IBM® Netezza® Analytics
Release 3.0.1.0

Netezza Package for R Developer's Guide

Revised: March 31, 2014

Part Number 00J2223-03 Rev. 3

Note: Before using this information and the product that it supports, read the information in Notices and Trademarks on page 81.

Contents

4 Netezza Analytics Library for R

5 Netezza Matrix Package

List of Tables

Preface

Audience for This Guide

This guide is designed for users who are interested in using IBM PureData™ System for Analytics with R Language functionality. Before using this package, you should have a thorough understanding of mathematics and statistics, and R language skills. Further, you should be familiar with the basic operation and concepts of IBM PureData System for Analytics and the NPS.

Purpose of This Guide

This guide describes the Netezza Package for R GUI, which consists of the following libraries:

- Netezza Analytics Library for R
- Netezza R Library
- Netezza Matrix Library for R

This guide also describes how researchers, developers, business analysts, and other specialists can use these libraries as tools to leverage the capabilities of IBM PureData System for Analytics with IBM Netezza Analytics in their work.

Netezza Analytics Support

If you are having trouble using Netezza Analytics or any of its components:

- Retry the action, carefully following the instructions in the documentation.
- Send an email to FEEDBACK_INZA@wwpdl.vnet.ibm.com.
  In the body of the email, add the following information:
  - The currently installed version of Netezza Analytics
  - A detailed description of the problem

If you have a question about the IBM Netezza Developer Network (NDN program), such as membership or training, send your question to ndnsupport@wwpdl.vnet.ibm.com.

NPS Appliance Support

If you are having trouble using the IBM Netezza appliance and the issue is not specifically related to use or development of Netezza Analytics, do the following steps:

1. Retry the action, carefully following the instructions in the documentation.
2. Go to the IBM Netezza Knowledge Base at https://knowledge.netezza.com. Enter your support user name and password. You can search the knowledge base or the latest updates to the product documentation. Click **Netezza HelpDesk** to submit a support request.
3. If you are unable to access the IBM Netezza Knowledge Base, you can also contact IBM Netezza Support at the following telephone numbers:
   - North American Toll-Free: +1.877.810.4441
   - United Kingdom Free-Phone: +0.800.032.8382
   - International Direct: +1.508.620.2281

For details about your support plan choices and coverage, see your IBM Netezza maintenance agreement.

Comments on Netezza Analytics or the Documentation

Let us know what you like or dislike about Netezza Analytics functionality. Send an email to FEEDBACK_INZA@wwpdl.vnet.ibm.com and include the following information:

- Appropriate version numbers (NPS, Netezza Analytics, nzcm, registered functions)
- Your specific comments
- Your name, organization and email address

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Send an email to netezzadoc@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.

Conventions

The following conventions apply:

- In the technical literature, both the guides and reference guides, the term “Analytic Executable” or “AE” is used. In marketing materials, the term “User-Defined Analytic Process” or “UDAP” is used. The terms User-Defined Analytic Process and UDAP are synonymous with the terms Analytic Executable and AE.
- Upper case for SQL commands; for example INSERT, DELETE

- In some instances to denote parameter names, argument names, or other named references, bold font is used.
- In file names and commands, a term surrounded by angle brackets (< >) indicates a placeholder that should be replaced with the actual text such as a revision number, database name, or user name.
- In code samples, a single *backslash* (“\”) at the end of a line denotes a line continuation and should be omitted when using the code at the command line, a SQL command or in a file.

CHAPTER 1

Introduction to R and the R Environment

Introduction to R

R is a language as well as an environment that is primarily used for statistical computing and related graphic creation. It is similar to and based upon the S language and environment, which is developed at Bell Laboratories and can be considered as a different implementation of S.

R provides a wide variety of statistical techniques such as linear and nonlinear modeling, classic statistical tests, time-series analysis, classification, and clustering. It also includes graphical techniques. Because of its focus on statistical techniques, it works well as a tool for implementing Analytic Executables (AEs) and for performing data analytics by using Netezza Analytics.

R is freely available under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on Windows, MacOS, and a wide variety of UNIX platforms and similar systems, including FreeBSD and Linux. Because of the public nature of the code and the ability for outside parties to develop packages, R is highly extensible. For a detailed description of the packages, see R Packages and CRAN.

Note: Downloading, installing, and working with Open Source R and all other required packages is subject to the terms and conditions that are mentioned in the appropriate license files of those packages.

The R Environment

R is more than a language. Is it an integrated suite of software facilities for data manipulation, calculation, and graphical display. Therefore, the term *environment* is used to characterize R as a coherent system rather than a collection of isolated tools.

R is designed around a true computer language, allowing users to add additional functionality by defining new functions. For computationally intensive tasks, C, C++, and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

R Packages and CRAN

As a GNU project, extensions to the base R language and environment, typically referred to as *packages*, are freely available. Packages exist to expand the usage for many different types of projects and functionality, with more packages being developed all the time. Certain packages are supplied with the R distribution. Many more packages are available through the Comprehensive R Archive Network (CRAN) family of Internet sites covering a wide range of modern statistics. CRAN is a network of File Transfer Protocol (FTP) and Web servers around the globe that store identical, up-to-date versions of code and documentation for R.

In this document, references to CRAN-style packages indicate that the packages were developed by using the methodology and standards of the CRAN project. However, because these packages are specific to Netezza Analytics, they are not officially submitted to CRAN but published on NDN instead.

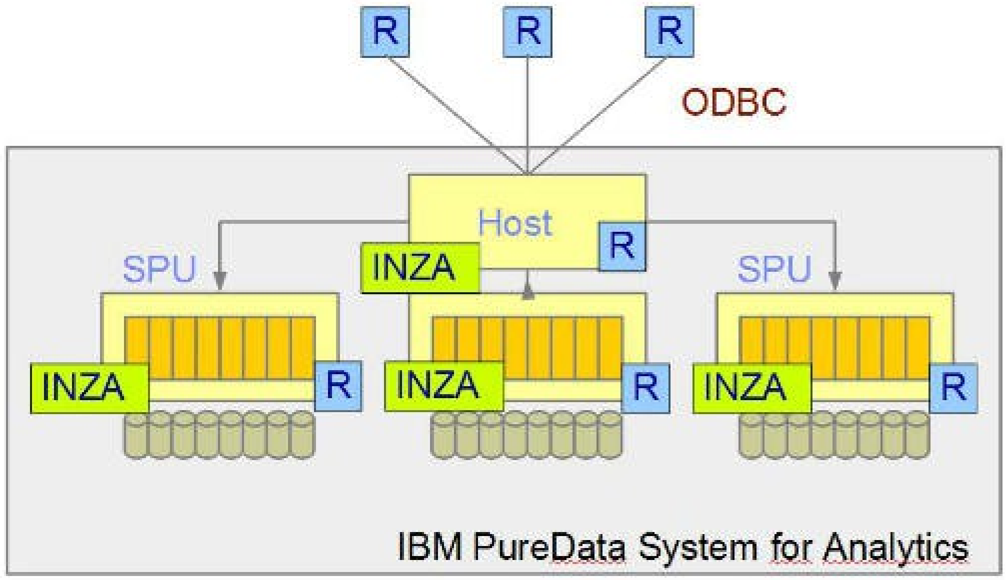
Overview of the R Netezza Analytics Functionality

Using R on IBM PureData System for Analytics, offers a solution that allows users to get the best of both worlds, the flexibility and richness of the R language and the scalability and enterprise strength integration of IBM PureData System for Analytics. Figure 1 shows the basic architecture of this integration. One instance of R is running on each of potentially many clients connecting to the data warehouse. Additionally, there are several instances of R running in the database.

IBM PureData System for Analytics stores data in partitions distributed on blades. R is running on these blades allowing to access and process data in parallel in each partition. Additionally, there is an R instance running on the host.

Both the R instances on the server, and the instance on the client are extended by packages that provide the seamless integration between both. Users mostly interact with the client instance of R, just as if working locally. The client packages then manage the communication with the server to actually execute analytics queries on the server and on the blades.

Figure 1: Basic architecture of R and IBM PureData System for Analytics



There are several ways for the R client to interact with the database server.

Data in the database can be queried from the R client instance in a transparent way by defining local data structures that serve as proxies. Upon access to these local proxy objects, internally SQL is generated and invoked to transfer the data from the server to the client. Thus, the user can work with remote data in a similar way as with local data and still profit from pushing projections and selections into the database. Also, simple operations on the database can be invoked through R wrappers, allowing R users who are not familiar with SQL to perform these operations. This functionality is described in Netezza R Library.

Instead of loading the data to the client, the R user can invoke own R functions or transformation on the server, where they are executed either on the host, or in parallel on the data slices. This is achieved by transferring the user R code to the server, where it is interpreted by the server side R instances. Because user code often depends on additional libraries, these can be installed on the server to provide the required environment. How to execute user-defined code on IBM PureData System for Analytics is described in The Netezza R Library .

Instead of using own algorithms for statistics or matrix tasks, the user can rely on high performance implementations of many standard algorithms as implemented in Netezza Analytics. These algorithms can be invoked through R wrapper functions that return local R objects of the same type or of similar type as the original R algorithms for these tasks. For details about this functionality, see Netezza Analytics Library for R and Netezza Matrix Package.

Finally, there are scenarios, in which R code should be integrated in an environment that does not use R client instances, but rather SQL based tools, such as IBM Cognos. In these cases, R code can be deployed into user-defined functions that can be invoked from any SQL-based client. The client application does not need to be aware of the fact that these UDFs are actually backed by R. For details about these scenarios, see the *IBM Netezza Analytics User-Defined Analytic Process Developer's Guide*.

R Server Instances on the NPS

While integration tools for using R with Netezza Analytics are provided in the Netezza Analytics installation, an R server package is not distributed with the product.[1]

To make use of R for AEs and analytic activities, a system administrator must install an R package on the NPS before the provided tools for interfacing with R can be used. For information about how to install R on the server, refer to the instructions on NDN or contact your sales representative for further support.

The Netezza Client Packages for R

The following libraries create the Netezza Package for R:

- The Netezza R Library with functions to connect to the Performance Server, to operate on tables in the database and to run R user-defined code in the database
- The Netezza Analytics Library for R with functions that allow using in-database procedures to process large data sets
- The Netezza Matrix Library, with functions to allow operations on large matrices in the database

Netezza R Library

The Netezza R Library is a standard CRAN-style R package. It provides capabilities for working with the Netezza system from an R client, allowing you to operate on tables in-database and run your R functions in-database.

Netezza Analytics Library for R

The Netezza Analytics Library for R is a standard CRAN-style R package for accessing Netezza Analytics in-database analytics from an R client. Netezza Analytics contains a set of built-in analytic routines implementing widely-used statistical and data-mining algorithms. These routines are designed to run in the database. Processing is fast and capable of handling extremely large amounts of data.

1 For information about how to install R, see the IBM developerWorks Netezza Developer Network (NDN) community. You need to register first at developerWorks (www.ibm.com/developerWorks). Search for “NDN” to locate the Netezza Developer Network community. Follow the instructions in the overview page to get access to the private part of the community.

Netezza Matrix Library

The *Netezza Matrix Library* contains R functions and operators that can be used to work with matrices stored in the database and to invoke the Netezza Matrix Engine from an R client. The Netezza Matrix Library provides a wide variety of matrix capabilities ranging from simple matrix addition to complex linear algebra operations, such as singular value decomposition (SVD).

CHAPTER 2
Configuring the Local Machine

Introduction

The R environment must be configured on the local machine before R functionality can be used on IBM PureData System for Analytics. Configuration includes preparing the ODBC connection between the local machine and the NPS. It also includes installing a number of additional R packages that are not included in the base R installation but that are required to work with Netezza Analytics.

The following sections describe how to configure the ODBC Drivers and how to configure the local machine to work with R on the NPS through the R GUI for Windows.

ODBC Driver Configuration

Before a connection can be made between the local machine and R on the NPS, an ODBC connection must be made. For detailed information about how to install the ODBC drivers for IBM PureData System for Analytics, see the installation instructions of the NPS ODBC driver that is available on Fix Central.

