODELCLASS	VARCHAR(32)	Type of tree model, always classification.
MAXSPLIT	SMALLINT	Maximal number of splits from a node.
DEPTH	SMALLINT	Depth of the tree.
MISSINGVALUE-STRATEGY	VARCHAR(32)	Strategy for handling the case where a predicate evaluates to UNKNOWN. Possible values are 'lastPrediction', 'nullPrediction', 'defaultChild', and 'weightedConfidence'.
MISSINGVALUE-PENALTY	DOUBLE	Factor to be applied to the confidence every time the default child (or surrogate node) strategy has to be applied.
NUMLEAVES	BIGINT	Number of leaf nodes contained in the model.
NUMNODES	BIGINT	Number of nodes contained in the model.

NZA_META_<model_name>_NODES

The NZA_META_<model_name>_NODES table contains all tree nodes in the model together with some explanatory information. The table further encodes the tree structure because it contains the predecessor (parent) node for every node.

The table contains one line for each tree node and has the following layout:

Table 12: Columns of the NZA_META_<model name>_NODES table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has NODEID 1.
NAME	NVARCHAR(100)	Name of the node (defaults to NODEID).
DESCRIPTION	NVARCHAR(10000)	Textual node description (defaults to NULL).
SIZE	DOUBLE	Number of data records in the node.
RELSIZE	DOUBLE	Relative size of the node: Size of the current node divided by the size of root node (NODEID 1).
ISLEAF	BOOLEAN	Indicates whether the node is a leaf node.
PARENT	BIGINT	NODEID of parent node.
CLASS	<target column type>	Prediction made for records in this node.
IMPURITY	DOUBLE	The class impurity measure of this node.
DEFAULTCHILD	BIGINT	NODEID of default child node, which is used as successor if the predicate cannot be evaluated. If MISSINGVALUE-STRATEGY is not ‘defaultChild‘ this value is NULL.

NZA_META_<model_name>_PREDICATES

The NZA_META_<model_name>_PREDICATES table contains all simple predicates and all simple set predicates in the model. The predicate of a given node indicates which condition must be true to be reached from its parent. The predicate of the root node is true.

The table has the following layout:

Table 13: Columns of the NZA_META_<model name>_PREDICATES table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model having the predicate. The root node has

Column	Data Type	Purpose
		NODEID 1.
COLUMNNAME	NVARCHAR(128)	Name of the column referenced by the predicate. It is NULL when the predicate is 'true', 'false', 'isMissing' or 'isNotMissing'.
OPERATOR	VARCHAR(16)	Operator used by the predicate. Possible operators are 'true' and 'false', 'equal', 'notEqual', 'lessThan', 'lessOrEqual', 'greaterThan', 'greaterOrEqual', 'isMissing', and 'isNotMissing' for simple predicates, and 'isIn' or 'isNotIn' for simple set predicates.
VALUE	NVARCHAR(16000)	Column value referenced by the predicate. It is NULL when COLUMNNAME is NULL.

NZA_META_<model_name>_COLUMNS

The NZA_META_<model_name>_COLUMNS contains all columns that are used by the data mining algorithm. From this table, you can determine, which columns are required for the model application.

The table contains one line for each model column and has the following layout:

Table 14: Columns of the NZA_META_<model name>_COLUMNS table

Column	Data Type	Purpose
COLUMNNAME	NVARCHAR(128)	Name of an input column
DATATYPE	VARCHAR(64)	SQL data type of COLUMNNAME
OPTYPE	VARCHAR(16)	Operational type of COLUMNNAME: possible values are ‘categorical‘, ‘ordinal‘ and ‘continuous‘.
USAGETYPE	VARCHAR(16)	Usage type: possible values are ‘ignored‘, ‘active‘, ‘predicted‘, ‘supplementary‘,‘frequencyWeight‘, ‘analysisWeight‘,and ‘group‘.
COLUMNWEIGHT	DOUBLE	A priori factor of contribution to the model training process relative to other columns. The default value is 1.0.

Column	Data Type	Purpose
IMPORTANCE	DOUBLE	Indicates the column's importance for the data mining model as determined by the algorithm. The value is between 0 and 1.
OUTLIERTREATMENT	VARCHAR(16)	Outlier treatment: possible values are 'asIs', 'asMissingValues' and 'asExtremeValues'
LOWERLIMIT	DOUBLE	Lower limit of valid values in a numeric field. Null indicates negative infinity.
UPPERLIMIT	DOUBLE	Upper limit of valid values in a numeric field. Null indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the outlier limits are contained in the valid value range or not: possible values are 'openClosed', 'openOpen', 'closedOpen', and 'closedClosed'.

NZA_META_<model_name>_COLUMN_STATISTICS

The NZA_META_<model_name>_COLUMN_STATISTICS contains statistical information about the active columns for each node. The contents and the existence of this table depend on the value of the parameter statistics when the model is built.

The table has the following layout:

Table 15: Columns of the NZA_META_<model name>_COLUMN_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has `NODEID` 1.
COLUMNNAME	NVARCHAR(128)	Name of an input column.
CARDINALITY	BIGINT	Number of distinct values. The value is null if the column is continuous.
MODE	NVARCHAR (16000)	Most frequent discrete value in `COLUMNNAME` for `NODEID`.
MINIMUM	DOUBLE	Minimum value. The value is null if the column is not numeric.

Column	Data Type	Purpose
MAXIMUM	DOUBLE	Maximum value. The value is null if the column is not numeric.
MEAN	DOUBLE	Mean value. The value is null if the column is not numeric.
VARIANCE	DOUBLE	Variance. The value is null if the column is not numeric.
VALIDFREQ	BIGINT	Frequency of valid values.
MISSINGFREQ	BIGINT	Frequency of missing values.
INVALIDFREQ	BIGINT	Frequency of invalid values.
IMPORTANCE	DOUBLE	Importance of the column for data records in `NODEID` towards predicting the target.

NZA_META_<model_name>_DISCRETE_STATISTICS

The NZA_META_<model_name>_DISCRETE_STATISTICS table contains statistical information about the values of active categorical columns for each node. The contents of this table depend on the value of the parameter statistics when the model is built.

The table has the following layout:

Table 16: Columns of the NZA_META_<model name>_ DISCRETE_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has `NODEID` 1.
COLUMNNAME	NVARCHAR(128)	Name of input column.
VALUE	NVARCHAR(16000)	Value occurring in COLUMNNAME. If the column is numeric, the value is a string representation of the true column value.
COUNT	DOUBLE	Number of occurrences of records with `VALUE.`
RELFREQUENCY	DOUBLE	Percentage of occurrences of records with `VALUE`. The relative frequency is based on all records, including those with invalid or

Column	Data Type	Purpose
		null values.
DEVIATION	DOUBLE	RELFREQUENCY(NODEID) -RELFREQUENCY(1)

NZA_META_<model_name>_NUMERIC_STATISTICS

The NZA_META_<model_name>_NUMERIC_STATISTICS contains statistical information about the values of active continuous columns for each node. The contents and the existence of this table depend on the value of the parameter statistics when the model is built.

The table has the following layout:

Table 17: Columns of the NZA_META_<model name>_NUMERIC_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has NODEID 1.
COLUMNNAME	NVARCHAR(128)	Name of a numeric input column.
FROMVALUE	DOUBLE	Lower interval boundary; null indicates negative infinity.
TOVALUE	DOUBLE	Upper interval boundary; null indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the interval limits are contained. Possible values are ‘openClosed‘, ‘openOpen‘, ‘closedOpen‘, and ‘closedClosed‘.
COUNT	DOUBLE	Number of occurrences of records in the interval.
RELFREQUENCY	DOUBLE	Percentage of occurrences of records in the interval. The relative frequency is based on all records, including those with invalid or null values.
DEVIATION	DOUBLE	RELFREQUENCY(NODEID) -RELFREQUENCY(0)
MEAN	DOUBLE	Mean of all values in the interval.

Column	Data Type	Purpose
VARIANCE	DOUBLE	Variance of all values in the interval.

CHAPTER 15

Nearest Neighbors

The nearest neighbor family of classification and regression algorithms is frequently referred to as *memory-based* or *instance-based learning*, and sometimes also as *lazy learning*. These terms correspond to the main concept of this approach, which is to replace model creation by memorizing the training data set and using it appropriately to make predictions.

Background

The basic nearest neighbor (NN) algorithm makes classification or regression predictions for an arbitrary instance x by identifying a training instance $x_{NN} \in T$ that is closest to x and returning its class label $c\ x_{NN}$ or target function value $f\ x_{NN}$ as the predicted class label or target function value for x .

The *k*NN algorithm extends this idea by permitting a specified number $k \geq 1$ of closest training instances to be used rather than just one. For the classification task the predicted class label is determined by the voting of these nearest neighbors, that is, the majority class label in the set of the selected k instances is returned. For the regression task their target function values are averaged to achieve the predicted value to return. The choice of k provides an opportunity to control the tradeoff between overfitting prevention (which may be important particularly for noisy data) and resolution (capability to yield different predictions for similar instances). It usually has to be individually adjusted for a particular data set, with typical values ranging from 1 to several dozen.

The voting mechanism used in the *k*NN algorithm for the classification task makes it possible to specify class weights, to make the prediction process more sensitive to some classes. Specifying a vector of numerical weights for each class instructs the algorithm to weight the training instances of particular classes accordingly when voting. In this manner, the *k*NN algorithm can be made cost-sensitive by assigning each class $d \in C$ a weight $w_d = [d]$, where $[d]$ is the misclassification cost of predicting any class $d' \neq d$ for instances of true class d .

The *k*NN algorithms essentially make two decisions:

memory representation—how to store the training set so that the search for the nearest neighbors can be organized efficiently

distance calculation—how to measure the distance (dissimilarity) between instances

Only the distance calculation is discussed here, as memory representation is an implementation issue that for IBM Netezza In-Database Analytics is resolved by the database implementation environment.

The most commonly used distance measure for the *k*NN algorithm is the Euclidean distance, calculated for instances x_1 and x_2 , described by attributes $a_1, a_2 ,... ,a_n$, as follows:

$$x_1 , x_2 = \sum_{i=1} \sqrt{{}^{n}\ a_i\ x_1 - a_i\ x_2 \quad {}^{2}} \tag{63}$$

This assumes continuous attributes, but can be also used with mixed sets containing some continuous and some discrete attributes, by replacing the difference $a_i\ x_1 \ - a_i\ x_2$ for discrete attribute a_i by 0 if $a_i\ x_1 \ = a_i\ x_2$ or 1 otherwise. The Euclidean distance can be considered a special case of the Minkowski metric for $p=2$:

$$_p\ x_1 , x_2 = \sum_{i=1}^{n} |a_i\ x_1 - a_i\ x_2 |^{p\ p}\ {}^{\frac{1}{}} \tag{64}$$

The value of p can be used to control the relative impact of larger differences on the calculated distance. Two other common special cases are obtained for $p=1$ and $p=\infty$:

$$_1\ x_1 , x_2 = \sum_{i=1}^{n} |a_i\ x_1 - a_i\ x_2\ | \tag{65}$$

$$_\infty\ x_1 , x_2 = \max_{i=1,\ \ ,n} |a_i\ x_1\ - a_i\ x_2\ | \tag{66}$$

The latter is also known as the *maximum distance* and former is also known as the *Manhattan distance*. The related *Canberra distance* uses a slightly different formula:

$$_{1\ '}\ x_1 , x_2 = \sum_{i=1}^{n} \frac{|a_i\ \ x_1 - a_i\ x_2\ |}{|a_i\ x_1\ |\ |a_i\ x_2\ |} \tag{67}$$

All these distance measures can be applied to instances described with discrete attributes in the same way as the Euclidean distance.

Distance functions based on attribute value differences may yield misleading neighbor selection when the attributes used to describe the data differ significantly in their ranges or dispersion. One or a few attributes with the largest range or dispersion may have the dominating impact on the calculated distance and make other attributes negligible, which is not necessarily desirable. To prevent this problem, it is recommended to apply standardization or normalization transformations to continuous attributes before using them for the *k*NN algorithm with such distance functions.

In some applications it may be more appropriate to judge similarity or dissimilarity based on correlations rather than differences of their attribute values. This is justified if instances considered similar may have even substantially different attribute values which exhibit the same “low-high pattern,” that is, the same attributes tend to have high or low values for the compared instances. This is the case, for example, for text documents described by attributes defined as word frequencies, where it is not the difference of word occurrence counts that matters, but the common subsets of the most frequent and/or the least frequent words. Useful similarity measures in such situations are the linear or rank correlation and the cosine distance of the attribute value vectors for the instances. The latter is the cosine of the angle between the attribute value vectors, and is calculated as:

$$\cos(x_1, x_2) = \frac{\sum_{i=1}^{n} a_i(x_1) \cdot a_i(x_2)}{\|a(x_1)\| \cdot \|a(x_2)\|} \qquad (68)$$

where $\| a(x) \|$ designates the Euclidean norm of the vector of attribute values for instance x :

$$\|a(x)\| = \sqrt{\sum_{i=1}^{n} a_i^2(x)} \qquad (69)$$

This is actually a closeness or similarity rather than distance measure, as it is maximized for the most similar instances for which their attribute value vectors are nearly parallel, pointing to roughly the same direction.

Searching for nearest neighbors is the computational bottleneck of the *k*NN algorithm that may reduce its utility for large data sets. There are two major types of approaches to overcoming this limitation:

improved memory organization—using appropriate data structures for storing training instances that permit efficient search, for example, k -d trees

data set subsampling—using heuristic techniques to limit the number of instances for which the distance function must be calculated

Improved memory organization is applicable for in-memory implementations and not applicable in the case of in-database implementation. Data subsampling is therefore adopted, using the *core set* subsampling technique. It reduces the amount of computation required for *k*NN prediction several times without affecting the accuracy significantly.

Applications

The *k*NN algorithm often yields highly accurate predictions, competitive with the most accurate models available. This accuracy potential makes it attractive for applications that demand high accuracy but do not require the availability of an explicit human-readable model. Their quality, however. depends largely on the distance measure. Therefore, the *k*NN algorithm is best suited to applications where sufficient domain knowledge is available to support the selection of an appropriate measure.

By following the lazy learning principle, the *k*NN algorithm postpones all computation needed to generate predictions until they are required. This makes the prediction process much more expensive computationally than for other algorithms since operating by creating a model is relatively cheap to apply. This limits the effectiveness of using the nearest neighbor approach in applications that demand frequent predictions based on large data sets, particularly online real-time prediction with a stream of continuously arriving new instances. It is therefore a better choice for applications where predictions are requested sparingly, but whose accuracy is of extreme importance. However, with core set-based speedup techniques, the *k*NN algorithm may still be useful for large data sets and frequently requested predictions as well.

Available Functionality

The implementation of the *k*NN algorithm provides the following functionality via the **KNN** and **PREDICT_KNN** stored procedures:

classification (voting) and regression (averaging) predictions

support for class weights during classification

optional standardization of continuous attributes (enabled by default)

a choice of predefined distance measures: Euclidean, Manhattan, Canberra, maximum, and user defined measures via UDFs

support for both continuous and discrete attributes in difference-based distance calculation (the difference between discrete values is 0 if equal and 1 otherwise)

rows from the input table containing NULL values are ignored.

Examples

To illustrate the *k*NN algorithm, it is applied to the classification task on the *CensusIncome* data. Although the actual work is performed by the algorithm only when predictions are requested, for consistency with other algorithms the “model building” procedure must be called first as demonstrated below:

```
CALL nza..KNN('intable=nza..CensusIncome_train, id=id,
target=income, model=ci_knn');
```

The model specified via the model argument contains a table with a copy of the provided input table, with rows containing NULL values removed and an appropriate distribution applied. This is the model representation for the *k*NN algorithm, that is the “memorized” training set. To apply such a model, call the corresponding prediction procedure:

```
CALL nza..PREDICT_KNN('intable=nza..CensusIncome_test, model=ci_knn, id=id,
target=income, distance=euclidean, outtable=CensusIncome_income_eu3nn, k=3');
```

It is in the prediction procedure where the opportunity to specify the value of *k* and distance measure is available. The above call uses *k* =3 and the Euclidean distance. Continuous attributes are standardized before distance calculation. Standardization is recommended, so it is enabled by default, but can be disabled using the **stand=FALSE** argument.

It may take a considerable time to generate *k*NN predictions, since for each test set instance the training set is searched for its nearest neighbors. The amount of computation needed for this search is greatly reduced using the fast mode. Fast mode is enabled by default but can be switched off by specifying the **fast=FALSE** argument, which forces an exact, exhaustive nearest neighbor search (recommended only for very small data sets).

The predictions can be evaluated by calculating the misclassification error:

```
CALL nza..CERROR('pred_table=CensusIncome_income_eu3nn,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');
```

Optionally, the calculation can be sped up using core set subsampling (enabled by default) and the confusion matrix:

```
CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income_eu3nn, id=id,
target=income, matrixTable=ci_income_eu3nn_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_eu3nn_cm');
```

To compare, the predictions for k=5 instead of k=3 can be generated and evaluated:

```
CALL nza..PREDICT_KNN('intable=nza..CensusIncome_test, model=ci_knn, id=id,
target=income, distance=euclidean, outtable=CensusIncome_income_eu5nn, k=5');

CALL nza..CERROR('pred_table=CensusIncome_income_eu5nn,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income_eu5nn, id=id,
target=income, matrixTable=ci_income_eu5nn_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_eu5nn_cm');
```

and the predictions generated using the Manhattan instead of Euclidean distance:

```
CALL nza..PREDICT_KNN('intable=nza..CensusIncome_test, model=ci_knn, id=id,
target=income, distance=manhattan, outtable=CensusIncome_income_mh3nn, k=3');
```

```
CALL nza..CERROR('pred_table=CensusIncome_income_mh3nn,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income_mh3nn, id=id,
target=income, matrixTable=ci_income_mh3nn_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_mh3nn_cm');
```

Using all three parameter setups shows that the high-income class tends to be incorrectly detected, resulting in a low true positive rate value, with the high-income class considered positive. Use class weights to make the algorithm more sensitive to this class, as demonstrated for decision trees. The following SQL query creates the weights table with the high-income class weighted 4 more times than the low-income class:

```
CREATE TABLE CensusIncome_weights AS SELECT DISTINCT income as class, CASE
WHEN income='50000+.' THEN 4 ELSE 1 END AS weight FROM
nza..CensusIncome_train;
```

This weights table can then be used for *k*NN prediction as follows, assuming the second of the three parameter setups demonstrated above:

```
CALL nza..PREDICT_KNN('intable=nza..CensusIncome_test, model=ci_knn, id=id,
target=income, weights=CensusIncome_weights, distance=manhattan,
outtable=CensusIncome_income_mh3nnw, k=3');

CALL nza..CERROR('pred_table=CensusIncome_income_mh3nnw,
true_table=nza..CensusIncome_test, pred_id=id, true_id=id, pred_column=class,
true_column=income');

CALL nza..CONFUSION_MATRIX('intable=nza..CensusIncome_test,
resulttable=CensusIncome_income_mh3nnw, id=id, target=income,
matrixTable=ci_income_mh3nnw_cm');

CALL nza..CMATRIX_STATS('matrixTable=ci_income_mh3nnw_cm');
```

The true positive rate is considerably increased, at the cost of increased false positive rate (decreased precision).

To illustrate the application of the *k*NN algorithm to regression, the following sequence of calls use the *WineQuality* data set to generate and evaluate predictions with k=3 and the Euclidean distance measure:

```
CALL nza..KNN('intable=nza..WineQuality_train, model=wq_knn, id=id,
target=quality');

CALL nza..PREDICT_KNN ('intable=nza..WineQuality_test, model=wq_knn, id=id,
target=quality, distance=euclidean, k=3, outtable=WineQuality_quality_3nn');
```

```
CALL nza..MSE('pred_table=WineQuality_quality_3nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

```
CALL nza..MAE('pred_table=WineQuality_quality_3nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

The same can be repeated with k=1 and k=5:

```
CALL nza..PREDICT_KNN ('model=wq_knn, intable=nza..WineQuality_test, id=id,
target=quality, distance=euclidean, k=1, outtable=WineQuality_quality_1nn');
```

```
CALL nza..MSE ('pred_table=WineQuality_quality_1nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

```
CALL nza..MAE ('pred_table=WineQuality_quality_1nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

```
CALL nza..PREDICT_KNN ('model=wq_knn, intable=nza..WineQuality_test, id=id,
target=quality, distance=euclidean, k=5, outtable=WineQuality_quality_5nn');
```

```
CALL nza..MSE ('pred_table=WineQuality_quality_5nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

```
CALL nza..MAE ('pred_table=WineQuality_quality_5nn,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

For easier comparison of the predicted and true target attribute value, the following SQL query creates an auxiliary view:

```
CREATE VIEW WineQuality_knn_pred AS SELECT P3NN.class AS pred_value_3nn,
P1NN.class AS pred_value_1nn, P5NN.class AS pred_value_5nn, T.quality as
true_value
FROM WineQuality_quality_3nn P3NN, WineQuality_quality_1nn P1NN,
WineQuality_quality_5nn P5NN, nza..WineQuality_test T
WHERE P3NN.id=T.id AND P1NN.id=T.id AND P5NN.id=T.id;
```

The correlation between the predictions of each regression tree and the true target attribute values can then be calculated:

```
CALL nza..CORR('intable=WineQuality_knn_pred,
incolumn=pred_value_3nn; true_value');
```

```
CALL nza..CORR('intable=WineQuality_knn_pred,
incolumn=pred_value_1nn;true_value');
```

```
CALL nza..CORR('intable=WineQuality_knn_pred,
incolumn=pred_value_5nn;true_value');
```

The results may appear disappointing, but better models may not result even with considerably more refined algorithms since the quality attribute is not easily predictable in the *WineQuality* data set.

Finally, the following calls demonstrate the effects of disabling two useful and usually recommended features of the *k*NN algorithm, attribute standardization and core set-based subsampling, also known as fast mode. These experiments are performed with k =5 , which appeared to outperform k=3 and k=1 according to the results obtained before.

```
CALL nza..PREDICT_KNN ('intable=nza..WineQuality_test, model=wq_knn, id=id,
target=quality, distance=euclidean, k=5, stand=FALSE,
outtable=WineQuality_quality_5nn_nostand');

CALL nza..MSE ('pred_table=WineQuality_quality_5nn_nostand,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL nza..MAE ('pred_table=WineQuality_quality_5nn_nostand,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL nza..PREDICT_KNN ('intable=nza..WineQuality_test, model=wq_knn, id=id,
target=quality, distance=euclidean, k=5, fast=FALSE,
outtable=WineQuality_quality_5nn_nofast');

CALL nza..MSE ('pred_table=WineQuality_quality_5nn_nofast, pred_column=class,
pred_id=id, true_table=nza..WineQuality_test, true_column=quality,
true_id=id');

CALL nza..MAE ('pred_table=WineQuality_quality_5nn_nofast, pred_column=class,
pred_id=id, true_table=nza..WineQuality_test, true_column=quality,
true_id=id');
```

Disabling attribute standardization increases the error substantially, due to the considerable differences in the ranges, means, and standard deviations of the attributes in the *WineQuality* data set. Switching off fast mode, on the other hand, reduces the error marginally, which is also to be expected. This improvement is achieved at the cost of increased computation time, which is noticeable even for such a small data set. It takes about 3 times longer to generate predictions without the fast mode enabled, and for larger data sets the effect is usually much more substantial.

CHAPTER 16

Linear Regression

Background

Linear regression is a simple but very useful and commonly applied approach to the regression task, even though it only performs direct modeling of linear relationships. It is the thing that limits its applicability – a linear model representation – that makes it fast, efficient, and easy to use (compared to more refined regression algorithms).

Linear regression adopts a particularly simple form of a parametric model representation, in which the model's output for an instance x is calculated as:

$$h\ x\ = w_0\ \sum_{i=1}^{n} w_i\, a_i\ x\ = \sum_{i=0}^{n} w_i\, a_i\ x \qquad (70)$$

where $w_0 , w_1 ,\ldots , w_n$ are *model parameters* (also called weights) and a_0 is an attribute defined as constantly equal 1 , introduced for notational convenience. With this representation, the model applied to any instance calculates the linear combination (or a dot product, more precisely speaking) of its attribute values and parameter values. The parameter w_i corresponding to attribute a_i can be interpreted as representing its impact on the model's prediction with all other attributes held fixed.

Creating a linear regression model based on a given data set, commonly referred to as fitting a model to the data, consists of model parameter estimation, that is, identifying model parameters that optimize an adopted quality measure, typically the mean square error. Linear regression algorithms may differ in the particular technique applied to parameter estimation, with the *least squares* method being the most common choice. It is based on linear algebra transformations of the training set, represented by a matrix with $|T|$ rows, corresponding to instances, and $n\ 1$ columns, corresponding to attributes $a_0 , a_1 , , a_n$ (with the artificial attribute a_0 constantly equal 1 included). To make it possible, all attributes have to be continuous. Any discrete attributes in the original data need therefore to be numerically encoded via a number of binary attributes (treated as continuous) indicating their different possible discrete values.

Let A denote such a numerical matrix training set representation, $\boldsymbol{w}$ denote the column vector of model parameters $w_0 ,w_1 , ,w_n$, and $\boldsymbol{y}$ denote the column vector of target function values for training instances, corresponding to consecutive rows of A . Then the perfect parameter vector would satisfy the following overdetermined matrix equation:

$$A \cdot \boldsymbol{w} = \boldsymbol{y} \tag{71}$$

A least mean squares solution of this equation, leading to a parameter vector that minimizes the mean square error, can be obtained as follows:

$$A^T \cdot A \cdot \boldsymbol{w} = A^T \cdot \boldsymbol{y} \tag{72}$$

$$w = \ A^T \cdot A^{-1} \cdot A^T \cdot \boldsymbol{y} \tag{73}$$

That is, by inverting the $n\ 1 \times n\ 1$ matrix $A^T \cdot A$. The calculation of model parameters is typically accompanied by testing the statistical significance of the model's fit to the data (using the *F*-test) as well as the statistical significance of particular attributes (by using the *t*-test to verify whether the corresponding model parameters significantly differ from 0).

Compared to more complex nonlinear regression, linear regression might appear inferior as being capable of modeling linear or nearly-linear relationships only. While this limitation is important, the following equally important advantages of linear regression still make it attractive for many applications:

efficiency: the parameter estimation process is computationally efficient

ease of use: no complex algorithm setup is required

interpretability: model parameters are (relatively) easy to interpret, at least compared to nonlinear regression

robustness: the parameter estimation process is not prone to false local minima of the mean square error

Their common root is at the simplicity of the parameter estimation process for linear models, which boils down to the minimization of a quadratic function (since the mean square error is quadratic with respect to model parameters).

The limitation resulting from the linear representation can be partially alleviated by applying one of the following two strategies, making it possible to approximate nonlinear relationships with linear models:

data decomposition: decompose the data set into a number of regions and fit separate linear models to these regions, thus creating a piecewise-linear representation,

data transformation: transform the data by creating a new set of attributes, nonlinearly dependent on the original ones, which may make it possible to approximate the target function linearly.

Applications

Linear regression can be applied to all regression tasks as the first reasonable attempt to create a regression model. For many real-world domains the quality of linear models may turn out to be satisfactory, but otherwise it at least provides a solid baseline for further modeling attempts. It may be particularly worthwhile to consider whenever its advantages (efficiency, easy of use, interpretability, and robustness) are important and there is no strong evidence for the nonlinearity of the relationship between the target function and attributes or other strategies (data decomposition or data transformation) are used to overcome the nonlinearity. It is particularly common to see linear models applied for financial prediction, demand forecasting, production volume prediction, etc.

Available Functionality

The Netezza Analytics implementation of linear regression provided by the **LINEAR_REGRESSION**, **PREDICT_LINEAR_REGRESSION** stored procedures covers the following functionality:

fitting linear model with or without the intercept term

multiple regression

fitting the model based on colinear or nearly-colinear attributes (using Moore-Penrose pseudoinversion)

standard LSE procedure using QR decomposition

model diagnostics

support for discrete and continuous attributes

The input table can contain NULL values, that is, instances with missing attribute values, but since all such values are replaced by 0.0, it is highly recommended to remove or impute them prior to model fitting or prediction. The Data Imputation section describes and demonstrates how the latter can be accomplished. You can obtain similar functionality by using the Generalized Linear Models procedure with the identity link setting and the Gaussian distribution setting. If you have data sets with few attributes, the procedure might then work faster.

Examples

Consider creating a linear regression model for predicting the quality attribute from the *WineQuality* data set. The following call builds a linear regression model and provides additional diagnostic information:

```
CALL nza..LINEAR_REGRESSION('intable=nza..WineQuality_train, id=id,
target=quality, model=wq_lr, intercept=TRUE,
calculateDiagnostics=TRUE, useSVDSolver=TRUE');
```

The linear regression model coefficients and the values of diagnostic measures can be inspected by looking into the model table (the name of the model table takes the form nza_meta_<*name of the* *model*>_model):

```
SELECT var_name, value, pval FROM nza_meta_wq_lr_model;
```

Apart from the values of model coefficients, the model table provides diagnostic measures when these measures are requested by the caller.

Model coefficients are described by the following characteristics:

- the coefficient value (the **VALUE** column)
- the attribute (predictor variable) column (the **VAR_NAME** column)
- the target function (predicted variable) column name (the **PREDICTED_NAME** column)
- the identifier of the attribute (predictor variable) in the coefficient matrix (the **VAR_ID** column)
- the identifier of the target function (predicted variable) in the coefficients matrix (the **PREDICTED_ID** column)
- the identifier of the attribute value (predictor variable level) for discrete attributes (the **LEVEL_ID** column)
- the identifier of the target function value (predicted variable level) for discrete attributes (the **PREDICTED_LEVEL_ID** column)

The following measures are diagnostic measures:

- the standard deviation of the fitted coefficients (the **ST_DEV** column)
- the t-test statistic (and corresponding p-value) for the hypothesis that a given coefficient is equal to 0.0 (the **TVAL** and **PVAL** columns)

The constructed linear regression model can be applied to new data using following call:

```
CALL nza..PREDICT_LINEAR_REGRESSION('intable=nza..WineQuality_test, id=id,
model=wq_lr, outtable=wq_lr_pred');
```

provided that the table containing data has the same structure (attribute column names, not necessarily in the same order as original set) as the table used for model construction.

The *CensusIncome* data set, which contains some nominal variables, provides an example of a more complicated linear regression model:

```
CREATE TABLE CensusIncome_restr AS
SELECT * FROM nza..CensusIncome WHERE wage_per_hour>0;

CALL
                             nza..LINEAR_REGRESSION('int
                             able=CensusIncome_restr,
                             incolumn=CLASS_OF_WORKER;
                                   DETAILED_INDUSTRY_RECODE;
                                   DETAILED_OCCUPATION_RECODE;
                                   EDUCATION;
                                   ENROLL_IN_EDU_INST_LAST_WK;
                                   MARITAL_STATUS;
                                   MAJOR_INDUSTRY_CODE;
                                   MAJOR_OCCUPATION_CODE;
                                   RACE;
                                   HISPANIC_ORIGIN;
                                   SEX;
                                   MEMBER_OF_A_LABOR_UNION;
```

```
         REASON_FOR_UNEMPLOYMENT;
         FULL_OR_PART_TIME_EMPLOYMENT_STAT;
         TAX_FILER_STAT;
         REGION_OF_PREVIOUS_RESIDENCE;
         STATE_OF_PREVIOUS_RESIDENCE;
         DETAILED_HOUSEHOLD_AND_FAMILY_STAT;
         DETAILED_HOUSEHOLD_SUMMARY_IN_HOUSEHOLD;
         MIGRATION_CODE_CHANGE_IN_MSA;
         MIGRATION_CODE_CHANGE_IN_REG;
         MIGRATION_CODE_MOVE_WITHIN_REG;
         LIVE_IN_THIS_HOUSE_1_YEAR_AGO;
         MIGRATION_PREV_RES_IN_SUNBELT;
         COUNTRY_OF_BIRTH_FATHER;
         COUNTRY_OF_BIRTH_MOTHER;
         COUNTRY_OF_BIRTH_SELF;
         CITIZENSHIP;
         OWN_BUSINESS_OR_SELF_EMPLOYED;
         FILL_INC_QUESTIONNAIRE_FOR_VETERANS_ADMIN;
         VETERANS_BENEFITS;
         YEAR;
         INCOME;
         FAMILY_MEMBERS_UNDER_18;
     AGE:cont;
     WAGE_PER_HOUR:cont,
   coldeftype=nom,
   coldefrole=ignore,
target=wage_per_hour,
   model=census_restr_wage,
   id=id,
   calculateDiagnostics=TRUE');
```

Output Table Data Formats

NZA_META_<model_name>_MODEL Table

The NZA_META_<model_name>_MODEL table contains details about linear model coefficients as well as model diagnostic measures. The table contains one line for each continuous predictor and one line for each level of nominal predictor, at a given predicted variable.