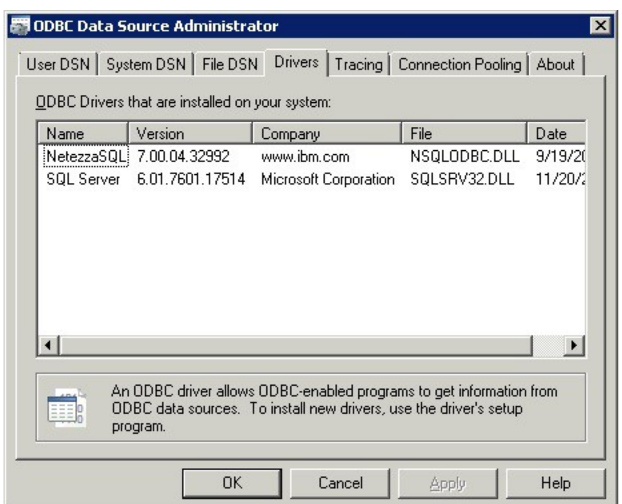
ODBC Driver Configuration for Windows

This section describes how to install and configure the ODBC driver for the 64-bit version of Windows and the 32-bit version of Windows.

1. Download the Windows ODBC drivers from Fix Central by doing the following steps:
   a. Click **Select product**.
   b. From the **Product Group** list, select **Information Management**.
   c. From the **Select from Information Management** list, select **IBM Netezza NPS Software and Clients**.

   d. From the **Installed Version** list, select the version of IBM Netezza NPS that you have installed.
   e. From the **Platform** list, select **Windows**, and then click **Continue**.
   f. Select **Browse for fixes**, and then click **Continue.**
   g. Select the corresponding fix pack for your IBM Netezza NPS version.
      The fix pack contains the *nz-winclient-vxxx.zip* file, where *xxx* is the corresponding version number.
   h. Extract the *nz-winclient-vxxx.zip* file and use one of the following files:
      - For 64-bit Windows, use the *nzodbc32bit4win64.exe* file.
      - For 32-bit Windows, use the *nzodbcsetup.exe* file.
2. After the download is completed, double-click the file name to launch the installer.
3. In the window that opens, select the language to use and click **OK**.
4. Follow the steps of the installer package by clicking **Next >** after each selection.
   The application installs all the necessary files on your computer. A rebooting might be required after installation.
5. Click **Done** to finish the installation; then close the installer application.
6. To check if the installation is completed correctly, open the **Control Panel** and select **Administrative Tools**.
7. From the list, select Data Sources (ODBC).
8. In the dialog box that opens, click the **Drivers** tab.
   NetezzaSQL should appear in the list as shown in the following figure.

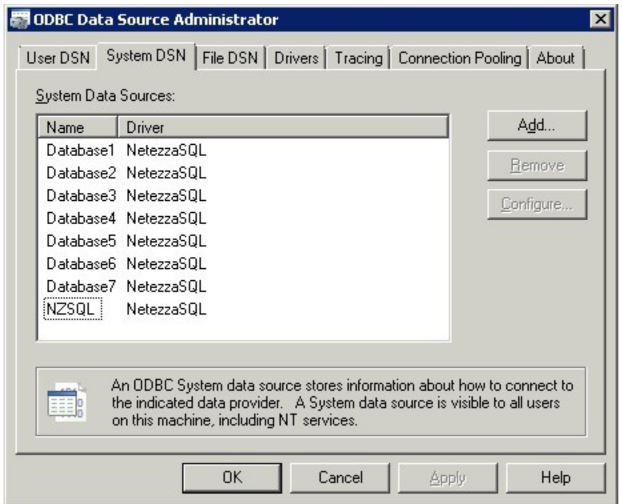
Figure 2: ODBC Data Source dialog box with Drivers Tab selected



9. Click the **System DSN** tab.

The NetezzaSQL driver named NZSQL should appear in the list as shown in the following figure.

Figure 3: ODBC Data Source dialog box with System DSN tab selected



If the local settings match, the installation is complete. If the local settings do not match, reinstall the driver.

Note: You can define custom DSNs in the **System DSN** tab, if necessary.

R Package Configuration

To run the R Language, additional packages must be installed through the R GUI.

Required Netezza Analytics-specific Packages

The following packages are specific to Netezza Analytics. These packages provide an interface between R and the NPS. For a description of these packages, see The Netezza Client Packages for R.

- **nzr<version_number>.zip**—The Netezza R Library. This basic R package is required to access the functionality of the Netezza R Library, the Netezza Analytics Library for R, and the Netezza Matrix Library. Therefore, it must be installed before any R package can be used.
- **nza<version_number>.zip**—The Netezza Analytics Library for R. Provides R functionality for analytics.
- **nzmatrix<version_number>.zip**—The Netezza Matrix Library. Provides R Library functionality for matrices.

These files are available from NDN.

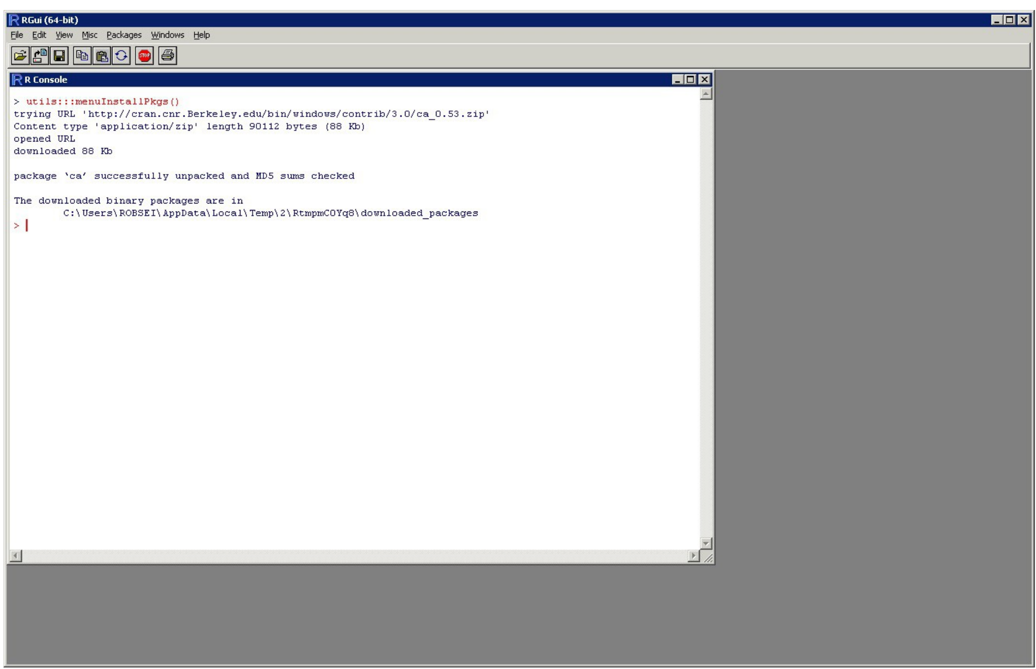
Required Standard Packages

For R to run properly, the following standard packages must be installed on the client. The packages are listed in alphabetical order.

- **arules**—Provides support for association rules
- **arulesViz**—Required for the visualization of association rules as provided in the *nza* package; not required for the installation of the Netezza Analytics packages *nzr*, *nzmatrix,* and *nza*.
- **bitops**—Provides functions for bitwise operations
- **ca**—Provides simple correspondence analysis, multiple correspondence analysis, and joint correspondence analysis
- **caTools**—Provides tools for moving window statistics, GIF, Base64, ROC AUC, and others
- **e1071**—Provides miscellaneous functions of the Department of Statistics (e1071)
- **MASS**—Provides support functions and Datasets for Venables and Ripley's MASS
- **rgl**—Provides a 3D visualization device system
- **RODBC**—Provides ODBC database access
- **tree**—Provides classification and regression trees
- **rpart**—Provides decision and regression trees
- **tree**—Provides classification and regression trees
- **XML**—Provides tools for parsing and generating XML within R

Note: When these packages are installed, dependent packages are also installed if required. Therefore, depending on the order in which the packages are installed, it might not be necessary to manually install each package. For example, when installing the **ca** package, the **rgl** package is automatically installed. Notifications regarding automatically installed dependencies appear in the R GUI console as shown in the following figure.

Figure 4: Installation notices and dependencies on R GUI console



Optional Standard Packages

The following standard packages are optional:

- **arulesViz**—Required for the visualization of association rules as provided in the *nza* package; not required for the installation of the Netezza Analytics packages *nzr*, *nzmatrix,* and *nza*

Installing the Packages

To install the nzr package, the nza package, and the nzmatrix package, do the following steps.

Note: First, you must install the nzr package because is needed to use the nza package and the nzmatrix package.

1. From the R GUI, click **Packages > Install package(s) from local zip files...**
   A dialog box with a list of the available packages opens.
2. Select the nzr package, and then click OK.
3. Repeat step 1 and step 2 to install the nza package and the nzmatrix package.

Acquiring R

Netezza plugins are supported for R GUI version 3.0.x for both x32 and x64. Appropriate versions of R can be downloaded from the official R website at http://www.r-project.org. Follow the installation instructions.

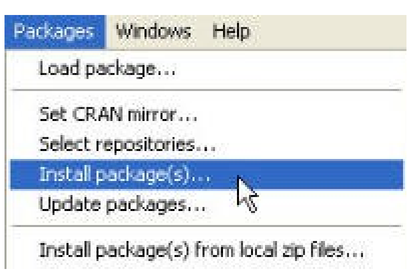
Configuration Instructions for Windows

The following description shows how to install the required packages, and the nzr, nza, and nzmatrix packages by using R GUI on Windows. Steps should be similar for a different platform or client.

To install the packages, do the following steps:

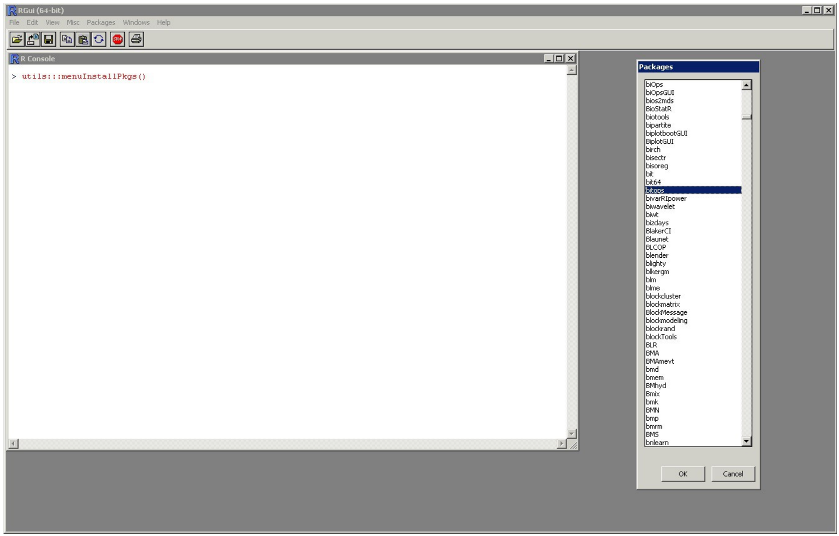
1. Update the R GUI with any appropriate CRAN package by selecting
   Packages > Install Package(s)...
   Note: Using the **Install Package(s)...** option causes the R GUI to make a connection to a CRAN server. Therefore, it might be necessary to select the server before this process can be completed. Using this option avoids the need to manually download the packages to the local machine.

Figure 5: The Packages window



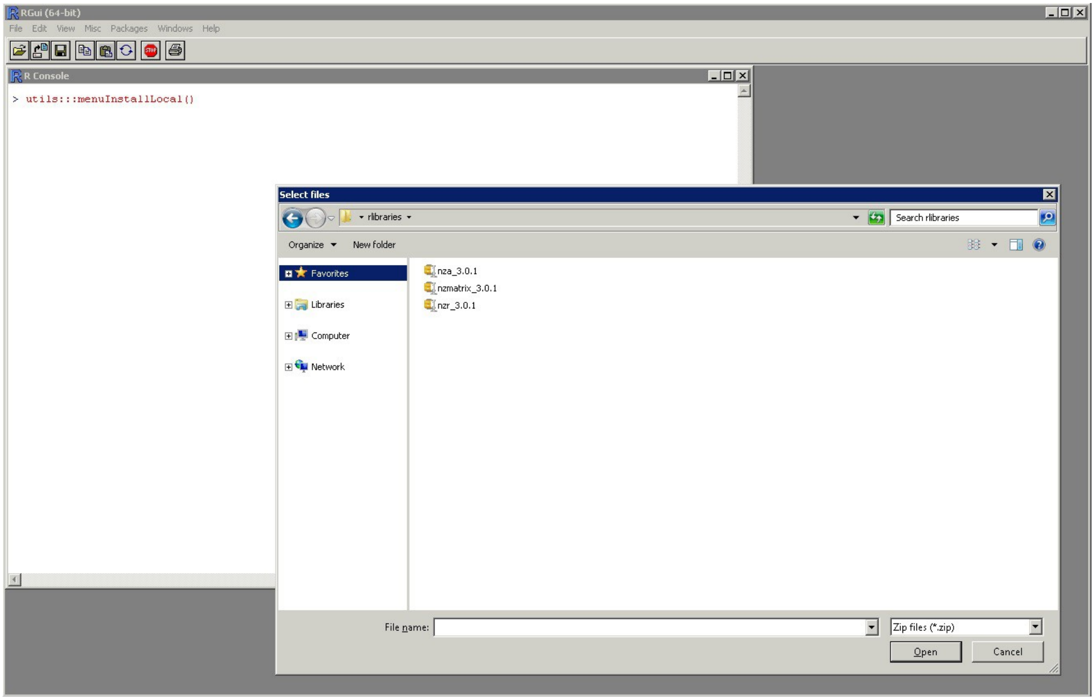
2. From the list of available packages, select the appropriate package, and then click **OK**.

Figure 6: Select a package to be installed



3. Repeat step 1 and step 2 for each package.
4. Download the libraries as needed.
5. After the download is completed, from the **Packages** window, select **Packages > Install package(s) from local zip files...**
6. Navigate to the zip file location on the local machine or network as shown in the following figure.

Figure 7: Select a package to be installed from a .zip file



7. After the file is located, double-click the file name in the window, or select it and click **Open**.
8. Repeat step 5, step 6, and step 7 for each package.

Installation Verification and ODBC Connectivity Check

After installing all Netezza R Library components and completing the configuration of the ODBC driver and the database setup for the Netezza Analytics Library for R, Netezza R Library, and Netezza Matrix Library components, the connectivity of the R GUI with the Netezza appliance must be verified. In the following description, it is assumed that the DSN NZSQL is defined and refers to a database. It is also assumed that the user on IBM PureData System for Analytics have the necessary rights to access the NZA database and to create new tables in the current database.

To verify the installation and configuration, you can use the following commands:

- To verify the Netezza R Library package install and proper configuration of the Netezza software run:
  ```
  library(nzr)
  ```
  This command loads the Netezza R Library libraries into the R GUI. After the libraries are loaded, run:
  ```
  demo(nzr)
  ```
  This command runs a script that demonstrates and checks the basic functionality of the Netezza R Library.
- To verify the Netezza Analytics Library for R package install and the configuration of the Netezza software run:

```
library (nza)
```

This command loads the Netezza Analytics Library for R and the Netezza R Library libraries into the R GUI. After the load is completed, run:

```
demo (nza)
```

This command runs the demo script to demonstrate and check the basic functionality of the Netezza Analytics Library for R.

- To verify the Netezza Matrix Library package install and the configuration of the Netezza software run:

```
library (nzmatrix)
```

This command loads the Netezza Matrix Library and the Netezza R Library libraries into the R GUI. After the load is completed, run:

```
demo (nzmatrix)
```

This command runs the demo script that demonstrates and checks the basic functionality of the Netezza Matrix Library.

Creating a Working Database

Before you start to do analytics by using the Netezza client packages for R, you must create a working database to store the result tables of the analysis. This working database must be enabled for Netezza Analytics.

Prerequisites: To enable your working database for Netezza Analytics after you create it, you must have the right to run the **create_inza_db.sh** script on your NPS machine. This script is in the ***/nz/export/ae/utilities/bin*** directory.

Important: Do not use system databases, such as SYSTEM, NZM, NZA, NZR, NZMSG, and NZRC to store the result tables.

The following example shows how to create the ANALYSIS_DB database. The database owner is DEVUSER.

To create the ANALYSIS_DB database, do the following steps:

1. Log on to your NPS machine and launch nzsql.
2. Run the following commands:
   a. `CREATE USER DEVUSER WITH PASSWORD '<password>';`
      where <password> is a password of your choice.
   b. `ALTER USER DEVUSER WITH IN GROUP inza_admins;`
   c. `CREATE DATABASE ANALYSIS_DB;`
   d. `ALTER DATABASE ANALYSIS_DB OWNER TO DEVUSER;`
   e. `\c ANALYSIS_DB`
   f. `GRANT ALL ADMIN TO DEVUSER;`
3. Quit nzsql by running the following command:
   `\q`
4. Change to the /nz/export/ae/utilities/bin directory by running the following command:

```
cd /nz/export/ae/utilities/bin
```

5. Enable the ANALYSIS_DB database for Netezza Analytics by running the following command:
```
./create_inza_db.sh ANALYSIS_DB
```

6. Enable the rights for the DEVUSER by running the following command:
```
./create_inza_db_developer.sh ANALYSIS_DB DEVUSER
```

Note: The INZA_DEVELOPERS group is for users who need to register new AEs, UDXs, and stored procedures.

CHAPTER 3

The Netezza R Library

Overview

The Netezza R library, which is contained in the NZR package, provides functionality to manipulate database objects, transfer data between client and server, and run user-defined code on the database server. It is also the base for the nza and the nzmatrix packages, which are described in subsequent chapters.

Data Types

To run R user-defined functions on the server requires a good understanding of how different data types in NPS are represented and processed by the Netezza R libraries.

Analytic Executables that are written in R support a number of data types, all of which have a direct equivalent in the Netezza DBMS. Some data types can be represented precisely in R, whereas some data types must be cast to a similar data type, such as a 64-bit integer stored as numeric or double in R. There are also data types that cannot be easily supported in R. The following table identifies all available data types and shows how they are supported in the R Language Adapter.

Table 1: Availability and support of data types in R

Netezza	Supported	R	Comments
FIXED	Yes	Character	
VARIABLE	Yes	Character	
NATIONAL_FIXED	Yes	Character	UTF-8
NATIONAL_VARIABLE	Yes	Character	UTF-8
BOOL	Yes	Logical	
INT8	Yes	Integer	
INT16	Yes	Integer	

INT32	Yes	Integer	
INT64	Yes	Double	loss of precision
NUMERIC32	No		
NUMERIC64	No		
NUMERIC128	No		
FLOAT	Yes	Double	
DOUBLE	Yes	Double	
DATE	Yes	Integer	
TIME	Yes	POSIXct	
TIMETZ	Yes	List	elements: *time* and *zone*
TIMESTAMP	Yes	Double	loss of precision
INTERVAL	Yes	List	elements: *time* and *month*

Character Strings

Netezza columns of types VARIABLE and FIXED are represented in R as character vectors. The Netezza NATIONAL_FIXED and NATIONAL_VARIABLE data types result in UTF-8-encoded character strings, and are handled using Encoding(x) == "UTF-8".

Integers

Integers that are stored in one, two, and four bytes, that is, INT8, INT16 and INT32, are translated to integer vectors. When outputting data, the 4-byte-long integer value is cropped to an appropriate length.

Netezza INT64 columns are represented as numeric vectors in R. Thus, input values greater than $2^{53}-1$ might not be represented correctly. When outputting INT64 values from an R AE, the output value is cast to numeric, and then to 8-byte-long integer.

NA values are not allowed when outputting data to the NPS system when the output format is an integer type. R allows for NA values in the case of integers, but internally, these values are represented as an arbitrarily chosen sequence of bits—currently the largest negative 32-bit integer—which cannot be translated to Netezza integers. If NULL should be output instead, the setOutputNull function can be used.

Boolean

The BOOL data type is represented as logical.

NA values are not allowed when outputting data to the NPS system when the output format is Boolean. If NULL should be output instead, the setOutputNull function can be used.

Floating Point

Netezza supports two floating-point numeric formats, FLOAT and DOUBLE, which are both standardized and described in “IEEE 754 Standard for Floating-Point Arithmetic.” In R, columns of these types are cast as double; similarly, outputting one of these data types means casting from double.

Numeric

Netezza numeric data types are NUMERIC32, NUMERIC64, and NUMERIC128. Currently, none of the NUMERIC types are supported in R. To avoid errors, convert the data types to REAL before using R functions on the data.

Date and Time

Netezza defines a number of data and time formats: DATE, TIME, TIMETZ, TIMESTAMP, and INTERVAL.

Date

The DATE data type is stored as a 4-byte integer and represents the number of days before (-) or after (+) 1/1/2000 (January 1, 2000). In R, it is stored as an integer value.

minimal value	-730,119 (1/1/0001)
maximal value	2,921,939 (12/31/9999)

Time

The TIME data type is stored as an 8-byte integer and represents the number of microseconds between midnight and one microsecond before midnight. In R, it is stored as a double value, but only the integer portion is taken into account, whereas the fractional portion is ignored.

minimal value	0 (00:00:00.000000)
maximal value	86,399,999,999 (23:59:59.999999)

Time with Timezone

The TIMETZ data type consists of two fields: the standard TIME field and a timezone field that is a 4-byte integer representing the offset in seconds, sign reversed. For example, the offset of “+1 hour” is stored as -3600 . The offset must be a whole number of minutes, that is, $\text{offset} \bmod 60 = 0$.

offset minimal value	-46800 (+ 13:00:00)
offset maximal value	46740 (-12:59:00)

Timestamp

The TIMESTAMP data type is an 8-byte integer representing the number of microseconds before (-) or after (+) 00:00:00.0, 1/1/2000. In R, it is stored as data type double, which means that for some values greater than $2^{53}-1$, the rounding error might affect the value that is returned to Netezza.

minimal value	-63,082,281,600,000,000 (00:00:00, 1/1/0001)
maximal value	252,455,615,999,999,999 (23:59:59.999999, 12/31/9999)

Interval

The INTERVAL data type consists of a 4-byte integer—the number of months, signed—and an 8-byte integer—the number of microseconds, signed. A configuration of both a positive (+) or negative (-) months value as well as a positive (+) or negative (-) microseconds values are possible and supported by Netezza. In R, these values are represented as an integer (months) and double (microseconds). The microsecond use means that, as with TIMESTAMP, rounding errors must be taken into account.

Notes:

- The microsecond value can be as large as the INT64 data type allows and overflows into negatives without error. In R, the microsecond value is stored as type numeric and cast to INT64 when it is sent to Netezza. Because the numeric type allows values that are larger than the allowed maximum INT64 value, it is important that values are not larger than the maximum INT64. This rule applies also to the minimal microsecond value.
- A month is always considered to contain 30 days.
- The months and microseconds values are stored separately; they do not exchange information.

months minimal value	3,000,000 (-250,000 years)
months maximal value	3,000,000 (250,000 years)
microseconds minimal value	*none* (minimal signed *INT64*)
microseconds maximal value	*none* (maximal signed *INT64*)

Connection to the NPS System

Before you can work with the NPS system, a connection must be established. This section contains a brief overview of functions that allow you to connect to the NPS.

The two most frequently used functions are **nzConnect()** and **nzConnectDSN()**. These functions pass user-supplied credentials or a predefined Database Source Name (DSN) string respectively. If called with the default value of the **verbose** parameter, they also output the R version and the Netezza R Library-related packages that are installed on the NPS system.

```
#load nzr package
library(nzr)

# connect to a Netezza database "mm" on the "TT4-R040" machine
nzConnect("user", "password", "TT4-R040", "mm")

#Installed version of '  r_ae  ' cartridge:  3.0.1.35826

#On the spus: R 3.0.2
#On the host: R 3.0.2
#
# or:
```

```
nzConnectDSN('NetezzaSQL', verbose=FALSE)
```

To stop a connection, you must call the **nzDisconnect()** function, which removes the hidden variable. Two other functions that are useful for testing connections are **nzCheckConnection()** and **nzIsConnected()**. The **nzCheckConnection()** function throws an error if a connection is not open. The **nzIsConnected()** function returns a boolean value that indicates whether there is a connection to the NPS system.