Table 18: Columns of the V_NZA_COMPONENTS view

Primary key columns: VAR_ID, LEVEL_ID, PREDICTED_ID

Column Name	Column Type	Description
VAR_ID	INTEGER	Index of the attribute in table used for model construction. Special variables are encoded using negative values: ► 1: model intercept ► 2: R^2 ► 3: RSS (Residual Sum of Squares)

Column Name	Column Type	Description
		► 4: estimator of variance of predicted variable
VAR_NAME	NVARCHAR	Name of the attribute in input table, or name of the special variable: (Intercept), [R^2], [RSS], or [Y_VAR_EST].
LEVEL_ID	INTEGER	Identifier of predictor level in the dictionary created for nominal attributes. For continuous attributes it is always set to 1. For model diagnostics variable it is set to 0.
LEVEL_NAME		Name of the nominal variable level.
PREDICTED_ID		Index of the predicted variable (distinguishes groups of coefficients in case of multiple regression).
PREDICTED_NAME		Name of the predicted variable.
LEVEL_NAME		Name of the nominal variable level.
PREDICTED_ID		Index of the predicted variable (distinguishes groups of coefficients in case of multiple regression).
PREDICTED_NAME		Name of the predicted variable.
PREDICTED_LEVEL_ID		Always 1.
PREDICTED_LEVEL_NAME		Not used.
VALUE		Value of the coefficient or special variable.
ST_DEV		Standard deviation of given model coefficient value (when calculation of diagnostics was requested, -1 otherwise).
TVAL		The value of the T test statistics for a given model coefficient (when calculation of diagnostics was requested, -1 otherwise).
PVAL		The p-value of the two sided T test for a given model coefficient (when calculation of diagnostics was requested, -1 otherwise).

CHAPTER 17

Regression Trees

Regression trees are decision trees adapted to the regression task, which store numeric target attribute values instead of class labels in leaves, and use appropriately modified split selection and stop criteria.

Background

As with decision trees, regression tree nodes decompose the data into subsets, and regression tree leaves correspond to sufficiently small or sufficiently uniform subsets. Splits are selected to decrease the dispersion of target attribute values, so that they can be reasonably well predicted by their mean values at leaves. The resulting model is piecewise-constant, with fixed predicted values assigned to regions to which the domain is decomposed by the tree structure.

Creating and using regression tree models involves three major algorithmic subtasks:

- regression tree growing
- regression tree pruning
- regression tree prediction

Since they are direct analogs of their decision tree counterparts, only the differences are highlighted here.

Growing

The purpose of growing is to create a regression tree from a given data set by appropriately selecting splits to minimize the target attribute dispersion and assigning target values to leaves when no further splits are required or possible. The three key operations performed when growing regression trees are the same as for decision trees:

- stop criteria
- target value assignment

split selection

Stop criteria for regression trees prevent applying further splits when:

the number of instances in the corresponding subset is less than a specified minimum

the level of the current node is greater than a specified maximum

the improvement of target value dispersion to the best available split is less than a specified minimum

Target values, assigned to both leaves and internal nodes, are mean target function values for the corresponding subsets of instances. Candidate splits, as with decision splits, are evaluated based on the average dispersion in the subsets obtained after the split.

A natural target value dispersion measure, used for both stop criteria and split selection, is the variance. For node $\mathbf{n}$ it is calculated as:

$$s_f(\mathbf{n}) = \frac{1}{|T_{\mathbf{n}}|} \sum_{x \in T_{\mathbf{n}}} \left(f(x) - m_f(\mathbf{n}) \right)^2 \qquad (74)$$

where $T_{\mathbf{n}}$ denotes the subset of training instances associated with node $\mathbf{n}$, and:

$$m_f(\mathbf{n}) = \frac{1}{|T_{\mathbf{n}}|} \sum_{x \in T_{\mathbf{n}}} f(x) \qquad (75)$$

is the mean target function value in node $\mathbf{n}$.

Pruning

Regression tree pruning may be used for overfitting prevention, just like for decision trees. The predictive utility of all nodes of a previously grown regression tree is verified and those that do not improve the expected prediction quality on new data are replaced by leaves. The decision is based on pruning criteria. The same reduced error pruning (REP) algorithm as for decision trees is employed, with a separate pruning set used to compare the predictive quality of nodes and leaves that could replace them. The mean square error is the most commonly used quality indicator, but other regression quality measures discussed in the Model Diagnostics section could be also adopted for this purpose, including the linear or rank correlation.

Prediction

Regression tree prediction consists of using a previously grown regression tree to generate predictions for a data set. It is performed by applying the splits from the tree nodes to propagate instances from the data set down to the corresponding leaves. Regression tree prediction takes a regression tree and a data set on input and uses the tree to predict the target attribute values for the data set.

Missing Value Support

Missing value support for regression trees is implemented using the same fractional instance technique described in the Missing Value Support section of Decision Trees. For data sets with no missing values, the behavior remains unchanged. Data sets with missing values that could not be processed can be used for both regression tree model creation and prediction. This increases computational expense, but keeps the impact of missing values on model and prediction quality as low as possible.

Applications

The main advantage of regression trees as regression models is their human-readability. The tree structure not only predicts target function values, but also explains which attributes are used and how they are used to arrive at these predictions. Regression trees are usually applied when this advantage is important and outweighs the disadvantage related to the piecewise-constant approximation that they provide to the target function. One consequence of the model’s output being a step function is that small changes of attribute values may yield no changes of the model’s output at all as long as the change is within a single “step,” or substantial changes when moving to a neighboring “step.” This is inconvenient in some applications, where the regression model is expected to gently respond to input changes.

Available Functionality

The implementation of regression trees provided by the **REGTREE**, **PREDICT_REGTREE**, and **PRINT_MODEL** stored procedures covers the following functionality:

- top-down regression tree growing
- support for discrete and continuous attributes
- binary equality-based splits for discrete attributes where *attribute = value*
- binary inequality-based splits for continuous attributes (*attribute ≤ value*),
- split selection based on target attribute dispersion (variance, with more definable via UDFs)
- several stop criteria
  - sufficiently small target attribute dispersion
  - not enough instances, less than a specified minimum required for a split
  - not enough improvement of target attribute dispersion, less than a specified minimum required for a split
  - reaching a maximum allowed tree depth
- regression tree pruning using the reduced error pruning algorithm with the mean square error, coefficient of determination, linear correlation, or rank correlation as quality indicators,
- target attribute mean and variance prediction
- regression tree structure printing

If the input table contains NULL attribute values (for example, in instances with missing attribute values), the fractional instance technique is automatically applied to handle them. The additional computational expense depends on the particular data set and the number of missing values, but typically there is a 5-30% time increase compared to the same data sets with missing values removed or imputed. If this computational expense is undesirable, instances with missing values should be removed or missing values imputed. The Data Imputation section describes and demonstrates how the latter can be accomplished.

Consider that the computational effort needed to grow a regression tree depends on the number of nodes that must be created, which is controlled by the stop criteria. Using appropriate parameter settings for the minimum number of instances required for a split, the minimum dispersion improvement required for a split, or the maximum tree depth may save computation time. Default settings result in moderately complex trees.

Examples

Consider creating regression trees for predicting the quality attribute from the *WineQuality* data set. The following calls build three regression trees, differing in the applied stop criteria and pruning.

```
CALL nza..REGTREE('intable=nza..WineQuality_train, id=id,

target=quality, model=wq_regtree1, minsplit=200, minimprove=0.025');

CALL nza..REGTREE('intable=nza..WineQuality_train, id=id,

target=quality, model=wq_regtree2, maxdepth=8, minimprove=0.025');

CALL nza..REGTREE('intable=nza..WineQuality_train,
valtable=nza..WineQuality_prune, id=id, target=quality, model=wq_regtree3,

minimprove=0.025');
```

In each case the algorithm was instructed to stop when the variance reduction due to the best split fell below 0.025. Additionally, for the first tree, no splitting was attempted after reaching less than 200 instances. For the second tree a maximum depth of 8 was specified. This yields a somewhat larger tree. The third tree was grown without restrictions for the number of instances being split or tree depth, but with pruning applied (which turns out to yield a tree that is actually quite similar to the first). This was achieved by specifying the pruning data set via the **valtable** argument. With this argument specified, the **REGTREE** procedure calls both **GROW_REGTREE** and **PRUNE_REGTREE**, whereas the previous calls are equivalent to **GROW_REGTREE** alone.

More readable output is produced by the **PRINT_REGTREE** procedure:

```
CALL nza..PRINT_MODEL('model=wq_regtree1');


CALL nza..PRINT_MODEL('model=wq_regtree2');


CALL nza..PRINT_MODEL('model=wq_regtree3');
```

The printout obtained for the last tree is presented below:

```
 -- regression tree: "WQ_REGTREE3" --
ALCOHOL <= 10.8
| VOLATILE_ACIDITY <= 0.25
 | VOLATILE_ACIDITY <= 0.205
 | | DENSITY <= 0.99784
 | | | if true then class value -> 5.9153094462541
 | | | CITRIC_ACID <= 0.3
 | | | | if true then class value -> 7.0357142857143
 | | | | if false then class value -> 6.0333333333333
 | | if false then class value -> 5.6896551724138
 | if false then class value -> 5.3559748427673
 ALCOHOL <= 11.733333333333
 | FREE_SULFUR_DIOXIDE <= 13
 | | if true then class value -> 5.125
 | | VOLATILE_ACIDITY <= 0.47
 | | | if true then class value -> 6.2088452088452
 | | | if false then class value -> 4.25
 | FREE_SULFUR_DIOXIDE <= 10
 | | if true then class value -> 5.7727272727273
 | | PH <= 3.29
 | | | RESIDUALSUGAR <= 1.7
 | | | | VOLATILE_ACIDITY <= 0.58
 | | | | | DENSITY <= 0.99035
 | | | | | | if true then class value -> 6.3636363636364
 | | | | | | if false then class value -> 5.85
 | | | | | if false then class value -> 5.25
 | | | | FREE_SULFUR_DIOXIDE <= 20
 | | | | | if true then class value -> 6.2666666666667
 | | | | | VOLATILE_ACIDITY <= 0.15
 | | | | | | if true then class value -> 7.8
 | | | | | | ALCOHOL <= 12.4
 | | | | | | | if true then class value -> 6.5752212389381
 | | | | | | | if false then class value -> 6.8989898989899
 | | | if false then class value -> 6.967032967033
```

Each line in this textual tree representation corresponds to a node or a leaf, and the indentation reflects the tree level. For a node the split condition is printed; for a leaf, the assigned class label is printed. Each node is followed by its left and right subtrees. This is exactly the same representation as for decision trees.

The following calls apply all the three regression trees created above to the test set, storing predictions in the **WineQuality_quality1**, **WineQuality_quality2**, and **WineQuality_quality3** tables:

```
CALL nza..PREDICT_REGTREE('model=wq_regtree1,
intable=nza..WineQuality_test, outtable=WineQuality_quality1, var=TRUE');

CALL nza..PREDICT_REGTREE('model=wq_regtree2,
intable=nza..WineQuality_test, outtable=WineQuality_quality2, var=TRUE');

CALL nza..PREDICT_REGTREE('model=wq_regtree3,
intable=nza..WineQuality_test, outtable=WineQuality_quality3, var=TRUE');
```

If the **id** argument skipped in calls to **PREDICT_REGTREE**, as above, the column specified during model creation is used as the unique instance identifier. The achieved prediction quality can be evaluated using the MSE and MAE errors:

```
CALL                    nza..MSE('pred_table=Winequality_quality1,
pred_column=class,  pred_id=id,  true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL                    nza..MAE('pred_table=WineQuality_quality1,
pred_column=class,  pred_id=id,  true_table=nza..Winequality_test,
true_column=quality, true_id=id');

CALL                    nza..MSE('pred_table=WineQuality_quality2,
pred_column=class,  pred_id=id,  true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL                    nza..MAE('pred_table=WineQuality_quality2,
pred_column=class,  pred_id=id,  true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL                    nza..MSE('pred_table=WineQuality_quality3,
pred_column=class,  pred_id=id,  true_table=nza..WineQuality_test,
true_column=quality, true_id=id');

CALL                    nza..MAE('pred_table=WineQuality_quality3,
pred_column=class,  pred_id=id,  true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

Despite size differences, the quality of predictions generated using the trees does not differ substantially, although the pruned tree appears to outperform the other two.

For easier comparison of the predicted and true target attribute value, the following SQL query creates an auxiliary view:

```
CREATE VIEW WineQuality_regtree_pred AS SELECT P1.class AS pred_value1,
P2.class AS pred_value2, P3.class AS pred_value3, T.quality as true_value
FROM WineQuality_quality1 P1, WineQuality_quality2 P2, WineQuality_quality3
P3, nza..WineQuality_test T WHERE P1.id=T.id AND P2.id=T.id AND P3.id=T.id;
```

Using this view, you can calculate the correlation between the predictions of each regression tree and the true target attribute values:

```
CALL nza..CORR('intable=WineQuality_regtree_pred,

incolumn=pred_value1;true_value');

CALL nza..CORR('intable=WineQuality_regtree_pred,

incolumn=pred_value2;true_value');

CALL nza..CORR('intable=WineQuality_regtree_pred,

incolumn=pred_value3;true_value');
```

The predictions do not correlate very strongly with the true values, which indicates that these regression trees failed to fully capture the relationship between the wine quality and physicochemical features. This result indicates that the relationship is not sufficiently strong in the available data set. Of the three trees, again the last one (pruned) turns out the best, and the second one (the largest tree) is the worst, although the differences are rather minor.

The same evaluation process can be repeated using the training set only:

```
CALL nza..PREDICT_REGTREE('model=wq_regtree1, intable=nza..WineQuality_train,
outtable=WineQuality_train_quality1, var=TRUE');

CALL nza..PREDICT_REGTREE('model=wq_regtree2, intable=nza..WineQuality_train,
outtable=WineQuality_train_quality2, var=TRUE');

CALL nza..PREDICT_REGTREE('model=wq_regtree3, intable=nza..WineQuality_train,
outtable=WineQuality_train_quality3, var=TRUE');

CREATE VIEW WineQuality_regtree_train_pred AS SELECT P1.class AS pred_value1,
P2.class AS pred_value2, P3.class AS pred_value3, T.quality as true_value
FROM WineQuality_train_quality1 P1, WineQuality_train_quality2 P2,
WineQuality_train_quality3 P3, nza..WineQuality_train T WHERE P1.id=T.id AND
P2.id=T.id AND P3.id=T.id;

CALL nza..CORR('intable=WineQuality_regtree_train_pred,
incolumn=pred_value1;true_value');

CALL nza..CORR('intable=WineQuality_regtree_train_pred,
incolumn=pred_value2;true_value');

CALL nza..CORR('intable=WineQuality_regtree_train_pred,
incolumn=pred_value3;true_value');
```

Clearly, the obtained training set-based quality estimates are in no way reliable estimates of the expected new data prediction quality, but they show how well the models actually fit the training data. Not surprisingly, their predictions are somewhat (but not vastly) better than those of the test set. For example, the second tree, which was the worst in the previous example, now appears to be the best. This is a clear indication that it is actually overfitted.

The following example creates corrupted copies of the training, pruning, and test subsets of the *WineQuality* data set to demonstrate the missing value handling capability of the Netezza regression tree algorithm. In each case, 5% of values for two selected attributes of the *WineQuality* data set are removed:

```
CREATE TABLE WineQuality_train_miss AS SELECT * FROM nza..WineQuality_train;
UPDATE WineQuality_train_miss SET alcohol=NULL WHERE random()<0.05;
UPDATE WineQuality_train_miss SET volatile_acidity=NULL WHERE random()<0.05;


CREATE TABLE WineQuality_prune_miss AS SELECT * FROM nza..WineQuality_prune;
UPDATE WineQuality_prune_miss SET alcohol=NULL WHERE random()<0.05;
UPDATE WineQuality_prune_miss SET volatile_acidity=NULL WHERE random()<0.05;


CREATE TABLE WineQuality_test_miss AS SELECT * FROM nza..WineQuality_test;
UPDATE WineQuality_test_miss SET alcohol=NULL WHERE random()<0.05;
UPDATE WineQuality_test_miss SET volatile_acidity=NULL WHERE random()<0.05;
```

The two affected attributes are those most frequently used by the previously created trees, so the modifications will definitely degrade the model and prediction quality.

The following call creates a regression tree using the modified training and pruning data set, with same parameter setup as previously for the last, most successful tree:

```
CALL nza..REGTREE('intable=WineQuality_train_miss,
valtable=WineQuality_prune_miss, id=id, target=quality,
model=wq_regtree3miss, minimprove=0.025');
```

It takes slightly longer than before for the original data with no missing values. Here is how the resulting tree can be applied to the modified test set:

```
CALL nza..PREDICT_REGTREE('model=wq_regtree3miss,
intable=WineQuality_test_miss, outtable=WineQuality_quality3miss, var=TRUE');
```

You can verify this by printing the tree:

```
CALL nza..PRINT_REGTREE('model=wq_regtree3');
```

You will see that it still uses the **alcohol** and **volatile_acidity** attributes. Evaluating the predictions using the mean square error and the mean absolute error reveals some degradation of prediction quality (as expected):

```
CALL nza..MSE('pred_table=WineQuality_quality3miss,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

```
CALL nza..MAE('pred_table=WineQuality_quality3miss,
pred_column=class, pred_id=id, true_table=nza..WineQuality_test,
true_column=quality, true_id=id');
```

Output Table Data Formats

The following tables are generated when you build regression trees:

- NZA_META_<model_name>_MODEL
- NZA_META_<model_name>_NODES
- NZA_META_<model_name>_PREDICATES
- NZA_META_<model_name>_COLUMNS
- NZA_META_<model_name>_COLUMN_STATISTICS
- NZA_META_<model_name>_DISCRETE_STATISTICS
- NZA_META_<model_name>_NUMERIC_STATISTICS

NZA_META_<model_name>_MODEL

The NZA_META_<model_name>_MODEL table contains information about the entire tree model.

The table contains only one line and has the following layout:

Table 19: Columns of the NZA_META_<model name>_MODEL table

Column	Data Type	Purpose
MODELCLASS	VARCHAR(32)	Type of tree model, always regression.
MAXSPLIT	SMALLINT	Maximal number of splits from a node.
DEPTH	SMALLINT	Depth of the tree.
MISSINGVALUE-STRATEGY	VARCHAR(32)	Strategy for handling the case where a predicate evaluates to `UNKNOWN`. Possible values are 'lastPrediction', 'nullPrediction', 'defaultChild', and 'weightedConfidence'.
MISSINGVALUE-PENALTY	DOUBLE	Factor to be applied to the confidence every time the default child (or surrogate node) strategy has to be applied.
NUMLEAVES	BIGINT	Number of leaf nodes contained in the model.
NUMNODES	BIGINT	Number of nodes contained in the model.

NZA_META_<model_name>_NODES

The NZA_META_<model_name>_NODES table contains all tree nodes in the model together with some explanatory information. The table further encodes the tree structure because it contains the predecessor (parent) node for every node.

The table contains one line for each tree node and has the following layout:

Table 20: Columns of the NZA_META_<model name>_NODES table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has `NODEID` 1.
NAME	NVARCHAR(100)	Name of the node (defaults to `NODEID`).
DESCRIPTION	NVARCHAR(10000)	Textual node description (defaults to `NULL`).
SIZE	DOUBLE	Number of data records in the node.
RELSIZE	DOUBLE	Relative size of the node: Size of the

Column	Data Type	Purpose
		current node divided by the size of root node (NODEID 1).
ISLEAF	BOOLEAN	Indicates whether the node is a leaf node.
PARENT	BIGINT	NODEID of parent node.
CLASS	<target column type>	Prediction made for records in this node.
IMPURITY	DOUBLE	The class impurity measure of this node.
DEFAULTCHILD	BIGINT	NODEID of default child node, which is used as successor if the predicate cannot be evaluated. If MISSINGVALUE-STRATEGY is not ‘defaultChild‘ this value is NULL.

NZA_META_<model_name>_PREDICATES

The NZA_META_<model_name>_PREDICATES table contains all simple predicates and all simple set predicates in the model. The predicate of a given node indicates which condition must be true to be reached from its parent. The predicate of the root node is true.

The table has the following layout:

Table 21: Columns of the NZA_META_<model name>_PREDICATES table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model having the predicate. The root node has NODEID 1.
COLUMNNAME	NVARCHAR(128)	Name of the column referenced by the predicate. It is NULL when the predicate is 'true', 'false', 'isMissing' or 'isNotMissing'.
OPERATOR	VARCHAR(16)	Operator used by the predicate. Possible operators are 'true' and 'false', 'equal', 'notEqual', 'lessThan', 'lessOrEqual', 'greaterThan', 'greaterOrEqual', 'isMissing', and 'isNotMissing' for simple predicates, and 'isIn' or 'isNotIn' for simple set predicates.
VALUE	NVARCHAR(16000)	Column value referenced by the predicate.

Column	Data Type	Purpose
		It is NULL when COLUMNNAME is NULL.

NZA_META_<model_name>_COLUMNS

The NZA_META_<model_name>_COLUMNS contains all columns that are used by the data mining algorithm. From this table, you can determine, which columns are required for the model application.

The table contains one line for each model column and has the following layout:

Table 22: Columns of the NZA_META_<model name>_COLUMNS table

Column	Data Type	Purpose
COLUMNNAME	NVARCHAR(128)	Name of an input column
DATATYPE	VARCHAR(64)	SQL data type of COLUMNNAME
OPTYPE	VARCHAR(16)	Operational type of COLUMNNAME: possible values are ‘categorical‘, ‘ordinal‘ and ‘continuous‘.
USAGETYPE	VARCHAR(16)	Usage type: possible values are ‘ignored‘, ‘active‘, ‘predicted‘, ‘supplementary‘,‘frequencyWeight‘, ‘analysisWeight‘,and ‘group‘.
COLUMNWEIGHT	DOUBLE	A priori factor of contribution to the model training process relative to other columns. The default value is 1.0.
IMPORTANCE	DOUBLE	Indicates the column’s importance for the data mining model as determined by the algorithm. The value is between 0 and 1.
OUTLIERTREATMENT	VARCHAR(16)	Outlier treatment: possible values are ‘asIs‘, ‘asMissingValues‘ and ‘asExtremeValues‘
LOWERLIMIT	DOUBLE	Lower limit of valid values in a numeric field. Null indicates negative infinity.
UPPERLIMIT	DOUBLE	Upper limit of valid values in a numeric field. Null indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the outlier limits are

Column	Data Type	Purpose
		contained in the valid value range or not: possible values are ‘openClosed‘, ‘openOpen‘, ‘closedOpen‘, and ‘closedClosed‘.

NZA_META_<model_name>_COLUMN_STATISTICS

The NZA_META_<model_name>_COLUMN_STATISTICS contains statistical information about the active columns for each node. The contents and the existence of this table depend on the value of parameter statistics when the model is built.

The table has the following layout:

Table 23: Columns of the NZA_META_<model name>_COLUMN_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has `NODEID` 1.
COLUMNNAME	NVARCHAR(128)	Name of an input column.
CARDINALITY	BIGINT	Number of distinct values. The value is null if the column is continuous.
MODE	NVARCHAR (16000)	Most frequent discrete value in `COLUMNNAME` for `NODEID`.
MINIMUM	DOUBLE	Minimum value. The value is null if the column is not numeric.
MAXIMUM	DOUBLE	Maximum value. The value is null if the column is not numeric.
MEAN	DOUBLE	Mean value. The value is null if the column is not numeric.
VARIANCE	DOUBLE	Variance. The value is null if the column is not numeric.
VALIDFREQ	BIGINT	Frequency of valid values.
MISSINGFREQ	BIGINT	Frequency of missing values.
INVALIDFREQ	BIGINT	Frequency of invalid values.

Column	Data Type	Purpose
IMPORTANCE	DOUBLE	Importance of the column for data records in NODEID towards predicting the target.

NZA_META_<model_name>_DISCRETE_STATISTICS

The NZA_META_<model_name>_DISCRETE_STATISTICS table contains statistical information about the values of active categorical columns for each node. The contents and the existence of this table depend on the value of the parameter statistics when the model is built.

The table has the following layout:

Table 24: Columns of the NZA_META_<model name>_ DISCRETE_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has NODEID 1.
COLUMNNAME	NVARCHAR(128)	Name of input column.
VALUE	NVARCHAR(16000)	Value occuring in COLUMNNAME. If the column is numeric, the value is a string representation of the true column value.
COUNT	DOUBLE	Number of occurrences of records with VALUE.
RELFREQUENCY	DOUBLE	Percentage of occurrences of records with VALUE. The relative frequency is based on all records, including those with invalid or null values.
DEVIATION	DOUBLE	RELFREQUENCY(NODEID) -RELFREQUENCY(1)

NZA_META_<model_name>_NUMERIC_STATISTICS

The NZA_META_<model_name>_NUMERIC_STATISTICS contains statistical information about the values of active continuous columns for each node. The contents and the existence of this table depend on the value of the parameter statistics when the model is built.

The table has the following layout:

Table 25: Columns of the NZA_META_<model name>_NUMERIC_STATISTICS table

Column	Data Type	Purpose
NODEID	BIGINT	Index value of the node in the tree model. The root node has `NODEID` 1.
COLUMNNAME	NVARCHAR(128)	Name of a numeric input column.
FROMVALUE	DOUBLE	Lower interval boundary; null indicates negative infinity.
TOVALUE	DOUBLE	Upper interval boundary; null indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the interval limits are contained. Possible values are ‘openClosed‘, ‘openOpen‘, ‘closedOpen‘, and ‘closedClosed‘.
COUNT	DOUBLE	Number of occurrences of records in the interval.
RELFREQUENCY	DOUBLE	Percentage of occurrences of records in the interval. The relative frequency is based on all records, including those with invalid or `null` values.
DEVIATION	DOUBLE	RELFREQUENCY(NODEID) -RELFREQUENCY(0)
MEAN	DOUBLE	Mean of all values in the interval.
VARIANCE	DOUBLE	Variance of all values in the interval.

CHAPTER 18

K-Means Clustering

Background

The *k*-means algorithm is the most widely-used clustering algorithm that uses an explicit distance measure to partition the data set into clusters.

The main concept behind the *k*-means algorithm is to represent each cluster by the vector of mean attribute values of all training instances assigned to that cluster, called the cluster's *center*. There are direct consequences of such a cluster representation:

- the algorithm handles continuous attributes only, although workarounds for discrete attributes are possible
- both the cluster formation and cluster modeling processes can be performed in a computationally efficient way by applying the specified distance function to match instances against cluster centers

The algorithm operates by performing several iterations of the same basic process. Each training instance is assigned to the closest cluster with respect to the specified distance function, applied to the instance and cluster center. All cluster centers are then re-calculated as the mean attribute value vectors of the instances assigned to particular clusters. The cluster centers are initialized by randomly picking *k* training instances, where *k* is the desired number of clusters. The iterative process should terminate when there are either no or sufficiently few changes in cluster assignments. In practice, however, it is sufficient to specify the number of iterations, typically a number between 3 and 36.

The range of reasonable distance (or dissimilarity) measures is the same as discussed for the *k*NN algorithm in the Nearest Neighbors section, including all the standard difference-based measures: Euclidean, Manhattan, Canberra, and Maximum. For k-means, two additional distance measures are available--Norm_Euclidean and Mahalanobis. The Norm_Euclidean is the default measure used for k-means clustering.

Normalized Euclidean Distances

For clustering data when the variables of input data have different scales, the normalized Euclidean distance option is better suited than Euclidean distance. The normalized Euclidean distance is:

$$d_n(x, y) = \sqrt{\left(\sum (x_i - y_i)^2 / \sigma_i^2\right)} \qquad (76)$$

where, s_i is standard deviations for categorical variables, (xi-yi) is 0 if xi=yi and 1 otherwise with s_i= 1.

Mahalanobis Distances

The Mahalanobis Distance (MD) is a useful distance measure for cluster analysis and classification. Unlike the Euclidean distance, the MD takes the correlations of the input data into account and is scale-invariant. However, calculation of the MD can be expensive, especially if there is a large number of dimensions, due to the calculation of inverse covariance matrices and matrix multiplications.

The MD is an abstract distance between two entities.

MD with same set of variables

For a group of observations with the same set of (possibly correlated) variables, the MD between an observation $x=(x_1, x_2, x_3, \dots, x_n)^T$ and the center of a group $\boldsymbol{G}$ with mean $u=(u_1, u_2, u_3, \dots, u_n)^T$ and covariance matrix S is defined as:

$$\boldsymbol{D}_M(x, \boldsymbol{G}) = \sqrt{(x-u)^T S^{-1} (x-u)} \qquad (77)$$

For categorical variable values, $(x_i - u_i)$ is a generalized deviation of categorical variable x_i.

Automatic Transformation

For *k*-means, distances are applied to a pair consisting of an instance and a cluster center rather than two instances. Beyond that, it may be necessary to transform continuous attributes by standardization or normalization for difference-based distance measures, to avoid the potentially misleading effect of substantially different ranges or dispersions. A **transform** option is available in the NZA..KMEANS algorithm to automatically transform one or more columns of input data when the input variables have different scales. The following auto_transform options are supported:

L – leave as is

S – standardize

N – normalize

Discrete Attributes

As with nearest neighbors, handling discrete attributes requires appropriate distance measures. The standard difference-based measures can be modified for this purpose by replacing the value difference used for continuous attributes by a *discrete difference*: 0 when the compared values are equal and 1 otherwise. The cluster representation must be modified, as well. Instead of mean attribute values the most frequent values (modes) can be used for discrete attributes in cluster centers.

The randomness of cluster initialization may yield different results in several independent runs, and it is a widespread practice to invoke the algorithm several times to choose the most satisfactory clustering with respect to quality criteria or preferences for a given application.

The clustering model created by the *k*-means algorithm is fully specified by the list of created cluster centers as well as with the distance measure used.

Statistics

Statistical information may not be necessary for certain applications. To save space and time, statistics are not collected by default. An additional parameter is supported by all model building procedures that support collection of enriched statistics. This parameter governs the level of detail of the statistics.

Regardless of the statistics settings, all information needed to score the model is stored. For the k-means algorithm, this includes the distance function, mean values for each cluster and continuous column, and modal values for each categorical column.

If statistics=all[:N] or statistics=values:N, then all statistical information is stored down to the individual column value level. Discrete statistics are limited to at most N distinct values. The default value of N is 100. The default statistics parameter is none.

If statistics=columns, all statistical information is stored down to the individual column level, omitting value level statistics. NZA_META_<model_name>_DISCRETE_STATISTICS and NZA_META_<model_name>_NUMERIC_STATISTICS are not created.

If statistics=none, only statistical information necessary for scoring is stored.

Applications

The *k*-means algorithm usually compares well to more refined and computationally expensive clustering algorithms with respect to the quality of results. Its capability to work with arbitrary distance functions, which may, if necessary, incorporate domain-specific knowledge, makes it convenient for many applications. For practical applications of the algorithm, the range of possible values of the *k* parameter is sufficiently small to be examined by running the algorithm several times with different values of *k*. The extreme situations of partitioning the data into more than a dozen clusters, or a few dozen is not useful. A given application is likely to narrow down the range to a few values.

Available Functionality

The IBM Netezza In-Database Analytics contains the implementation of the *k* -means package algorithm and are exposed as the **PREDICT_KMEANS** stored procedures with the **KMEANS** and following features:

support for both continuous and discrete attributes

in difference-based distance calculation—the difference between discrete values is assumed to be 0 if they are equal and 1 otherwise

in cluster center representation—modes (the most frequent values) are used instead of means for discrete attributes

a choice of predefined distance functions: Normalized Euclidean (the default), Euclidean, Manhattan, Canberra, Mahalanobis, maximum, and other user-defined functions via UDFs

stop criterion satisfied on convergence or after a specified maximum number of iterations performed

a choice to auto-transform a list of columns of input data

cluster membership prediction for new data

Rows from the input table containing NULL values are ignored.

Examples

To illustrate the *k*-means algorithm, apply it to create a clustering model for the *CensusIncome* data set. The income attribute, representing the class, is not used for clustering. This is achieved by the following call, specifying it as the target attribute to be ignored by the algorithm:

```
CALL nza..KMEANS('intable=nza..CensusIncome_train, id=id, target=income, k=5,
maxiter=3, distance=euclidean, model=ci_km5c, outtable=ci_km5m_out');
```

This call uses the Euclidean as a distance measure and creates 5 clusters within at most 3 iterations. Five output tables are created. One output table is specified via the **outtable** argument, which contains cluster membership information for each instance from the 'intable' training data set with the distance from the cluster's center. Four meta tables are specified via the **model** argument:

```
NZA_META_<model>_MODEL
NZA_META_<model>_CLUSTERS
NZA_META_<model>_COLUMNS,
NZA_META_<model>_COLUMN_STATISTICS
```

The NZA_META_<model>_MODEL contains information pertaining to the entire clustering model. The NZA_META_<model_name>_CLUSTERS contains all clusters in the model together with some cluster information (cluster centers, which are mean attribute values for each cluster, along with the cluster size and the sum of squared distances between cluster members and the center). The NZA_META_<model>_COLUMNS contains all columns used by the Kmeans clustering and scoring. The NZA_META_<model>_COLUMN_STATISTICS contains column statistics information.