Details

Either **nzConnect()** or **nzConnectDSN()** must be called before any other Netezza R Library function can be called because a connection function sets a hidden variable, the existence of which is checked in other Netezza R Library functions. Before connecting to a different machine, an existing connection must be stopped by using **nzDisconnect(),** or by setting the **force** argument in **nzConnect()** to **TRUE**.

The function signature with default arguments is:

```
nzConnect(user, password, machine, database, force = FALSE, queryTimeout = 0,
    loginTimeout = 0, verbose = TRUE)

nzConnectDSN(dsn, force = FALSE, verbose = TRUE)
```

Arguments Description

- **user**—the name of a database user, given as a string
- **password**—the password of a database user, given as a string
- **machine**—the name or IP address of a Netezza machine, given as a string
- **database**—the name of a database, given as a string
- **force**—optional parameter to force a connection; the default value is **FALSE**
- **queryTimeout**—optional parameter indicating timeout for a query; the default value is 0, which means “no timeout”
- **loginTimeout**—optional parameter indicating login timeout; the default value is 0, which means “no timeout”
- **verbose**—optional parameter indicating verbose mode
- **dsn**—the DSN string as defined in the local ODBC configuration

Managing Data with the Netezza Library for R

This section describes Netezza R Library data types and some basic functions that allow data manipulation.

nz.data.frame

The most important and frequent construct is the object of the class nz.data.frame. The function nz.data.frame() creates a pointer to a table on the Netezza system. This pointer can later be used to run data transformations with **nzApply,** or **nzRun,** or data mining algorithms. It does not store any data in local memory but rather provides metadata that can be used to determine the correct table subset (columns, or rows, or both) where the user code should run. It is the standard output of the majority of data manipulation functions in the Netezza R Library.

```
nzConnect("user", "password", "TT4-R040", "mm")
# create a reference to the table adult
nzadult = nz.data.frame("adult")

#Show the reference
nzadult
#SELECT
#ID,AGE,WORKCLASS,FNLWGT,EDUCATION,EDUCATION_NUM,MARITAL_STATUS,OCCUPATION,RELA#T
IONSHIP,RACE,SEX,CAPITAL_GAIN,CAPITAL_LOSS,HOURS_PER_WEEK,INCOME FROM ADULT
```

The **nz.data.frame** class implements a number of methods for extracting a subset of its data, gathering meta-info similar to **data.frame**, and working with parallel data-processing algorithms. From Version 7.0.3, NPS can be configured to work with schemas. Tables with schemas can be referenced in the same way, using the schema name in addition to the table name.

```
# create a reference to the table adult
nzadult = nz.data.frame("a.adult")
```

There is currently no support for cross-database access.

Both schemas and tables can be case-sensitive. In this case, they need to be put into to double quotes. The following is an example for accessing table "Adult" (case-sensitive) in schema A (not case-sensitive)

```
# create a reference to the table Adult
nzadult = nz.data.frame('A."Adult"')
```

Columns in database tables are always treated as case-sensitive in R. Column names that are not defined as case-sensitive are transformed to default-case.

[,], $ and dim

A subset of columns, or rows, or both can be specified by using the **[,]** operator.

A limitation is that rows cannot be referenced by their numbers because there is no continuous row numbering on the Netezza system. Instead, you must specify value-based conditions, such as

```
nzdf2 <- nzadult[nzadult$ID>20,]
```

The result of each selection can be partially checked by using the **dim()** function because at this stage, only metadata has been transferred to R. This function returns the number of rows and columns.

```
# there are 15 columns and 32561 rows in the "adult" table
dim(nzadult)
#[1] 32561    15

# selecting columns 5,6 and 7
t1 <- nzadult[,5:7]
#SELECT EDUCATION,EDUCATION_NUM,MARITAL_STATUS FROM ADULT
# t1 has only 3 columns
dim(t1)
#[1] 32561     3
```

```
# selecting columns by their names
(t2 <- nzadult$AGE)
#SELECT  AGE  FROM  ADULT
# t2 has only 1 column
dim(t2)
#[1] 32561     1

# selecting rows satisfying condition AGE>30 and EDUCATIONNUM=10
# from the subset of columns number 1,2,5,7 and 16
t3 <- nzadult[nzadult[,2]>30 & nzadult[,6] == 10,c(1,2,6,7,15)]
t3
#SELECT ID,AGE,EDUCATION_NUM,MARITAL_STATUS,INCOME FROM ADULT WHERE  ( AGE >
#'30' )  AND  ( EDUCATION_NUM = '10' )
# there are fewer rows and fewer columns than in the initial table
dim(t3)
#[1] 4226    5
```

head, tail

To get a sample of the data, you can use the **head()** and **tail()** functions. The functions pull the specified data from the start or end of the data set.

```
head(t3,4)
#   ID AGE EDUCATION_NUM     MARITAL_STATUS INCOME
#1  28  54            10 Married-civ-spouse  large
#2  56  43            10 Married-civ-spouse  large
#3  92  37            10           Divorced  small
#4 140  49            10 Married-civ-spouse  large

tail(nzadult[,1:4])
#         ID AGE WORKCLASS FNLWGT
#32556 32538  30   Private 345898
#32557 32542  41   Private 202822
#32558 32546  39 Local-gov 111499
#32559 32550  43 State-gov 255835
#32560 32554  32   Private 116138
#32561 32558  40   Private 154374
```

as.data.frame

To look at the complete data set, it must be downloaded from the Netezza system by using **as.data.frame.** Because *adult* is a large data set, in the following example one of the data frames created above, (t3), is used instead.

```
reg_df <- as.data.frame(t3)
head(reg_df)
#   ID AGE EDUCATION_NUM     MARITAL_STATUS INCOME
#1  28  54            10 Married-civ-spouse  large
#2  56  43            10 Married-civ-spouse  large
#3  92  37            10           Divorced  small
#4 140  49            10 Married-civ-spouse  large
#5 204  42            10      Never-married  small
#6 264  59            10 Married-civ-spouse  small
class(reg_df)
#[1] "data.frame"
```

as.nz.data.frame

Another useful data manipulation function is **as.nz.data.frame**. It creates an *nz.data.frame* object from a different R object. Then, a Netezza system table is created, and the passed data is inserted into this table. The created object points to the newly created system table.

This example shows how an *nz.data.frame* object can be created from another R object, in this case from a data.frame iris.

```
data(iris)
if (nzExistTable("nziris")) nzDeleteTable("nziris")
d = as.nz.data.frame(iris, "nziris")
d
#SELECT  Sepal_Length,Sepal_Width,Petal_Length,Petal_Width,Species FROM nziris
class(d)
#[1] "nz.data.frame"
#attr(,"package")
#[1] "nzr"
```

The *iris* data set is now stored in an Netezza system table NZIRIS. If the second argument is not specified, the table name is randomly generated.

Details

The function signatures with default arguments are:

```
nz.data.frame(table1, case.sensitive = NULL)
as.data.frame(x1, row.names=NULL, optional=FALSE, max.rows=NULL,
              order.by=TRUE, ...)
as.nz.data.frame(x2, table2 = NULL, distributeon = NULL, fast=TRUE)
```

Arguments Description

- **table1**—the name of a table available on the Netezza system in the currently-used database
- **row.names**—not used, included for compatibility
- **optional**—not used, included for compatibility
- **x1**—object to be coerced to *data.frame*
- **max.rows**—optional argument; maximum number of rows to be transferred to the client
- **order.by**—optional argument; denotes whether ordering should be used
- **x2**—object to be coerced to *nz.data.frame*
- **table2**—optional argument; table name; if not provided, the function selects a name
- **distributeon**—optional argument; column name; data distribution on the Netezza system is based on this column
- **fast**—optional argument; if set to **FALSE**, when creating a table, multiple inserts are performed; this option requires the data to be stored locally in a temporary file.

Running SQL Code

With the Netezza R Library package, it is possible to run any SQL query on the Netezza system and return its results into the R client by using the nzQuery() and nzScalarQuery() functions. The nzQuery() function returns a *data.frame* object with the query results, whereas the nzScalarQuery() function returns the query result forced to a single scalar value.

The following examples show both functions.

```
t = nzQuery("SELECT * FROM _V_DUAL_DSLICE")
t
#   DSID
#1     1
#2     3
#3     4
#4     2
t = nzScalarQuery("SELECT COUNT(*) FROM _V_DUAL_DSLICE")
t
#[1] 4
```

For debugging, the Netezza R Library package provides the **nzDebug()** function, which switches debugging on or off, so that some functions available in the package print additional debug information.

This example repeats a sample that is shown previously in this section, this time with debugging switched on. For more information about debugging, see More on Debugging.

```
nzDebug(TRUE)
nzadult = nz.data.frame("adult")
#select current_schema
#SELECT CAST(COUNT(*) AS INTEGER) AS field FROM _v_obj_relation WHERE objname =
#'ADULT' AND schema ='ADMIN'
#select current_schema
#SELECT attname AS field FROM _V_RELATION_COLUMN WHERE name = 'ADULT' AND #schema
='ADMIN' ORDER BY ATTNUM
```

Details

For the query functions, all parts of the input query are concatenated with paste(..., sep="") and the result is passed to the Netezza system. The nzDebug() function sets up a global variable **.nzDebug** with a value that is equal to the value of the onoff parameter. The nzDependencies() function accepts a vector containing names of packages to check.

```
nzQuery(..., as.is = TRUE)
nzScalarQuery(..., as.is = TRUE)
nzDebug(onoff = TRUE)
```

Arguments Description

- **...**—any number of query parts passed to **paste**
- **as.is—**denotes whether R should leave the result column as is or run the default RODBC-type conversions
- **onoff**—turn debugging on (**TRUE**) or off (**FALSE**)
- **types**—a vector containing the names of packages to check

Netezza SQL Command Wrappers

Two Netezza SQL (nzsql) wrappers are used frequently. To check if a table exists, use the nzExistTable() function. To delete a table, use the nzDeleteTable() function. These are wrappers of nzsql commands or basic functions.

```
if (nzExistTable('tmpTable')) nzDeleteTable('tmpTable')
```

Other wrappers on nzsql commands or common functions include:

- **nzCreateView**
- **nzDropView**
- **nzJoin**
- **nzJoin.permanent**
- **nzMerge**
- **nzTruncateTable**

For a detailed description of these functions, see the Netezza R Library package manual.

Running User-Defined Functions

User-defined functions can run either on each row, or on each group of rows given a grouping column. The first case is covered by nzApply(), the second functionality is realized by nzTAapply() function. There are also two more flexible functions, nzRun() and nzRunHost() that allow users to iterate through the data manually. Because the latter functions use essentially the same interface that is used for implementing UDFs in R, see the *IBM Netezza Analytics User-Defined Analytic Process Developer's Guide* for more details.

nzApply

The nzApply() function applies a user-provided function to each row of a given distributed data frame (nz.data.frame). For each processed row, it expects at most one result row (vector, list) that is inserted into the output nz.data.frame. An example is presented below:

```
data(iris)

if (nzExistTable('iris')) {nzDeleteTable('iris')}
       d <-as.nz.data.frame(iris)

f <- function(x) { return(sqrt(x[[1]])) }

if (nzExistTable('apply_output')) nzDeleteTable('apply_output')

r <- nzApply(d[,1], NULL, f, output.name='apply_output',
                 output.signature=list(SQUAREROOT=NZ.DOUBLE))
head(r)
#  SQUAREROOT
#1   2.645751
#2   2.626785
#3   2.366432
#4   2.366432
#5   2.366432
#6   2.366432

# this exists also as an overloaded apply method and the following
# returns the same result
nzDeleteTable('apply_output')
r <- apply(d[,1], NULL, f, output.name='apply_output',
       output.signature=list(SQUAREROOT=NZ.DOUBLE))
```

When you apply a function to a table, be careful with data types. Either specify the exact subset of columns the types of which match the types that are expected by the function, or add casting of the columns to the desired format for the given function:

```
f <- function(x) { return(sqrt(as.numeric(x[[1]]))) }
if (nzExistTable('apply_output')) nzDeleteTable('apply_output')
r <- nzApply(d, NULL, f, output.name='apply_output',
                 output.signature=list(SQUAREROOT=NZ.DOUBLE))
head(r)
#  SQUAREROOT
#1   2.258318
#2   2.213594
#3   2.213594
#4   2.258318
#5   2.258318
#6   2.258318
```

nzTApply

The nzTApply() function applies a user-provided function to each subset (group of rows) of a given distributed data frame (nz.data.frame). The subsets are determined by a specified index column. The results of applying the functions are put into a data frame. In the example below, the same nz.data.frame as in the nzApply() example is used. The example contains the *iris* data set.

```
print(d)
#SELECT  Sepal_Length,Sepal_Width,Petal_Length,Petal_Width, Species FROM nziris
# the following lines do the same - compute the mean value
# in every group
nzTApply(d, d[,5], mean)
nzTApply(d, 'Species', mean)
nzTApply(d, 5, mean)
# Sepal_Length Sepal_Width Petal_Length Petal_Width Species    Species
#1        6.588       2.974        5.552       2.026     nan  virginica
#2        5.006       3.428        1.462       0.246     nan     setosa
#3        5.936       2.770        4.260       1.326     nan versicolor
```

Details

The output of these functions depends on whether **output.name** and **output.signature** are specified. For nzApply(), an object of class *data.frame* is returned. The object has the same number of columns as the sequences that are returned from **fun**. If the **output.name** is not provided, no table is created on the IBM PureDatea for Analytics system. For nzTApply(), if an **output.name** is provided, the **output.signature** must also be specified. The **output.signature** parameter can be used to avoid receiving a sparse table and to set the desired output columns types; if the parameter is provided, **fun** must return values that can be cast to these types.