You can use the PRINT_KMEANS procedure to inspect the built model. For example:

```
CALL nza..PRINT_KMEANS('model=ci_km5c, mode=clusters');
```

Sample output might be:

```
| CLUSTERID | NAME | SIZE   | RELSIZE             | WITHINSS         | DESCRIPTION |
+-----------+------+--------+---------------------+------------------+-------------+
| 1         | 1    | 3603   | 0.025733508556409   | 89551928909.02   |             |
| 2         | 2    | 6650   | 0.047495928920378   | 3965395356.2001  |             |
 | 3          | 3   | 464    | 0.0033140016570008 | 1430862970383    |               |
| 4          | 4   | 126396 | 0.90275119275491   |  15122880034.032 |               |
| 5          | 5   | 2899   | 0.020705368111305  |  307725114544.09 |               |
```

The created cluster model can be inspected by directly looking into the **ci_km5c** cluster table:

```
SELECT * FROM nza_meta_ci_km5c_clusters ORDER BY clusterid;
```

The table contains one row for each cluster and one column for each attribute used for clustering with the mean attribute value for continuous attributes and the most frequent value for discrete attributes. The following additional diagnostic columns are also included:

clusterid—the cluster identification number

size—the number of training instances in the cluster

withinss—the sum of squared distances between training instances assigned to the cluster and the cluster center

A more readable cluster description can be obtained by selecting only the diagnostic columns and skipping the columns representing cluster centers:

```
SELECT clusterid, size, withinss FROM nza_meta_ci_km5c_clusters ORDER BY
clusterid;
```

The **ci_km5m_out** members table contains one row for each training instances and the following columns:

id—the instance identifier

cluster_id—the identifier of the cluster to which the instance is assigned

distance—the distance between the instance and the cluster center, according to the distance measure used for the clustering

It can be used to calculate additional indicators based on the assignment of training instances to clusters. For example, the following query can be used to determine the distribution of the **income** attribute, not used for clustering, within the obtained clusters:

```
SELECT cluster_id, income, count(*)
FROM ci_km5m_out O, nza..CensusIncome_train T
WHERE O.id=T.id
GROUP BY cluster_id, income
ORDER BY cluster_id, income;
```

The following example demonstrates how to describe the distribution of the continuous **age** attribute within the clusters by moments:

```
CREATE VIEW ci_km5m_age AS
SELECT cluster_id, age
FROM ci_km5m_out O, nza..CensusIncome_train T
WHERE O.id=T.id;

CALL nza..MOMENTS('intable=ci_km5m_age, incolumn=age, by=cluster_id,
outtable=ci_km5m_age_moments');
```

```
SELECT * FROM ci_km5m_age_moments;
```

Refer to the section on data exploration algorithms for more details on distribution moments:

Since the cluster sizes of the clustering obtained with *k*=5 are extremely unbalanced, with one very large cluster dominating the others, it may be reasonable to consider a smaller value of *k*. The following code repeats the example presented above for *k*=2.

```
CALL nza..KMEANS('intable=nza..CensusIncome_train, id=id, target=income, k=2,
maxiter=3, distance=euclidean, model=ci_km2c, outtable=ci_km2m_out');

SELECT clusterid, size, withinss FROM nza_meta_ci_km2c_clusters ORDER BY
clusterid;

SELECT cluster_id, income, count(*) FROM ci_km2m_out O,
nza..CensusIncome_train T WHERE O.id=T.id GROUP BY cluster_id, income ORDER
BY cluster_id, income;

CREATE VIEW ci_km2m_age
AS SELECT cluster_id, age
FROM ci_km2m_out O, nza..CensusIncome_train T
WHERE O.id=T.id;

CALL nza..MOMENTS('intable=ci_km2m_age, incolumn=age, by=cluster_id,
outtable=ci_km2m_age_moments');

SELECT * FROM ci_km2m_age_moments ORDER BY cluster_id;
```

After adjusting the k value, the clusters are much more balanced.

The clustering model created based on the training set can be applied to new data. This is demonstrated by the following call, using the same distance and transform options from the model, which generates cluster membership assignments for the CensusIncome test set using the clustering created for k=5.

The k-means clustering options used to build the k-means model (distance, transform, etc.), and statistics of columns and clusters (cluster's variance and covariance information, cluster's mean values, etc.) are saved in meta tables NZA_META_<model>_CLUSTERS, NZA_META_<model>_COLUMNS, and NZA_META_<model>_COLUMN_STATISTICS for k-means scoring, regardless of whether the statistics collection is enabled or not. The same measures used for building the clusters are used to score and predict new clusters.

Note: The distance option is no longer required when calling PREDICT_KMEANS. The same distance option used to build the model is used for scoring. The distance option is accepted, for compatibility reasons, but the value specified is ignored.

```
CALL nza..PREDICT_KMEANS('intable=nza..CensusIncome_test, id=id,
outtable=ci_km5m_test, model=ci_km5c');
```

The output table contains one row for each instance from the specified data set and the same columns described above. It can be used to determine the distribution of the **income** attribute within the obtained clusters on the test set as follows:

```
SELECT cluster_id, income, count(*)
FROM ci_km5m_test O, nza..CensusIncome_test T
WHERE O.id=T.id
GROUP BY cluster_id, income
ORDER BY cluster_id, income;
```

As discussed previously, it may be reasonable to standardize or normalize continuous columns when applying the k-means algorithm with difference-based distance functions. Use transform=S to standardize all continuous columns; use transform=N to normalize all continuous columns. For example, entering transform=age:N;income:N normalizes only age and income columns and leaves the other columns as is.

```
CALL nza..KMEANS('intable=CensusIncome_train, id=id, target=income, k=2,
maxiter=3, distance=euclidean, model=ci_std_km2c, outtable=ci_std_km2m_out,
transform=S');

SELECT clusterid, size, withinss
FROM nza_meta_ci_std_km2c_clusters
ORDER BY clusterid;


SELECT cluster_id, income, count(*)
FROM ci_std_km2m_out O, nza..CensusIncome_train T
WHERE O.id=T.id
GROUP BY cluster_id, income
ORDER BY cluster_id, income;
```

Note: Columns with constant values do not contribute to clustering. These columns cannot be transformed because the variance value is 0. Auto-transformation may generate an error message if one or more columns cannot be transformed due to its variance or a mean value close to 0. The error message includes column name, column's mean value, and column's variance value. For example:

```
NOTICE: WARNING: Transformation of column 'BONUS' failed, mean(BONUS)=1000,
variance(BONUS)=0.
```

You can either selectively list each transformed column individually or preprocess the intable to exclude columns that cannot be transformed.

Output Table Data Formats

NZA_META_<model name>_MODEL Table

The NZA_META_<model name>_MODEL contains information pertaining to the entire clustering model. The table contains one line. Note that information common to all model types, such as OWNERID can be found in NZA_META_MODELS. Following are the table columns.

Table 26: Columns of the NZA_META_<model name>_MODEL table

Column	Data Type	Purpose
MODELCLASS	VARCHAR(32)	Type of clustering model, either center-based or distribution-based.
COMPARISONTYPE	VARCHAR(16)	Type of record comparison, either distance or similarity.
COMPARISONMEASURE	VARCHAR(32)	Function used to aggregate individual column distances (or similarities), for example, squaredEuclidean.
NUMCLUSTERS	INTEGER	Number of clusters contained in the model.

NZA_META_<model name>_CLUSTERS Table

The NZA_META_<model name>_CLUSTERS table contains a list of all clusters in the model and some cluster information. The table contains one line for each cluster. Following are the table columns.

Table 27: Columns of the NZA_META_<model name>_CLUSTERS table

Column	Data Type	Purpose
CLUSTERID (primary key column)	INTEGER	Index value of the cluster in the cluster model. If CLUSTERID is 0, the row pertains to all input records.
NAME	NVARCHAR(100)	Name of the cluster. The default value is CLUSTERID. Use the function set_ClusterName to change the value of this field
DESCRIPTION	NVARCHAR(10000)	Textual cluster description. The default value is NULL. Use the function set_ClusterName to change the value of this field
SIZE	BIGINT	Number of data records in the cluster.
RELSIZE	DOUBLE	Relative size of the cluster: SIZE/SIZE(0).
WITHINSS	DOUBLE	A measure of the cluster homogeneity: the sum of squared distances between records of the cluster and the cluster center.

NZA_META_<model name>_COLUMNS Table

The NZA_META_<model name>_COLUMNS table contains all columns used by the data mining algorithm: the user can determine from this table which columns are required for model application. The table contains one line for each model column.

Note that the table may contain both columns from the original user input table and columns internally created by the auto-transform option. For model application, only the original columns are required because the transformed values are determined by the scoring algorithm. Parameter values for the transformations, such as the mean value of a column, can be found in NZA_META_<model name>_COLUMN_STATISTICS. For some mining functions only a subset of the usage types is possible.

Following are the table columns.

Table 28: Columns of the NZA_META_<model name>_COLUMNS table

Column	Data Type	Purpose
COLUMNNAME (primary key column)	NVARCHAR(128)	Name of an input column.
DATATYPE	VARCHAR(64)	SQL data type of the column.
OPTYPE	VARCHAR(16)	Operational type of the column. Possible values are: categorical, ordinal, and continuous.
USAGETYPE	VARCHAR(16)	Usage type. Possible values are ignored, active, predicted, supplementary, frequencyWeight, analysisWeight,and group.
COLUMNWEIGHT	DOUBLE	A priori factor of contribution to the model training process relative to other columns. The default value is 1.0.
AUTOTRANSFORM	CHAR(1)	Method of column transformation applied automatically by the KMEANS procedure. Possible values are: 'S' for standardization, 'N' for normalization, and 'L' or null for leave as is.
TRANSFORMEDCOLUMN	NVARCHAR(128)	Column name after transformation, Value is null if the column has not been transformed.
COMPAREFUNCTION	VARCHAR(16)	Function used to compare a pair of column values. Possible values are: absDiff, delta, and equal.
IMPORTANCE	DOUBLE	Indicates the column’s importance for the data mining model as determined by the algorithm.

Column	Data Type	Purpose
		The value is between 0 and 1.
OUTLIERTREATMENT	VARCHAR(16)	Outlier treatment. Possible values are: asIs, asMissingValues, and asExtremeValues.
LOWERLIMIT	DOUBLE	Lower limit of valid values in a numeric field. Null indicates negative infinity.
UPPERLIMIT	DOUBLE	Upper limit of valid values in a numeric field. Null indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the outlier limits are contained in the valid value range or not. Possible values are: openClosed, openOpen, closedOpen, and closedClosed.
STATISTICSTYPE	VARCHAR(16)	Indicates which type of statistics are available for the column. Possible values are: ▶ discrete--statistics available from NZA_META_<model name>_DISCRETE_STATISTICS, ▶ numeric--statistics available from NZA_META<model name>_NUMERIC_STATISTICS, ▶ column--statistics available only from NZA_META_<model name>_COLUMNS ▶ null--no statistics available

NZA_META_<model name>_COLUMN_STATISTICS Table

The NZA_META_<model name>_COLUMN_STATISTICS contains one line for each cluster. Following are the table columns.

Table 29: Columns of the NZA_META_<model name>_COLUMNS table

Column	Data Type	Purpose
CLUSTERID (primary key column)	INTEGER	Index value of the cluster in the cluster model. If CLUSTERID is 0, the row pertains to all input records.
COLUMNNAME (primary key column)	NVARCHAR(128)	Name of an input column.
CARDINALITY	BIGINT	Number of distinct values. The value is NULL if

Column	Data Type	Purpose
		the column is continuous.
MODE	NVARCHAR (16000)	Most frequent discrete value in the column for CLUSTERID.
MINIMUM	DOUBLE	Minimum value. The value is NULL if the column is not numeric.
MAXIMUM	DOUBLE	Maximum value. The value is NULL if the column is not numeric.
MEAN	DOUBLE	Mean value. The value is NULL if the column is not numeric.
VARIANCE	DOUBLE	Unbiased sample variance. The value is NULL if the column is not numeric.
VALIDFREQ	BIGINT	Number of valid values.
MISSINGFREQ	BIGINT	Number of missing values.
INVALIDFREQ	BIGINT	Number of invalid values.
IMPORTANCE	DOUBLE	Normalized chi-square value, which indicates if the column distribution in the cluster is significantly different from the overall column distribution. The normalized chi-square value is the factor by which the chi-square value differs from the chi-square value that is sufficient for 99.99% significance (considering degrees of freedom). *This field is currently not supported.* Value: null

NZA_META_<model name>_COVARIANCES Table

The NZA_META_<model name>_COVARIANCES table represents the covariance matrix and its inverse. These matrices are symmetric, that is, cov(c1, c2) = cov(c2, c1). Therefore, only one of the two combinations is stored. Note that if one of the columns is categorical (or both), covariance is not defined, and the values of COVARIANCE and INVERSE pertain to a generalized covariance. The table has one line for each cluster. Following are the table columns.

Table 30: Columns of the NZA_META_<model name>_COVARIANCES table

Column	Data Type	Purpose
CLUSTERID (primary key column)	INTEGER	Index value of the cluster in the cluster model. If CLUSTERID is 0, the row pertains to all input records.
COLUMNNAME1 (primary key column)	NVARCHAR(128)	Name of a first input column.
COLUMNNAME2 (primary key column)	NVARCHAR(128)	Name of a second input column.
COVARIANCE	DOUBLE	If COLUMNAME1 = COLUMNNAME2, the variance of the first input column. Otherwise, the covariance of the first and the second input column.
INVERSE	DOUBLE	Values of the inverse of the covariance matrix.

NZA_META_<model name>_DISCRETE_STATISTICS Table

The NZA_META_<model name>_DISCRETE_STATISTICS table provides statistics about a cluster. Following are the table columns.

Table 31: Columns of the NZA_META_<model name>_DISCRETE_STATISTICS table

Column	Data Type	Purpose
CLUSTERID (primary key column)	INTEGER	Index value of the cluster in the cluster model. If CLUSTERID is 0, the row pertains to all input records.
COLUMNNAME	NVARCHAR(128)	Name of an input column.
VALUE	NVARCHAR(16000)	Value occurring in the input column. Note that the column may be numeric, in which case the value is a string representation of the true column value.
COUNT	BIGINT	Number of occurrences of records with VALUE.
RELFREQUENCY	DOUBLE	Percentage of occurrences of records with VALUE. The relative frequency is based on all records, including those with invalid or null

Column	Data Type	Purpose
		values.
DEVIATION	DOUBLE	RELFREQUENCY(CLUSTERID) -RELFREQUENCY(0)

NZA_META_<model name>_NUMERIC_STATISTICS Table

The NZA_META_<model name>_NUMERIC_STATISTICS table provides statistics about a cluster. Following are the table columns.

Table 32: Columns of the NZA_META_<model name>_NUMERIC_STATISTICS table

Column	Data Type	Purpose
CLUSTERID (primary key column)	INTEGER	Index value of the cluster in the cluster model. If CLUSTER_ID is 0, the row pertains to all input records.
COLUMNNAME (primary key column)	NVARCHAR(128)	Name of a numeric input column.
FROMVALUE (primary key column)	DOUBLE	Lower interval boundary. NULL indicates negative infinity.
TOVALUE (primary key column)	DOUBLE	Upper interval boundary. NULL indicates positive infinity.
CLOSURE	VARCHAR(12)	Indicates whether the interval limits are contained. Possible values are openClosed, openOpen, closedOpen, and closedClosed.
COUNT	BIGINT	Number of occurrences of records in the interval.
RELFREQUENCY	DOUBLE	Percentage of occurrences of records in the interval. The relative frequency is based on all records, including those with invalid or NULL values.
DEVIATION	DOUBLE	RELFREQUENCY(CLUSTERID) -RELFREQUENCY(0)
MEAN	DOUBLE	Mean of all values in the interval.
VARIANCE	DOUBLE	Variance of all values in the interval.

CHAPTER 19

Divisive Clustering

The divisive clustering algorithm is a computationally efficient, top-down approach to creating hierarchical clustering models. Conceptually, it can be thought of as a wrapper around the k -means algorithm (with a specialized method for initial centroid setting), running the algorithm several times to divide clusters into subclusters. The internal *k*-means algorithm assumes a fixed $k = 2$ value.

The divisive clustering algorithm may return different results for the same data set and the same random generator seed when you use different input data distribution or a different number of dataslices. This is due to the behavior of the random number generator, which generates random sequences depending on the number of dataslices and data distribution. The algorithm returns the same model when you use the same machine, the same input data distribution, and the same random seed.

Background

The cluster formation process of the divisive clustering algorithm begins with a single cluster containing all training instances, then the first invocation of k -means divides it into two subclusters by creating two descendant nodes of the clustering tree. Subsequent invocations divide these clusters into more subclusters, and so on, until a stop criterion is satisfied. Stop criterion can be specified by the maximum clustering tree depth or by the minimum required number of instances for further partitioning. The resulting hierarchical clustering tree can be used to classify instances by propagating them down from the root node, and choosing at each level the best matching sub-cluster with respect to the instance's distance from sub-cluster centers.

The internal *k*-means process of the divisive clustering algorithm operates using the ordinary k-means algorithm (with the modified initial centroid generation), discussed in the K-Means Clustering section, using a fixed value of *k*=2 and working with the subset of data from the parent cluster. The initial centroid generation consists two steps: random generation n>>k candidates and then selection of outermost pair of candidates. The cluster center representation and distance measures remain the same. The numbering scheme for clusters in a clustering tree is the same as decision trees: the root node is number 1, and the descendants of node number i have numbers $2\ i$ and $2i\ 1$.

Additionally, leaves, which are clusters with no subclusters, are designated by negative numbers.

Applications

The divisive clustering algorithm is primarily designed for applications where a cluster hierarchy is required for large data sets, for which the alternative agglomerative clustering approach is impractical. The need for a hierarchical clustering model may result from two needs.

To discover and symbolically represent the similarity patterns in the data more completely and in more detail than with flat clustering. Divisive clustering makes it possible to see what similarity-based cluster can be identified as well as also how diverse they are internally, and how they can be further decomposed. This is useful for applications where the clustering model is intended to represent some knowledge about the domain, derived from the analyzed data set. Consider, for example, hierarchical customer segmentation that provides more insight into different customer profiles, or hierarchical document clustering, which roughly represents a topic hierarchy with different levels of generality and specificity.

To enable using a single clustering model with varying “resolution.” Divisive clustering makes it possible to dynamically choose the clustering level of interest, separately for particular instances. This is particularly important when clustering is used to make inference about new instances, such as in hidden attribute prediction and anomaly detection applications, where adjusting the clustering level is a means of balancing the specificity and reliability of predictions.

Available Functionality

The implementation of the divisive clustering algorithm available in IBM Netezza In-Database Analytics via the **DIVCLUSTER** and **PREDICT_DIVCLUSTER** stored procedures provides the following functionality:

cluster formation via repetitive application of the *k*-means algorithm with *k*=2

stop criteria for hierarchy creation specified by the maximum clustering tree depth (number of cluster levels) or by the minimum number of instances in a cluster required for further partitioning

seed for random generator setting

stop criteria for a single *k*-means division satisfied on convergence or reaching a specified maximum number of iterations

a choice of predefined distance functions: Euclidean, Manhattan, Canberra, maximum, with more user-definable measures available via UDFs

cluster membership prediction for new data

Rows from the input table containing NULL values are ignored.

Examples

To illustrate the divisive clustering algorithm, follow the pattern of the *k*-means examples and exclude the **income** attribute, representing the class, which can be achieved by specifying it as the target attribute. The following call creates a 3-level clustering tree, using the Euclidean distance measure and performing no more than 10 *k*-means iterations to divide each cluster into subclusters.

```
CALL nza..DIVCLUSTER('intable=nza..CensusIncome_train, id=id,
target=income, maxiter=10, distance=euclidean, maxdepth=3, model=ci_dc3t,
outtable=ci_dc3m');
```

The model specified via the model argument is stored in one table (named NZA_META_<model>_MODEL) and consists of a cluster table NZA_META_CI_DC3T_CLUSTERS, containing cluster centers for each cluster from the created clustering tree. There is one column for each attribute with the mean attribute value for continuous attributes and the most frequent value for discrete attributes and the following additional columns:

clusterid—the cluster identification number

size—the number of training instances in the cluster

withinss—the sum of squared distances between training instances assigned to the cluster and the cluster center

You can view the contents of this table ordered with respect to the absolute value of the **clusterid** column, due to the convention of negating leaf numbers:

```
SELECT * FROM NZA_META_ci_dc3t_MODEL ORDER BY abs(cluster_id);
```

The other output table, specified via the **outtable** argument, provides cluster membership information for training instances for the leaves of the tree. It contains the following columns for each instances from the training set:

id—the instance identifier

cluster_id—the identifier of the leaf cluster to which the instance is assigned

distance—the distance between the instance and the cluster center, according to the distance measure used for the clustering

The resulting clustering tree is heavily unbalanced, with the majority of training instances assigned to a single leaf:

```
SELECT cluster_id, count(*)
FROM ci_dc3m
GROUP BY cluster_id
ORDER BY abs(cluster_id);
```

As demonstrated for the *k*-means algorithm, the cluster membership table can be used to calculate various indicators based on attribute distribution within clusters, meaningful for particular applications. The following example demonstrates how the mutual information can be calculated and how the $_2$ test can be performed to verify the relationship between cluster membership and the **income** attribute, which was not used for clustering. For details on the mutual information and the

$_{2}$ test, refer to the Data Exploration section.

```
CREATE VIEW ci_dc3m_income AS SELECT cluster_id, income FROM ci_dc3m M,
nza..CensusIncome_train T WHERE M.id=T.id;

CALL nza..MUTUALINFO('intable=ci_dc3m_income,
incolumn=income;cluster_id, outtable=ci_dc3m_income_mutinf');

SELECT * FROM ci_dc3m_income_mutinf;

CALL nza..CHISQ_TEST('intable=ci_dc3m_income,
incolumn=cluster_id;income, outtable=ci_dc3m_income_chisq');

SELECT * FROM ci_dc3m_income_chisq;
```

The results indicate a strong dependency between cluster membership and income.

The resulting hierarchical clustering model can be used for making predictions for new instances. This predicts cluster membership on a selected level of the clustering tree. Unless explicitly specified via the **level** argument, it defaults to the bottom level. The following call generates cluster membership predictions on the test set using the second level of the previously created 3-level clustering tree model:

```
CALL nza..PREDICT_DIVCLUSTER('model=ci_dc3t, level=2,
intable=nza..CensusIncome_test, id=id, outtable=ci_dc3m_test');
```

The format of the output table is the same as the training set cluster membership table created during model creation and can be used to calculate various indicators in the same way.

CHAPTER 20

TwoStep Clustering

TwoStep clustering is a data mining algorithm for large data sets. It is faster than traditional methods because it typically scans a data set only once before it saves the data to a clustering feature (CF) tree. TwoStep clustering can make clustering decisions without repeated data scans, whereas other clustering methods scan all data points, which requires multiple iterations. Non-uniform points are not gathered, so each iteration requires a reinspection of each data point, regardless of the significance of the data point. Because TwoStep clustering treats dense areas as a single unit and ignores pattern outliers, it provides high-quality clustering results without exceeding memory constraints.

The TwoStep algorithm has the following advantages:

It automatically determines the optimal number of clusters. You do not have to manually create a different clustering model for each number of clusters.

It detects input columns that are not useful for the clustering process. These columns are automatically set to supplementary. Statistics are gathered for these columns but they do not influence the clustering algorithm.

The configuration of the CF tree can be granular, so that you can balance between memory usage and model quality, according to the environment and needs.

Background

To cluster data, the TwoStep clustering algorithm does the following actions:

The algorithm scans all data and builds a clustering feature (CF) tree. This tree is built by arranging the input records in a way that similar records become part of the same tree node. If there is a memory issue, the tree is rebuilt with an increased threshold and outliers are removed.

The leaves of the CF tree are clustered hierarchically in memory. The clustering is done by calculating the n * (n-1) / 2 distances between each pair of leaves and merging the two clusters with the smallest distance. The process of calculating distances between clusters and merging the closest two is repeated until the root of the tree is reached. All data is contained in one cluster, thus forming a binary tree.

Starting with the root node of the binary tree, the child node with the worst quality is added. The process of adding child nodes continues until the number of clusters that is determined automatically or that is specified by the user is reached. The number of clusters that is determined automatically is also the optimal number of clusters.

The clustering result is refined by a final pass over the data where each record is assigned to the closest cluster. This behavior is similar to the behavior of the K-means algorithm.

Applications

The TwoStep algorithm takes the following steps:

In the preprocessing step, a CF-tree is built with a limited number of intermediate clusters. The build is controlled by the *maxleaves* parameter. Thus, you can exchange model quality with a high *maxleaves* value for good performance with a low *maxleaves* value according to your needs.

In the refinement step, the final model is built. For this reason, TwoStep is especially suitable for large data sets.

Unlike the K-Means and divisive clustering algorithms, the TwoStep algorithm can determine an optimal number of clusters. Furthermore, it supports log-likelihood distance. This distance is a distribution-based distance measure that is suitable for nominal and numerical attributes.

Available Functionality

The implementation of the TwoStep clustering algorithm is available in IBM SPSS In-Database Analytics through the **TWOSTEP, PREDICT_TWOSTEP, PRINT_TWOSTEP, EXPORT_PMML** or **PMML_MODEL** stored procedures.

TwoStep procedure

The TwoStep procedure generates a clustering model.

This model has the following elements:

Support for nominal and numerical attributes

Support for missing values

Predefined distance measures: Log-likelihood (the default), Euclidean, Norm_Euclidean

PREDICT_TWOSTEP Procedure

The PREDICT_TWOSTEP procedure applies a TwoStep clustering model on an input table.

This model has the following elements:

Cluster membership prediction for new data

Support for missing values

PRINT_TWOSTEP Procedure

The PRINT_TWOSTEP procedure prints details of a TwoStep clustering model.

PMML-related Procedures

The EXPORT_PMML and PMML_MODEL convert a TwoStep clustering model to PMML format.

Examples

To illustrate the TwoStep algorithm, a clustering model for the Iris data set is created in the following example.

The example uses the following assumptions and settings:

Norm_Euclidean distance is the distance measure.

For nominal attributes, statistics are limited to the 25 most frequent values.

For numerical attributes, discrete statistics are calculated only for attributes with 25 or fewer different values. If the number of values is greater than 25, continuous statistics are calculated by building buckets.

In this example, the following call is issued:

```
CALL nza..TWOSTEP('model=irisb, intable=nza..iris,
distance=norm_euclidean, id=id, statistics=values:25');
```

Unlike K-Means, you do not have to specify the wanted number of clusters to be found.

The call generates a clustering model that is represented by the following tables:

NZA_META_IRISB_MODEL

NZA_META_IRISB_CLUSTERS

NZA_META_IRISB_COLUMNS

NZA_META_IRISB_COLUMN_STATISTICS

NZA_META_IRISB_DISCRETE_STATISTICS

NZA_META_IRISB_NUMERIC_STATISTICS

Based on these tables, you can convert the "irisb" model to the PMML format and write it to the file /tmp/irisb.pmml by issuing the following call:

```
CALL nza..EXPORT_PMML('model=irisb, file=/tmp/irisb.pmml');
```

You can extract information about centers of cluster 1 and cluster 2 regarding attributes "CLASS" and "PETALLENGTH" by issuing the following call:

```
CALL nza..PRINT_BIRCH('model=irisb, mode=centers,
clusters=1;2,columns=class;petallength');
```

The call issues the requested information in tabular form, similar to the following table:

```
| CLUSTERID | COLUMNNAME   | CARDINALITY | MODE        | MINIMUM | MAXIMUM
| MEAN   | VARIANCE           | COUNT |
+-----------    +              +             +------------+---------+---------
| 1         |------------CLASS |------------3  | versicolor |         |
|        |                    | 50     |
| 1         | PETALLENGTH |              |            | 1       | 1.9
| 1.464 | 0.028364102564103  | 50     |
| 2         | CLASS        | 3             | setosa     |         |
|        |                    | 100    |
| 2         | PETALLENGTH |              |            | 3       | 6.9
| 4.906 | 0.671688           | 100    |
+-----------    +------------- +------------    +------------+---------+---------
```

You can predict cluster affiliation for the new iris_test data set, by issuing the following call:

```
CALL nza..PREDICT_TWOSTEP('model=irisb, intable=nza..iris_test,
id=id, outtable=irisb_out');
```

The call generates the irisb_out outtable that has one record for each record from the iris_test in-table. The iris_test intable is identified by the ID.

The table looks as follows:

```
  ID   | CLUSTER_ID|      DISTANCE
-----+             +-----------------
   2 |------------1| 1.4026800339739
  30 |            1| 1.1851532469099
  58 |            2| 2.5620498130415
  66 |            2| 1.3337242870809
```

You can read the table as follows:

The record from the iris_test intable with the ID 2 is assigned to cluster 1 of the irisb model. Its distance to the center of cluster 1 is 1.4026800339739. The distance was measured with the Norm_Euclidean measure, which was also used to generate the irisb model.

Output Table Data Formats

The models that are created by the TwoStep algorithm are stored in several tables, in the same manner as the K-Means models. For details of the TwoStep models, see the descriptions in the K-Means section as shown in the following tables.

TwoStep Procedure Output Tables

Table 33: Columns of the NZA_META_<model name>_MODEL table

Table Name (Link)	Description
NZA_META_<model name>_MODEL Table	Generic information on the model
NZA_META_<model name>_CLUSTERS	Generic information on the clusters

Table Name (Link)	Description
NZA_META_<model name>_COLUMNS Table	Generic information on the input columns that are used to build the model
NZA_META_<model name>_COLUMN_STATISTICS Table	Statistics on the input columns that are used to build the model
NZA_META_<model name>_DISCRETE_STATISTICS Table	Statistics on the values of the discrete input columns (nominal or numeric) that are used to build the model
NZA_META_<model name>_NUMERIC_STATISTICS Table	Statistics on the intervals of the numeric input columns that were discretized due to their number of values bigger than <n> This table is only filled in when parameter statistics is set to **all** or **values:<n>**, and there are numeric input columns containing more than <n> values.

PRINT_TWOSTEP Procedure

The PRINT_TWOSTEP procedure returns the following output, depending on the setting of the mode parameter:

Table 34: Columns of the NZA_META_<model name>_MODEL table

If mode is...	Output is of a SELECT statement on the...
clusters	NZA_META_<model name>_CLUSTERS Table
centers	NZA_META_<model name>_COLUMN_STATISTICS Table
statistics	Joined tables: • NZA_META_<model name>_DISCRETE_STATISTICS Table • NZA_META_<model name>_NUMERIC_STATISTICS Table This mode displays a complete result only if the model was built with statistics=values or statistics=all.

CHAPTER 21
Association Rules

Association rules mining is a popular method for discovering interesting and useful patterns in a large scale transaction database. The database contains transactions which consist of a set of items and a transaction identifier (e.g., a market basket). Association rules are implications of the form **X -> Y** where **X** and **Y** are two disjoint subsets of all available items. **X** is called the antecedent or LHS (left hand side) and **Y** is called the consequent or RHS (right hand side). Discovered association rules have to satisfy user-defined constraints on measures of significance and interest.

Background

The *Apriori* algorithm organizes the search for frequent itemsets by systematically considering itemsets of increasing size in consecutive iterations. Due to its method of calculation, the number of candidates identified by the *Apriori* algorithm may be overwhelming for extremely large data sets or a low support threshold. Because of this limitation, the FP-growth algorithm is provided in the IBM Netezza In-Database Analytics package instead.

The FP-growth algorithm avoids candidate generation as well as multiple passes through the data by creating a data structure called a *frequent pattern tree*, or FP-tree. This tree is a compact representation of the data set contents sufficient for finding frequent itemsets. Nodes of the tree represent single items and store their occurrence counts. Only items with sufficiently high support, frequent item-sets of size 1, are represented. Branches, called node-links, in the FP-tree connect nodes that represent items co-occurring for some instances in the data set. There is also a frequent item header table that points to nodes corresponding to particular items.

The tree is built by identifying all frequent items and their counts, then consecutively “inserting” each transaction to the tree. This requires exactly two scans of the data set, regardless of its size or support threshold level. The FP-tree is used to identity frequent itemsets using a *frequent pattern growth* process, which traverses the tree by following node-links in an appropriate way.

By avoiding explicit candidate generation, the FP-growth algorithm reduces the number of data set scans. It can also perform efficiently, regardless of the threshold support.

Applications

Unlike most other data mining tasks, algorithms used for association rule mining do not differ in their output for the same data and support threshold, since they find all sufficiently supported association rules. There is no space for an *inductive bias* as in classification, regression, or clustering algorithms. Where association rule mining algorithms differ is computational efficiency. The FP-growth algorithm, more efficient than the *Apriori* algorithm in most cases, is therefore applicable to all practical instantiations of the association rule mining tasks. It is particularly well-suited to applications where large data sets must be mined or many rules must be created, even with relatively low support values. This is where its efficiency advantages are most likely to manifest themselves and considerably reduce the computational time needed for frequent itemset and rule discovery. Although for small data sets its performance advantage over the *Apriori* algorithm may be small, it is still applicable and sufficiently fast, since such small data sets require minimal computation.

Available Functionality

The Netezza implementation of association rules mining algorithm is available through the ARULE stored procedure and the PREDICT_ARULE stored procedure that provide the following functionality:

association rules mining (including frequent itemset generation)

multiple transaction groups analysis at a time

a minimum support threshold specified via an absolute or relative value

a minimum confidence threshold

a user-specified maximum itemset size (the maximum length of association rule)

a user-specified data decomposition/parallelization level

application of existing association rules for new data

Examples

Consider creating significant and interesting association rules on exemplary RETAIL and RETAILG data sets. The RetailG table comes from retail table and is extended by one column grp which divides transactions into 3 groups. The support greater than 100 examples determines measure of significance and confidence greater than 0.5 measure of interestingness.

The following calls discover mentioned rules.

```
CALL nza..ARULE('model=ASSOC_A1K_05,intable=nza..Retail,
supporttype=absolute, support=100, maxsetsize=6, confidence=0.5');

CALL nza..ARULE('model=ASSOC_A1K_05_G,intable=nza..RetailG,
supporttype=absolute, support=100, maxsetsize=6, confidence=0.5, by=grp');
```

The call runs the algorithm on the Retail data set, requesting that frequent item-sets and rules (generated based on frequent item-sets) with absolute support of 100 or more, rules confidence of

0.5 and more, and maximum number of rule's items (maxsetsize) of 6 and less, with instructions to store output model in the few tables described below.

To inspect the built model, the PRINT_ARULE procedure could be used. For example:

```
CALL nza..PRINT_ARULE('model=ASSOC_A1K_05, minsize=3,
minconf=0.8, minlift=1.2, minconv=1.2');

CALL nza..PRINT_ARULE('model=ASSOC_A1K_05_G, minsize=3, minconf=0.8,
minlift=1.2, minconv=1.2');
```

In the example above, the additional rule selection filter is used to limit the number of presented rules. The minimum number of a rule's items (minsize), confidence (minconf), lift (minlift), and conviction are applied. This allows you to decrease the number of presented rules from 100 to 12 for the most interesting ones for the *ASSOC_A1K_05* model and from 51 to 5 for the *ASSOC_A1K_05_G* model.

The demonstrations above do not include advanced use of the **lvl** argument of the algorithm, which controls the level of data decomposition and computation parallelization within distinct groups. It is set to 1 by default. However, if out of memory problems are encountered, it can be adjusted appropriately to the data set size and machine architecture. Note that a value of 0 performs no decomposition, which may yield the best performance for small data sets or high number of distinct groups.

Note: The examples for the PRINT_ARULE stored procedure do not contain all available parameters. For example, to replace item names by meaningful product names, you can use the namemap parameter. To generate an output table in addition to textual output, you can use the outtable parameter. For a detailed description of all parameters, see the *IBM Netezza In-Database Analytics Reference Guide.*

Output Table Data Formats

As a result, when the ARULE algorithms run, the following tables are generated:

NZA_META_<model_name>_GROUP Table

NZA_META_<model_name>_RULE Table

NZA_META_<model_name>_ITEM Table

NZA_META_<model_name>_ITEMSET Table

NZA_META_<model_name>_GROUP Table

The NZA_META_<model_name>_GROUP table contains parameters used for rules generation for a group. The table contains the following columns.

Table 35: Columns of the NZA_META_<model name>_GROUP table

Column	Data Type	Purpose
GID (primary key column)	BIGINT	The group unique identifier.
GROUP_NAME	<type from intable>	Group name coming from intable.
NUMBEROFTRANSACTIONS	BIGINT	Number of transactions (baskets of items) contained in the input data.
MAXNUMBEROFITEMSPERTA	BIGINT	Number of items contained in the largest transaction.
AVGNUMBEROFITEMSPERTA	DOUBLE	Average number of items contained in a transaction.
NUMBEROFITEMS	BIGINT	Number of different items contained in the input data.
NUMBEROFITEMSETS	BIGINT	Number of itemsets contained in the model.
NUMBEROFRULES	BIGINT	Number of rules contained in the model.
MINIMUMSUPPORT	BIGINT	Minimum relative support value (#supporting transactions / #total transactions) satisfied by all rules.

NZA_META_<model_name>_RULE Table

The NZA_META_<model_name>_RULE table contains discovered association rules as well as a set of “interestingness” measures. The table contains the following columns.

Table 36: Columns of the NZA_META_<model name>_RULE table

Column	Data Type	Purpose
GID (primary key column)	BIGINT	The group unique identifier.
RHS_SID	BIGINT	Right-hand side itemset ID (refers to *ITEMSETS* table)
LHS_SID	BIGINT	Left-hand side itemset ID (refers to *ITEMSETS* table).
RHS_SIZE	INTEGER	Size of right-hand side itemset.
LHS_SIZE	INTEGER	Size of left-hand side itemset.

Column	Data Type	Purpose
SUPPORT	DOUBLE	Support of the rule, that is, the relative frequency of transactions that contain A and C: [support(A->C)= support(A+C)]
CONFIDENCE	DOUBLE	Confidence of the rule: [confidence(A->C) = support(A+C) / support(A)]
LIFT	DOUBLE	Lift value of the rule. If the XML attribute is specified explicitly in the rule, the following equation must hold true: [lift(A->C) = confidence(A->C) / support(C)
CONVICTION	DOUBLE	Conviction value of the rule. The frequency that the rule makes an incorrect prediction: [conviction(A->C) = (1-support(C))/(1-confidence(A->C))
AFFINITY	DOUBLE	Affinity of the rule, a measure of the transactions that contain both the antecedent and consequent, compared to those that contain the antecedent or the consequent (union). [affinity(A->C) = support(A+C) / [support(A) + support(C) - support(A+C)]
LEVERAGE	DOUBLE	Leverage value of the rule, a measure of the difference between the observed frequency of A+C and the frequency that would be expected if A and C were independent: [leverage(A->C) = support(A->C) - support(A)*support(C)]

NZA_META_<model_name>_ITEM Table

The NZA_META_<model_name>_ITEM table contains a dictionary of items. The table contains the following columns.

Table 37: Columns of the NZA_META_<model name>_ITEM table

Column	Data Type	Purpose
ITEM (primary key column)	BIGINT	The item unique identifier.
ITEM_NAME	<type from the intable>	Item name coming from original input table, defined by *intable* parameter.

NZA_META_<model_name>_ITEMSET Table

The NZA_META_<model_name>_ITEMSET table contains frequent itemsets that are discovered by the FPGrowth algorithm and that are limited by the *support* and *maxsetsize* parameters. The number of item columns depends on the *maxsetsize* parameter. The default value of this parameter is 6, the maximum value is 64. The table contains the following columns.

Table 38: Columns of the NZA_META_<model name>_ITEM table

Column	Data Type	Purpose
SID (primary key column)	BIGINT	An identifier to uniquely identify an itemset
GID	BIGINT	An identifier to uniquely identify a group
ITEM1	BIGINT	First item identifier
ITEM2 to ITEM<n>	BIGINT	*N* item identifier, depending on the specified max rule size
SIZE	INTEGER	The number of items contained in this itemset
SUPPORT	DOUBLE	The relative support of the itemset
LIFT	DOUBLE	A positive number indicating the ratio between the actual support of the itemset and its expected or predicted support

CHAPTER 22

Sequential Patterns and Rules

Sequential patterns mining is a method for discovering interesting and useful patterns in a large-scale transaction database. Similar to association rules mining, the database contains transactions that consist of a set of items and a transaction identifier, for example, a market basket. Additionally, the transactions are grouped to form a sequence that is ordered by the transaction ID. There can be sequences of transactions for a specified customer, or sequences of transactions per store, or whatever group of transactions is available in the database.

Sequential patterns are frequent patterns that are available in one or several successive transactions of many input sequences. The pattern notation (a b) means items a and b are often found together in the same transaction. The pattern notation (a)(b) means that items a and b are often found together in two different transactions of the same input sequence, where b is in a later transaction than a.

Starting from the sequential patterns, sequential rules are implications of the form **X -> Y** where **X** and **Y** are two sequential patterns that are found. **X** is called the body of the rule, and **Y** is called the head of the rule. Discovered sequential patterns and rules must satisfy user-defined constraints on measures of significance and interest.

Background

The *PrefixSPan* algorithm organizes the search for frequent patterns by systematically considering itemsets of increasing size in consecutive iterations. The *PrefixSPan* algorithm does multiple passes through the data and creates iteratively several *prefix trees*, one per prefix. A prefix is a frequent item. All prefix trees together are a compact representation of the data set contents. Nodes of the tree represent single frequent items, and store their occurrence counts and the transaction time in which they are found. The path from the root node of the tree to the node represents a sequential pattern.

Inside transactions, only sorted patterns are searched: the items are sorted by their ID and a pattern (a b) is searched only if the ID of b is bigger than the ID of a. This method reduces the complexity of the search algorithm and improves the overall performance. It is proven and obvious that patterns (a and (b a) are equivalent so that searching only for pattern (a b) is sufficient to get the correct result in less time.

Applications

Unlike most other data mining tasks and similar to association rules mining, algorithms that are used for sequential patterns mining do not differ in their output for the same data and support threshold. The reason is that these algorithms find all sufficiently supported sequential patterns and rules. There is no inductive bias as in classification algorithms, regression algorithms, or clustering algorithms.

However, sequential patterns mining algorithms differ in computational efficiency. The *PrefixSPan*algorithm proved to be more efficient in most cases and is therefore applicable to all practical instantiations of the sequential patterns mining tasks. This algorithm works fast for small and big data sets, and it is well-suited for finding a few or many frequent patterns. However, the combinatorial aspect of finding a huge number of frequent patterns iteratively might have the disadvantage to increase the temporary tables up to several times the size of the original input table. Temporary tables map the frequent patterns to the input sequences.

Available Functionality

The Netezza implementation of the sequential patterns mining algorithm is available through the SEQRULES stored procedure.

This procedure provides the following functionality:

Sequential patterns and rules mining

A name mapping table to present item names instead of item codes

A minimum support threshold that is specified through an absolute value or relative value

A minimum confidence threshold

A maximum length of sequential patterns

A minimum time and a maximum time between transactions of an input sequence that is to be considered

A maximum time between the first item and the last item of a sequential pattern that is found

Examples

Consider creating significant and interesting sequential rules on exemplary RETAILS data sets. The RETAILS table comes from the retail table and is extended by an SID column that groups transactions into sequences of transactions. The minimum support is set to 3% of the total number of input sequences. The minimum confidence is set to 0.5.

The following calls discover the mentioned rules.

```
CALL nza..SEQRULES('model=retails_seq, intable=nza..retails,
sid=sid, tid=tid, item=item, minsupport=0.03, minconf=0.5');
```

The sequential patterns and rules that are found are stored in the output model tables, which are described in Output Table Data Formats.

To inspect the built model, use the following PRINT_SEQRULES stored procedure or the PRINT_MODEL stored procedure. You can print the patterns or the rules. Additionally, you can filter out patterns of rules according to their support, lift, confidence, length, time, or items that they contain or that they do not contain. You can sort the result and truncate it to a specified number of rows.

```
CALL nza..PRINT_SEQRULES('model=retails_seq, minlen=3,
maxtime=30000, sort=confidence;lift');
```

```
CALL nza..PRINT_MODEL('model=retails_seq, type=patterns,
minlen=3, itemsin=32;41, maxlift=2, limit=20');
```

If some of the rules or patterns are not useful to you, you can prune the sequential patterns model by using almost the same parameters that you use for printing. You can first print the rules by using the filter parameters; and when the resulting output suits your needs, you can prune the model to exactly those rules and patterns.

For example, the following call prunes the model, that is, restricts its content, to exactly the rules that were used in the first call of PRINT_SEQRULES. The sort parameter is not used in PRUNE_SEQRULES.

```
CALL nza..PRUNE_SEQRULES('model=retails_seq, minlen=3, maxtime=30000');
```

The patterns and rules that are pruned are not deleted from the model, but marked as inactive.

The following call activates these patterns and rules again.

```
CALL nza..PRUNE_SEQRULES('model=retails_seq, reset=true');
```

Finally, you can use the model to detect or predict the purchase behavior of other customers. Only non-pruned patterns and non-pruned rules are considered. When you apply the model on new input sequences of transactions, you can detect the patterns in these input sequences. Moreover, you can detect rules the body of which but not the head of which is in the input sequence. By using these rules, you can predict that this customer probably buys the items that are contained in the head of the rule in the future.

The following call predicts the rules that apply to the same input table. The call delivers only the most confident rule per input sequence.

```
CALL nza..PREDICT_SEQRULES('model=retails_seq, intable=nza..retails,
outtable=retails_out, type=rules, sort=confidence, limit=1');
```

Output Table Data Formats

The SEQRULES stored procedure generates the following tables:

NZA_META_<model_name>_SEQRULES Table
NZA_META_<model_name>_SEQPATTERNS_STATISTICS Table
NZA_META_<model_name>_SEQPATTERNS Table
NZA_META_<model_name>_ITEMSETS Table
NZA_META_<model_name>_ITEMS Table

NZA_META_<model_name>_SEQRULES Table

The NZA_META_<model_name>_SEQRULES table contains discovered sequential rules and a set of interestingness measures. The table contains the following columns.

Table 39: Columns of the NZA_META_<model name>_SEQRULES table

Column	Data Type	Purpose
RULEID (primary key column)	INTEGER	The unique identifier of the rule
SEQUENCEID	INTEGER	The sequential pattern ID that corresponds to this rule For example, the ID consists of the sequence of the body and the head (refers to *SEQPATTERNS* table)
BODYID	INTEGER	The sequential pattern ID of the body (refers to *SEQPATTERNS* table)
HEADID	INTEGER	The sequential pattern ID of the head (refers to *SEQPATTERNS* table).
CONFIDENCE	DOUBLE	The confidence of the rule: [confidence(A->C) = support(A+C) / support(A)]
AVGTIME	DOUBLE or	The average time between the body and the head of

Column	Data Type	Purpose
	INTERVAL	this rule
PRUNED	BOOLEAN	The rule is marked as pruned or not pruned

NZA_META_<model_name>_SEQPATTERNS_STATISTICS Table

The NZA_META_<model_name>_SEQPATTERNS_STATISTICS table contains the set of interestingness measures for the sequential patterns that are found. The table contains the following columns.

Table 40: Columns of the NZA_META_<model name>_SEQPATTERNS_STATISTICS table

Column	Data Type	Purpose
SEQUENCEID (primary key column)	INTEGER	The unique ID for the sequential pattern
LENGTH	INTEGER	The number of items in the sequential pattern
COUNT	BIGINT	The number of input sequences containing this sequential pattern
SUPPORT	DOUBLE	The fraction of the input sequences containing this sequential pattern
LIFT	DOUBLE	The lift of the sequential pattern lift(a b) = support(a b) / (support(a) * support(b))
AVGTIME	DOUBLE or INTERVAL	The average duration of the sequential pattern It is NULL when the column *tid* is neither numeric, nor of type date/time.
DEVTIME	DOUBLE or INTERVAL	The standard deviation of the average duration of the sequential pattern It is NULL when the column *tid* is neither numeric, nor of type date/time.
PRUNED	BOOLEAN	The rule is marked as pruned or not pruned

NZA_META_<model_name>_SEQPATTERNS Table

The NZA_META_<model_name>_SEQPATTERNS table contains discovered sequential patterns. The table contains the following columns.

Table 41: Columns of the NZA_META_<model name>_SEQPATTERNS table

Column	Data Type	Purpose
SEQUENCEID (distribution column)	INTEGER	The unique ID for the sequential pattern
SEQUENCENUM	INTEGER	The rank of the itemset in the sequential pattern
ITEMSETID	INTEGER	The itemset ID that belongs to the sequential pattern (refers to *ITEMSETS* table)

NZA_META_<model_name>_ITEMSETS Table

The NZA_META_<model_name>_ITEMSETS table contains frequent itemsets that are discovered in the sequential patterns. The table contains the following columns.

Table 42: Columns of the NZA_META_<model name>_ITEMSETS table

Column	Data Type	Purpose
ITEMSETID (primary key column)	INTEGER	The unique itemset ID
ITEMID	INTEGER	An item ID that belongs to this itemset

NZA_META_<model_name>_ITEMS Table

The NZA_META_<model_name>_ITEMS table contains a dictionary of items. The table contains the following columns.

Table 43: Columns of the NZA_META_<model name>_ITEMS table

Column	Data Type	Purpose
ITEMID (primary key column)	INTEGER	The unique item ID
ITEM	Type of the input item column	The item value in the original data
ITEMNAME	NVARCHAR(1024)	The display name of the item If no mapping table is indicated, the item value is displayed by default.
COUNT	BIGINT	The number of input sequences that contain this item

Column	Data Type	Purpose
SUPPORT	DOUBLE	The fraction of the input sequences that contain this item

CHAPTER 23

Time Series Forecasting

Background

Many types of business-relevant or scientific data have values that change over time. Some typical examples are:

- Daily sales figures for a store
- Energy consumption readings from household electric meters
- Price per gallon at a local gas station

It is often useful to analyze the behavior of such changes, both to describe the development over time, specifically for a particular trend and seasonality, as well as to predict unknown values of the series, usually for the future. A typical area of application is supply chain management, where future needs may be predicted based on past trends.

A time series is a sequence of numerical data values, measured at successive—but not necessarily equidistant—points in time. Examples are daily stock prices, monthly unemployment counts, or annual changes in global temperature. The two main goals of time series analysis are to understand the underlying patterns which are represented by the observed data and to make forecasts.

Time Series Types

There are two different types of time series:

- **Distinct Time Series**—Values belong to exact points in time, for example a time series of individual temperature readings
- **Aggregated Time Series**—Values are aggregated across some interval in time, for example a time series of monthly average temperatures or daily sales figures

Aggregated time series can be considered as distinct time series, where the aggregated values belong to a specified point within the time interval. The interval typically chosen is either the beginning, the middle, or the end. In the following example, aggregated time series are arbitrarily considered as distinct time series with values at the beginning of the interval. The time series of monthly average temperatures might be, for example:

```
 2011-11-30 0:00      52.7
 2010-12-31 0:00      null
 2011-01-31 0:00      45.0
 2011-02-28 0:00      44.3
2011-03-31  0:00      51.2
2011-04-30  0:00      55.3
2011-05-31  0:00      69.7

2011-06-30  0:00      68.8
```

Time series can be in general decomposed into a trend, seasonality and random noise.

A *trend* is a linear or nonlinear function of time that does not repeat in time. An example of this is the increase of the annual average world temperature during the last 150 years.

The *seasonality* is a periodic function, that is, a function that regularly repeats itself in time. An example is the cyclic behavior in monthly recorded temperature data in Germany.

The *random noise* is attributed to unpredictable fluctuations in the data.

Time Series Algorithms

NZ Analytics provides three independent time series modeling algorithms:

Exponential Smoothing

ARIMA

Seasonal Trend Decomposition

In addition to these modeling algorithms, Spectral Analysis determines seasonal behavior of time series.

The implemented algorithms decompose a time series into a trend and a seasonal component and analyze them in order to build a descriptive model. This model is used for prediction.

Unlike other data mining algorithms, the time series algorithm does not include separate modeling and scoring phases. Typically, a time series is extrapolated into the future for some period of time, or up to some future point in time that is known at the time of model creation, referred to as the *forecast horizon*. Later, when forecasts are needed for other, unforeseen points in time, there are typically more historic time values available due to continued data collection. Therefore, a new, more accurate model should be created. Since a typical time series consist of relatively small numbers of time values, creating a new model should not take much time.

Data Requirements

All time series algorithms in IBM Netezza Analytics are based on equidistant input time series. If the data provided by the user is not equidistant it is preprocessed to establish a set of equidistant time values.

A time series of length n is equidistant if it contains values at all of the following points in time and none other:

$t_i = t_0 \quad i*s$ for $i=0, \ \dots \ , n-1$ with a fixed step size s .

In general, s does not need to be a fixed interval in real time, but any two subsequent values in the series must be “logically” equidistant, that is, they are equidistant in terms of the business semantics. A time series might, for example, capture monthly airline passengers arriving at an airport:

```
 2010-11-30 12:00    3,500,000
 2010-12-31 12:00    3,900,000
 2011-01-31 12:00    3,700,000
 2011-02-28 12:00    3,400,000
2011-03-31  12:00    4,500,000
2011-04-30  12:00    3,900,000
2011-05-31  12:00    5,800,000

2011-06-30  12:00    6,000,000
```

These time values can be considered equidistant, even if not all months have the same duration, since each one is measuring data for a calendar month, which is considered an equidistant measurement. If the times are converted to integers, for example counting the number of months since January 1970, the transformed time series now looks like this:

```
3,500,000
3,900,000
3,700,000
3,400,000
4,500,000
3,900,000
5,800,000
6,000,000
```

In a similar example, assume the daily price of the IBM stock is recorded. There is no data recorded on Saturdays, Sundays, or public holidays. Logically, the Monday value comes immediately after the Friday value and is the same distance as Monday is to Tuesday. Mapping real time to logical time is not supported, so for such scenarios, the user must perform a transformation before calling the TIMESERIES procedure. If the data is not transformed, it is interpolated as described in the Interpolation section.

Interpolation

If a time series is equidistant with the exception of some missing values, interpolation is used to estimate those values that are missing.

Using the example of of airline passengers arriving in a given month, where the month value is an integer counted from January 1970:

```
3,500,000
3,900,000
3,700,000
3,400,000
4,500,000
3,900,000
```

```
5,800,000
6,000,000
```

If, for instance, the value at time 505 were missing, it could be estimated as 3,650,000 using linear interpolation. In this case, interpolation results in the required equidistant time series.

Further, consider a time series consisting of time values at irregular points in time, for example the following series of temperature readings:

```
2011-07-24   7:00    57
2011-07-24  14:00    75
2011-07-24  21:00    72
2011-07-25   7:15    59
2011-07-25  14:00    77
2011-07-25  20:55    74
2011-07-27   7:00    60
2011-07-27  14:00    78
2011-07-27  22:00    74
```

There are two possible ways of dealing with this situation. One is for the application to calculate aggregates, in this case probably daily aggregates according to a formula based on semantic knowledge of the data. This might result in

```
2011-07-24     24:00    69
2011-07-25     24:00    71
2011-07-26     24:00    null
2011-07-27     24:00    72
```

Alternatively, the algorithm can treat the series as a distinct series and determine a suitable step size s . In this case, the step size determined by the algorithm might be 8 hours, so that the following points in time “belong” to the equidistant series.

```
2011-07-24   6:00
2011-07-24  14:00    75
2011-07-24  22:00
2011-07-25   6:00
2011-07-25  14:00    77
2011-07-25  22:00
2011-07-26   6:00
2011-07-26  14:00
2011-07-26  22:00
2011-07-27   6:00
2011-07-27  14:00    78
2011-07-27  22:00    74
```

Only a small number of values are known. However, by using other known values at off times the missing values can be generated using interpolation.

At the end of this process, the result is always an equidistant time series. Once the time series is created, exact values for points in time are no longer needed. They are internally replaced by index values starting at 1. Thus, for the Time Series algorithms, a time series is described as a sequence v_i, where $i=1,\dots,n$.

Exponential Smoothing

Exponential Smoothing uses an approach whereby the influence of observations decreases over time, exponentially. Similarly, addition, trend and seasonality are taken into account.

The basic form of the formula, used for stationary series without seasonality, calculates a smoothed value s_i for each element in the series as:

(ES-N;N) $s_i = v_i \quad 1- \quad s_{i-1}$ with smoothing constant

If this is inserted recursively, the result is a formula that shows the exponential decay of historic influence:

$$s_i = \left[v_i \quad 1- \quad v_{i-1} \quad 1- {}^2 v_{i-2} \quad 1- {}^3 v_{i-3} \quad 1- {}^4 v_{i-4} \ldots \right]$$

If seasonality with period p is taken into account, still assuming no trend, the formula becomes:

(ES-N;A)

$s_i = v_i - o_{i-p} \quad 1- \quad s_{i-1}$ with o_k being the seasonal oscillations

$o_i = v_i - s_i \quad 1- \quad o_{i-p}$ with seasonal smoothing constant

If seasonal oscillations change over time, the seasonal smoothing constant ensures that older values are considered less than more recent ones. The time series model records the most recent “smoothed” oscillations.

Multiplicative seasonality is treated in the same way, dividing by rather than subtracting the oscillation:

(ES-N;M)

$$s_i = \frac{v_i}{o_{i-p}} 1- \quad s_{i-1}$$

$$o_i = \frac{v_i}{s_i} 1- \quad o_{i-p}$$

The Multiplicative, or exponential, trend without seasonality is handled in the following way:

(ES-M;N)

$s_i = v_i \quad 1- \quad s_{i-1} t_{i-1}$ with t_i being the smoothed trend

$$t_i = \frac{s_i}{s_{i-1}} 1- \quad t_{i-1}$$ with trend smoothing constant

Analogous to seasonal treatment, if the trend changes over time, the trend smoothing constant ensures that older values are considered less than more recent ones. The time series model records the most recent “smoothed” trend.

When the damped trend is modeled, the algorithm tries to learn a systematic change of trend. In the case of damped additive trend, the formulas are:

(ES-DA;N) $s_i = v_i \quad 1- \quad s_{i-1} \quad t_{i-1}$ with trend damping parameter

$${}_i = s_i - s_{i-1} \quad 1- \quad t_{i-1}$$

The other combinations of trend and seasonality are performed similarly. As an example, these are the formulas for the most frequently used damped additive model with multiplicative seasonality:

(ES-DA;M)

$$s_i = \frac{v_i}{o_{i-p}} \quad 1- \quad s_{i-1} \quad t_{i-1}$$

$$t_i = s_i = s_{i-1} \quad 1- \quad t_{i-1}$$

$${}_i = \frac{v_i}{s_i} \quad 1- \quad o_{i-p}$$

Predictions are made by application of $f_{i\,\max+H} = \left(s_{i\,\max} + \sum_{k=1}^{H} \phi^k t_{t\,\max}\right) o_{i\,\max-p+k}$.

The special case of additive trend is often referred to as Holt-Winters method.

ARIMA

AutoRegressive Integrated Moving Average (ARIMA) is a three-part model, with an autoregressive, and integrative and a moving average term.

The autoregressive portion of the model determines how v_i linearly depends on past observations $_{i-1}, v_{i-2}, \dots, v_p$. AR(p) models, that is, autoregressive models of order *p,* are defined as:

(AR) $\quad v_i = \sum_{j=1}^{p} \quad {}_j v_{i-j}$ with weights $\quad {}_j$ used to minimize the forecast error

The moving average portion of the model is the weighted sum of past errors. MA(q) models, that is, autoregressive models of order *q* are defined as:

(MA) $\quad v_i = \sum_{j=1}^{q} {}_j \quad {}_{i-j}$

where ${}_i$ denotes the difference between the estimated and the observed values at time i . Again, the coefficients ${}_j$ are used to minimize the forecast error.

Combining AR(p) and MA(q) yields models known as ARMA models:

(ARMA) $\quad v_i = \sum_{j=1}^{p} {}_j v_{i-j} \quad \sum_{j=1}^{q} {}_j \quad {}_{i-j}$

To model seasonality with periodicity S , the seasonal AR-polynomial and the seasonal MA-polynomial are included in the ARMA(p,q) model. The resulting model is referred to as ARMA(p,q)(P,Q)S in the literature. However, for disambiguation and to avoid case sensitivity, this document uses ARMA(p,q)(SP,SQ)S.

(SAR) $A_{SP}=1-\sum_{j=1}^{SP} a_j v_{i-j}$ with seasonal weights a_j

(SMA) $B_{SQ}=1-\sum_{j=1}^{SQ} b_j v_{i-j}$ with seasonal weights b_j

Since ARMA(p,q)(SP,SQ)S models can be applied only to stationary time series, it must be determined whether the time series is actually stationary. If it is not, it is differentiated d times for the non-seasonal part and SD times for the seasonal part until it becomes stationary. The transformed time series does not contain a trend. The values for p , d , q , SP , SD and SQ are chosen based on statistical tests and the evaluation of BIC. Using the stationary derivatives, the ARMA(p,q)(SP,SQ)S model is estimated using the Levenberg-Marquardt-algorithm. This approach minimizes the sum of squares of the one-step-forecast errors (residuals).

Seasonal Trend Decomposition

Seasonal Trend Decomposition removes periodic behavior and selects a basic shape for the trend, for example a quadratic function. Basic shapes have a number of parameters, or coefficients, whose values are determined so that the mean squared error of the residuals, that is, the differences between the fitted and the observed values of the time series, are minimized.

Consider the following models to capture seasonality using period p :

Additive Seasonality	$Y_t = trend\ t\ \ s_{t \bmod p}$
Multiplicative Seasonality	$Y_t = trend\ t\ s_{t \bmod p}$

To model the trend, one of the following functions $trend\ t$ is fitted as shown in Table 44.

Table 44: Trend types and formulas

Trend Name	**Formula**
Linear Trend	$trend\ t = c_o\ \ c_1\ t$
Quadratic Trend	$trend\ t = c_o\ \ c_1\ t\ \ c_2\ t^2$
Cubic Trend	$trend\ t\ = c_o\ \ c_1 t\ \ c_2\ t^2\ \ c_3\ t^3$
Logarithmic Trend	$trend\ (t) = c_o + c_1 \ln (t)$
Exponential Trend	$trend\ t\ = c_0 c_1 e^{c_2 t}$
Hyperbolic Trend	$trend\ t\ = c_o\ \ \frac{1}{c_1 t}$

Optimization of the sum of squared errors is used to determine the best combination of seasonality model and trend function with its coefficients.

For the evaluation of time series models, two criteria are used.

The *Akaike Information Criterion* (AIC) is defined as:

(AIC) $$AIC=\ln \frac{u'\,u}{n} \quad 2\frac{p}{n}$$

The *Schwartz Information Criterion* (BIC)is defined as:

(BIC) $$BIC=\ln \frac{u'u}{n} \ \ln n \quad \frac{p}{n}$$

In both cases, $u\ t$ is the function giving the residuals, that is, the differences between fitted and observed values, and p is the number of parameters of the model.

Spectral Analysis

Spectral analysis detects the frequencies of periodic behavior by performing a Fourier Transformation, which transforms the series from the time domain into a distribution of frequencies. This transformed series, called a *periodogram*, shows the contribution of the different frequency components: If there is a “peak” at some frequency f, that is, the periodogram has a local maximum at f, the series probably contains an f-frequency oscillation.

The periodogram is partitioned into sections with one significant peak each, disregarding very high and very low frequencies. The relative weight of each section is calculated as a sum of periodogram values in that section divided by the sum of all periodogram values. Consequently, all relative weights sum to 100% and each section is represented by one frequency value, the weighted average of frequencies in the section.

If there is a peak at frequency f, it is likely to reappear at frequencies 2f, 3f, and so on. The algorithm adds the weights of these multiples to the respective base frequency and deletes the higher frequencies.

Sections with less than 5% weight are removed to reduce noise.

Unless provided by the user, the frequency with the largest weight is taken into account as the seasonality of the time series. The highest frequency corresponds to a period of m values in the time series.

Building a Time Series Model

You build a time series model by executing the TIMESERIES stored procedure. This stored procedure invokes the time series forecasting process and, based on the specified algorithm, performs the calculations and writes the data out to the appropriate tables. The stored procedure requires an input table with a specific format. During processing, the stored procedure creates a number of algorithm-specific tables that store data resulting from the time series analysis. Finally, if specified, user-named output tables are created that contain forecast data or seasonally adjusted data.

TIMESERIES Procedure

The following examples illustrate use of the TIMESERIES stored procedure to build time series models.

Examples

To illustrate the timeseries algorithm, build a timeseries model for the *curves* data set. The *curves* data set contains six different curves defined by their coordinates (x,y) and identified uniquely by a name in the column curve.

```
CALL nza..TIMESERIES('model=curves_es, intable=nza..curves, by=curve,
time=x, target=y' );
```

This call, by default, uses the Exponential Smoothing algorithm and automatically determines the best parameters for each curve. No output table is created; you can explore the model either by calling the PRINT_TIMESERIES or PRINT_MODEL stored procedures (see Printing a Time Series Model), or by looking directly in the model tables.

The next call can be used when an experienced user wants to force the Exponential Smoothing algorithm to use a given parameter, in this case the damped additive trend. Additionally to the previous call, this call creates a user output table, *curves_esda_adjusted,* that contains the six curves of the *curves* data set after they have been seasonally adjusted. Seasonally adjusted data can be useful for graphically displaying the data trend after the seasonal influence has been removed.

```
CALL nza..TIMESERIES('model=curves_esda, intable=nza..curves, by=curve,
time=x, target=y, seasadjtable=curves_esda_adjusted, algorithm=esmoothing,
trend=DA');
```

The next call indicates how an experienced user calls the ARIMA algorithm with given parameters, either as value (d=2) or as limit (p<=5). Additionally, the call creates a user output table, *curves_arima_forecast,* that contains the future predictions of the six curves of the *curves* data set.

```
CALL nza..TIMESERIES('model=curves_arima, intable=nza..curves, by=curve,
time=x, target=y, outtable=curves_arima_forecast, algorithm=ARIMA, p<=5, d=2,
q<=5');
```

Input Table Data Format

Most data mining algorithms require an input table containing one row per entity under investigation. When performing customer segmentation, one row for each customer is required as input to the clustering algorithm. In most cases such a format must be generated as a preprocessing step, for example by aggregation.

However, a time series is an object that cannot easily be stored in a single row. It consists of a number of *time value pairs,* numeric values belonging to particular points in time. Such time-value pairs can be stored in a single row of a table.

In many instances, large numbers of related time series are analyzed. In order to capture many time series in a single input table, an additional column is used to contain a time series ID.

Therefore, the input table contains up to three columns; the <ts id> column may not exist if the table contains a single time series.

Column names are arbitrary and must be provided when the algorithm is called.

Table 45: Input table column summary

Column	Data Type	Purpose
<ts id>	<ts id type>	Time series ID
<time>	<ts time type>	Time value
<value>	<ts value type>	Value of time series <ts id> at time <time>

Detailed Column Descriptions

<ts id>—A column containing time series ids; it can be of any type referenced as <ts id type> in this document. It is recommended that <ts id> is used as a distribution key, so that the complete data forming a time series is stored on the same SPU.

<time>—A column containing time values. The datatype of the value must be either time-related or one of the accepted numeric types. Valid datatypes are **TIME**, TIME WITH TIME ZONE (alias **TIMETZ)**, **DATE**, or **TIMESTAMP**, **SMALLINT**, **INTEGER**, **BIGINT**, **FLOAT(p)** with $1 \leq p \leq 15$, or **NUMERIC(p, s)** with $1 \leq p \leq 38$ and $0 \leq s \leq p$. These datatypes are referred to as <ts time type> elsewhere in this document.

Note that values of type **TIMETZ** are processed internally in the order in which they occur in real time. If time exists as separate date and time columns, the user must create a view in which they are combined as timestamps.

<value>—A column containing time series values at times <time>; The datatype of the value must be one of the following numeric types: **SMALLINT**, **INTEGER**, **BIGINT**, **FLOAT(p)** with $1 \leq p \leq 15$, or **NUMERIC(p, s)** with $1 \leq p \leq 38$ and $0 \leq s \leq p$ and is referred to as <ts value type> elsewhere in this document.