If the **fun** function causes errors, the debugger mode can be used to investigate conditions where errors occur. For more detailed information, see More on Debugging. When option **debugger.mode=TRUE**, then the result table is not stored in the Netezza system**.** Instead, for every group a diagnostic test is called, and the environment for the first group that causes an error is transported to the local R client and opened in the R debugger.

Consider the following R code:

```
nziris = nz.data.frame('iris')
FUN5 = function(x) {
```

```
        if(min(x[,1]) < 4.5) cov(0) else min(x[,1]) }
nzTApply(nziris, 5, FUN5, debugger.mode=T)
```

While in debug mode, the function nzTApply() returns a summary for group processing. This summary is presented in a table with the following columns:

- The first column contains the outcome or error description
- The second column contains the type of outcome (try-error in case of error)
- The third column contains the group name for which the given result is returned

In this example, there are three groups, where one group produces an error.

```
Found 1 error
values type group
1 101 integer virginica
2 supply both 'x' and 'y' or a matrix-like 'x' try-error setosa
3 51 integer versicolor
```

Then, for the first group that caused an error, a dumped environment is downloaded from the remote SPU to the R client and opened in the R debugger. For more detailed information, see More on Debugging.

```
nzApply(X, MARGIN, FUN, output.name = NULL, output.signature = NULL,
         clear.existing = FALSE, ...)
nzTApply(X, INDEX, FUN = NULL, output.name = NULL, output.signature = NULL,
         clear.existing = FALSE, debugger.mode = FALSE, ..., simplify = TRUE)
```

Arguments Description

The following arguments are used with the nzApply() and nzTAapply() functions.

- **X**—input data frame
- **MARGIN**—currently not used but the argument is required; NULL must be passed
- **FUN**—user-defined function
- **FUN** can return a scalar value or a row. It receives a subset of the input data in a form of a *data.frame* with columns names in lower case.
- **output.name**—name of the output table created on the Netezza system
- **output.signature**—data types for output table columns; if not provided, a generic (sparse) table is created
- **clear.existing**—if **TRUE**, delete the output table if it currently exists
- **debugger.mode**—if **TRUE**, nzTApply works in debugger mode
- **...**—these arguments are passed to **fun**
- **simplify** – not used, included for compatibility
- **INDEX**—the value used to index the data set where **INDEX** may be supplied as of the following items:
  - A character string the value of which must be present among columns of **X**
  - **A**n integer not greater than the number of columns of **X**

CRAN

The Comprehensive R Archive Network[2] (CRAN) is a network of servers around the world that store open-source R distributions, extensions, documentation, and binaries. The repository has grown from only 12 packages in 1997 to over 2500 packages currently. Most of the mirror servers are hosted on universities across the world, creating an active, open-source community. The repository is extensively used by the R community, due to the large number of add-on packages, which are generally available under the GPL license. Users can take advantage of the CRAN repository and download the chosen packages, whereas they should consider that these packages are completely external to Netezza.

CRAN packages can be installed on the NPS. The Netezza R Library package provides tools for installing and managing CRAN packages on the Netezza appliance.

nzInstallPackages, nzIsPackageInstalled

To install a package, use nzInstallPackages(). Note that on both, the Host and the SPUs, a two-step installation process is the default. The function output (installation log) for a successful installation of a package is presented below.

```
nzConnectDSN('NetezzaSQL')
nzInstallPackages("http://cran.r-project.org/src/contrib/bitops_1.0-4.1.tar.gz")

#Host:
#Installing: /nz/export/ae/workspace/nz/r_ae/bitops_1.0-4.1.tar.gz
#* installing to library '/nz/export/ae/languages/r/2.10/host/lib64/R/library'
#* installing *source* package 'bitops' ...
#** libs
#/nz/export/ae/sysroot/host/bin/i686-rhel4-linux-gnu-gcc -std=gnu99
#-I/nz/export/ae/languages/r/2.10/host/lib64/R/include  -m32    -fpic  -m32 -c
#bit-ops.c -o bit-ops.o
#/nz/export/ae/sysroot/host/bin/i686-rhel4-linux-gnu-gcc -std=gnu99
#-I/nz/export/ae/languages/r/2.10/host/lib64/R/include  -m32    -fpic  -m32 -c
#cksum.c -o cksum.o
#/nz/export/ae/sysroot/host/bin/i686-rhel4-linux-gnu-gcc -std=gnu99 -shared
#-m32 -L/nz/export/ae/sysroot/host/lib -L/nz/export/ae/sysroot/host/usr/lib
#-L/nz/export/ae/sysroot/host/lib -o bitops.so bit-ops.o cksum.o
#-L/nz/export/ae/languages/r/2.10/host/lib64/R/lib -lR
#** R
#** preparing package for lazy loading
#** help
#*** installing help indices
#** building package indices ...
#* DONE (bitops)
#SPUs:
#Installing: /nz/export/ae/workspace/nz/r_ae/bitops_1.0-4.1.tar.gz
#test: ==: binary operator expected
#test: ==: binary operator expected
#* installing to library /nz/export/ae/languages/r/2.10/spu/lib64/R/library
#* installing *source* package bitops ...
#** libs
#gcc -std=gnu99 -I/nz/export/ae/languages/r/2.10/spu/lib64/R/include  -m32
#-fpic  -m32 -c bit-ops.c -o bit-ops.o
#gcc -std=gnu99 -I/nz/export/ae/languages/r/2.10/spu/lib64/R/include  -m32
#-fpic  -m32 -c cksum.c -o cksum.o
#gcc -std=gnu99 -shared -m32 -L/nz/export/ae/sysroot/spu/lib
#-L/nz/export/ae/sysroot/spu/usr/lib -liconv -o bitops.so bit-ops.o cksum.o
```

2 http://cran.r-project.org/

```
#-L/nz/export/ae/languages/r/2.10/spu/lib64/R/lib -lR
#** R
#** preparing package for lazy loading
#** help
#*** installing help indices
#** building package indices ...
#* DONE (bitops)

To verify package installation, use nzIsPackageInstalled().
nzIsPackageInstalled(bitops)
# host spus
# TRUE TRUE
nzIsPackageInstalled(RODBC)
# host  spus
# TRUE FALSE
```

Details

The nzInstallPackages() function sends the specified CRAN package to the Netezza appliance and installs this package.

- If the **pkg** parameter value starts with **http://**, it is assumed to be a web address. The package is then downloaded from the specified URL and sent to the Netezza appliance.
- If the **pkg** parameter value is a local file, it is sent to the Netezza appliance.

After the file is sent, it is installed on the Netezza system, which involves compiling the C/Fortran code. After the installation and compilation is completed, the installation log is displayed on the screen.

The nzIsPackageInstalled() function checks whether a package is installed on the Netezza appliance Host and SPUs. If the package is found on the Host or SPUs, a message is displayed on the screen. If the package is found in the specified locations, the return value is **TRUE.** If the package is not found in the specified locations, the return value is **FALSE**.

```
nzInstallPackages(pkg, installOnSpus = TRUE)
nzIsPackageInstalled(package)
```

Arguments Description

The following arguments are used with the nzInstallPackages() and nzIsPackageInstalled() functions.

- **pkg**—local file path or a web address; web addresses must begin with “http://”
- **installOnSpus**—optional argument, when **FALSE**, the package is not installed on SPUs
- **package**—name of the package to be checked

External packages can be used on the client and on the server. In the following example, the external **gam** package, which is downloaded from CRAN, is used to build a GAM model on the client. This model then uploaded to the server and applied in-database to the records of an IBM PureData System for Analytics table. This package is installed and loaded on both, the NPS and on client machines, and is used to build the model *model1*. The **pred** function, which uses this package, is applied on the NPS system to an *nz.data.frame*.

```
nzInstallPackages("http://cran.r-project.org/src/contrib/akima_0.5-4.tar.gz")
#(... output log from installation omitted for clarity)
nzInstallPackages("http://cran.r-project.org/src/contrib/gam_1.04.tar.gz")
#(... output log from installation omitted for clarity)
install.packages("gam")
library(gam)
library(nzr)
nzConnect("user","password","tt4-r040","nza")

#
# model is build in R locally on the client
#
model1 = gam(Sepal.Length~Petal.Length+Petal.Width, iris, family=gaussian)

nzIris = nz.data.frame("iris")
pred <- function(x, model1) {
  require(gam)
  predict(model1, data.frame(Petal.Length=as.numeric(x[[2]]),
          Petal.Width=as.numeric(x[[3]])))
}

#
# then the model is applied to all rows in the database
#
nzApply(nzIris, FUN=pred, model1=model1)
```

Storing R objects in Database Tables

The Netezza R library allows you to store serialized versions of your objects in database tables. As NPS does not support large objects (LOBS), objects are stored across several database rows. To make management of such objects easier, the nzr package introduces the object class *nz.list.*

nz.list is aligned with lists in R, although objects are stored remotely instead of locally. An nz.list object is a reference to a specially formatted database table.

To create an nz.list, you can use the following command:

```
nzl <- nz.list("MYNEWLIST",createTable=T, indexType="character");
```

This command creates a table MYNEWLIST and a local stub object *nzl*. The **indexType character** parameter indicates that the index column for the list is represented by varchar in the database. The **indexType** integer would use an integer column for this purpose.

Objects can be stored to the remote list by using the [] and $ operators.

```
#Store an object
nzl['myKey'] ← 1:100000
nzl$myKey ← 1:100000

#Read an object
nzl['myKey']
nzl$myKey

#Delete an object
nzl['myKey'] ← NULL
nzl$myKey ← NULL
```

The *names* function returns all the keys in a list; the *length* function returns the length of a list.

Arguments Description

The following arguments are used with the nz.list constructor:

- **tableName**—The name of the database table containing the list. This can be an existing table or a new one (see createTable).
- **createTable**—optional argument, if **TRUE**; a new table is created if it does not yet exist
- **indexType**—if a new table is created, it indicates the type of index to use, *integer* or *character*

More on Debugging

Most operations are done by calling SQL code that operates on data in the database. When debugging, the SQL code that is being called might be of interest. To turn on verbose information, you can use the **nzDebug()** function.