Since only a single value is allowed at any one time, the *<ts id>* and *<time>* columns together form a primary key. If <time> has type TIMETZ, no two values can indicate identical times, for example. 2:00 PM GMT and 3:00 PM CET.

In addition, the user may provide a table describing the time series. This table consists of three columns with primary key <ts id>:

Table 46: User-defined time series table columns

Column	Data Type	Purpose
<ts id>	<ts id type>	The time series ID.

Column	Data Type	Purpose
TSNAME	NVARCHAR(100)	The name of the time series.
DESCRIPTION	NVARCHAR(10000)	The description of the time series.

Output Table Data Formats

The following information is stored by the time series algorithm:

- Information pertaining to the entire time series model
  - Generic model information independent of the model type, such as the algorithm used or the creation date
- Information pertaining to individual time series
  - Time series name and, optionally, a description
  - First and last time value used to model the time series
  - Number of equidistant steps in the time series
  - Global forecasting accuracy measures
  - Seasonality, periods and their strengths
  - Trend information
  - For some algorithms, several algorithm-specific measures
- Information for particular points in time
  - Interpolated time series value
  - Seasonally adjusted value
  - Forecast value and standard deviation of forecast error

As a result, when the Time Series Algorithms run, one or more of the following tables are generated, depending on which algorithm is used and the parameter values.

This table holds information about the entire time series model, including all time series:

- NZA_META_<model_name>_MODEL Table

These tables are independent of the algorithm used and hold information pertaining to entire time series:

- NZA_META_<model_name>_SERIES Table
- NZA_META_<model_name>_PERIODS Table
- NZA_META_<model_name>_SEASONALITYDETAILS Table

These tables contain algorithm-specific details pertaining to entire time series:

- NZA_META_<model_name>_EXPODETAILS Table
- NZA_META_<model_name>_ARIMADETAILS Table
- NZA_META_<model_name>_ARMADETAILS Table

NZA_META_<model_name>_STDDETAILS Table

These tables are independent of the algorithm used and hold information for individual points in time:

NZA_META_<model_name>_INTERPOLATED Table

NZA_META_<model_name>_FORECAST Table

NZA_META_<model_name>_SERIES Table

The NZA_META_<model_name>_SERIES contains information pertaining to the entire time series. The table contains one line for each time series in the input data with the following columns.

Table 47: Columns of the NZA_META_<model name>_SERIES table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
NAME	NVARCHAR(100)	The name of the time series.
DESCRIPTION	NVARCHAR(10000)	The description of the time series.
FROMTIME	<ts time type>	Time of the earliest time value used.
TOTIME	<ts time type>	Time of the latest time value used.
INTERPOLATIONMETHOD	VARCHAR(32)	The method of interpolation used, either 'linear', 'cubicspline' or 'exponentialspline'. The value is NULL if no interpolation was performed.
STEPS	INTEGER	The number of equidistant time points used between from and to
SEASONS	INTEGER	The number of seasons in a period or NULL if no seasonality was used.
SEASONALITYTYPE	VARCHAR(32)	Either 'additive', 'multiplicative' or NULL if no seasonality was used.
STEPSIZE	DOUBLE	Difference between two steps or two seasons. The value is provided in the unit given in UNIT.
UNIT	VARCHAR(32)	The unit of the stepsize, either 'MILLISECOND', 'SECOND', 'MINUTE', 'HOUR' (if <ts time type> is TIME), 'DAY', 'WEEK', 'MONTH', 'QUARTER', 'YEAR' (if <ts time

Column	Data Type	Purpose
		type> is DATE), or all of the above if <ts time type> is TIMESTAMP. For numeric types, it is NULL.
PHASE	INTEGER	The season index of the last point in history. This value may be interpolated. It is NULL if no seasonality was used.
STDERROR	DOUBLE	The global standard deviation of the forecast error.
RMSE	DOUBLE	The root mean squared error of the forecasts.
ERRORCODE	CHAR(10)	The error code for the series, with a prefix ANL- for analytics and a 5-digit number for the particular condition, null, if the time series was successfully processed. For a description of error codes, see the Events and Error Conditions section.

NZA_META_<model_name>_PERIODS Table

The NZA_META_<model_name>_PERIODS table contains information pertaining to periodic behavior of the time series found by spectral analysis; the table contains one line for each time series and period with the following columns.

Table 48: Columns of the NZA_META_<model name>_PERIODS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
PERIOD (primary key column)	DOUBLE	The detected period of seasonality.
SEASONS	INTEGER	The number of equidistant steps in a period in interpolated series.
FREQUENCY	DOUBLE	The detected frequency of seasonality.
UNIT	VARCHAR(32)	The unit of the PERIOD and FREQUENCY, either 'MILLISECOND', 'SECOND', 'MINUTE', 'HOUR' (if <ts time type> is TIME), 'DAY', 'WEEK', 'MONTH',

Column	Data Type	Purpose
		'QUARTER', 'YEAR' (if <ts time type> is DATE), or all of the above if <ts time type> is TIMESTAMP, for numeric types, NULL.
WEIGHT	DOUBLE	The weight of periodic behavior with frequency FREQUENCY. The value is between 0 and 1.
HARMONICWEIGHT	DOUBLE	The contribution of the harmonic peaks within WEIGHT, calculated as the area under the harmonic peaks divided by the total area under the Fourier spectrum; the part of WEIGHT contributed by the higher harmonics.

NZA_META_<model_name>_SEASONALITYDETAILS Table

The NZA_META_<model_name>_SEASONALITYDETAILS table contains details about the periodic behavior of the time series used by the exponential smoothing algorithm. The table contains one line for each time series and season with the following columns.

Table 49: Columns of the NZA_META_<model name>_SEASONALITYDETAILS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
SEASON (primary key column)	INTEGER	The season index, from 1 to SEASONS as contained in table NZA_META_<model_name>_SERIES
OSCILLATION	DOUBLE	The oscillation at SEASON; depending on the type of seasonality, the oscillation can be either a value that is added, for example, an additive term, or a value that is multiplied, for example, a factor.

NZA_META_<model_name>_EXPODETAILS Table

The NZA_META_<model_name>_EXPODETAILS table contains information pertaining to exponential smoothing models. The table contains one line for each time series with the following columns.

Table 50: Columns of the NZA_META_<model name>_EXPODETAILS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
SMOOTHEDVALUE	DOUBLE	The smoothed value of the time series at the last point in history. This value may be interpolated.
ALPHA	DOUBLE	The smoothing parameter for the level.
TRENDTYPE	VARCHAR(32)	The trend type. Valid values are 'none', 'additive', 'dampedadditive', 'multiplicative', and 'dampedmultiplicative'.
SMOOTHEDTREND	DOUBLE	The smoothed trend of the time series at the last point in history. This point may be interpolated. The value is NULL if TRENDTYPE is 'none'.
GAMMA	DOUBLE	The smoothing parameter for the trend. The value is NULL or 0 if TRENDTYPE is 'none'.
PHI	DOUBLE	The damping parameter for the trend. The value is NULL or 0 if TRENDTYPE is not 'dampedadditive' or 'dampedmultiplicative'.
DELTA	DOUBLE	The smoothing parameter for the seasonality. The value is NULL or 1 if no seasonality was used.

NZA_META_<model_name>_ARIMADETAILS Table

The NZA_META_<model_name>_ARIMADETAILS table contains information pertaining to ARIMA models. The table contains one line for each time series with the following columns.

Table 51: Columns of the NZA_META_<model name>_ARIMADETAILS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
INTERCEPT	DOUBLE	The constant additive offset of the original time series.
P	SMALLINT	The AR(p) degree of the ARIMA model.
D	SMALLINT	The differentiation degree of the ARIMA(p,d,q)

Column	Data Type	Purpose
		model.
Q	SMALLINT	The MA(Q) degree of the ARIMA model.
SP	SMALLINT	The seasonal AR degree of the ARIMA model.
SD	SMALLINT	The seasonal differentiation degree of the ARIMA model. The value is either 0 or 1.
SQ	SMALLINT	The seasonal MA degree of the ARIMA model.
AIC	DOUBLE	The value of the Akaike information criterion.
BIC	DOUBLE	The value of the Bayesian information criterion, which is also known as the Schwarz information criterion.
LJUNGBOX	DOUBLE	The Ljung-Box statistics value.
LJUNGBOXP	DOUBLE	The Ljung-Box statistics p-value.

NZA_META_<model_name>_ARMADETAILS Table

The NZA_META_<model_name>_ARMADETAILS table contains information about the ARMA portion of ARIMA models. The table contains one line for each time series and autoregressive, moving average or autocorrelation coefficient, with the following columns.

Table 52: Columns of the NZA_META_<model name>_ARMADETAILS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
TYPE (primary key column)	CHAR(4)	The lag type. Valid values are 'AR', 'MA', 'SAR', 'SMA', 'ACF' or 'PACF'.
INDEX (primary key column)	SMALLINT	The positive integer value of lag <= P (if 'AR'), <= Q (if 'MA'), <= SP (if 'SAR'), <= SQ (if 'SMA')
COEFFICIENT	DOUBLE	The numeric value of the correlation coefficient for type TYPE and INDEX. The range is [-1, 1].
CRITICALVALUE	DOUBLE	For models of type 'ACF' or 'PACF', the critical

Column	Data Type	Purpose
		value of the correlation coefficient.
STANDARDERROR	DOUBLE	The incertitude for the value of the coefficient.

NZA_META_<model_name>_STDDETAILS Table

The NZA_META_<model_name>_STDDETAILS table contains information pertaining to seasonal trend decomposition models. The table contains one line for each time series with the following columns.

Table 53: Columns of the NZA_META_<model name>_STDDETAILS table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID.
FITFUNCTION	VARCHAR(32)	The best-fitting fit function. Possible values are 'linear', 'quadratic', 'cubic', 'logarithmic', 'exponential' and 'hyperbolic'.
INTERCEPT	DOUBLE	The constant term in the fit function
COEFFICIENT1	DOUBLE	The multiplicative coefficient of the first non-constant term of the fitting function
COEFFICIENT2	DOUBLE	The multiplicative coefficient of the second non-constant term of the fitting function. This value is used only for quadratic and cubic functions.
COEFFICIENT3	DOUBLE	The multiplicative coefficient of the third non-constant term of the fitting function. This value is used only for cubic functions.
EXPONENT	DOUBLE	The exponent constant of the first non-constant term of the fitting function. This value is used only for exponential functions.

NZA_META_<model_name>_INTERPOLATED Table

The NZA_META_<model_name>_INTERPOLATED table contains interpolated time series values. The table contains one row for each time series and interpolated point in time with the following columns.

Table 54: Columns of the NZA_META_<model name>_INTERPOLATED table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID
INDEX (primary key column)	INTEGER	The index value of a logically equidistant time series.
TIME	<ts time type>	The corresponding time value in the format of the input TIME column.
INTERPOLATED	<ts value type>	The interpolated value of the time series.

For the interpolation method, see the NZA_META_<model_name>_MODEL Table.

NZA_META_<model_name>_FORECAST Table

The NZA_META_<model_name>_FORECAST table holds forecast values. The table contains one line for each time series and point in time for which a forecast has been made, with the following columns.

Table 55: Columns of the NZA_META_<model name>_FORECAST table

Column	Data Type	Purpose
TSID (primary key column)	<ts id type>	The time series ID
TIME (primary key column)	<ts time type>	Time value at which forecast is made
FORECAST	<ts value type>	Expected value of time series
STANDARDERROR	DOUBLE	Standard deviation of forecast error

The table may include TIME values at times before the last time in history, that is the value of the to parameter, if these were contained in the parameter forecasttimes, but the corresponding FORECAST values are NULL. STANDARDERROR may be NULL if it is not possible to estimate the expected error. This is usually an indication of low confidence in the FORECAST value.

If <ts time type> is TIME, the maximum value of TIME is 23:59:59.999999, if it is TIMETZ, the maximum value is 23:59:59.999999-12:59+13:00.

<output> Table

If the user supplied a table name in the outtable parameter when the TIMESERIES stored procedure was executed, a copy of the NZA_META_<model_name>_FORECAST table is created in a table bearing the specified name. The table contains forecast values, and contains one line for each time series and point in time for which a forecast has been made. Columns are as follows.

Table 56: Columns of the <output> table

Column	Data Type	Purpose
<ts id> (unique key column)	<ts id type>	The time series ID.
<time> (unique key column)	<ts time type>	The time value at which the forecast is made.
<value>	<ts value type>	The expected value of the time series.
STANDARDERROR	DOUBLE	The standard deviation of the forecast error.

Note that the first column is omitted if a value for the by parameter was not specified.

<seasadjtable> Table

If the user supplied a table name in the seasadjtable parameter when the TIMESERIES stored procedure was executed, an additional table is created bearing the specified name. The table contains seasonally adjusted time series values, and contains one line for each time series and point in time. Columns are as follows.

Table 57: Columns of the <seasadjtable> table

Column	Data Type	Purpose
<ts id> (unique key column)	<ts id type>	The time series ID.
<time> (unique key column)	<ts time type>	The time value in the format of the input TIME column.
ADJUSTEDVALUE	<ts value type>	The seasonally adjusted value of the time series.

Note that the first column is omitted if a value for the by parameter has not been supplied. Note further that time points may be different from the original time points.

Printing a Time Series Model

Use the PRINT_TIMESERIES stored procedure to display a time series model.

PRINT_TIMESERIES Procedure

The PRINT_TIMESERIES stored procedure displays a time series model.

Example

You created different time series models for the *curves* data set in the section Building a Time Series Model. Use the following call to display those models as tables of the original data and the forecast values of the curve "linear":

```
CALL nza..PRINT_TIMESERIES('model=curves_arima, history=true,
series=linear');
```

Sample output could be:

```
| TSID   | TIME | HISTORY  | FORECAST |
|--------+------+----------+----------|
| linear | 1    | 2        |          |
| linear | 2    | 4        |          |
| linear | 3    | 6        |          |
| linear | 4    | 8        |          |
| linear | 5    | 10       |          |
| linear | 6    | 12       |          |
| linear | 7    | 14       |          |
| linear | 8    | 16       |          |
| linear | 9    | 18       |          |
| linear | 10   | 20       |          |
| linear | 11   | 22       |          |
| linear | 12   | 24       |          |
| linear | 13   | 26       |          |
| linear | 14   | 28       |          |
| linear | 15   | 30       |          |
| linear | 16   | 32       |          |
| linear | 17   | 34       |          |
| linear | 18   | 36       |          |
| linear | 19   | 38       |          |
| linear | 20   | 40       |          |
| linear | 21   | 42       |          |
| linear | 22   | 44       |          |
| linear | 23   | 46       |          |
| linear | 24   | 48       |          |
| linear | 25   | 50       |          |
| linear | 26   | 52       |          |
| linear | 27   | 54       |          |
| linear | 28   | 56       |          |
| linear | 29   | 58       |          |
| linear | 30   | 60       |          |
| linear | 31   | 62       |          |
| linear | 32   | 64       |          |
| linear | 33   | 66       |          |
| linear | 34   | 68       |          |
| linear | 35   | 70       |          |
| linear | 36   | 72       |          |
| linear | 37   | 74       |          |
| linear | 38   | 76       |          |
| linear | 39   | 78       |          |
| linear | 40   | 80      |          |
| linear | 41   | 82      |          |
| linear | 42   | 84      |          |
| linear | 43   | 86      |          |
| linear | 44   | 88      |          |
| linear | 45   | 90      |          |
| linear | 46   | 92      |          |
| linear | 47   | 94      |          |
| linear | 48   | 96      |          |
| linear | 49   | 98      |          |
| linear | 50   | 100     |          |
| linear | 51   |         | 102      |
| linear | 52   |         | 104      |
| linear | 53   |         | 106      |
| linear | 54   |         | 108      |
| linear | 55   |         | 110      |
| linear | 56   |         | 112      |
| linear | 57   |         | 114      |
| linear | 58   |         | 116      |
| linear | 59   |         | 118      |
| linear | 60   |         | 120      |
| linear | 61   |         | 122      |
| linear | 62   |         | 124      |

| linear | 63   |         | 126      |
```

The PRINT_TIMESERIES procedure also offers a basic plotter function to graphically display the original data and the forecast values. The time axis points to the bottom of the page, and the target axis points to the right. The following call displays the curve “sinus” as a plotted graph:

```
CALL nza..PRINT_TIMESERIES('model=curves_arima, history=true, series=sinus,
plot=true');
```

Sample output could be:

```
=================================================
sinus
=================================================
     |0                                      40.10^-1
     |+--------- +         +         + ---------+|->
 1   |                               +
 2   |                                      +
 3   |                                          +
 4   |                                      +
 5   |                               +
 6   |                     +
 7   |           +
     |  +
9|+
10 |+
 11   |          +
 12   |                    +
 13   |                              +
 14   |                                     +
 15   |                                        +
 16   |                                     +
 17   |                              +
 18  |                    +
 19  |          +
 20  |  +
 21  |+
 22  |  +
 23  |          +
 24  |                    +
 25  |                              +
 26  |                                     +
 27  |                                        +
 28  |                                     +
 29  |                              +
 30  |                    +
 31  |          +
 32  |  +
 33  |+
 34  |  +
 35  |          +
 36  |                    +
 37  |                              +
 38  |                                     +
 39  |                                        +
 40  |                                     +
 41  |                              +
 42  |                    +
 43  |          +
 44  |  +
 45  |+
 46  |  +
 47  |          +
 48  |                    +
 49  |                              +
 50  |                                     +
 51  |                                        x
 52  |                                     x
 53  |                              x
 54  |                    x
 55  |          x
 56  |  x
 57  |x
 58  |  x
 59  |          x
 60  |                    x
 61  |                              x
 62  |                                     x
     |
     V  time
(+) history data     (i) interpolated value     (x) forecast value
```

Events and Error Conditions

The following errors might occur during time series analysis. Some errors pertain only to a subset of the time series, in which case the other time series are processed normally. The resulting model table **NZA_META_<model_name>_SERIES** may contain an error code for each individual time series.

The following table shows the possible error codes.

Table 58: Time series error codes

Return Code	Event or Error Condition / Description	Response
NZATS-0100	The time series contains duplicate time values. Times in a time series must be unique.	The user must remove all duplicates.
NZATS-0101	The time series contains too many missing values. No more than 90% of the values of a time series may be missing.	The user should supply values for some of the missing data or remove some of the time points having missing values.
NZATS-0102	The time series is too short for analysis. A time series must contain at least 2 values.	The user should provide more values, for example by increasing the span between “from” and “to.”
NZATS-0103	The time series is too short to be differentiated. After the differentiation step of the ARIMA algorithm, a time series must still be longer than the number of seasons. The minimum required value is d + (SD+1)*(number of seasons).	The user should provide more values of the time series.
NZATS-0104	The time series is too short to be differentiated. After the differentiation step of the ARIMA algorithm, a time series must still contain enough values. The minimum required value is d+2.	The user should provide more values of the time series.
NZATS-0105	The time series is too short to be differentiated. After the seasonal differentiation step of the ARIMA algorithm, a time series must still be longer than the number of seasons. The minimum required value is (SD+1)*(number of seasons).	The user should provide more values of the time series.
NZATS-0106	The time series is too short for seasonal analysis, that is, the time series is shorter than the number of seasons. The minimum required value is the number of seasons.	The user should provide more values of the time series.
NZATS-0107	The time series is too short for exponential	The user should provide more

Return Code	Event or Error Condition / Description	Response
	smoothing analysis with seasonality. That is, the time series is shorter than twice the number of seasons. The minimum required value is twice the number of seasons.	values of the time series.
NZATS-0108	The time series is too short for ARIMA analysis. The minimum required value is 8.	The user should provide more values of the time series.
NZATS-0109	The time series contains too many time points that are close together. The average distance between two time points is below the precision of a double and the calculated interpolation step is 0.	The user should provide time values that are further apart.
NZATS-0110	The time series is too short for spectral analysis. The minimum required value is 10.	The user should provide more values of the time series.
NZATS-0112	The time series is too long for Seasonal Trend Decomposition analysis. There can be at most 2^32 / (number of seasons) values.	The user should provide fewer values of the time series, for example by decreasing the span between “from” and “to.”
NZATS-0113	A damped multiplicative trend cannot be used for these data. The data contain a negative trend and a damp factor less than 1. This does not allow using a damped multiplicative trend.	The user should use a different value of the trend parameter or omit the parameter to have the system determine a suitable value.
NZATS-0114	A multiplicative trend is applied to 0 or negative data. The data contain 0 or negative values. A multiplicative trend can only be applied to positive time series.	The user should use a different value of the trend parameter or omit the parameter to have the system determine a suitable value.
NZATS-0115	Seasonality and period are inconsistent. Seasonality none can only be used with a period of 0. Seasonality additive or multiplicative can only be used when period is greater than one stepsize.	The user should use a consistent combination of the period and seasonality parameters.

CHAPTER 24

Generalized Linear Models

Background

Linear models have been found useful for modeling many real-world phenomena due to their simplicity for training and for model application. True linear models assume a Gaussian distribution of the response variable and a linear impact of the control variables on the response variable. However, in practice, these assumptions cannot always be met.

An extension called Generalized Linear Models (GLM) has therefore been developed to handle these issues. The non-linear relationship is modeled via the link function, while a set of diverse distributions is allowed for the response variable. Of special interest here are the heavy-tailed and nominal (discrete-valued) distributions that are not covered by linear models.

GLM has application in areas such as biology, biopharmaceuticals, engineering, actuarial science and quality assurance.

Using Generalized Linear Models

Assume that the user has a set of records, each of them describing a separate event, independent of the other events. Within an event (a record) is a set of *predictor* variables and a *response* variable. The goal is to find a model that allows the user to predict the value of the response variable based on the known values of the predictor variables.

The GLM framework assumes that the predictor variables influence the response variable in the following way: The predictor variables are transformed by a *transformation principle* to *factors*. These can be used directly as “factors” or they can be transformed to factors using a transformation matrix.

The factors combined linearly, using the *beta parameter vector*, provide a value called *theta* (θ). This , transformed by the *inverted link function*, provides a mean value of the response variable under the given value of predictor variables. This means that it can be assumed that the response variable is influenced both by the predictor variables and a non-observable noise process for which only the distribution type it follows is known. The *distribution type* is assumed to be parameterized by the mean value. So the actual value of the response variable is a value randomly selected under this distribution parameterized by the aforementioned mean, produced via the inverted link function.

To use the algorithm, the user must specify:

which variables are the predictor variables

what are the principles of transformation from predictor variables to factors

which is the response variable

which is the link function (from the predefined set)

which is the distribution type that the response variable follows

The algorithm iteratively seeks the best-fitting model that is the beta vector. Eventually, if the link function and/or distribution are generic, and the user did not specify the respective generic parameters, the link function parameters/distribution parameters may be also sought. The error is represented by the sum of squares of differences between the prediction and the real value of the response variable. Therefore, the maximum number of iterations should also be specified.

The results are coefficients (beta) combining the predictors to the quantity θ . If necessary, the missing parameters of generic distribution/link function may also be returned.

GLM Algorithms

IBM Netezza In-Database Analytics makes two independent GLM algorithms available:

Iteratively Reweighted Least Square (IRLS)

Parallel Stochastic Gradient Descent (PSGD)

Iteratively Reweighted Least Square Algorithm

The *Iteratively Reweighted Least Square* (IRLS) algorithm is currently considered a de facto industry standard.

Initial values of the linear coefficients combining the factors to the θ value are calculated by performing the traditional linear regression on the factors and the value of the link function applied to the response. Once the initial values are known, an iterative process is then run to compute the final values.

First a *working response* is computed, which is equal to the θ obtained from the current coefficients and the predictors. This value is increased or decreased proportional to the difference between the response value and its current expected value. The increased or decreased value is then multiplied by the derivative of the θ on the mean. This accounts for the error in fitting the prediction to the response. Next, working weights of cases (records) are computed. The weights are lower for cases with higher absolute value of the derivative of the θ on the mean (derivative of the link function) and for higher expected variance of the case.

Once the working responses and working weights are obtained, weighted linear regression is performed.

A new iteration is then started unless stopping criteria are matched; stopping criteria are the number of iterations performed and the improvement achieved.

Parallel Stochastic Gradient Descent Algorithm

The *Parallel Stochastic Gradient Descent* (PSGD) algorithm is a more recent development and it is subject to a number of restrictions. It works only for a few distribution types and link functions. The major difference as compared to IRLS is the reduced computational effort: no correlation matrices are computed and the decision to change linear coefficients is done as each record is parsed rather than by a complete initial pass by the means of records. For well-randomized and large data sets, the user can expect a speed advantage over IRLS.

Distribution Types

The types of distributions governing the “error” of the response variable are the following: Bernoulli, Gaussian (normal), Poisson, binomial, negative binomial, wald (inversegaussian), and gamma.

The Bernoulli, binomial, Poisson, and negativebinomial distributions are nominal, that is, discrete-valued, distributions.

The Gaussian (normal), wald (inverse gaussian) and gamma distributions are continuous, that is, real-valued, distributions.

Bernoulli Distribution

The Bernoulli distribution implies that the response variable takes on values 0 and 1. It models a single toss of a manipulated coin that does not have the same chance of heads and tails. The inverted link function maps θ to the chance of heads.

Binomial Distribution

Binomial distribution is a special case that needs to be treated separately. It implies that there are actually two discrete response variables taking non-negative integers. It models tossing a number of times a manipulated coin that does not have the same chance for heads and tails. The one of the variables counts the number of heads, the other of sum of heads and tails (named trials). The inverted link function maps in this case θ to the “chance” (share) of heads.

Poisson Distribution

The Poisson distribution implies that the response variable takes on non-negative integer values. The inverted link function maps θ to the mean of the variable distributed according to the Poisson distribution. The distribution is driven by the *Poisson process*. In this process the average rate of success over time is known. The probability of a single success within a time interval is proportional to the length of the interval and is independent of the probability outside of the interval. If the interval gets shorter, the probability of more than 1 success goes toward zero.

Negative Binomial Distribution

The Negative Binomial distribution means that the response variable take on non-negative integer values. It models the toss a manipulated coin that does not have the same chance for heads and tails until a predefined number of heads is reached. The response variable is the count of tails. The inverted link function maps θ to the chance of heads.

Gaussian Distribution

The Gaussian distribution, also called the “normal” distribution, is claimed to occur in many natural processes. Specifically, in the case of a multitude of independent processes generating continuous numbers from a distribution, then their average tends to follow normal distribution. The inverted link function maps θ to the mean value of the distribution of the response variable, while the standard deviation of the response variable is deemed to be the same for any value of θ .

Wald Distribution

The Wald distribution, also called the *Inverse Gaussian distribution*, represents the first passage time for Brownian motion. The inverted link function maps θ to the mean value of the distribution of the response variable, while the distribution parameter λ , which influences standard deviation but not the mean of the response variable is deemed to be the same for any value of θ .

Gamma Distribution

The Gamma distribution is the distribution of a sum of random variables that are exponentially distributed. Also exponentially distributed are time intervals between successes in the Poisson experiment. The inverted link function maps θ to the mean value of the distribution of the response variable, while the distribution parameter scale of the response variable is deemed to be the same for any value of θ .

Link Functions

Link functions handle the non-linearity of the relationship between the θ and the mean of response variable (*meanresp*). A number of link functions have been implemented: identity, inverse, logit, log, complementary log-log (cloglog), gaussit, cauchit, sqrt, invsquare, loglog, log-complement (logc), negbin, oddspower and power.

Identity Link Function

The Identity link function is of the form $\theta = meanresp$, that is, the mean of the response variable.

Inverse Link Function

The Inverse link function is of the form $\theta=\frac{1}{meanresp}$.

Log Link Function

The Log link function is of the form: $\theta=\ln(meanresp)$.

Logit Link Function

Logit link function is of the form: $\theta=\ln\left(\frac{\mu}{1-\mu}\right)$.

Probit Link Function

The Probit link function family is understood as an inverse of any cumulative probability distribution function, It is mainly used when the distribution function of GLM is either Bernoulli or binomial. IBM Netezza In-Database Analytics implements cauchit and gaussit here.

Gaussit Link Function

The Gaussit link function, sometimes called probit in a narrow sense, is the standard normal distribution inverse $\theta=qnorm(meanresp)$.

Cauchit Link Function

The Cauchit link function is the inverse of the Cauchy distribution $\theta=\ln(mean)$.

Log-log Link Function

The log-log link function (loglog) is in the form $\theta=-\log(-\log(meanresp))$. This is a “non-canonical” link function and is not commonly used.

Complementary Log-log Link Function

The Complementary log-log link function (cloglog) is of the form $\theta=\log(-\log(1-meanresp))$. This is a “non-canonical” link function and is not commonly used.

Log-complement Link Function

The log-complement link function (logc) is of the form $\theta=\ln(1-meanresp)$. This is a “non-canonical” link function and is not commonly used.

Odds Power Link Function

The odds power link function (oddspow) is of the form $\theta=\frac{\left(\frac{meanresp}{(1-meanresp)}\right)^{\alpha}-1}{\alpha}$ where the α parameter is somehow fixed. For example, α may be set equal to 1. This is a “non-canonical” link function and is not commonly used.

Power Link Function

The power link function is of the form $\theta=meanresp^{\lambda}$, where λ is somehow fixed. This is a “non-canonical” link function and is not commonly used.

Negbin Link Function

The negbin link function is of the form $\theta=\ln\left(\frac{k}{meanresp+1}\right)$, where parameter k is somehow fixed. This is a “non-canonical” link function and is not commonly used.

Input Data Format

The GLM algorithms require an input table containing one row per entity under investigation. Rows must have unique IDs and are deemed to represent independent observations and/or experiments.

A column must be identified that contains values to be used as the response variable. In the case of binomial distribution, two response variables must exist. For continuous distributions, the response variable must to be of type DOUBLE, or able to be converted to DOUBLE. For nominal distributions, INTEGER type should be used.

If not specified explicitly by the user through *“incolumn” (*or globally by *“coldeftype” and “coldefrole”* parameters), numerical predictor variables are treated as continuous, while non-numeric predictor variables are treated as discrete. A user could also define the input table column type and role by the table named and passed by the *colPropertiesTable* parameter. The proper dictionary is automatically created. The user can change these defaults using the column_properties stored procedure. This procedure creates a table by setting the colPropertiesTable parameter.

Rules for removing predictor dimensions for nominal attributes in GLM

If there are k nominal attributes, such as A_1 , ... , A_k, exactly one level predictor dimension is removed from the input table for any subset of the k-1 nominal attributes. Removed vectors are linearly dependent and can be constructed with remaining vectors. More precisely, if exactly one predictor dimension is removed from nominals A_2 , ... , A_k, then the removed level predictor dimension of A_j is equal to the sum of all level vectors of the nominal attribute A_1 minus the remaining level vectors of nominal attribute A_j. The sum of all level vectors of attribute A_1 is the vector of ones, that is, entries that are equal to 1, for example, [1, 1, 1].

This formula works because there must be exactly one cell equal to 1 in each row cast to any nominal attribute. If the intercept column is included in the model, exactly one level predictor dimension is removed from all nominal attributes A_1 , ... , A_k.

The removed level vector of A_j can be constructed in the following way:

The intercept vector, which is the level of ones, minus all remaining level vectors of nominal attribute A_j. Removed level vectors do not change the rank of the input matrix, which means that the quality of the solution is ensured.

Example:

The input table T consists of one nominal **Class** column.

There are three levels of the **Class** column: C1, C2, and C3.

If the build model includes intercept, there will be the following model coefficients:
Class:C2, Class:C3, and intercept.

One predictor, Class:C1, was removed. Note that any of the three predictors Class:C1, Class:C2, and Class:C3 can be removed.

Building a GLM Model

GLM Procedure

The GLM procedure builds the Generalized Linear Model.

PRINT_GLM Procedure

The PRINT_GLM procedure prints the built Generalized Linear Model.

PREDICT_GLM Procedure

The PREDICT_GLM procedure applies the built Generalized Linear Model to generate regression predictions.

Transformation Principles

Transformation principles relate to the values that are supplied for the interaction parameters for the GLM Stored procedure.

The general principle is that the formula of the transformation principle consists of names of predictors, the one argument function name '^' and the ';' and '*' and '=' symbols;

The semi-colon (;) separates factors in the incolumn and/or in the interaction parameter. Within each of group the symbol * separates constituents. A constituent is a predictor name or, in case of continuous only, a predictor name followed by ^ and a natural number.

If the constituent is a name of a continuous predictor, then it represents an expression identical with the name of this predictor, if followed by ^ and a number, the expression predictor raised to the respective power.

If the constituent is a name of a nominal predictor then it represents a set of expressions:

```
if <predictor name>=<level value_i> then 1 else 0 end if
```

for all distinct values of <predictor name> .

The operator * combines two sets of expressions {e1,e2..,en}, {g1,g2,..,gm} to a set of expressions {e1*g1, e2*g2,…,en*gm}.

After applying the * operator within a group of factors, a set of expressions is obtained which is the set of factors within the group.

incolumn Parameter Values

These examples describe the transformation principle permitted in a incolumn parameter. For these examples, let x1,x2,x3… denote continuous predictor names; n1,n2, n3,… denote nominal predictors.

Example 1

```
incolumn= x1;x2;x3
```

The factors used in GLM are x1, x2, x3.

Example 2

```
incolumn= x1:cont;x2:cont;x3:cont
```

The continuous predictors that are used in GLM are x1, x2, 3.

Example 3

```
incolumn= n1:nom;
```

There are as many factors as there are values of n1 in the data; each factor takes the value of 1 or 0, depending on the value of the predictor n1. For example, if n1 takes values va, vb and vc, then n1="va", n1="vb" and n1="vc". If n1="vb" in a record, then the factors take the values: (n1="va")=0, (n1="vb")=1 and (n1="vc")=0.

Example 4

```
incolumn= n1;n2
```

There are as many factors as there are values of n1 and of n2 in the data.

interaction Parameter Values

These examples describe the transformation principle permitted in the interaction parameter. For these examples, let x1, x2, x3… denote continuous predictor names; n1, n2, n3,… denote nominal predictors.

Example 1

```
interaction=x1*x2
```

There is only one additional factor: the product of x1*x2.

Example 2

```
x1*x2;x2*x3
```

There are two additional factors: the product of x1*x2 and product of x2*x3.

Example 3

```
n1*n2
```

This means that there will be as many factors as there are values in the dot product of the sets of values of n1 and of n2 in the data.

Example 4

```
n1*x1
```

There are as many factors as there are values in n1 and each factor takes the value of x1 or of zero, depending on the value of the predictor n1. For example, if n1 takes values va, vb and vc, then the factors are (n1="va")*x1, (n1="vb")*x1 and (n1="vc")*x1. If n1="vb" in a record, and x1=133, then the factors take the values 0, 133, 0 respectively.

Example 5

```
n1*x1^2
```

There are as many factors as there are values in n1 and each factor takes the value of x1 or of zero, depending on the value of the predictor n1. For example, if n1 takes values va, vb and vc, then the factors are (n1="va")*(x1^2), (n1="vb")*(x1^2) and (n1="vc")*(x1^2). If n1="vb" in a record, and x1=133, then the factors take the values 0,017689,0 respectively.

Examples

The functionality of the Generalized Linear Model implementation is illustrated by the examples using the *WineQuality* data set.

Consider creating a GLM model for predicting the quality attribute from the *WineQuality* data set:

```
CALL nza..glm('intable=nza..WineQuality_train, incolumn=
fixed_acidity ; volatile_acidity ; citric_acid ; residualsugar ; chlorides
; free_sulfur_dioxide ; total_sulfur_dioxide ; density ; ph ; sulphates ;
alcohol ,
 intercept=TRUE, id=id, target=quality, model=glm_wq, maxit=2,
coldefrole=ignore
, family=gaussian, link=identity');
```

We have assumed there that the error distribution is gaussian and the link function is identity.

The GLM model coefficients and the values of diagnostic measures can be inspected by calling the PRINT_GLM procedure:

```
call nza..PRINT_GLM('model=glm_wq');
```

We notice here that the column citric_acid and total_sulfur_dioxide do not play any role for the model as their p-values are far beyond 0.05.

The constructed GLM model can be applied to new data using following call:

```
CALL nza..PREDICT_GLM('intable=nza..WineQuality_test, id=id, model=glm_wq,
outtable=out_wq');
```

Consider now creating a GLM model for predicting the quality attribute from the *WineQuality* data set when we restrict to only quality levels 5 (which we encode 0) and 7 (which we encode 1). Note that we first drop the formerly created tables and models.

```
call nza..drop_model('model=glm_wq');
drop table out_wq;

create table wq57 as select *,case when quality=5 then 0 else 1 end
qdiff from nza..WineQuality_train where quality=5 or quality=7;
CALL nza..glm('intable=wq57, incolumn=fixed_acidity ; volatile_acidity ;
residualsugar ; chlorides ; free_sulfur_dioxide ;
density ; ph ; sulphates ; alcohol , intercept=TRUE, id=id, target=qdiff,
model=glm_wq, maxit=2, coldefrole=ignore, family=bernoulli, link=logit');
```

The GLM model coefficients and the values of diagnostic measures can be inspected by calling the PRINT_GLM procedure:

```
call nza..PRINT_GLM('model=glm_wq');
```

The constructed GLM model can be applied to new data using following call:

```
create table wq57ts as select *,case when quality=5 then 0 else 1 end qdiff
from nza..WineQuality_test where quality=5 or quality=7;
CALL nza..PREDICT_GLM('intable=wq57ts, id=id,
model=glm_wq, outtable=out_wq');
```

To see how well the prediction fits the true value, run:

```
select count(*), qdiff, round(pred) as predicted from wq57ts t join out_wq
p on t.id=p.id group by qdiff,round(pred) order by qdiff,round(pred);
```

The functionality of the model with interactions is illustrated in the examples below.
First a multiplicative interaction:

```
CALL nza..glm('intable=wq57, incolumn=
fixed_acidity ; volatile_acidity ; residualsugar ; chlorides ;
free_sulfur_dioxide ; density ; ph ; sulphates ; alcohol, interaction=density *
sulphates , intercept=TRUE, id=id, target=qdiff, model=glm_wq_i1, maxit=2,
coldefrole=ignore, family=bernoulli, link=logit');
```

```
CALL nza..PREDICT_GLM('intable=wq57ts, id=id, model=glm_wq_i1,
outtable=out_wq_i1');
select count(*), qdiff, round(pred) as predicted from wq57ts t join out_wq_i1
p on t.id=p.id group by qdiff,round(pred) order by qdiff,round(pred);
select count(*), qdiff,pred from wq57ts t join out_wq p on t.id=p.id group by
qdiff,round(pred) order by qdiff,round(pred);
```

The following example with powers creates two new columns—cube of density and square of sulphates.

```
CALL nza..glm('intable=wq57, incolumn=
fixed_acidity ; volatile_acidity ; residualsugar ; chlorides ;
free_sulfur_dioxide ;  density ;  ph  ; sulphates ; alcohol
, interaction=density^3;  sulphates^2 ,
 intercept=TRUE, id=id, target=qdiff, model=glm_wq_i2, maxit=2,
coldefrole=ignore
, family=bernoulli, link=logit');
```

Following is another example of power transformations:

```
CALL nza..glm('intable=wq57, incolumn=
  volatile_acidity ; residualsugar ;  free_sulfur_dioxide ;
density ; alcohol; fixed_acidity ; residualsugar,
interaction=fixed_acidity^2 ; residualsugar^2
,
 intercept=TRUE, id=id, target=qdiff, model=glm_wq_i3,
maxit=2, coldefrole=ignore
, family=bernoulli, link=logit');
```

Output Table Data Formats

As a result, when the GLM algorithms run, the following tables are generated:

- NZA_META_<model_name>_MODEL Table
- NZA_META_<model_name>_FADIC Table
- NZA_META_<model_name>_DICTIONARY Table
- NZA_META_<model_name>_GLMDIC Table
- NZA_META_<model_name>_PCOVMATRIX Table
- NZA_META_<model_name>_PPMATRIX Table
- NZA_META_<model_name>_RESIDUALS Table
- NZA_META_<model_name>_STATS Table

NZA_META_<model_name>_MODEL Table

The NZA_META_<model_name>_MODEL table contains details about the linearly combined factors of the GLM model. Following are the table columns.

Table 59: Columns of the NZA_META_<model name>_MODEL table

Column	Data Type	Purpose
FAC_ID (primary key column)	INTEGER	Identifier of the factor.
BETA	DOUBLE	The beta value of the linear part.
STD_ERROR	DOUBLE	The standard deviation of beta value.
TEST	DOUBLE	Test statistics for beta significance (difference from zero).
P_VALUE	DOUBLE	Significance of the above test statistics.
CLASS_ID	INTEGER	Class, if the parameter (predictor variable) is discrete.
DF	BIGINT	Degrees of freedom.

NZA_META_<model_name>_FADIC Table

The NZA_META_<model_name>_FACDIC table contains the list of all factors. Following are the table columns.

Table 60: Columns of the NZA_META_<model name>_FADIC table

Column	Data Type	Purpose
FAC_ID (primary key column)	INTEGER	Identifier of the factor.
FAC_EXPRESSION	NVARCHAR	An expression representing the factor. The expression may be derived from the PPMATRIX table.

NZA_META_<model_name>_DICTIONARY Table

The NZA_META_<model_name>_DICTIONARY table contains information about the input variables. Following are the table columns.

Table 61: Columns of the NZA_META_<model name>_DICTIONARY table

Column	Data Type	Purpose
ATTNUM (primary key column)	SMALLINT	Column ID of the input variable.

Column	Data Type	Purpose
ATTNAME	NVARCHAR	Name of the variable/column.
DICNAME	NVARCHAR	Name of the table that contains the dictionary for the variables. Only used for of nominal variables.
LEVELS	BIGINT	Number of levels. Only used for nominal variables.
ATTAVG	DOUBLE	Average value. Only used for continuous variables.
ATTSTD	DOUBLE	Standard deviation. Only used for continuous variables.

NZA_META_<model_name>_GLMDIC Table

The NZA_META_<model_name>_GLMDIC table contains the list of all factors. Following are the table columns.

Table 62: Columns of the NZA_META_<model name>_GLMDIC table

Column	Data Type	Purpose
GLM_ID (primary key column)	INTEGER	Column ID of the variable (GLM run specific).
NAME	NVARCHAR	Name of the variable/column.
COLROLE	BOOLEAN	True=predictor, FALSE=response variable.

NZA_META_<model_name>_PCOVMATRIX Table

The NZA_META_<model_name>_PCOVMATRIX table contains the covariance values. Following are the table columns.

Table 63: Columns of the NZA_META_<model name>_PCOVMATRIX table

Column	Data Type	Purpose
FAC_RAW (primary key column)	INTEGER	Factor ID or a special number identifying the target variable.
FAC_COL (primary key column)	INTEGER	Factor ID or a special number identifying the target variable

Column	Data Type	Purpose
VALUE	DOUBLE	Covariance between factors indicated at row and column IDs.

NZA_META_<model_name>_PPMATRIX Table

The NZA_META_<model_name>_PPMATRIX table defines the factors IDs. Following are the table columns.

Table 64: Columns of the NZA_META_<model name>_PPMATRIX table

Column	Data Type	Purpose
FAC_ID (primary key column)	INTEGER	Identifier of the factor.
GLM_ID	INTEGER	GLM specific identifier – the “covariate” (only predictors used here).
POWER	DOUBLE	If the covariate is unrelated to factor or is not continuous, then the cell is set to NULL Otherwise cell value is set to the exponent that the covariate is raised to in the dependency expression. Example: Let the GLM_ID identify the continuous *work* variable. Let FAC_ID identify a factor of the form if [jobcat=”professional”) then 1 else 0 endif * work ^2 Therefore, this factor is correlated to the *work* covariate, and the number that should be entered in the cell is 2 because *work* is present at second power in the expression.
LEVEL	INTEGER	If the covariate is unrelated to factor or is not nominal, then the cell is set to NULL. Otherwise, the cell value is set to the level of the covariate used in the dependency expression. Example: Let the GLM_ID identify the nominal variable jobcat, with levels professional, clerical, skilled, unskilled, which got the level identifiers 1,2,3,4 respectively in the dictionary _DIC. Let FAC_ID identify a factor of the form if [jobcat=”professional”) then 1 else 0 endif * work ^2 Therefore, this factor is correlated to the covariate

Column	Data Type	Purpose
		jobcat, and the number that should be entered in the cell is 1 because jobcat is professional in the dependency expression.

NZA_META_<model_name>_RESIDUALS Table

The NZA_META_<model_name>_RESIDUALS table contains the covariance values. Following are the table columns.

Table 65: Columns of the NZA_META_<model name>_RESIDUALS table

Column	Data Type	Purpose
RECORDID (primary key column)	INTEGER RESIDUALS	Observation unique identifier.
RAW	DOUBLE	So-called response residual for each observation, that is, the difference between observed value and fitted (predicted by model) value.
PEARSON	DOUBLE	So-called Pearson residual for each observation, that is, the response residual normalized with the estimated standard deviation for the observation.
DEVIANCE	DOUBLE	So-called Deviance residual for each observation, that is, the **sign**([response residual]) * **sqrt**([observation weight] * **deviance**([real value],[fitted value]))

NZA_META_<model_name>_STATS Table

The NZA_META_<model_name>_STATS table contains statistic information. Following are the table columns.

Table 66: Columns of the NZA_META_<model name>_STATS table

Column	Data Type	Purpose
STATISTIC	NVARCHAR	Name of the computed statistics.
VALUE	DOUBLE	Value of the computed statistics.

CHAPTER 25
Model Deployment

Model Deployment provides the ability to export analytics models from one Netezza database and import them into a database residing on a physically separate Netezza system. This functionality is crucial for projects such as rolling out one or more analytics models from a development or validation system to a production system. (While the systems need not be physically separate, there are simpler mechanisms for copying models between databases that reside on the same appliance.)

Background

The basis of the model deployment functionality is IBM Netezza Analytics model management, which gives users control over model administration. Any user who has the necessary privileges to read analytics models on the source system and to create analytics models on the target system can transfer models between the two Netezza systems. As a result, model deployment is not confined to database administrators; any user can have privileges to deploy, with the IBM Netezza Analytics security architecture controlling model access.

Netezza analytics also provides functionality to export analytic models into PMML files. PMML focuses on the exchange of data mining models between different products, however, and therefore is not the best choice for model deployment, even if an import functionality were available. (See PMML Support, for more information on PMML in Netezza Analytics.) Model deployment requires an exact copy of all types of models (model content, model metadata, and model privileges). PMML may not work in conjunction with all types of Netezza analytics-supported algorithms, resulting in certain metadata conversion into PMML to require proprietary PMML extensions. Furthermore, converting models to and from PMML is time-consuming. For these reasons, Netezza analytics includes a model deployment solution based on a proprietary import/export format.

The model deployment functionality is available using stored procedures and a shell script. The shell script provides functionality above that which can be provided by stored procedures (for example, file operations). The procedures and command line utilities can:

export one or more analytic models to file(s)

inspect the contents of these file(s)

import one or more analytic models from these file(s) into the target database. The model import and export comprises the model contents as well as their metadata.

All types of analytic models, provided they have been registered in the metadata management system, can be imported and exported. The contents of all managed components of a model is included in the exported files. Note that the source (export) and target (import) Netezza systems must have the same system case, (that is, both must be lowercase systems or both must be uppercase systems).

Permissions and Access

Because database security and file system security are two independent security systems, the security functions of the database cannot be reflected on the file system level. For example, while access to models is restricted to database users with the corresponding privileges, access to file contents is restricted to system users with the corresponding permissions. Therefore, once a model is exported to file(s), users other than the database users may have access to the model data. Any user who can read the exported file(s) can list the models in these files or import them into a target database.

Model privileges can also be exported and imported, but only by the database administrator.

Applications

Movement of one or more analytic models from a test system to a physically separated production system is a typical usage scenario for the model deployment functionality. In simpler cases, where both databases are within the same appliance, you would use the COPY_MODEL procedure. However, the COPY_MODEL procedure cannot be used to transfer models to a target database when the systems are physically separated. The following is a sample use case for model deployment.

Sample Deployment Scenario

A data mining analyst is using Netezza analytics to analyze customer data and is ready to deploy analytic models produced in a test environment into a production environment. The production environment is physically separated from the test environment. Simply, the analyst needs to export the models on the test system, move the exported file(s) to the production system, import the models on to the target, and finally to manage privileges.

The steps for this scenario are:

On the test system, the analyst exports all models to be deployed to a set of files.

Using the transport medium of choice (network, memory stick, etc.), the analyst moves the file(s) to the production system.

On the production system, the analyst imports all or selected models from the file(s) into the Netezza analytic-enabled production database. Optionally, models can be overwritten on model import.

The analyst evaluates and rectifies model privileges. By default, the user who initiates the export and import is the new owner of the models. If that user is a database administrator, all model privileges can be transferred from the source to the target system. If alternate privileges are desired, the administrator can issue individual GRANT_MODEL or REVOKE_MODEL calls.

Available Functionality

The implementation of the model deployment function is provided by the **EXPORT_MODEL**, **IMPORT_MODEL**, and LIST_MODEL procedures, as well as an included shell script. Each is described below.

NZA..EXPORT_MODEL

The NZA..EXPORT_MODEL procedure allows you to specify one or more analytics models to be exported (using the **model** parameter). When specifying more than one model name, each model must be separated with a semicolon. Only models from the current database can be exported. Mixed case model names must be double-quoted; unquoted names are converted to system case. If model=all is specified, all models in the current database are exported. To export a model, the user must have the LIST and SELECT privileges on the model.

You can also use a **where** parameter to filter models to be exported from the current database. All columns available in the V_NZA_MODELS view can be used in the where parameter .

When specifying the **directory** parameter (mandatory), an existing directory where the export files are stored, note that the directory must be writable by the database process (owned by user nz). The directory must be specified as absolute path name, relative path names are not allowed.

For each model, the export procedure writes a series of files (a file set), dependent on the number and kind of components of the model. Because it is possible that the directory will receives hundreds of files, it is best that you specify an empty directory. To simplify organization, all export files are prepended with a common prefix (based on the mandatory **name** parameter). By making this prefix different for each call of the EXPORT_MODEL procedure, you can reuse the directory for different calls of EXPORT_MODEL.

Note: While the EXPORT_MODEL procedure does not produce a single archive, the shell script interface, described below, can combine all exported files into a single archive file.

The ***acl*** parameter is used to export model privileges also. This parameter can be used by the ADMIN user only.

The **overwrite** parameter determines if conflicting files in the specified directory are deleted before the new files are written. If the directory already contains a file set with the same name as the file set to be written, this existing file set is deleted if overwrite = true, or the export is aborted if overwrite = false.

The EXPORT_MODEL procedure writes several files into a user-specified directory by a process belonging to the database. This process is owned by user nz and, consequently, the files are owned by nz. For this reason, user nz must have the right to write into the specified target directory. In addition, by default the written files have open read access. If this is not desired, you must change the file permissions manually when EXPORT_MODEL has finished.

The procedure returns the number of models that were exported successfully.

NZA..IMPORT_MODEL

The NZA..IMPORT_MODEL procedure allows you to specify one or more analytics models to be imported (model parameter). When specifying more than one model name, they must be separated with semicolons. Models are always imported into the current database. Mixed case model names must be double-quoted;unquoted names are converted to system case. (Note that the system case of the export machine must match the system case of the import machine.) If model=all is specified, all models in the file set are imported.

You can also use the **where** parameter to filter models to be imported into the current database . All columns available in the V_NZA_MODELS view can be used in the where parameter.

Use the **directory** parameter (mandatory) to specify the directory that contains the files that were previously exported. The directory must be specified as an absolute path name.

Use the **name** parameter to identify the desired set of files. Each file set written into a directory can be identified by its name (that is, the prefix prepended during export). If the specified directory contains only the result of one EXPORT_MODEL operation, the file set can be clearly detected and you need not specify the name parameter.

Use the **acl** parameter to import model privileges (provided they were exported before). This parameter can be used by the ADMIN user only. If a user or group whose privileges are to be imported does not exist, the privileges of this user or group are not imported. Note that users and groups are not created automatically.

By default, models in the current database are not overwritten on model import. If you do wish to overwrite models, set the **overwrite** parameter to true. Models with the same name as an imported model are deleted before the model is imported. Any user who can create a new analytics model in a database can also import models from the file set into that database. The current user becomes the owner of the imported models. To overwrite a model, a user needs the DROP privilege on the model.

If the ADMIN user calls the import procedure, the ADMIN can specify the owner parameter. This parameter sets an alternative owner for all imported models; the new owner must exist on the target system.

The procedure returns the number of models that were imported successfully.

NZA..LIST_MODELS

The LIST_MODELS procedure used for model deployment is the same as that for the rest of the Netezza analytics, with the additional ability to list also models from a file set. For model deployment, therefore, LIST_MODELS has two additional parameters. The **directory** parameter, when specified, activates the deployment view for the procedure. The directory must be specified as an absolute path name. The **name** parameter identifies the file set in this directory; if the specified directory contains only the result of one EXPORT_MODEL operation, the file set can be clearly detected and you need not specify the name parameter. The LIST_MODELS procedure lists the names and some properties of the models in the file set. No special privileges are needed to list the models.

LIST_MODELS cannot list the models in an archive file created by the shell script (described below). To list archived models, use the list mode of the shell script.

Shell Script

The model deployment functionality includes a command line utility, **nzmodel**, that can perform the export and import of models and display the contents of an export file set. In addition, it can create an archive file instead of file sets. It can perform the model export or model import from outside the database, and provides these two features not available with the stored procedure:

Using the **-f** parameter, the result of the export process can be a single file, not a file set. This single archive file contains all files from the file set and can be imported directly without manually extracting the file set. This not only simplifies the transfer of the export result to another location, but also better preserves the integrity of the file set.

Copy mode allows models to be copied directly from one Netezza host to another, without any intermediate files.

Restriction: Only the Linux user who started the NPS database may execute the **nzmodel** script. Typically, this user is the **nz** user.

The following is the syntax for the shell script:

```
nzmodel -e [-avo] [-z netezza-host] [-d database] [-u user]
[-p password]
           [-w where-clause] [-f file/directory] [-n name] [model...]
nzmodel -i [-avo] [-z netezza-host] [-d database] [-u user] [-p password]
           [-w where-clause] [-O owner] [-f file/directory] [-n name]
[model...]
nzmodel -l [-v] [-f file/directory] [-n name]
nzmodel -c [-avo] [-z source-netezza-host] [-d source-
           database] [-u source-user] [-p source-password]
           [-Z target-netezza-host] [-D target-database]
           [-U target-user] [-P target-password]
           [-w where-clause] [-O owner] [model ...]
nzmodel -h
```

The following table defines the parameters of the shell script:

Table 67: Options for the model deployment shell script

Option /alt. name	Explanation
-e / --export	Export model(s) to an archive file or directory. See Export (option -e) for more detail.
-i / --import	Import model(s) from an archive file or directory. See Import (option -i) for more detail.
-l / --list	List file or directory contents (model names and properties).See List (option -l) for more detail.

Option /alt. name	Explanation
-c / --copy	Copy models from the local database/host to another database/host. See Copy (option -c) for more detail.
-h / --help	Show help.
-a / --acl	Export/import model privileges with the model (ADMIN only).
-o / --overwrite	Overwrite output files (export) or models in the database (import).
-v / --verbose	Display verbose mode.
-d / --db	Name of the (source) database.
-D / --tdb	Name of the target database.
-z / --host	Host name or address of the (source) database.
-Z / --thost	Host name or address of the target database.
-p / --password	Password for the (source) database.
-P / --tpassword	Password for the target database.
-u / --user	User name for the (source) database.
-U / --tuser	User name for the target database.
-w / --where	A where-clause to filter models for export from the source database or models for import from the archive file or directory.
-O / --owner	The owner for all imported models; by default the importing user is the model owner.
-f / --file	The name of the archive file ~~name~~ used for export or import (tgz format). Alternatively an (existing) directory name can be specified.
-n / --name	A name for the file set (only if a directory is specified).

Export, Import, List, and Copy Option Details

The following sections provide additional detail on the options.

Export (option -e)

In export mode, the script exports one or more models to a single file or to a file set in a directory.

(see the description of NZA..EXPORT_MODEL for more information). It is possible to export models from a database on another host (using -z). Note that in this case, all resulting files are written on that host, not on the local Netezza system. If one of the parameters (host, database, user, or password) is not specified, its value is taken from the corresponding environment variable.

The models to be exported can be specified using a **where** clause, or their names can be listed at the end of the command line (space-separated). If “all” is specified in the model name list, all models are exported. The final list of exported models is constructed by merging the model set specified by the where clause with the models specified by name. The where clause and the list of model names can select only models in the current database (option -d). The where clause can be constructed in the same way as the where parameter in the NZA..LIST_MODELS procedure.

The result can be written to an archive file, or a set of single files. (The archive file is a zipped form of the file set.) If an existing directory is specified with the **-f** option, the result is written as a set of files into that directory. In that case, you must also specify the **-n** option to define a name for the file set. In all other cases, one archive file is written. If the specified file already exists, it is either overwritten without notice (if the **-o** option was specified) or the user is asked if the file should be overwritten.

Import (option -i)

In import mode, the script imports one or more models from a single file or a file set into a database (see the description of NZA..IMPORT_MODEL for more information) .It is possible to import models into a database on another host (using -**z**). Note that in this case, all import files must exist on that host, not on the local Netezza system. If one of the parameters (host, database, user, or password) is not specified, its value is taken from the corresponding environment variable.

The models to be imported can be specified using a **where** clause, or their names can be listed at the end of the command line (space-separated). If “all” is specified in the model name list, all models are imported. The final list of imported models is constructed by merging the model set specified by the where clause with the models specified by name. Models can be imported only in the specified import database (option **-d**), regardless of the original export database. The where clause can be constructed in the same way as the where parameter in the NZA..LIST_MODELS stored procedure.

The file/directory parameter (option **-f**) can be any of the following:

An archive file created by a model export operation.

A directory that contains an exported file set. If the directory contains more than one exported file set, you must specify the **-n** parameter must to select the desired file set. If you do not know the name of the exported file set, you can derived it from the file names (because the name of a file set is part of each file name).

The import can fail if an imported model has the same name as an existing model. In this case it fails if the overwrite option is not specified or the user does not have privileges to drop the existing model.

List (option -l)

In list mode, the script lists the names and some properties of the models in the archive file or the file set. You can specify the file/directory and the name parameter in the same way as you would with import mode.

Copy (option -c)

In copy mode, the script copies one or more models from a database on the local host directly to a database on another host. The models can be specified in the same way as with the EXPORT_MODEL procedure, using the where clause and/or a list of model names. Although the copy mode allows you to specify a source host (option -z), currently the source host must always be the local host.

Examples

Stored Procedure Examples

To export the two models – DECTREE_MODEL and LinRegModel – into the existing directory /tmp/export, you execute:

```
call nza..export_model('model=dectree_model;"LinRegModel",

                        directory=/tmp/export, name=production');
```

The names of all created files in /tmp/export start with **production** (as specified by the **name** parameter).

The export command above does not export model privileges. If you are logged in as user ADMIN, you can also export privileges. Using the **overwrite** option replaces the existing file set with a new one:

```
call nza..export_model('model=dectree_model;"LinRegModel",
                        directory=/tmp/export, name=production,
                        overwrite=true, acl=true, verbose=true');
```

Suppose you have several models whose names starts with DECTREE. If you want to export them and also the model LinRegModel, you could do so as follows:

```
call nza..export_model('model="LinRegModel", directory=/tmp/export,
                        where=MODELNAME LIKE ''DECTREE%'',
                        name=production, overwrite=true');
```

The model sets defined by the **model** and the **where** parameter are unified.

To list the models exported to /tmp/export, use this command:

```
call nza..list_models('directory=/tmp/export');
```

To import the exported models into another database, log in to target database and call the IMPORT_MODEL procedure:

```
call nza..import_model('model=dectree_model;"LinRegModel",

                        directory=/tmp/export');
```

Note that you do not need to specify the **name** parameter, because /tmp/export contains only one export file set (“production”).

The import command above does not import model privileges. If you are logged in as user ADMIN, you can also import privileges (provided you have exported them previously). Use the overwrite option to overwrite models:

```
call nza..import_model('model=dectree_model;"LinRegModel",
                        directory=/tmp/export,
                        overwrite=true, acl=true, verbose=true');
```

If the target machine has a user PETER, you can make PETER the owner of all imported models (you must be logged in as ADMIN):

```
call nza..import_model('model=dectree_model;"LinRegModel",
                        directory=/tmp/export, owner=peter,
                        overwrite=true, acl=true, verbose=true');
```

Shell Script Examples

To export the two models DECTREE_MODEL and LinRegModel, located in database TEST, into the existing directory /tmp/export, you execute:

```
nzmodel -e -d test -f /tmp/export -n product

        dectree_model \"LinRegModel\"
```

The names of all created files in /tmp/export start with **product**. Note that default values for the host, the user, and the password are taken from environment variables NZ_HOST, NZ_USER, NZ_PASSWORD.

Since the Linux shell interprets double quotes itself, you must use the escape char (\) to prevent this, since the double quotes are needed in the database for mixed case names.

Instead of writing a file set, you can also create a single archive file:

```
nzmodel -e -d test -f /tmp/export/product.tgz
        dectree_model \"LinRegModel\"
```

Here, only the file product.tgz is created.

The following lists the models exported to /tmp/export:

```
nzmodel -l -f /tmp/export
```

The following lists the models exported to /tmp/export/product.tgz:

```
nzmodel -l -f /tmp/export/product.tgz
```

The following imports the models exported to /tmp/export into the database TESTIM:

```
nzmodel -i -d testim -f /tmp/export dectree_model \"LinRegModel\"
```

The following imports **all** models exported to /tmp/export/product.tgz into the database TESTIM:

```
nzmodel -i -o -d testim -f /tmp/export/product.tgz all
```

You can use the copy option to copy models directly from the local host/database to another host/database:

```
nzmodel -c -d test -H productionhost -D testim -U admin -P

        password dectree_model \"LinRegModel\"
```

Here we specified the username and password of the target machine (productionhost)explicitly. This command will work only if you can use nzsql to connect to the target machine.

CHAPTER 26

PMML Support

Overview

The Predictive Model Markup Language (PMML) is widely accepted as the standard for exchange of data mining models and is defined by the Data Mining Group (DMG).[2] Netezza Analytics PMML supports export of *k*-means clustering models, and limited support for decision trees, association rules, and naïve Bayes algorithms

PMML uses XML to represent mining models. The structure of the models is described by an XML Schema. One or more mining models can be contained in a PMML document. Consumers typically apply imported models to their own data or analyze the models. An example could be to visualize model contents.

PMML General Information

The following sections provide general information on Netezza Analytics PMML support.

PMML Header Element

Any PMML model requires a *version* attribute as well as a *Header* element and it may optionally have a *MiningBuildTask* element. Following is the PMML definition of Header:

```
<xs:element name="Header">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Extension" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element minOccurs="0" ref="Application"/>
      <xs:element minOccurs="0" maxOccurs="unbounded"
                  ref="Annotation"/>
      <xs:element minOccurs="0"
    ref="Timestamp"/> </xs:sequence>
    <xs:attribute name="copyright" type="xs:string" use="required"/>
```

[2] Details can be found at http://dmg.org/.

```
    <xs:attribute name="description"
  type="xs:string"/> </xs:complexType>
</xs:element>
```

The header of Netezza PMML models contains the following values:

Table 68: Header values of Netezza PMML models

Field	Value
Extension	None
Application	<Application name="**IBM Netezza Analytics**" version="*x.**x***" /> (where *x.x* is the version of Netezzza Analytics running on the system)
Annotation	None
Timestamp	<Timestamp>*model creation time*</Timestamp>, where *model creation time* is DATECREATED from NZA_METADATA_MODELS
Copyright	"Copyright *copyright statement* All Rights Reserved", where *copyright statement* is COPYRIGHT from NZA_METADATA_MODELS
Description	"*description*", where *description* is DESCRIPTION from NZA_METADATA_MODELS

MiningBuildTask Element

The PMML element MiningBuildTask consists only of Extensions and defines no semantics. It is meant to describe the configuration of the training run that produced the model. Netezza PMML models contain the following:

```
<MiningBuildTask>
  <Extension name=params>
    <X-Parameter … />
    …
  </Extension>
  <Extension name=colparams>
    <X-ColumnProperty … />
    …
  </Extension>
</MiningBuildTask>
```

X-Parameter and X-ColumnProperty Elements

The X-Parameter element is defined as follows:

```
<xs:element name="X-Parameter">
  <xs:complexType>
    <xs:attribute name="taskseq"  type="xs:integer" use="optional"/>
    <xs:attribute name="name"     type="xs:string"  use="required"/>
    <xs:attribute name="type"     type="xs:string"  use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

The X-ColumnProperty element is defined as follows:

```
<xs:element name="X-ColumnProperty">
  <xs:complexType>
    <xs:attribute name="colname" type="xs:string" use="required"/>
    <xs:attribute name="property" type="xs:string" use="required"/>
    <xs:attribute name="type"     type="xs:string"  use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

The data are those contained in NZA_META_PARAMS and NZA_META_COLPROPS. If colname is *, the properties pertain to all columns for which an X-Column-Property is not in the PMML document.

Data Dictionary

Any PMML must have a data dictionary describing all available input columns. Netezza PMML models simply contain a list of DataField elements. Each of them contains the following attributes:

```
<xs:attribute name="name" type="FIELD-NAME" use="required"/>
<xs:attribute name="optype" type="OPTYPE" use="required"/>
<xs:attribute name="dataType" type="DATATYPE" use="required"/>
```

The following type definitions apply:

```
<xs:simpleType name="OPTYPE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="categorical"/>
    <xs:enumeration value="ordinal"/>
    <xs:enumeration value="continuous"/>
  </xs:restriction>
</xs:simpleType>
```

Of the DATATYPE values defined in PMML, only the ones described below are used:

```
<xs:simpleType name="DATATYPE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="string"/>
    <xs:enumeration value="integer"/>
    <xs:enumeration value="float"/>
    <xs:enumeration value="double"/>
    <xs:enumeration value="boolean"/>
    <xs:enumeration value="date"/>
    <xs:enumeration value="time"/>
    <xs:enumeration value="dateTime"/>
  </xs:restriction>
</xs:simpleType>
```

If column statistics are contained in the model, continuous fields also contain Interval elements, categorical fields contain Value elements listing all valid values.

Of the possible subelements and attributes of Interval defined in PMML, only the ones described below are used:

```
<xs:element name="Interval">
  <xs:complexType>
    <xs:attribute name="closure" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="openClosed"/>
          <xs:enumeration value="openOpen"/>
          <xs:enumeration value="closedOpen"/>
          <xs:enumeration value="closedClosed"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="leftMargin" type="NUMBER"/>
    <xs:attribute name="rightMargin" type="NUMBER"/>
  </xs:complexType>
</xs:element>
```

Of the possible subelements and attributes of Value, only the ones described below are used:

```
<xs:element name="Value">  <xs:complexType>
    <xs:attribute name="value" type="xs:string"
  use="required"/> </xs:complexType>
</xs:element>
```

Note: The order of intervals and values that is defined in the data dictionary defines also the order of statistics values in succeeding statistics elements.