The following example checks which three SQL **SELECT** commands are used in the **nzTApply()** function.

```
nzDebug(TRUE)
nzTApply(nzIris, "CLASS", function(x) mean(x))
#SELECT filename FROM TABLE WITH
#FINAL(nzr..placefile('base64text','QQoyCjEzMzM3N...
#SELECT UPPER(attname) AS field  FROM _V_RELATION_COLUMN
#                                            WHERE UPPER(name) = UPPER('IRIS')
#SELECT ae_output_t.* FROM (SELECT row_number() OVER(PARTITION BY CLASS ORDER #BY
CLASS) AS nzrn, count(*) OVER (PARTITION BY CLASS) AS nzcnt, from_alias.* #FROM
(SELECT ID,SEPALLENGTH,SEPALWIDTH,PETALLENGTH,PETALWIDTH,CLASS FROM IRIS) #AS
from_alias) AS outer_from, TABLE WITH FINAL
#(nzr..r_udtf(ID,SEPALLENGTH,SEPALWIDTH,PETALLENGTH,PETALWIDTH,CLASS,CLASS,
#nzrn, nzcnt, 'WORKSPACE_PATH=file579be4f1')) AS ae_output_t
#     ID SEPALLENGTH SEPALWIDTH PETALLENGTH PETALWIDTH CLASS      CLASS
#1 125.5       6.588      2.974       5.552      2.026   nan  virginica
#2  25.5       5.006      3.418       1.464      0.244   nan     setosa
#3  75.5       5.936      2.770       4.260      1.326   nan versicolor
nzDebug(FALSE)
```

The **nzTApply()** function provides another way of debugging. Using the **debugger.mode** argument allows the user to download the environment where an error occurred. If an error occurs during data processing in the database, the corresponding data set is downloaded to the client and opened with the debugger command.

Note: To use this method of debugging, the **nzserver** package must be installed in the R client.

While the package name must be available in the workspace, the installed package might be empty.

To prepare an empty **nzserver** stub, use the following command:

```
tmp<-NULL;package.skeleton('nzrserver','tmp')
```

The package can then be installed. The specific installation procedure depends on the operating system. For more information, see Writing R Extensions.

```
FUN2debug = function(x) if(min(x[,1]) < 4.5) cov(0) else min(x[,1])
nzTApply(nzIris, "CLASS", FUN2debug, debugger.mode=T)
```

```
# Found 1 error
#
#                                          values      type      group
#1                                           101   integer  virginica
#2 supply both 'x' and 'y' or a matrix-like 'x' try-error     setosa
#3                                            51   integer versicolor
#
#
# Recalling environment for group setosa
#
# Take environment no. 11 and check for the args variable
#
# Message:  supply both 'x' and 'y' or a matrix-like 'x'Available environments
#had calls:
#1: dispatcher()
#2: try(handleConnection(), silent = TRUE)
#3: tryCatch(expr, error = function(e) {
#4: tryCatchList(expr, classes, parentenv, handlers)
#5: tryCatchOne(expr, names, parentenv, handlers[[1]])
#6: doTryCatch(return(expr), name, parentenv, handler)
#7: handleConnection()
#8: runWrapper()
#9: nzrsrv.tapply(userData$fun, userData$args, userData$cols)
#10: process.cell(data)
#11: do.call(fun, c(list(x = data), args))
#12: function (..., FUN2s)
#13: tryCatch(FUN2s(...), error = function(e) {
#14: tryCatchList(expr, classes, parentenv, handlers)
#15: tryCatchOne(expr, names, parentenv, handlers[[1]])
#
#Enter an environment number, or 0 to exit  Selection: 11
#Browsing in the environment with call:
#   do.call(fun, c(list(x = data), args))
#Called from: debugger.look(ind)
#Browse[1]> args
#$x
#   id sepallength sepalwidth petallength petalwidth  class
#1   4         4.6        3.1         1.5        0.2 setosa
#2  31         4.8        3.1         1.6        0.2 setosa
#3  27         5.0        3.4         1.6        0.4 setosa
#4  23         4.6        3.6         1.0        0.2 setosa
```

C H A P T E R 4

Netezza Analytics Library for R

The Netezza Analytics Library for R package is a standard CRAN-style R package. In this section, basic functions for using in-database analytics directly from the R client are reviewed.

System Prerequisites and Installation

To use the Netezza Analytics Library for R package, R must be available on the client machine and Netezza Analytics must be installed and registered on the NPS system. For more information about how to install Netezza Analytics, see the corresponding section in the *IBM Netezza Analytics Administrator's Guide*.

Introduction

The R environment offers a large number of functions for data analysis, model validation, model visualization, and data preprocessing. However, in the base R installation outside of the Netezza environment, the following bottlenecks might occur when processing large data sets:

- **Memory limit**—In the base 32-bit R installation, users are limited to 4 GB or 2GB of RAM, depending on the operating system.
- **Processing speed**—In the base installation, only one thread is allowed. As a result, even if R is working on a multicore machine, the time-consuming steps are not done at full speed. Although libraries that enable parallel computation exist, they require sophisticated configuration.
- **Method of accessing large data sets**—In databases that are larger than several terabytes, the data sets are stored in a set of virtualized disks. Importing the data set to R in chunks and processing it step-by-step is not optimal. In most cases, it is much faster to run the analytic routines closer to the data instead of bringing the data to the R client for analysis.

This section describes how to use Netezza Analytics to do analytics for large data sets in R.

- Netezza Analytics contains several built-in analytic routines for statistical and data mining algorithms. Because these algorithms are registered and executable from the database, they are fast and work close to the data. The results from these procedures, such as fitted models, model predictors, and so on, are then downloaded from the database to R. Then, the outcomes are transformed into R classes and made accessible in R for subsequent steps, such as processing or visualization.

- Netezza Analytics contains routines for computing data aggregates in the database. These aggregates, which are usually much smaller than the data they stem from, can be computed in the database and then downloaded to R, where the rest of the computation is done. For many algorithms, this method of precomputing certain sufficient statistics in the database, then transferring them to R, and performing the remaining computation in R, greatly increases efficiency.

Documentation and Help

The following sections describe the supported wrappers for different Netezza Analytics functions, their main arguments and parameters. The sections also provide simple examples of how to use these functions. For a full list of options and parameters for each of these functions, use the R build-in help system. For example, to invoke the help page for nzGlm(); you can use the following command:

```
> help(nzGlm)
```

As most of the functions that are provided by the NZA package, are wrappers around Netezza Analytics functions, you can also take a look at the *IBM Netezza Analytics In-Database Analytics Developer's Guide* and the *IBM Netezza Analytics In-Database Analytics Reference Guide* for details. For example, if you are, interested in details about the *link* parameter for the nzGlm() function, the section on GLM in the *IBM Netezza Analytics In-Database Analytics Developer's Guide* gives you a detailed description of individual link functions that are available.

Wrappers for Built-in Analytics

Decision Trees

The Netezza Analytics package for the NPS provides a set of sophisticated algorithms. A simple example of a decision tree algorithm is presented below. In the subsequent sections, examples of other statistical and data mining algorithms are shown.

Assume that the *adult* data set[3] is stored in the database **MM** in the table **ADULT**. In the code snippet below, a connection to the database is created, then a pointer to the table with data is made. Next, the decision tree model is fitted and finally, the fitted model is downloaded to R. All steps are transparent to the R user.

The first step is to make a connection to the database.

3 Refer to http://archive.ics.uci.edu/ml/datasets/Adult for more information.

```
# loads necessary packages
library(nza)

# connects to the mm database
nzConnect("user", "password", "TT4-R040", "mm")

# creates a pointer to the data set stored in the table adult
nzadult = nz.data.frame("adult")
```

The *nzadult* R object is a pointer to the table **ADULT** on the NPS system. It is an object of the class *nz.data.frame* with overloaded functions like print(), [,] and others. As described in The Netezza R Library , it corresponds to a standard data.frame object, but is stored remotely in the database.

The R functions that are wrappers on NPS analytic routines can take this pointer as an argument. This example uses the function nzDecTree() from the Netezza Analytics Library for R package that builds the classification tree. The nzDecTree() function is an R wrapper that prepares an SQL query that remotely calls the Netezza Analytics **DECTREE** stored procedure. The procedure runs remotely and the final model is returned to R. Next, the model is converted to an object of the tree class that has a similar structure to the objects that are created with the R tree() function.

Below is an example of building a decision tree for predicting the variable *income*, based on the variables *age*, *sex* and *hours per week*. The **ID** column from data set **ADULT** is specified in the *id* parameter.

```
# build a tree using built-in analytics
adultTree = nzDecTree(INCOME~AGE+SEX+HOURS_PER_WEEK, nzadult, id="ID")
```

The function output is stored in the database while the function nzDecTree() transforms it by default to an R object of the class *tree*, which is specified by the package of the same name. Therefore, overloaded functions such as print(), plot() or predict() work with this object. It is possible to print or visualize the tree in R, even if it was fitted for a large data set with millions of rows.

```
# plot and print the tree
plot(adultTree)

print(adultTree)
#node), split, n, deviance, yval, (yprob)
#      * denotes terminal node
#
# 1) root 32561 0 small ( 0.24081 0.75919 )
#   2) AGE < 27 8031 0 small ( 0.03213 0.96787 ) *
#   3) AGE > 27 24530 0 small ( 0.30913 0.69087 )
#     6) SEX=Female 7366 0 small ( 0.15096 0.84904 ) *
#     7) SEX < >Female 17164 0 small ( 0.37701 0.62299 )
#      14) HOURS_PER_WEEK < 41 10334 0 small ( 0.29988 0.70012 ) *
#      15) HOURS_PER_WEEK > 41 6830 0 small ( 0.49370 0.50630 )
#        30) AGE < 35 1925 0 small ( 0.34649 0.65351 ) *
#        31) AGE > 35 4905 0 large ( 0.55148 0.44852 ) *
```

The fitted model can be applied to another data set. If the data set is stored in a database table, massive data transfer can be avoided by using the overloaded function predict() to do classification inside the NPS system.

```
# use the previously created tree for prediction of the same data
adultPred = predict(adultTree, nzadult)

# download the prediction results into a data.frame
```

```
head(as.data.frame(adultPred))
#  ID CLASS
#1  1 small
#2  2 small
#3  3 small
#4  4 small
#5  5 small
#6  6 small
```

As an alternative to the returned object of class *tree*, nzDecTree might also return an object of class *rpart*, as specified in the package of the same name. This object also supports overloaded functions of *rpart* such as print(), plot() or predict().

```
adultRpart = nzDecTree(INCOME~AGE+SEX+HOURS_PER_WEEK, nzadult, id="ID",
       format="rpart")

plot(adultRpart)

print(adultRpart)
#n= 32561
#
#node), split, n, loss, yval, (yprob)
#      * denotes terminal node
#
# 1) root 32561 NA small (0.240809.... 0.759190....)
#   2) AGE< 27 8031 NA small (0.032125.... 0.967874....) *
#   3) AGE>=27 24530 NA small (0.309131.... 0.690868....)
#     6) SEX=Female 7366 NA small (0.150963.... 0.849036....) *
#     7) SEX=<other> 17164 NA small (0.377010.... 0.622989....)
#      14) HOURS_PER_WEEK< 41 10334 NA small (0.299883.... 0.700116....) *
#      15) HOURS_PER_WEEK>=41 6830 NA small (0.493704.... 0.506295....)
#        30) AGE< 35 1925 NA small (0.346493.... 0.653506....) *
#        31) AGE>=35 4905 NA large (0.551478.... 0.448521....) *
```

Regression Trees

The example below demonstrates regression trees. The basic idea is the same as for decision trees. In this code snippet the **WEATHERR** data set is used. The variable of interest is *grade*, a continuous variable. The mean values of the variable *grade* are stored in each of the corresponding leaves of the regression tree.