Transformation Dictionary and Local Transformations

Netezza PMML models do not contain transformation dictionaries. Netezza PMML models may contain local transformations. The details pertaining to k-means local transformation are described in the k-means section, Local Transformations.

Mining Models

The main section in a PMML model consists of one or more models of particular types. The XML element for each of the model types starts with an optional Extension elements, designed to hold proprietary (non-standard) information that must not be relevant for scoring. Netezza Analytics uses the models' Extension elements to hold model metainformation. The extension looks as follows:

```
<Extension extender='metadata' name='<item>' value='<val>'/>
where <item> can be any one of the column names in V_NZA_MODELS of which
the information is not otherwise contained in the PMML document, such as
USERCATEGORY. <val> is the corresponding value.
```

Clustering Model

Netezza clustering models have the following form:

```
<xs:element name="ClusteringModel">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Extension" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="MiningSchema"/>
      <xs:element ref="Output" minOccurs="0" /> <xs:element
      ref="ModelStats" minOccurs="0"/> <xs:element
      ref="ModelExplanation" minOccurs="0"/> <xs:element
      ref="LocalTransformations" minOccurs="0" /> <xs:element
      ref="ComparisonMeasure"/>
      <xs:element ref="ClusteringField"
                               minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="Cluster" maxOccurs="unbounded"/>
      <xs:element ref="ModelVerification" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="modelName" type="xs:string"
                                 use="optional"/>
      <xs:attribute name="functionName" type="MINING-FUNCTION"
                                 use="required"/>
      <xs:attribute name="algorithmName" type="xs:string"
                               use="optional"/>
      <xs:attribute name="modelClass"
        use="required"> <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="centerBased"/>
            <xs:enumeration value="distributionBased"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="numberOfClusters" type="INT-NUMBER"
                               use="required"/>
    </xs:complexType>
  </xs:element>
```

The generic elements are described in the following subsections. Netezza generates k-means models for clustering. For details of the k-means models, see K-means Clustering.

Mining Schema and Mining Field

Most data mining models contain a mandatory MiningSchema element describing the subset of the data dictionary actually used by the model as well as further column properties.

The mining schema consists of a set of MiningField elements defined in the following way by PMML:

```
<xs:element name="MiningField">
  <xs:complexType>
    <xs:attribute name="name" type="FIELD-NAME" use="required" />
    <xs:attribute name="usageType" type="FIELD-USAGE-TYPE"
                                default="active" />
    <xs:attribute name="optype" type="OPTYPE" /> <xs:attribute
    name="importance" type="PROB-NUMBER" /> <xs:attribute
    name="outliers" type="OUTLIER-TREATMENT-METHOD"
```

```
                                default="asIs" />
    <xs:attribute name="lowValue" type="NUMBER" />
    <xs:attribute name="highValue" type="NUMBER" />
    <xs:attribute name="missingValueReplacement" type="xs:string" />
    <xs:attribute name="missingValueTreatment"
                                type="MISSING-VALUE-TREATMENT-METHOD" />
    <xs:attribute name="invalidValueTreatment" type="INVALID-VALUE-
                                TREATMENT-METHOD"
                                default="returnInvalid" />
  </xs:complexType>
</xs:element>
```

Besides all required attributes, Netezza PMML provides optype, importance, and missingValueTreatment. The description of the importance attribute and the missingValueTreatment attribute is provided independently for each of the algorithms.

Output Fields

The output fields describe a set of result values that can be computed when the model is applied to new data. In particular, the output fields specify names, types and rules for selecting specific result features. This information can be used by a scoring engine to determine the results that can be calculated from the model.

The Output element consists of a set of OutputField elements. See Output Fields in the k-means section for more information.

Model Statistics

PMML models may contain statistical information about some of the input fields in an element. The model statistics consists of a set of UnivariateStatistics elements. The description of which univariate statistics are contained in a k-means model is described in Model Statistics.

Model Explanation

Netezza may generate ModelExplanation elements. See the k-means Model Explanation for more details.

Model Verification

Model verification is intended to allow scoring engines to verify that their scores are correct, or rather identical to those intended by the model producer. Model verification is optional.

K-means Clustering

The PMML functionality supported by IBM Netezza In-Database Analytics allows users to export the data for K-Means models.

Clustering Model

The **ClusteringModel** element describes the clustering model as a set of clusters. This includes information needed for scoring; *k*-means needs, in particular, the center vectors of each cluster. Details of the sub-elements contained in **ClusteringModel** are described in the subsequent sections of this chapter.

ClusteringModel may have the following attributes:

```
<xs:attribute      name="modelName" type="xs:string" use="optional"/>
<xs:attribute      name="functionName" type="MINING-FUNCTION"
use="required" />
<xs:attribute      name="algorithmName" type="xs:string" use="optional"/>
<xs:attribute name="modelClass"
  use="required"> <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="centerBased"/>
      <xs:enumeration value="distributionBased"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute      name="numberOfClusters" type="INT-NUMBER" use="required"/>
```

Table 69: ClusteringModel attributes

Attribute	Description
modelName	the name string provided by the user for the model, when calling the *k*-means function
functionName	is "clustering"
algorithmName	is "Kmeans"
modelClass	is "centerBased"
numberOfClusters	the number of clusters contained in the model

Mining Schema and Mining Field

K-means Clustering supports the following usage types:

```
<xs:simpleType name="FIELD-USAGE-TYPE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="active" />
    <xs:enumeration value="supplementary" />
  </xs:restriction>
</xs:simpleType>
```

The mining fields of k-means models contain the attributes importance and missingValueTreatment. Importance is the value contained in column **IMPORTANCE** of

NZA_META_<model_name>_COLUMNS.

MissingValueTreatment is always by imputation of mean or mode. Therefore, **asMean** or **asMode** is used as values.

Output Fields

For k-means clustering, the output section looks as follows:

```
<xs:element name="Output">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="OutputField" minOccurs="1"
                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Only the following attributes of OutputField entries are contained:

```
<xs:element name="OutputField">
    <xs:complexType>
      <xs:attribute name="name" type="FIELD-NAME" use="required" />
      <xs:attribute name="optype" type="OPTYPE" /> <xs:attribute
      name="dataType" type="DATATYPE"/> <xs:attribute
      name="feature" type="RESULT-FEATURE" />
   </xs:complexType>
  </xs:element>
```

K-means models contain output fields having result features **clusterID** and **clusterAffinity**. Cluster affinity is defined such that it contains the distance of a row from its cluster center. Here is the Output section:

```
<Output>
  <OutputField  name="cluster_id" optype="categorical"
                datatype="integer" feature="clusterId" />
  <OutputField name="distance" optype="continuous"
                datatype="double" feature="clusterAffinity" />
</Output>
```

Model Statistics

K-means PMML models may contain statistical information about active and supplementary input fields.

Depending on the value of the statistics parameter provided when the model was built, the **UnivariateStatistics** elements contain some or all of the following information:

```
<xs:element name="UnivariateStats">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Counts" minOccurs="0"/>
      <xs:element ref="NumericInfo" minOccurs="0"/>
      <xs:element ref="DiscrStats" minOccurs="0"/>
      <xs:element ref="ContStats"
    minOccurs="0"/> </xs:sequence>
    <xs:attribute name="field" type="FIELD-
  NAME"/> </xs:complexType>
</xs:element>
```

Counts looks as follows:

```
<xs:element name="Counts">
  <xs:complexType>
    <xs:attribute name="totalFreq" type="NUMBER" use="required"/>
    <xs:attribute name="missingFreq" type="NUMBER"/>
    <xs:attribute name="invalidFreq" type="NUMBER"/>
    <xs:attribute name="cardinality" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
```

NumericInfo, which may be provided for continuous fields, looks as follows:

```
<xs:element name="NumericInfo">
  <xs:complexType>
    <xs:attribute name="minimum" type="NUMBER"/>
    <xs:attribute name="maximum" type="NUMBER"/>
    <xs:attribute name="mean" type="NUMBER"/>
    <xs:attribute name="standardDeviation" type="NUMBER"/>
  </xs:complexType>
</xs:element>
```

Note that the optional attributes median and interquartile range are not available, neither are quantiles. **DiscrStats** looks as follows:

```
<xs:element name="DiscrStats">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="INT_ARRAY" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="modalValue"
  type="xs:string"/> </xs:complexType>
</xs:element>
```

The array contains the frequencies for each of the values in the corresponding field. They are provided in the order in which the field values are listed in the Data Dictionary. **ContStats** looks as follows:

```
<xs:element name="ContStats">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Interval" minOccurs="0" maxOccurs="unbounded"/>
      <xs:group ref="FrequenciesType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="totalValuesSum" type="NUMBER"/>
    <xs:attribute name="totalSquaresSum" type="NUMBER"/>
  </xs:complexType>
</xs:element>
```

with:

```
<xs:group name="FrequenciesType">
  <xs:sequence>
    <xs:group ref="NUM-ARRAY" minOccurs="3"
  maxOccurs="3"/> </xs:sequence>
</xs:group>
```

where the three arrays hold the frequencies, sum of values, and sum of squared values for each interval. For a definition of interval, see the Data Dictionary section.

Model Explanation

For k-means models, Netezza Analytics does not generate a **ModelExplanation** element. Note that cluster statistics are contained in Partition elements in the clusters.

Local Transformations

Whenever categorical columns are present, *k*-means models contain an element, **LocalTransformations**, which contains one **DerivedField** for each categorical mining field.

The derived fields map the categories v1, v2, …, vn (in the order in which they appear in the data dictionary) to integers 1, 2, …, n. This is necessary, because center vectors in a clustering model can only have numeric entries, while Netezza Analytics *k*-means models use modes to characterize cluster centers.

Each DerivedField element has the following form:

```
<DerivedField name="DF" displayname="OF"
              optype="continuous" dataType="integer"
  <MapValues outputColumn="DF" defaultValue="0”>
    <FieldColumnPair field="int
 column="cat"/>
    <InlineTable>
      <row><cat>v1</cat><int>1</int></row>
      <row><cat>v2</cat><int>2</int></row>
      ...
      <row><cat>vn</cat><int>n</int></row>
    </InlineTable>
  </MapValues>
</DerivedField>
```

where OF is the original field name and DF is a uniquely generated name such as OF_INT.

More entries in Local Transformations may become necessary to capture automatic data normalization or standardization once it is available.

Comparison Measure

Netezza k-means models generate a **ComparisonMeasure** element. The comparison measure describes how to aggregate the values obtained by comparing the column value of a record with the corresponding component of a center vector.

Netezza k-means models use distance-based comparison measures **(kind='distance')**. Depending on user input to the algorithm for the distance function, the following elements are contained in the comparison measure:

```
euclidean           <euclidean/>
manhattan           <cityBlock/>
canberra            <cityBlock/>
                      <Extension name="compareFunction" value="canberra"/>
                      </cityBlock>

maximum                  <chebychev/>
```

For other functions (user-defined via UDFs), PMML generation is not supported, because the scoring engine is unlikely to have access to the UDF. Even in the special case, where the UDF implements one of the other distance measures supported by PMML, for example the Minkowski distance, a further UDF would be needed to tell the algorithm about the nature of the user-defined distance function.

Note that canberra cannot truly be modeled in PMML 4.0. Canberra uses a sum (chebychev) as comparison measure, however it requires a compare function that is not available in PMML. The function is a normalized absolute difference, where the absolute difference of values is divided by the sum of the individual field values.

The attributes **compareFunction** specifying the default for the mining fields, as well as **minimum** and **maximum** in **ComparisonMeasure** are not used by Netezza Analytics PMML. Instead, the compare function is explicitly set in **ClusteringField** whenever it is not the absolute difference, for example for categorical fields.

Clustering Field

Clustering fields refer to mining fields or derived fields. There is a clustering field for each continuous mining field, as well as for each field derived from a categorical mining field.

The **ClusteringField** element describes specific features for columns in a clustering model. Netezza PMML only uses the following attributes:

```
<xs:attribute name="field" type="FIELD-NAME" use="required"/>
<xs:attribute name="compareFunction" type="COMPARE-FUNCTION"
              use="optional"/>
```

For comparison measures euclidean, manhattan, and maximum, the compare function is absdiff for originally continuous fields, and delta for the fields derived from categorical fields. The compare function for originally continuous fields may be omitted, as it's the default value.

Cluster

For each cluster contained in the k-means model, there is an element **Cluster**:

```
<xs:element name="Cluster">
  <xs:complexType>
    <xs:sequence>
```

```
      <xs:group ref="NUM-ARRAY" minOccurs="0"/>
      <xs:element ref="Partition" minOccurs="0">
      <xs:element ref="Covariances" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="size" type="xs:nonNegativeInteger"
                                     use="optional"/>
  </xs:complexType>
</xs:element>
```

Each cluster starts with an array of center coordinates; the order of the values is the order in which the fields occur in ClusteringFields. This is straightforward in the case of continuous fields, but does not naively work for categorical fields, as the Netezza Analytics *k*-means model stores the modes, which may be non-numeric.

For non-numeric clustering fields, derived fields are necessary in order to generate numeric field values from the categories. The definition of these derived fields is done as described in the Local Transformations section, above.

With F the number of clustering fields and c_i the *i*th center coordinate, the arrays have the following form:

<Array n="F" type="real">c_1, c_2, …, c_F</Array>

The **Partition** element contains statistics describing the cluster members. It describes the distribution of each mining field, i.e. it uses the original rather than the derived fields.

If needed, covariances are also contained in the Cluster element.

```
<xs:element name="Covariances">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Matrix"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The sequence of rows/columns corresponds to the sequence in MiningSchema. Note that **Covariances** does not contain information about fields from the Transformation Dictionary.

CHAPTER 27

Utility Functions – Probability Distributions

Introduction

A cumulative probability distribution in a single, real valued variable F(x) formally assigns each x a value in a range [0,1] so that if (x1<x2) then F(x1)<=F(x2_).

Informally F(x) means the probability that a random variable takes the value x or less.

As already described in Generalized Linear Models, there exist a number of probability distribution types, like the already mentioned nominal (or discrete) ones: the Bernoulli, binomial, Poisson, and negativebinomial distributions, or the continuous ones like the Gaussian (normal) , wald (inverse gaussian) and gamma distributions.

They are characterized by specific parameters and we can be interested in the cumulative distribution F(x), distribution density f(x) or distribution mass m(x) or the inverted cumulative distribution IF(x).

The relationship between these functions is as follows: If F(x)=y and F(x) is strictly increasing at x, then IF(y)=x. If F(x) is continuous then almost everywhere f(x)=F'(x). In case of a discrete distribution with mass assigned to integers, m(x)=F(x)-F(x-1).

Using Probability Distribution Functions

We are frequently interested in finding out what kind of distribution is followed by a particular attribute. In majority of data analysis applications we would prefer normal distribution, but if a distribution differs from it, the attribute may be transformed to become a normal one.

For example if attribute a follows the distribution z, but we want it to follow the normal distribution n, then we would perform a transformation:

```
a'(x)=IF_n(F_z(a(x)))
```

Note that if attribute a follows the distribution z, then:

```
SELECT count(*) from atable where a < IF_z(0.1) ;
```

Should return a number being 10% of all the records. Same checks may be done at deciles (0.10.2,0.3…,0.9) or percentiles (0.01, 0.02,…, 0.98,0.99). If the results roughly agree, we have some confidence that we know what distribution is followed in the data.

But the most prominent usage of the probability distribution functions is hypothesis testing. Within nzAnalytics procedures like ANOVA, Spearman's correlation, Chi-square tests and t-tests, Wilcoxon and other tests, Generalized Linear Models and other incorporate usage of various probability distribution functions like normal, chi square, t-Student, F-Fisher and other.

And of course they are used to generate samples from distributions other than uniform.

Functions Related with each Distribution

The general form of a call to a probability function is:

```
<type><name><tail>(<argument>[,<parameters>])
```

The type can be either P (meaning cumulative), D (meaning density or mass function) , Q (meaning inverse cumulative).

The name can be BERN, BETA, BINOM, CAUCHY, CHISQ, EXP, F, FISK, GAMMA, GEOM, HYPER, LNORM, LOGIS, NWW, NBINOM, NORM, NORM3P, POIS, T, UNIF, WALD, WEIBULL, or WILCOX.

The tail can be either an empty string (meaning low taikl) or _H (meaning high tail).

For example:

PNORM(x) means F(x) for a standard normal distribution

PNORM_H(x) means 1- PNORM(x)

DNORM(x) means the density function of standard normal distribution

QNORM(p) means the inverse cumulative standard normal distribution function

QNORM_H(p) means 1-QNORM(p)

Distributions and their Parameters

The following sections briefly describe the available distributions.

BERN - Bernoulli Distribution (discrete)

The Bernoulli distribution implies that the response variable takes on values 0 and 1. It models a single toss of a manipulated coin that does not have the same chance of heads and tails. This is a one parameter distribution (the percentage of heads).

BETA - Beta Distribution

The Beta distribution is generic distribution, closely related to other distributions like th Fisher distribution via closed-form mathematical relations. As the beta distribution has a bounded range with positive density (from 0 to 1), processes with implicit lower and upper limits are modelled using it. It has two shape parameters.

BINOM - Binomial Distribution (discrete)

Binomial distribution models tossing a number of times a manipulated coin that does not have the same chance for heads and tails.

It is driven by two parameters: the percentage of heads and the number of trials.

CAUCHY - Cauchy Distribution

The Cauchy distribution describes the cross-section of resonant nuclear scattering, derived from the probability of a resonant state with a known lifetime.

It has two parameters: location and scale.

CHISQ - Chi-square Distribution

If you have variables X1,...,Xn each of them following standard normal distribution, the variable X=X1^2+X2^2+...+Xn^2 follows the chi-square distribution with n degrees of freedom.

It has one parameter: the number of degrees of freedom.

EXP - Exponential Distribution

The exponential distribution may be encountered in behavior of technical systems for example in case of time between failures, provided that the probability of failure is very low and does not change over time.

It has one parameter: the scale.

F - Fisher Distribution

The Fisher distribution, also called Fisher-Snedecor distribution or Fisher F-Distribution, describes the distribution of a quotient of two variables following a Chi-square distribution, both scaled by their number of degrees of freedom.

It has two parameters, the two degrees of freedom.

FISK - Fisk Distribution

The Fisk distribution may be used in hydrology in models of precipitation or stream flow rates.

It has two parameters: median and shape.

GAMMA - Gamma Distribution

The Gamma distribution is the distribution of a sum of random variables that are exponentially distributed. Also exponentially distributed are time intervals between successes in the Poisson experiment.

It has two parameters: shape and inverted scale.

GEOM - Geometric Distribution (discrete)

The cumulative Geometric distribution returns the probability that in a series of Bernoulli trials you will have x or less failures before the first success.

It has one parameter: the probability of success.

HYPER - Hypergeometric Distribution (discrete)

In an urn containing wu white balls and bu black balls, we draw N balls without replacement. Drawing a white ball is a success (1), a black ball is a failure (0). The cumulative Hypergeometric distribution calculates the probability that we get x or less successes in the N trials.

Wu, bu and N are the parameters of the distribution.

LNORM - Log-Normal or Galton Distribution

The properties of living tissues, like weight or skin area, tend to follow the Galton distribution.

It has two parameters: the median on logarithmic scale and the shape on logarithmic scale.

LOGIS - Logistic Distribution

The logistic distribution is used in describing the spread of

epidemics It ha two parameters: mean and scale.

MWW - Mann-Whitney-Wilcoxon Distribution (discrete)

We have a set of observations of variable x split into two sets. We assume that the values of the variable x are distributed in both sets in the same way, and are independent.

We assume that x is ordinal, so that we can rank all values of x. The statistics uStat is calculated as the sum of ranks of objects belonging to the first set or to the second set, whichever is lower.

The MWW distribution describes the distribution of uStat

It has two parameters: the number of items in the first set and the total number of items.

NBINOM - Negative Binomial Distribution (discrete)

The Negative Binomial distribution means that the response variable take on non-negative integer values. It models the toss a manipulated coin that does not have the same chance for heads and tails until a predefined number of heads is reached. The response variable is the count of tails.

It has two parameters: the number of successes needed to stop the series of Bernoulli trials and the success probability.

NORM - Standardized Normal or Gaussian Distribution

The Gaussian distribution, also called the “normal” distribution, is claimed to occur in many natural processes. Specifically, in the case of a multitude of independent processes generating continuous numbers from a distribution, then their average tends to follow normal distribution.

Its standardized version assumes that it has a mean of 0 and standard deviation of 1.

It has no parameters.

NORM3P - Normal or Gaussian Distribution

The Gaussian distribution, also called the “normal” distribution, is claimed to occur in many natural processes. Specifically, in the case of a multitude of independent processes generating continuous numbers from a distribution, then their average tends to follow normal distribution.

It has two parameters: the mean and the standard deviation.

POIS - Poisson Distribution (discrete)

The Poisson distribution implies that the response variable takes on non-negative integer values . The distribution is driven by the Poisson process. In this process the average rate of success over time is known. The probability of a single success within a time interval is proportional to the length of the interval and is independent of the probability outside of the interval. If the interval gets shorter, the probability of more than 1 success goes toward zero.

It has one parameter: the mean.

T- t-Student Distribution

Considering n independent random variables Z1,...,Zn distributed according to the normal distribution with mean mu and a fixed variance, we create a new variable Z as their average. If s is the standard deviation from the sample, then the variable X = (Z-mu)/(s/sqrt(n)) follows the t-Student distribution with n-1 degrees of freedom.

It has one parameter: the number of degrees of freedom.

UNIF - Uniform Distribution

Uniform distribution has two parameters: the lower bound and the upper bound. Its density is a constant between these bounds.

WALD - Wald Distribution

The Wald distribution, also called the Inverse Gaussian distribution, represents the first passage time for Brownian motion.

It has two parameters: location and shape.

WEIBULL - Weibull Distribution

The Weibull distribution gives a distribution of time between failures x, for which the failure rate is proportional to a power of time. The shape parameter kshape may be understood as follows:

If it is lower than 1, then the failure rate decreases over time,

If it is equal to 1, then the failure rate is constant over time,

If it is greater than 1, then the failure rate increases over time ("aging process").

The Exponential distribution is a special case of the Weibull distribution:

Weibull distribution has two parameters: scale and shape.

WILCOX - Wilcoxon Distribution (discrete)

Given two variables x and y measured for the same objects (items), we split the objects into two sets: the first set contains objects where x>y and the second set objects where x<=y.

For each object we compute the rank of |x-y|. The statistics sStat is calculated as the sum of ranks of objects belonging to the first set or to the second set, whichever is lower.

This statistics follows the Wilcoxon distribution.

The Wilcoxon distribution has one parameter: number of items.

Usage Examples

You may be interested in having a look at how well the petallength of setosa in the iris database fits the normal distribution. Run:

```
select avg(petallength), stddev(petallength)
from nza..iris where class = 'setosa';
```

The result will show that mean and standard deviation of petallength are 1.464 and 0.17351115943645 respectively.

Then run:

```
select rank() over (order by petallength asc) as r, r/50. as pp,
nza..pnorm3p(petallength, 1.464 , 0.17351115943645) ppnorm from
nza..iris where class = 'setosa';
```

to see that the correspondence is only rough (in the ideal case, the pp and ppnorm columns would be equal).

```
  r |    pp    |       ppnorm
----+          +--------------------
  1 |---------0.020000| 0.0037456162171375
  2 | 0.040000 |   0.01795929726821
  3 | 0.060000 |  0.064065203430695
  3 | 0.060000 |  0.064065203430695
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
  5 | 0.100000 |    0.1722824316448
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 12 | 0.240000 |   0.35611888863086
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 24 | 0.480000 |   0.58218228015053
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 38 | 0.760000 |   0.78342456624959
 45 | 0.900000 |   0.91310767209118
 45 | 0.900000 |   0.91310767209118
 45 | 0.900000 |   0.91310767209118

 45 | 0.900000 |   0.91310767209118
```

```
     49 |  0.980000  |   0.99401125247379
     49 |  0.980000  |   0.99401125247379
    (50 rows)
```

A different way to inspect how well the data follows a distribution is to compute the mean and variance of petallength:

```
select avg(petallength), variance(petallength) from nza..iris ;
```

The SELECT results in a mean that is approximately 3.75 and true variance of 3.11.

Now create a table, iripetall, describing the probability for a given plant to have its actual petallength, given that the true mean is 3.75 and true variance is 3.11.

Use the call:

```
CALL nza..CUMULATIVE('intable=nza..iris, id=id, type="n",
incolumn=petallength, outtable=iripetall, mean=3.75, variance=3.11');
```

It will look like this:

```
      id  |         pnorm         | petallength | mean | variance
    -----+                       +             +------ +
       4 | ------------------0.10100286923082 |-------------1.5| 3.75 |----------3.11
       8 |    0.10100286923082 |            1.5 | 3.75 |        3.11
      12 |    0.11139303484419 |            1.6 | 3.75 |        3.11
      16 |    0.10100286923082 |            1.5 | 3.75 |        3.11
      20 |    0.10100286923082 |            1.5 | 3.75 |        3.11
      24 |    0.12252669600944 |            1.7 | 3.75 |        3.11
      28 |    0.10100286923082 |            1.5 | 3.75 |        3.11
      32 |    0.10100286923082 |            1.5 | 3.75 |        3.11
      36 |  0.074092479803748  |            1.2 | 3.75 |        3.11
```

If you are interested in outliers, use the following call to see them:

```
select * from iripetall where pnorm<0.001 or pnorm> 0.999;
```

The output contains zero rows because the data fit quite well (there is not even a record with pnorm below 0.025 or above 0.975).

With the following call, you can determine the frequencey of plants in the population with a petallength close to a given plant from the sample in our table. (This assumes the probability distribution is the same, inthis case, normal.)

```
CALL nza..density('intable=nza..iris, id=id, type="n",
incolumn=petallength, outtable=iripetalld, mean=3.75, variance=3.11');
```

Here is a fragment of the output table iripetalld:

```
     id |        dnorm       | petallength | mean | variance
    ----+-------------------+             +------ +
      4 | 0.17678070212437 |-------------1.5| 3.75 |----------3.11
      8 | 0.17678070212437 |            1.5| 3.75 |        3.11
     12 | 0.18973901890261 |            1.6| 3.75 |        3.11
```

The density for plant number (id) 8 is 0.17678. Therefore, the probability of having a plant with petallength in the range 1.5-0.01 to 1.5+0.01 would be approximately 0.02*0.17678=0.0035356.

To generate a sample of 32 random numbers following the sample Fisk distribution, with median = 5 and shape = 2, do the following:

```
Create table randfisk (idf integer);
Insert into randfisk values(1);
Insert into randfisk select * from randfisk;
Insert into randfisk select * from randfisk;
Insert into randfisk select * from randfisk;
Insert into randfisk select * from randfisk;
Insert into randfisk select * from randfisk;
Select nza..qfisk(random(),5,2) from randfisk;
```

The last line is essential. The function “random()” generates random numbers ranging from 0 to 1. Qfisk considers this as the percentage at which to compute a value. The output looks as follows:

```
  qfisk
------------------
  6.4816069851943
  11.109999638123
   73.47798676712
 0.44251277274839
  4.3609914099502
  3.9652110974754
  8.9253326501444
  5.3863011022385
  18.889358419282
    1.91102815974
   18.88928575491
  12.092956478697
  2.9398120974769
  3.8557807477485
  5.4003170922859
  3.9099985788724
  2.2127362446343
  1.7368954145846
  8.5545143301408
  11.250158024616
  8.6580376466144
  9.8508742789332
  5.0378666446805
  5.1411129557278
  4.6351839618808
  1.9331837885486
   3.894388650915
   4.887109365305
  4.8174315751124
  5.0469902608837
  3.6759954053399
  1.7428535568487
(32 rows)
```

CHAPTER 28
Bulk Algorithms

By using bulk algorithms, you can process data sets in parallel. The data set is split into a partition of data sets that are based on a group.

A typical example is a table that contains the energy consumption over time for all customers of a utility company. With bulk algorithms, you can compute one model for each group, which is one model for each customer in the example. These models are built in parallel and independently from each other on each data slice. Creating the models in this way is much faster than building them sequentially on the host or outside the database.

You can even further improve the performance of the model creation by distributing the data into data slices according to the group field, for example, the customer ID. Thus, the data of one group is stored on one data slice.

Bulk algorithms are bulk matrix operations, bulk linear regression, and bulk principal component analysis.

Bulk Matrix Operations

Background

By using bulk matrix operations, you can work on many small matrixes in parallel. For example, bulk matrix operations can help you find the optimum linear regression model when you have many independent variables to choose from. Instead of using step-by-step linear regressions, you compute many regression tasks for different subsets of independent variables. From the created bulk of linear regression models, you can then select the model with the best results. To increase the processing speed, bulk matrix operations are based on user-defined table functions (UDTFs) with user-defined aggregates. This way, you can combine the matrix operations in bulks without having to store the intermediate results.

Applications

Bulk matrix operations can be applied to scenarios for which you want to shortly do a set of matrix operations on one matrix or on many small matrices.

Available Functionality

The following bulk matrix operations are available:

- Matrix operations for one matrix:
  - Inversion
  - Transposition
  - Scaling by a numeric factor
  - Singular value decomposition (SVD)
  - Eigenvalue decomposition
  - Matrix determinant
- Matrix operations for two matrices:
  - Sum of two matrices
  - Matrix multiplication of two matrices
  - Element-wise multiplication of two matrices
  - Kronecker product to solve linear equations Ax = b

Notes:

- For a single matrix operation:
  - Matrices of sizes up to 10,000 x 10,000 are supported.
  - The size of the matrix must be less than 100,000,000 cells.
- For multiple matrix operations:
  - Matrices of sizes up to 7,000 x 7,000 are supported.
  - The size of the matrix must be less than 50,000,000 cells.
- Matrices can be created from tables and vice versa.
- If the data set that you use for a bulk matrix operation is incomplete, an empty data set is returned. For example, the data set might be incomplete if the cell from the first row in the first column of the first matrix is missing.

Examples

The following examples are available for bulk matrix operations:

- Example—Matrix representation for bulk operations
- Example—Introductory example for bulk matrix operations
- Example—Input data reordering and redistribution

Example—Matrix representation for bulk operations

The matrix operations supports dense matrices that are stored in the database in row-column-value (RCV) format:

row (INT4) – row identifier 1..N

col (INT4) – column identifier 1..M

val (DOUBLE) – cell value

For input matrices, a special form of *sparse* matrix is supported. The first cell and the last cell of such a matrix is presented even if the value is zero. For the first cell, row and col is equal to 1. For the last cell, row is equal to the maximum row index, and col is equal to the maximum column index.

To do bulk matrices operations, you must extend the RCV format by additional fields.

The input table must be extended by id_task and id_matrix fields:

id_task (INT4)
Identifies the input group of matrices. The
sequence is from 1 up to 2147483647.

id_matrix (INT2)
Defines the order of matrices within the group of
matrices. The sequence is from 1 up to 32767.

row (INT4)
Is the row index, where the sequence starts from 1.

col (INT4)
Is the column index, where the sequence starts from 1.

val (DOUBLE) Is
the cell value.

The id_task field defines the identifier of the group of matrices that are used in a separate computational process. If the id_matrix value is set to 1, the operations are done on single matrices.

To convert the matrix to RCV format, you can use the MATRIX2RCV(NVARCHAR(ANY)) procedure. The input of the procedure is a comma-separated list of parameters in the form of <parameter>=<value>.

Input parameters that are passed to the UDTF are:

inmatrix
The name of the input matrix in block-cyclic format

outtable
The name of the output table that contains the matrix in RCV format

id_task

Optional: id_task value

id_matrix

Optional: id_matrix value

Example:

```
call nzm..shape('2,3,5,7',3,3,'A');
call nzm..print('A');
call nzm..matrix2rcv('inmatrix=A,outtable="A_RCV"');
select * from "A_RCV";
call nzm..matrix2rcv('inmatrix=A,outtable="A_RCV",id_task=2');
select * from "A_RCV";
call nzm..matrix2rcv('inmatrix=A,outtable="A_RCV",id_task=2,id_matrix=2');
select * from "A_RCV";
drop table "A_RCV";
call nzm..delete_matrix('A');
```

Note: In the second and third call of matrix2rcv, the input matrix A is added to the existing A_RCV table.

To convert an RCV table into an nzmatrix object, you can use the RCV2MATRIX(NVARCHAR(ANY)) procedure. The output matrix is specified by id_task and id_matrix. The input of the procedure is a comma-separated list of parameters in the form of <parameter>=<value>.

Input parameters that are passed to the UDTF are:

intable
The name of the input table that contains the matrix in RCV format

outmatrix
The name of output matrix in block-cyclic format

id_task

Optional: id_task value

id_matrix

Optional: id_matrix

value Example:

```
call nzm..shape('2,3,5,7',3,3,'A');
call nzm..print('A');
call nzm..matrix2rcv('inmatrix=A,outtable="A_RCV"');
call nzm..rcv2matrix('intable="A_RCV",outmatrix=B');
call nzm..print('B');
call nzm..matrix2rcv('inmatrix=B,outtable="A_RCV",id_task=2');
call nzm..rcv2matrix('intable="A_RCV",outmatrix=C,id_task=2');
call nzm..print('C');
call nzm..matrix2rcv('inmatrix=C,outtable="A_RCV",id_task=2,id_matrix=2');
call nzm..rcv2matrix('intable="A_RCV",outmatrix=D,id_task=2,id_matrix=2');
call nzm..print('D');
drop table "A_RCV";
call nzm..delete_matrix('A');
call nzm..delete_matrix('B');
call nzm..delete_matrix('C');
call nzm..delete_matrix('D');
```

Example–Introductory example for bulk matrix operations

It is assumed that you have two matrices, A and B, and that you want to add them by yielding a matrix C.

Matrix A is as follows:

```
12
14
16
```

Matrix B is as follows:

```
27
23
20
```

To hold them, you must create database tables in the following format:

```
Create table A (row int4, col int4, val double);
Create table B (row int4, col int4, val double);
```

Then, you insert the data into these tables.

```
Insert into A values(1,1,11);
Insert into A values(1,2,12);
Insert into A values(2,1,13);
Insert into A values(2,2,14);
Insert into A values(3,1,15);
Insert into A values(3,2,16);

Insert into B values(1,1,29);
Insert into B values(1,2,27);
Insert into B values(2,1,25);
Insert into B values(2,2,23);
Insert into B values(3,1,21);
Insert into B values(3,2,20);
```

To perform a bulk matrix operation, that is one task on two matrices in the example, you usually create a view or table by pulling together all tables that are part of the operation.

```
Create view AB_VIEW as
(select 1::int4 as id_task, 1::int2 as id_matrix,row,col,val from A)
union
(select 1::int4 as id_task, 2::int2 as id_matrix,row,col,val from B);
```

where:

A and B are to be processed in task 1

Matrix A plays the role of the first argument, that is matrix 1

Matrix B plays the role of the second argument that is matrix

2 Now, you can apply the sum_tf operation:

```
Create table C as
select o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from AB_VIEW) i,
TABLE(nzm..sum_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task)
```

The structure and interpretation of table C is identical to table A and table B.

If you have many similar tasks to do, that is you have matrices A1, A2, B1, B2, and you want to find the matrix C1=A1+B1 and C2=A2+B2, you prepare A1, A2, B1, B2 in the same way as matrix A and matrix B in the example.

Then, you create a view like the following view:

```
Create view MULTITASK_VIEW as
(select 1::int4 as id_task, 1::int2 as id_matrix,row,col,val from A1)
union
(select 1::int4 as id_task, 2::int2 as id_matrix,row,col,val from B1)
union
(select 2::int4 as id_task, 1::int2 as id_matrix,row,col,val from A2)
union
(select 2::int4 as id_task, 2::int2 as id_matrix,row,col,val from B2);
```

Now, you can apply the **sum_tf** operation:

```
create table C as
select i.id_task, o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from MULTITASK_VIEW) i,
TABLE(nzm..sum_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task);
```

Table C contains both matrices C1 and C2 that are stored in a single table. To identify rows and columns that belong to each of the table, use the id_task column of table C. You can also create appropriate views like the following view:

```
create view C1 as select row,col,val from C where id_task=1;

create view C2 as select row,col,val from C where id_task=2;
```

By using the same method, you can add operators to many matrices, for example, C1=A1+B1, C2=A2+B2,C3=A3+B3,...,Cn=An+Bn.

The following examples show similar matrix operations are similar in form, as you will see in the subsequent subsections.

The example about matrix representation and the example about input data reordering and redistribution show the structure of the tables that represent the matrices.

The other examples show how to apply different matrix operations.

Example—Input data reordering and redistribution

All matrix operations require correct data distribution and ordering to do correct matrix operations. To achieve this goal, you can use the special user-defined aggregate (UDA). The reorder_matrix(INT4) UDA is an analytic type aggregate that works in the incremental windows aggregate mode.

Call the *reorder_matrix(INT4)* UDA function before you do a matrix operation in the matrix operations execution chain**.**

You can use the following call for single matrix operations:

```
select id_task, row, col, val, nzm..reorder_matrix(id_task) over
(partition by id_task order by row desc, col desc) as
id_task_reorder from INP_TBL, <SINGLE_MATRIX_OPERATION >;
```

For operations on multiple matrices, add the id_matrix field to the call as shown in the following example:

```
select id_task, id_matrix, row, col, val, nzm..reorder_matrix(id_task) over
(partition by id_task order by id_matrix desc, row desc, col desc) as
id_task_reorder
from INP_TBL, <MULTIPLE_MATRIX_OPERATION>;
```

Note: Use the same data order for the SQL query that is shown in the example. If you change the matrix operation, an error might occur.

Examples—UDTF for single-matrix operations

All single operations are registered in the *NZM* database.

Example—Transposition

You can use an SQL query to do a matrix transposition operation. A specific UDTF is not required.

The following example shows such an SQL with row identifiers and column identifiers.

```
create table OUT_TBL as
select i.id_task, i.id_matrix, i.col as row, i.row as col, i.val from
INP_TBL i
distribute on (id_task);
```

The following examples refer to Example–Introductory example for bulk matrix operations.

It is assumed that you want to transpose matrix A by using the following call:

```
create table AT as
select  i.col as row, i.row as col, i.val from A i;
```

It is also assumed that you want to transpose all the matrices, for example, the matrices in the AB_VIEW, by using the following call:

```
create table AB_VIEW_T as
select i.id_task, i.id_matrix, i.col as row, i.row as col, i.val from AB_VIEW
i distribute on (id_task);
```

Example–Scaling

Matrix scaling multiplies all the elements by a scalar value.

You can use an SQL query to do a matrix scaling operation. A specific UDTF is not required.

The following example shows such an SQL where the value is multiplied by the scalar value.

```
create table OUT_TBL as
select i.id_task, i.row, i.col, i.val * scalar_value as val from INP_TBL i
distribute on (id_task);
```

where:

Scalar_value is a real number

Instead of the multiplication, you can process any arithmetical operation

The matrices that are to be scaled by the same scalar are contained in the INP_TBL table

Example–Inversion

Description: Calculates matrix inversion.
The matrix that is to be inverted must be a square matrix. It must also be invertible, that is the determinant must be non-zero. If M1 is the inversion of matrix M, then the matrix product M*M1=M1*M=I where I is a matrix with 1 on the diagonal and 0 elsewhere.

UDX name: inversion_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task, o.row, o.col, o.val from
(select *,nzm..reorder_matrix(id_task) over
(partition by id_task order by row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..inversion_tf(i.id_task_reorder,i.row,i.col,i.val)) o
distribute on (id_task);
```

Example–Pseudo-Inversion

Description: Calculates matrix pseudo-inversion.
The Moore-Penrose pseudo-inversion is computed by using the SVD method. The properties of the pseudo-inverted matrix are similar to the properties of the inverted matrix. The pseudo-inverted matrix is used when an inversion is not possible. For pseudo-inversion, operation parameters control the tolerance of the numerical precision of the inversion. Each input matrix must be a squared matrix.

UDX name: pseudo-inversion_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from (select
*,nzm..reorder_matrix(id_task) over (partition by id_task order by row
desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..pseudo_inversion_tf(i.id_task_reorder,i.row,i.col,i.val)) o
distribute on (id_task);
```

Note: Because of rounding errors or other inaccuracies, you must determine when a numerical value should be equal to zero, especially at singular points. To determine this condition, you can set or change the tolerance parameter of the UDTF.

By default, this parameter is set to 1.5e-8. After you do an SVD decomposition of the matrix M into U*S*V, a value in the S matrix is equal to zero, if its absolute value is lower than the maximum absolute value in the S matrix, multiplied by the tolerance value. The following example shows the setting of the tolerance value. In the example, the tolerance value is increased to 1.5e-6.

```
begin transaction;
      select nzm..init_parameters(1) from _v_dual_dslice;
      select nzm..set_parameter('tolerance','1.5e-6') from _v_dual_dslice;

      create table OUT_TBL as
      select i.id_task,o.row,o.col,o.val from
      (select *,nzm..reorder_matrix(id_task) over
      (partition by id_task order by row desc, col desc) as id_task_reorder
      from INP_TBL) i
      ,TABLE(nzm..pseudo_inversion_tf(i.id_task_reorder,i.row,i.col,i.val)) o
      distribute on (id_task);

      select nzm..clear_parameters() from
_v_dual_dslice; commit;
```

Example–XtX transformation

Description: Calculates the multiplication of a matrix X transposed and X for an input matrix X. For example, matrix X transposed * X. For each id_task, a single square matrix is returned. This operation is often called SSCP (sum of squares and cross products), and the name is used, for example, in linear regression.

UDX name: xtx_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table  OUT_TBL as
select i.id_task,o.row,o.col,o.val from (select *
,nzm..reorder_matrix(task_id) over (partition by id_task
order by row desc, col desc) as id_task_reorder from INP_TBL) i
,TABLE(nzm..xtx_tf(i.id_task_reorder, i.row, i.col, i.val)) o
distribute on (id_task);
```

Example–SVD decomposition

Description: Calculates the SVD decomposition matrix. For each id_task, the UDTF returns three matrices that are identified by the id_matrix field.

id_matrix=1: U matrix

id_matrix=2: S matrix (column vector)

id_matrix=3: Vt matrix

UDX name: svd_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

id_matrix (INT2)
Identified type of vector
(1-U matrix, 2-S matrix, 3-Vt matrix)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table  OUT_TBL as
select i.id_task,o.id_matrix,o.row,o.col,o.val from
(select *,nzm..reorder_matrix(id_task) over (partition by id_task
order by row desc, col desc) as id_task_reorder from INP_TBL) i
,TABLE(nzm..svd_tf(i.id_task_reorder,i.row,i.col,i.val)) o
distribute on (id_task);
```

Example—Eigenvalue decomposition

Description: Calculates the eigenvalue decomposition matrix for symmetric matrices. For non-symmetric matrices, an error occurs. For each id_task, the UDTF returns two matrices that are identified by id_matrix.

id_matrix=1: eigenvalue matrix

id_matrix=2: eigenvector

matrix UDX name: eigenvalue_tf

UDX type:

UDTF Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

id_matrix (INT2)
Identifies the type of the returned matrix according to eigenvalues or eigenvectors

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task,o.id_matrix,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over
(partition by id_task order by row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..eigenvalue_tf(i.id_task_reorder,i.row,i.col,i.val)) o
distribute on (id_task);
```

Example—Matrix determinant

Description: Calculates the determinant of a matrix. The matrix must be a square matrix. For each id_task,the UDTF returns a 1x1 matrix.

UDX name: determinant_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table  OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by row desc, col desc) as id_task_reorder from INP_TBL) i
,TABLE(nzm..determinant_tf(i.id_task_reorder,i.row,i.col,i.val)) o
distribute on (id_task);
```

Examples—UDTFs for operations on multiple matrices

All N-ary operations are registered in the *NZM* database.

Example—Sum

Description: Calculates the sum of two or more matrices.

UDX name: sum_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

id_matrix (INT2)
The matrix identifier of a matrix that is to be added. The id_matrix values must be consecutive; integers must start with 1. In different tasks, the number of matrices that are to be added can differ.

row (INT4)
Within the same task, the maximum row number of each input matrix must be identical

col (INT4)
Within the same task, the maximum col number of each input matrix must be identical

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table  OUT_TBL as select i.id_task, o.row, o.col, o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..sum_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task)
```

The example calculates the sum of all matrices with the same id_task and different id_matrix. Therefore, the number of resulting matrices will be identical to the number of different values of the id_task in the INP_TBL table, that is, you get one matrix for each id_task.

Example–Product

Description: Calculates the product of two or more matrices.

UDX name: product_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

id_matrix (INT2)
The matrix identifier of a matrix that is to be multiplied. The id_matrix values must be consecutive; integers must start with 1. In different tasks, the number of matrices that are to be multiplied can differ.

row (INT4)
Within the same task, the maximum row number of each input matrix must be identical with the maximum col number of the preceding input matrix

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from INP_TBL) i,
TABLE(nzm..product_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task)
```

Example—Kronecker product

Description: Calculates the Kronecker product of two or more matrices.

UDX name: kronecker_tf

UDX type: UDTF

Input:

id_task_reorder (INT4)
The id_task that is returned by the reorder_matrix UDA

id_matrix (INT2)
The matrix identifier of a matrix that is to be multiplied. The id_matrix values must be consecutive; integers must start with 1. In different tasks, the number of matrices that are to be

multiplied can differ.

row (INT4)

col (INT4)

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..kronecker_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task)
```

Example Solve

Description: Solves a linear equation Ax = B. For each id_task, the UDTF returns an x matrix (vector). To ensure a correct calculation, use two matrices for each id_task.

id_matrix=1: A matrix

id_matrix=2: B matrix (column

vector) UDX name: solve_tf

UDX type:

UDTF Input:

id_task (INT4)
The id_task that is returned by the reorder_matrix UDA

id_matrix (INT2)
The matrix identifier. 1 identifies matrix A; 2 identifies matrix B

row (INT4)

col (INT4)
Equal 1 is valid for everywhere for matrix B

val (DOUBLE)

Output:

id_task (INT4)

row (INT4)

col (INT4)
Equal 1 is valid everywhere

val (DOUBLE)

Calling method:

Input matrices that have this convention are stored in the INP_TBL table.

Output matrices are stored in the OUT_TBL table.

You must not change the names of the columns, functions, and so on that are used in the example. You can, however, change the names of the input table and the output table.

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..solve_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task)
```

Examples~~—~~User scenarios

Example~~—~~Calculate the bulk of sums for a few matrices

It is assumed that you have defined ten tasks for each of which you want to calculate the sum of five matrices in the INP_TBL table.

For each of the ten tasks, that is id_task=1 to ID_task=10, you have five matrices that are numbered from 1 to 5 by using id_matrix.

To calculate the sum of these matrices for each task, you can run the following query:

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..sum_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task);
```

The query writes the results of the bulk summing into OUT_TBL table. The results have ten different id_task numbers. For each task, only one matrix is stored.

Example~~—~~Calculate the bulk of chain for a few matrix operations

It is assumed that you have saved ten tasks, each of which has one matrix, in the INP_TBL table. For each of the ten tasks, that is id_task=1 to ID_task=10, one matrix is stored. Now, you want to calculate the bulk of INV(2.02*XtX+7.98) expressions.

To calculate these expressions separately for each id_task, you can run the following query:

```
create table OUT_TBL as
select i.id_task,y.row,y.col,y.val from
(select *, nzm..reorder_matrix(id_task) over (partition by id_task

order by id_task asc, row desc, col desc) as id_task_reorder
from INP_TBL) i
,TABLE(nzm..xtx_tf(i.id_task_reorder, i.row, i.col, i.val))
xtx ,TABLE(nzm..inversion_tf(i.id_task_reorder, xtx.row,
xtx.col, 2.02*xtx.val+7.98)) y
distribute on (id_task);
```

The results of the query are stored in the OUT_TBL table.

The processing order of the computation is as follows:

xtx(SSCP)

2.02*xtx.val+7.98

Inversion

If the number of parallel processors is sufficient, you can do each task in parallel.

Example—Calculate bulk matrix operations by using a shared matrix

It is assumed that you want to calculate a product of two matrices. The first matrix M1 is stored in INP_TBL1 table. Matrix M1 is unique for each id_task. The second matrix M2 is shared for each M1 matrix. Matrix M2 is stored in the INP_TBL2 table. To ensure that the computation is done efficiently, the M2 matrix in the INP_TBL2 table must be broadcasted to all data slices.

To calculate a product under these conditions, you can run the following query:

```
create table OUT_TBL as
select i.id_task,o.row,o.col,o.val from
(select *, nzm..reorder_matrix(id_task) over (partition by
id_task order by row desc, col desc) as id_task_reorder
from (
      select id_task,1 as id_matrix, row, col,
      val from INP_TBL1
      union all
      select ds.id_task, 2 as id_matrix, sh.row, sh.col, sh.val
      from INP_TBL2 sh cross join
            (select distinct id_task as id_task from INP_TBL1) ds
      ) i
,TABLE(nzm..product_tf(i.id_task_reorder,i.id_matrix,i.row,i.col,i.val)) o
distribute on (id_task);
```

The following query returns the instance of the M2 matrix for each id_task from the INP_TBL1 table:

```
select ds.id_task, 2 as id_matrix, sh.row, sh.col, sh.val
from INP_TBL2 sh cross join
      (select distinct id_task as id_task from INP_TBL1) ds
```

The value of id_matrix, id_matrix=2, means that the shared matrix is taken as the second argument of the product operation.

Bulk Linear Regression

Background

Linear regression models are useful to model many real-world phenomena because these models are easy to train and to apply. You can use these models in areas of biology, biopharmaceuticals, engineering, actuarial science, and quality assurance.

The bulk linear regression function implements the linear regression algorithm for bulk matrices. For bulk linear regression, multiple linear regression models for different data groups are created in parallel. For each input matrix, one linear regression model with different predictors is created. From the created bulk of linear regression models, you can optionally select the best model. By using a set of quality indicators, you can identify the model that suits your purpose best.

Applications

Bulk linear regression can be applied to the following main scenarios.

The data set can be split into logical subgroups, for example, one subgroup for each customer. In this case, a single model is created for each customer. The single model is independent from the models that are created for other customers.
A typical example is forecasting the energy consumption of individual households by creating an individual linear model, separately for each household.

Several models are created in parallel. From the created model, you can subsequently choose the best model. You can build these models on variants of the same data set, for example, you can include additional columns.

Available Functionality

The Netezza Analytics implementation of bulk linear regression extends the linear regression model that is provided by the LINEAR_REGRESSION, PREDICT_LINEAR_REGRESSION stored procedures. You can use the bulk linear regression to build many linear regression models for different data sets at the same time by using parallel machine architecture.

Bulk linear regression has the following functionality that is provided by user-defined table functions (UDTFs):

Fitting the bulk linear model with or without the intercept term

Multiple regressions

Fitting the model that is based on colinear attributes or nearly colinear attributes

Standard Least Square Estimation (LSE) procedure by using QR decomposition

Model diagnostics

Support for discrete and continuous attributes

The following procedures are available for data conversion:

Procedures that convert input data that is saved in table format to RCV format and vice versa

Procedures that convert each nominal attribute to a set of dummy single attributes and vice versa

Examples

Input parameters that are passed to the UDTF are:

id_task
The numbering of 'row' within each 'id_task' group must start with 1.

id_matrix

row

column

val

The following example shows how to create a model.

```
begin transaction;
select nzm..init_parameters(2) from _v_dual_dslice;
select  nzm..set_parameter('residuals','1')  from  _v_dual_dslice;

select nzm..set_parameter('intercept','1') from _v_dual_dslice;

create table M_MAT_TABLE as
      select i.id_task, o.id_matrix, o.row, o.col,
      o.val from
      (select *,
            nzm..reorder_matrix(id_task) over (partition by id_task
            order by id_matrix desc, row desc, col desc) as id_task_ro
            from
            (select 1 as id_matrix, id_task, row, col,
                  val from Y_MAT_TABLE
                  union all
                        select 2 as id_matrix, id_task, row, col,
                        val from X_MAT_TABLE
            ) a
      ) i,
      TABLE(nzm..lm_tf(i.id_task_ro,i.id_matrix,i.row,i.col,i.val))
o; commit;
```

Note: You must set the non-default parameters before you call the nzm..lm_tf UDTF by using the **set _parameter** parameter as shown in the example.

The following table shows the parameters for the nzm..lm_tf UDTF. These parameters control the model structure and the output:

Table 70: Parameters for the nzm..lm_tf UDTF

Parameter	Description	Values
residuals	Returns the residuals as additional output matrix.	0 - Default. Residuals are not returned. 1 - Residuals are returned.

Parameter	Description	Values
intercept	Automatically include intercepts in the model.	0 - Intercepts are not included. 1 - Default. Intercepts are included.

The *nzm..lm_tf* call requires two input matrices, Y and X, where Y represents response vectors and X represents predictors. These matrices are identified by the id_matrix value, where id_matrix = 1 is set for the Y matrix, and id_matrix = 2 is set for the X matrix.

By default, the nzm..lm_tf UDTF returns two matrices. If the **residuals** parameter is set to **1**, a third matrix is returned.

The following list shows the returned matrices:

ci-by-4 matrix with coefficients and characteristics, where:

ci is the number of input attributes including intercepts

Subsequent columns show the estimated coefficient, standard error, t-statistic, and the corresponding two-sided p-value

7-by-1 matrix with statistics of the fitted model, where:
Subsequent rows show the residual standard error, degrees of freedom, R2 and adjusted R2, F-statistic, the t-statistic, p-value, and the residual sum of squares

Optional: n-by-1 matrix with residuals, where n is the number of rows of the input matrix.

The following example shows how to apply the created model. After the models are built, you can apply them to new data, for example, to a bulk of matrices that are stored in the table. The model that is identified by id_task, is applied to the matrix that is also identified by id_task.

```
create table R_MAT_TABLE as
       select r.id_task, r.row, r.res+ m.val as
       val from
       (select z.id_task, z.row, sum(z.val*m.val) as res
             from Z_MAT_TABLE z right join M_MAT_TABLE m on (z.col=m.row-
             1) where z.id_task=m.id_task and m.id_matrix=1 and m.col=1
             group by z.id_task, z.row
        ) r, M_MAT_TABLE m
        where m.row=1 and m.col=1 and m.id_matrix=1 and m.id_task = r.id_task;
```

If the model that is stored in the M_MAT_TABLE table was built without intercepts, the application query looks as follows:

```
create table R_MAT_TABLE as
 select z.id_task, z.row, sum(z.val*m.val) as res
       from   M_MAT_TABLE_A   m   join   Z_MAT_TABLE   z   on
       (m.row=z.col)     where     z.id_task=m.id_task     and
       m.id_matrix=1 and m.col=1 group by z.id_task, z.row;
```

As a result of the model application, the R_MAT_TABLE table is created. The table contains the prediction separately for each task field and for each row. The task field is identified by id_task, and the row is identified by row_field.

The following example shows how to choose and apply the best model from a bulk of linear regression models. In the example, several linear models are built for each split. The best model is applied, for which then a residual sum of squares (RSS) is done. RSS measures the difference between the data and the model. If the RSS is small, the model and the data fit tightly together.

```
-- split data into train/test subset
create table winequality_train as select * from nza..winequality where id <
2467;
create table winequality_test as select * from nza..winequality where id >
2466;
-- conversion to RCV format
call nzm..simple2rcv_adv('outtable=winequality_train_x_rcv,
outmeta=winequality_train_x_rcv_meta, intable=winequality_train,
incolumnlist=.;-QUALITY,id=ID');
call nzm..simple2rcv_adv('outtable=winequality_train_y_rcv,
outmeta=winequality_train_y_rcv_meta, intable=winequality_train,
incolumnlist=QUALITY,id=ID');
call nzm..simple2rcv_adv('outtable=winequality_test_x_rcv,
outmeta=winequality_test_x_rcv_meta, intable=winequality_test,
incolumnlist=.;-QUALITY,id=ID');
call nzm..simple2rcv_adv('outtable=winequality_test_y_rcv,
outmeta=winequality_test_y_rcv_meta, intable=winequality_test,
incolumnlist=QUALITY,id=ID');
   create a split in data, split is auxiliary just for example
create table WINEQUALITY_TRAIN_X_RCV_4SPLIT as select
    (row/617 + 1)::integer as id_task,
    CASE WHEN row >= 617 THEN row - (row/617)::integer * 617 + 1 ELSE row END
as row,
    col, value as val
    from WINEQUALITY_TRAIN_X_RCV;

create table WINEQUALITY_TRAIN_Y_RCV_4SPLIT as
    select (row/617 + 1)::integer as id_task,
    CASE WHEN row >= 617 THEN row - (row/617)::integer * 617 + 1 ELSE row END
as row,
    col, value as val
    from WINEQUALITY_TRAIN_Y_RCV;

   create a linear model for each
split begin transaction;
select nzm..init_parameters(2) from _v_dual_dslice;
select nzm..set_parameter('residuals','1') from _v_dual_dslice;
select nzm..set_parameter('intercept','0') from _v_dual_dslice; -- without
intercept, easier model apply
create table WINEQUALITY_M_MAT_TABLE as select
    i.id_task, o.id_matrix, o.row, o.col,
    o.val from
    (select *, nzm..reorder_matrix(id_task) over (partition by id_task
order by id_matrix desc, row desc, col desc) as id_task_ro
        from
        (select 1 as id_matrix, id_task, row, col, val from
WINEQUALITY_TRAIN_Y_RCV_4SPLIT
        union all
        select 2 as id_matrix, id_task, row, col, val from
WINEQUALITY_TRAIN_X_RCV_4SPLIT) a ) i,
    TABLE(nzm..lm_tf(i.id_task_ro,i.id_matrix,i.row,i.col,i.val)) o;
```

```
commit;

-- apply the best model
drop table WINEQUALITY_R_MAT_TABLE1;
create table WINEQUALITY_R_MAT_TABLE1 as
    select z.row, sum(z.value*m1.val) as res
    from
    WINEQUALITY_TEST_x_RCV z
    right join
    WINEQUALITY_M_MAT_TABLE m1 on (z.col=m1.row)
    where m1.id_task= (select id_task as best_id from winequality_m_mat_table
where id_matrix=2 and row=3 order by val desc limit 1)
     and m1.id_matrix=1 and m1.col=1 group by m1.id_task, z.row;
-- RSS of the best model on test subset
select sum((y.value-r.res)*(y.value-r.res)) as resi from
WINEQUALITY_test_y_rcv y right join WINEQUALITY_R_MAT_TABLE1 r on (y.row =
r.row);
-- apply the worst model
drop table WINEQUALITY_R_MAT_TABLE2;
create table WINEQUALITY_R_MAT_TABLE2 as
    select z.row, sum(z.value*m1.val) as res
    from
    WINEQUALITY_TEST_x_RCV z
    right join
    WINEQUALITY_M_MAT_TABLE m1 on (z.col=m1.row)
    where m1.id_task= (select id_task as best_id from winequality_m_mat_table
where id_matrix=2 and row=3 order by val asc limit 1)
     and m1.id_matrix=1 and m1.col=1 group by m1.id_task, z.row;
-- RSS of the worst model on test subset
select sum((y.value-r.res)*(y.value-r.res)) as resi from
WINEQUALITY_test_y_rcv y right join WINEQUALITY_R_MAT_TABLE2 r on (y.row =
r.row);
```

Bulk Principal Component Analysis (PCA)

Background

The bulk PCA function implements the PCA algorithm for bulk matrices. For bulk PCA, multiple PCA models for different data groups are created in parallel. For each input matrix, one PCA model with different predictors is created. From the created bulk of PCA models, you can optionally select the best model. By using a set of quality indicators, you can identify the model that suits your purpose best.

Applications

Similar to bulk linear regression, bulk PCA can be applied when the input data can be split into logical chunks. Such input data can be data for individual customers, or when several models are evaluated in parallel.

Available Functionality

Bulk PCA has the following functionality:

- For each input matrix, the rotation matrix is returned.
- For each model, the standard deviation that is associated with each feature is returned.
- For each model that is applied to new data, the projected matrix is returned.
- The creation of the PCA models for different data groups is done in parallel.
- The following attributes and matrices are supported:
  - Continuous attributes
  - Matrices with up to 1000 attributes
  - Matrices with up to 10000 rows

Examples

The examples show the following operations:

- Initializing parameters by using the udf_init_parameters UDTF and setting parameters by using the udf_set_parameter UDTF
- Calculating a PCA rotation matrix and the distribution in percent of explained variables by using the nzm..pca_tf UDTF
- Standardizing a matrix by using the nzm..standardize UDTF

The following example shows how to initialize the creation of a model by using the udf_init_parameters UDTF and the udf_set_parameter UDTF:

```
begin transaction;
select nzm..init_parameters(1) from _v_dual_dslice;
select nzm..set_parameter('k','5') from _v_dual_dslice;
--and/or
--select nzm..set_parameter('p','95') from _v_dual_dslice;
```

The following example shows how to create a PCA rotation matrix by using the nzm..pca_tf UDTF

```
create table PCA_RES as
      select i.id_task, o.id_matrix, o.row, o.col,
      o.val from
      (select *,
            nzm..reorder_matrix(id_task) over (partition by id_task
            order by row desc, col desc) as id_task_ro from INP_TBL
      ) i,
      TABLE(nzm..pca_tf(i.id_task_ro,i.row,i.col,i.val)) o;
```

The following table shows the parameters for the nzm..pca_tf UDTF. These parameters control the size of the output matrices.

You can specify the maximum number of components in the PCA model in one of the following ways:

Directly by using a parameter k

By defining a minimum percentage of the accumulative variance that is explained by the components

Table 71: Parameters for the nzm..pca_tf UDTF

Parameter	Description	Values
k	Determines the number of components with the highest percentage of explained variation	
p	Determines the maximum percentage of explained variance that amounts to returning the first *kj* components (sorted by descending explained variance) with a cumulative percentage of explained variance not exceeding p	[0,100] Default: 100

Notes:

If you set both parameters, **k** and **p**, the minimum of k and *kj* (associated with p) is returned.

If you use the default values (*k* = *cj* or *p* = 100), loadings and standard deviations that are associated with all features are returned.

The nzm..pca_tf UDTF returns the following matrices:

cj-by-*kj* matrix with id_matrix = 1 with variable loadings, where *cj* that is the number of attributes, and *kj* that is the number of features

kj-by-1 matrix with the standard deviations with id_matrix = 2

You can apply all models to different associated data groups, or you can apply a single model to all data groups.

The following example shows how to apply all models by using the nzm..product_tf UDTF:

```
create table PCA_APP as
       select i.id_task, o.row, o.col, o.val
       from
       (select *,
            nzm..reorder_matrix(id_task) over (partition by
                  id_task order by id_matrix desc,
                  row desc, col desc) as id_task_ro
            from (select id_task, 2 as id_matrix, row, col,
                  val from PCA_RES pr where pr.id_matrix = 1
                  union all
                  select id_task,1 as id_matrix, row, col,
                  val from NEW_DATA) a
            ) i
      ,TABLE(nzm..product_tf(i.id_task_ro,i.id_matrix,i.row,i.col,
            i.val)) o;
```

The following example shows how to apply a single model by using the nzm..product_tf UDTF. The dimensions of each task from the NEW_DATA table must conform to the size of the selected model.

```
create table PCA_APP as
      select i.id_task, o.row, o.col, o.val
      from
      (select *,
            nzm..reorder_matrix(id_task) over (partition by
                  id_task order by id_matrix desc, row desc,
                  col desc) as id_task_ro
            from (select b.id_task, 2 as id_matrix, row, col,
                  val from PCA_RES a
                  CROSS JOIN
                  (select distinct id_task from PCA_RES) b
                  where a.id_task = 1 and a.id_matrix = 1
                  union all
                  select id_task,1 as id_matrix, row, col,
                  val from NEW_DATA) c
            ) i
      ,TABLE(nzm..product_tf(i.id_task_ro,i.id_matrix,i.row,i.col,
            i.val)) o;
```

The following example shows how to standardize a matrix by using the nzm..standardize UDTF.

```
create table MAT_STAND as
      select i.id_task, o.id_matrix, o.row, o.col,
      o.val from
      (select *,
            nzm..reorder_matrix(id_task) over (partition by id_task
            order by row desc, col desc) as id_task_ro from INP_TBL
      ) i,
      TABLE(nzm..standardize(i.id_task_ro,i.row,i.col,i.val)) o;
```

The following table shows the parameters for the nzm..standardize UDTF. These parameters control the schema of the standardization.

Table 72: Parameters for the nzm..standardize_tf UDTF

Parameter	Description	Values
center	Determines whether column centering is done	0 - Column centering is not done.

Parameter	Description	Values
		1 - Default. Column centering is done.
scale	Determines whether column scaling is done	0 - Column scaling is not done. 1 - Default. Column scaling is done.

The nzm..standardize UDTF returns the following matrices:

cj-by-*rj* matrix with standardized values of the input matrix with id_matrix = 1, where:

cj is the number of input attributes, for example, columns

rj is the number of input rows

cj-by-2 matrix with standardization coefficients with id_matrix = 2, where:

The first row contains means of columns or 0 if centering was not done

The second row has one of the following contents:

Root mean square if centering is not specified

Standard deviation of the column if centering is specified

1 if scaling is not specified

APPENDIX A

Notices and Trademarks

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR

IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
26 Forest Street
Marlborough, MA 01752 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. Copyright IBM Corp. (enter the year or years). All rights reserved.

Trademarks

IBM, the IBM logo, ibm.com and Netezza are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

NEC is a registered trademark of NEC Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Red Hat is a trademark or registered trademark of Red Hat, Inc. in the United States and/or other countries.

D-CC, D-C++, Diab+, FastJ, pSOS+, SingleStep, Tornado, VxWorks, Wind River, and the Wind River logo are trademarks, registered trademarks, or service marks of Wind River Systems, Inc. Tornado patent pending.

APC and the APC logo are trademarks or registered trademarks of American Power Conversion

Corporation.

Other company, product or service names may be trademarks or service marks of others.

Regulatory and Compliance

Regulatory Notices

Install the NPS system in a restricted-access location. Ensure that only those trained to operate or service the equipment have physical access to it. Install each AC power outlet near the NPS rack that plugs into it, and keep it freely accessible. Provide approved circuit breakers on all power sources.

Product may be powered by redundant power sources. Disconnect ALL power sources before servicing. High leakage current. Earth connection essential before connecting supply. Courant de fuite élevé. Raccordement à la terre indispensable avant le raccordement au réseau.

Homologation Statement

This product may not be certified in your country for connection by any means whatsoever to interfaces of public telecommunications networks. Further certification may be required by law prior to making any such connection. Contact an IBM representative or reseller for any questions.

FCC - Industry Canada Statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

This Class A digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe A respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

CE Statement (Europe)

This product complies with the European Low Voltage Directive 73/23/EEC and EMC Directive 89/336/EEC as amended by European Directive 93/68/EEC.

Warning: This is a class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

VCCI Statement

この装置は、情報処理装置等電波障害自主規制協議会 （VCCI） の基準に基づくクラスＡ 情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起越すことがあります。この場合には使用者が適切な対策を講ずるう要求されることがあります。