First, a connection to the database and a pointer to the NPS table are created.

```
# loads necessary packages
library(nza)

# connect to the nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to the WEATHERR table is created
weatherr = nz.data.frame("WEATHERR")
```

To build a regression tree remotely, the nzRegTree() function is used. It calls the Netezza Analytics stored procedure **REGTREE**. Only the final model, that is, the parameters of the fitted tree, is downloaded to R and transformed into an object of class *tree*, as specified by the package of the same name. This example builds a regression tree for predicting the variables *grade* based on all other variables within the data set. The **ID** column from data set **WEATHERR** is specified in the *id* parameter.

```
wTree = nzRegTree(GRADE~., data=weatherr, id="INSTANCE", minimprove=0.1,
minsplit=2, maxdepth=4)
```

Overloaded functions such as print() or plot() can be used to visualize the fitted tree.

```
# plot and print the tree
plot(wTree)

print(wTree)
#node), split, n, deviance, yval
#      * denotes terminal node
#
# 1) root 22 NA 2.636
#   2) OUTLOOK=sun 8 NA 3.875
#     4) TEMPERATURE < 72 6 NA 4.500
#       8) TEMPERATURE < 52 3 NA 4.000 *
#       9) TEMPERATURE > 52 3 NA 5.000 *
#     5) TEMPERATURE > 72 2 NA 2.000 *
#   3) OUTLOOK < >sun 14 NA 1.929
#     6) OUTLOOK=cloudy 6 NA 2.833
#      12) TEMPERATURE < 12 1 NA 2.000 *
#      13) TEMPERATURE > 12 5 NA 3.000 *
#     7) OUTLOOK < >cloudy 8 NA 1.250
#      14) HUMIDITY=low 2 NA 2.000 *
#      15) HUMIDITY < >low 6 NA 1.000 *
```

This pre-built model can be applied to another data set. If the data set is stored in the database, the overloaded predict() function can be used to apply the regression tree inside the NPS system. The predict() function calls the **PREDICT_REGTREE** stored procedure.

```
# make prediction using the model wTree on table weatherr
wPred = predict(wTree, weatherr, id="INSTANCE")
# wPred is a nz.data.frame and can easily be examined
head(wPred)
#  ID CLASS
#1  2     2
#2  6     1
#3 10     2
#4 14     1
#5 18     1
#6 22     1
```

As an alternative to the returned object of class *tree*, nzRegTree might also return an object of class *rpart*, as specified in the package of the same name. The functionality is equal to the *rpart* features in nzDecTree():

```
adultRpart = nzDecTree(INCOME~AGE+SEX+HOURS_PER_WEEK, nzadult, id="ID",
       format="rpart")

plot(adultRpart)

print(adultRpart)
#n= 32561
#
#node), split, n, loss, yval, (yprob)
#      * denotes terminal node
#
# 1) root 32561 NA small (0.240809.... 0.759190....)
#   2) AGE< 27 8031 NA small (0.032125.... 0.967874....) *
#   3) AGE>=27 24530 NA small (0.309131.... 0.690868....)
#     6) SEX=Female 7366 NA small (0.150963.... 0.849036....) *
#     7) SEX=<other> 17164 NA small (0.377010.... 0.622989....)
#      14) HOURS_PER_WEEK< 41 10334 NA small (0.299883.... 0.700116....) *
#      15) HOURS_PER_WEEK>=41 6830 NA small (0.493704.... 0.506295....)
#        30) AGE< 35 1925 NA small (0.346493.... 0.653506....) *
#        31) AGE>=35 4905 NA large (0.551478.... 0.448521....) *
```

One-way and Two-way ANOVA

Classic statistical functions such as ANOVA are available in Netezza Analytics. This section covers the one-way and two-way ANOVA.

In this sample, we will use the **WEATHERR** data set. We exclude the first instance so that there is the same number of observations for each parameter value in the variable **HUMIDITY**.

First, establish a connection to the database and create a pointer to the NPS **WEATHERR** table. Also, exclude the first row of the table.

```
# loads necessary packages
library(nza)

# connect to the nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to weatherr table is created
weatherr = nz.data.frame("weatherr")
# select all rows form the table whose INSTANCE is bigger than one
weatherr = weatherr[weatherr$INSTANCE>1, ]
```

The **HUMIDITY** and **OUTLOOK** columns in this table correspond to grouping variables. The column **TEMPERATURE** is a continuous variable. ANOVA is used to verify whether the mean value of the **TEMPERATURE** variable varies for different subgroups.

The nzAnova() function remotely executes the ANOVA algorithm. The function takes a *formula* object as the first argument.

If there is one variable on the right side of the formulas, the function calls the **ANOVA_CRD_TEST** algorithm of Netezza Analytics. If there are two variables on the right hand side, the function calls the **ANOVA_RBD_TEST** algorithm of Netezza Analytics. The function then transforms their results into an object of class *summary.aov*.

This example demonstrates the one-way ANOVA algorithm, where **HUMIDITY** is chosen as treatment variable.

```
nzAnova(TEMPERATURE~HUMIDITY, weatherr)
#           Df Sum Sq Mean Sq F value Pr(>F)
#HUMIDITY   2    275   137.3   0.265   0.77
#Residuals 18   9309   517.1
```

This example demonstrates the two-way ANOVA algorithm, where **HUMIDITY** is chosen as treatment variable and **OUTLOOK** is chosen as block variable.

```
nzAnova(TEMPERATURE~HUMIDITY+OUTLOOK, weatherr)
HUMIDITY   2    275   137.3   0.337 0.7189
OUTLOOK    3   3206  1068.7   2.627 0.0884 .
Residuals 15   6102   406.8
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

K-Means

Clustering methods are supported in the Netezza Analytics package.

The following *k*-means clustering example uses the **IRIS** data set. The data set contains data regarding three different *Iris* species, that is **setosa**, **versicolor,** and **virginica**. Thus, there are three clusters expected in the **IRIS** data set, each related to a different species.

First, establish a connection to the database and create a pointer to the NPS **IRIS** table.

```
# loads necessary packages
library(nza)

# connect to nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to IRIS table is created
nziris = nz.data.frame("iris")
```

The nzKMeans() function does clustering remotely by calling the stored procedure **KMEANS**. Cluster statistics, sizes, and mean values are the only data that is downloaded to R and transformed into a object of class *kmeans*, as described in the *stats* package.

The *k*-means algorithm splits the data into *k* clusters. The following example refers to *k=3* and *k=10*.

```
# K-Means for 2 clusters and euclidean distance
t3 = nzKMeans(nziris, k=3, id="ID")

# K-Means for 10 clusters and L1 distance
# also download item-cluster assignments for every item after computation
t10 = nzKMeans(nziris, k=10, distance="manhattan", id="id", getLabels=T)
```

To show clusters and their mean values, you can use the overloaded print() function.

```
print(t3)
#KMeans clustering with 3 clusters of sizes 50, 51, 49
#
#Cluster means:
#       CLASS PETALLENGTH PETALWIDTH SEPALLENGTH SEPALWIDTH
#1     setosa    1.464000   0.244000    5.006000   3.418000
#2 versicolor    4.264706   1.333333    5.915686   2.764706
#3  virginica    5.573469   2.032653    6.622449   2.983673
#
#Clustering vector:
#SELECT "ID","CLUSTER_ID","DISTANCE" FROM ADMIN.IRIS_MODEL59040
#
#Within cluster sum of squares by cluster:
#[1] 15.24040 32.93373 39.15551
#
#Available components:
#[1] "cluster"  "centers"  "withinss" "size"     "distance" "model"

# get used distance metric
t10$distance
#[1] "manhattan"
```

The returned object is implemented as a list and therefore, common list operations might be used to retrieve single components of the result.

```
names(t3)
#[1] "cluster"  "centers"  "withinss" "size"     "distance" "model"

t3$centers
#       CLASS PETALLENGTH PETALWIDTH SEPALLENGTH SEPALWIDTH
#1     setosa    1.464000   0.244000    5.006000   3.418000
#2 versicolor    4.264706   1.333333    5.915686   2.764706
#3  virginica    5.573469   2.032653    6.622449   2.983673

# nzKMeans' parameter getLabels was unset when computing t3
# hence t3$cluster is of class nz.data.frame
head(t3$cluster)
```

```
#  ID CLUSTER_ID  DISTANCE
#1  4          1 0.5188372
#2  8          1 0.0599333
#3 12          1 0.2513802
#4 16          1 1.2130919
#5 20          1 0.3989887
#6 24          1 0.3794628

# nzKMeans' parameter getLabels was TRUE when computing t10
# hence t3$cluster is of class data.frame
head(t10$cluster)
#1 2 3 4 5 6
#1 4 6 6 1 5
```

The fitted model can be applied to another data set. If the data set is stored in the database, you can use the overloaded predict() function to do classification on the NPS system. The function returns the distance to the closest cluster and its identifier.

```
res = predict(t3, nziris, id="ID")

head(res)
#  ID CLUSTER_ID  DISTANCE
#1  2          1 0.4381689
#2  6          1 0.6838070
#3 10          1 0.3665951
#4 14          1 0.9090611
#5 18          1 0.1509702
#6 22          1 0.3376270
```

The fitted model can be plotted by calling the overloaded plot() function. This function will either download a sample of points by their **DISTANCE,** or download all points to produce a matrix of scatterplots like the built-in function pairs() of R does.

```
# plot KMeans' result as matrix of scatterplots
plot(t10)
```

The function nzKMeans() also supports a *raw* output format. This format only downloads all tables that are created by the **KMEANS** algorithm of Netezza Analytics and stores them in a list of data.frames.

```
r = nzKMeans(nziris, k=3, id="ID", format="raw")
names(r)
#[1] "clusters"           "columns"            "column.statistics"
#[4] "model"              "centroids"          "modelname"
```

TwoStep

TwoStep is an alternative clustering algorithm to the *k*-means algorithm. Its main advantages are that it can determine the number of clusters automatically and that it can handle a mixture of categorical and continuous fields in a statistically sound way.

In the following TwoStep clustering example, the **IRIS** data set is used.

First, a connection to the database is established and a pointer to an NPS table is created.

```
# loads necessary packages
library(nza)

# connect to nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to IRIS table is created
```

```
nziris = nz.data.frame("iris")
```

Now the nzTwoStep function is used to cluster the data.

```
# TwoStep model for IRIS
t2 = nzTwoStep(nziris, id="id")
```

To show clusters and their mean values, you can use the overloaded print() function.

```
print(t2)
#TwoStep clustering with 3 clusters of sizes 50, 50, 50
#
#Cluster means:
#       CLASS PETALLENGTH PETALWIDTH SEPALLENGTH SEPALWIDTH
#1     setosa       1.464      0.244       5.006      3.418
#2  virginica       5.552      2.026       6.588      2.974
#3 versicolor       4.260      1.326       5.936      2.770
#
#Clustering vector:
#SELECT " ID "," CLUSTER_ID "," DISTANCE " FROM ADMIN.IRIS_MODEL74356
#
#Within cluster sum of squares by cluster:
#[1] 0.3713140 0.5413910 0.4277027
#
#Available components:
#[1] "cluster"  "centers"  "withinss" "size"     "distance" "model"
```

The result object provides information about cluster centers, cluster sizes, and sums of squares within each cluster. The distance attribute stores the information about which distance metric is used.

As in nzKMeans(), the output of nzTwoStep() is stored in a list. Therefore, all common list operations are available.

```
names(t2)
#[1] "cluster"  "centers"  "withinss" "size"     "distance" "model"

t2$size
#[1] 50 50 50

t2$distance
#[1] "loglikelihood"
```

The fitted model can be applied to another data set. If the data set is stored in the database, you can use the overloaded predict() function to do classification on the NPS system. The function returns the distance to the closest cluster and its identifier.

```
res = predict(t2, data, "ID")

head(res)
#  ID CLUSTER_ID   DISTANCE
#1  2          1 0.34278577
#2  6          1 0.54303152
#3 10          1 0.24827305
#4 14          1 0.67509943
#5 18          1 0.09122145
#6 22          1 0.21751088
```

The fitted model can be plotted by calling the overloaded plot() function. This function will either download a sample of points by their **DISTANCE**, or download all points to produce a matrix of scatterplots like the built-in function pairs() of R does.

```
# plot TwoStep's result as matrix of scatterplots
plot(t2)
```

The function nzTwoStep() also supports a *raw* output format. This format only downloads all tables that are created by the **KMEANS** algorithm of Netezza Analytics and store them in a list of data.frames.

```
r = nzKMeans(nziris, k=3, id="ID", format="raw")
names(r)
#[1] "clusters"            "columns"             "column.statistics"
#[4] "discrete.statistics" "model"               "numeric.statistics"
#[7] "modelname"           "cluster"
```

Naive Bayes

The following example shows a classification by using the Naive Bayes implementation. The current implementation supports only discrete values.

In the computations, the **ADULT** data set is used. Each row in the data set describes the properties of a particular person. The goal is to find a rule for predicting the income of this person, that is, whether the income is large or small. Only several discrete columns from ADULT, that is, **WORKCLASS**, **EDUCATION**, **SEX** and **INCOME** are used. As required by the algorithm, also the **ID** column is taken into account.

First, a connection to the database is established and a pointer to an NPS table is created.

```
# connect to nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to ADULT table is created
adult = nz.data.frame("adult")

# pick the needed columns from adult
adult = adult[ ,c("ID", "WORKCLASS", "EDUCATION", "SEX", "INCOME")]
```

To fit the Naive Bayes model, you can use the nzNaiveBayes() function. This function calls the **NAIVEBAYES** stored procedure. The only data that is downloaded to R includes the final model with priors and marginal distributions. The result data is then transformed into a naiveBayes object. This object has a similar structure to the object that is computed by using the package *e1071*.

Create a model that fits for the **class** variable, where all remaining variables are used as explanatory variables for the **INCOME** variable.

```
t = nzNaiveBayes(INCOME ~ ., adult, id="ID")
```

The print() function is overloaded. It presents *a priori* probabilities for all the classes and pairwise contingency tables that describe marginal distributions.

```
print(t)
#Naive Bayes Classifier for Discrete Predictors
#
#Call:
#nzNaiveBayes(INCOME ~ ., adult, id = "ID")
#
#A-priori probabilities:
#    small     large
#0.7591904 0.2408096
#
#Conditional probabilities:
#       WORKCLASS
#Y        Private Self-emp-not-inc State-gov Local-gov Federal-gov Self-emp-inc
```

```
#  small   19378             1817       945      1476         589          494
#  large    5154              724       353       617         371          622
#(...)
#
#       EDUCATION
#(...)
#
#       SEX
#(...)
```

The fitted model can be applied to another data set. The first 100 rows of the **ADULT** data set are used again as a test data set. The predictions made with the NaiveBayes classifier are compared with the real label values from the data set.

The following example shows a way of how to test the quality of the Naive Bayes model.

```
# pointer to the subset of adult table
adult100 = adult[adult$ID<=100, ]

# predict INCOME for adult100 data on Naive Bayes model t
# this computation will be calculated on NPS
nbPred = predict(t, adult100, id="ID")

# download prediction results and sort by ID column
pred = as.data.frame(nbPred)
pred = pred[order(pred[ ,1]), ]

# download original data and sort by ID column
t2 = as.data.frame(adult100[ ,c("ID", "INCOME")])
t2 = t2[order(t2[ ,1]), ]

# inspect both of the downloaded data
head(pred)
#  ID CLASS
#1  1 large
#2  2 large
#3  3 small
#4  4 small
#5  5 small
#6  6 small

head(t2)
#1   1  small
#75  2  small
#26  3  small
#51  4  small
#2   5  small
#76  6  small

# contingency table for actual and predicted INCOME parameter values
table(ACTUAL=t2[ ,2], PREDICTED=pred[ ,2])
#       PREDICTED
#ACTUAL  large small
#  large    11    14
#  small     5    70
```

Generalized Linear Models

The nzGlm() function wraps the **GLM** stored procedure for Netezza Analytics. The following example shows how to use this function. To predict the value for **SEPALLENGTH** for given **PETALLENGTH** and **PETALWIDTH** values, the **IRIS** data set is used.

First, a connection to the database is established and a pointer to an NPS table is created.

```
# loads necessary packages
library(nza)

# connect to nza database
nzConnect("user", "password", "TT4-R040", "nza")

# a pointer to IRIS table is created
nziris = nz.data.frame("iris")
```

Create a model that fits for the **SEPALLENGTH** variable, where **PETALLENGTH** and **PETALWIDTH** are used as explanatory variables only.

```
t = nzGlm(SEPALLENGTH ~ PETALLENGTH+PETALWIDTH, data=nziris, id="ID")
```

The functions print() and summary() are overloaded. They present a summary of the model.

```
print(t)
#Model Name
#IRIS_MODEL47515
#
#Call:nzGlm(form = SEPALLENGTH ~ PETALLENGTH + PETALWIDTH, data = iris,
#    id = "ID")
#
#
#Coefficients:
#   INTERCEPT  PETALLENGTH   PETALWIDTH
#-60018082238  38811202901  23865561738
#
#
#Residuals Summary:
#Pearson:             RSS: 0  df: 147        p-value: 0
#Deviance:            RSS: 0  df: 147        p-value: 0

summary(t)
#Call:nzGlm(form = SEPALLENGTH ~ PETALLENGTH + PETALWIDTH, data = iris,
#    id = "ID")
#
#
#GLM coefficients for model: "IRIS_MODEL47515"
#| Parameter   | Beta         | Std Error    | Test          | p-value      |
#| INTERCEPT   | -60018082237.637    | 6992.319359  | -8583429.783357     | 0
#      |
#| PETALLENGTH | 38811202901.378     | 5382.805712  | 7210218.0497 | 0     |
#| PETALWIDTH  | 23865561738.095     | 9101.838248  | 2622059.532135      | 0
#      |
#
# Residuals Summary:
#| Residual Type      | RSS          | df | p-value        |
#| Pearson     |0|147|0|
#| Deviance    |0|147|0|
```

The model can be applied to another data set. If the data set is stored in the database, you can use the overloaded predict() function to do classification on the NPS system.

```
pred = predict(t, nziris, "ID")

head(pred)
#  ID PRED
#1  2    0
#2  6    1
#3 10    1
#4 14    0
#5 18    1
#6 22    1
```

As the predict() function, the residuals() function and the fitted() function aim to imitate the behavior of *glm* objects.

```
res = residuals(t)
head(res)
#  1   2   3   4   5   6
#5.1 4.9 4.7 3.6 5.0 4.4

fit = fitted(t)
head(fit)
#1 2 3 4 5 6
#0 0 0 1 0 1
```

Time Series

The nzTs() function wraps the time series prediction functionality in Netezza Analytics. It supports the creation of one or several time series models on a table by using the following model types:

- Exponential Smoothing
- ARIMA
- Seasonal Trend Decomposition
- Spectral analysis

This function can create a single model by using the whole data that is available in the input table, or it can create several models by using the *by* grouping parameter.

The time series functionality in Netezza Analytics differs from other algorithms by not providing a separate predict function. Instead, the predictions are made during model fitting and stored as part of the time series model that is returned to the R client.

The following example shows how to use the time series function. The example uses the **CURVES** data set. This data set contains **X** and **Y** values that are stored in columns of the same name from different mathematical functions, such as *linear* or *quadratic,* as specified in a third column **CURVE**.

First, a connection to the database is established and a pointer to an NPS table is created.

```
# connect to nza database and create pointer to NPS table
nzConnect("user", "password", "TT4-R040", "nza")
curves = nz.data.frame('CURVES')
```

Then, the time series prediction functionality in Netezza Analytics is called. The function summary() prints statistics of each of the time series as described in the *IBM Netezza Analytics In-Database Analytics Developer's Guide*.

```
# loads necessary packages
library(nza)

# summarize every timeseries' results
tsModel = nzTs(data=curves, algorithm='esmoothing', time='x',
       target='y', by='curve')
summary(tsModel)
#Call:
#nzTs(data = curves, algorithm = "esmoothing", time = "x", target = "y",
#    by = "curve")
#
#Model:  CURVES_MODEL35471
#
#Algorithm:  Exponential smoothing
```

```
#Output:
#         TSID      TRENDTYPE SMOOTHEDTREND SMOOTHEDVALUE     ALPHA     GAMMA
#1  hyperbolic multiplicative  9.800000e-01    20.0000000 1.0000000 0.9999999
#2      linear       additive  2.000000e+00   100.0000000 0.8001000 0.0000000
#3      modulo       additive -6.505213e-19     0.2928968 0.6001000 0.0000000
#4       sinus dampedadditive -1.425499e-09     2.0000000 1.0000000 0.9999995
#5 logarithmic multiplicative  9.717819e-01     1.5051500 0.9999999 0.6805075
#6   quadratic       additive  9.800000e+01  2449.9999997 0.9999998 1.0000000
#       DELTA       PHI
#1 0.00000000 1.0000000
#2 0.00000000 1.0000000
#3 0.00000000 1.0000000
#4 0.04212114 0.5816227
#5 0.00000000 1.0000000
#6 0.00000000 1.0000000
```

Because the result of nzTs() is by default a list of nz.data.frames, you can use the common methods for lists or nz.data.frames.

```
names(tsModel)
#[1] "model"              "call"               "algorithm"
#[4] "series"             "periods"            "seasonalitydetails"
#[7] "interpolated"       "forecast"           "expodetails"

as.data.frame(tsModel$periods)
#    TSID UNIT    PERIOD  FREQUENCY SEASONS     WEIGHT HARMONICWEIGHT
#1 linear <NA> 13.337195 0.07497829      13 0.76237136      0.0000000
#2 linear <NA>  4.616534 0.21661273       5 0.14477519      0.0000000
#3 linear <NA>  2.753268 0.36320479       3 0.05911359      0.0000000
#4 modulo <NA> 10.013072 0.09986945      10 0.92266644      0.5929672
#5 modulo <NA>  1.992997 0.50175701       2 0.05624611      0.0000000
#6  sinus <NA> 11.951616 0.08367069      12 0.99852302      0.0000000
```

The subset *linear* from **CURVES** is used to do a basic time series prediction.

```
# select linear curve from curves data set
curves2 = curves[curves$CURVE == "linear"]

# build linear model using ARIMA algorithm
tsModel = nzTs(data=curves2, algorithm='ARIMA', time='x', target='y')

# show last known timeseries entries and predicted values
tail(curves)
#45 linear 45  90
#46 linear 46  92
#47 linear 47  94
#48 linear 48  96
#49 linear 49  98
#50 linear 50 100

head(tsModel$forecast)
#  TSID TIME FORECAST STANDARDERROR
#1    1   51      102             0
#2    1   52      104             0
#3    1   53      106             0
#4    1   54      108             0
#5    1   55      110             0
#6    1   56      112             0
```

Association Rules

The nzArule() function is a wrapper for the Netezza Analytics **ARULE** stored procedure. The function assumes that the transactions of interest are stored in the input table in the (TID, ITEMID) format.

nzArule() supports two different return format types. Use *arule* for output that is compatible to R package *arules*, or *raw* for downloading the raw results from the database. To visualize the assiciation rules, use the *arule* compatible return format and manually install the package *arulesViz.* This package is not a prerequisite of the package *nza* but it is compatible with the results of the nzArule() function.

First, a connection to the database is established and a pointer to an NPS table is created.

The following example uses the **RETAIL** data set.

```
# loads necessary packages
library(nza)

# connect to nza database and create pointer to NPS table
nzConnect("user", "password", "TT4-R040", "nza")

retail = nz.data.frame("retail")
```

Then, you can try to find association rules in the data set. By default, nzArule() returns an object of class *rules*. For further information about these objects, see the documentation about *arules*.

In the following example, the overloaded functions print(), summary(), inspect(), and sort() are used. These functions are described in the documentation about *arules*. In this data set, all items are encoded as numbers. You can run this example also with strings as item identifiers.

```
rules = nzArule(retail, "TID", "ITEM")

print(rules)
#set of 14 rules

summary(rules)
#set of 14 rules
#
#rule length distribution (lhs + rhs):sizes
#2 3
#8 6
#
#   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  2.000   2.000   2.000   2.429   3.000   3.000
#
#summary of quality measures:
#    support          confidence          lift
# Min.   :0.06127   Min.   :0.5094   Min.   :0.9698
# 1st Qu.:0.07280   1st Qu.:0.5788   1st Qu.:1.1579
# Median :0.09062   Median :0.6421   Median :1.2187
# Mean   :0.12253   Mean   :0.6447   Mean   :1.2246
# 3rd Qu.:0.11358   3rd Qu.:0.6868   3rd Qu.:1.3344
# Max.   :0.33055   Max.   :0.8168   Max.   :1.4210
#
#mining info:
#                                      data ntransactions support confidence
# SELECT " TID "," ITEM " FROM ADMIN.RETAIL         908576       5        0.5
#             model
# RETAIL_MODEL35732

inspect(sort(rules)[1:3])
#  lhs     rhs    support confidence     lift
#1 {48} => {39} 0.3305506  0.6916340 1.203273
#2 {39} => {48} 0.3305506  0.5750765 1.203273
#3 {41} => {39} 0.1294662  0.7637337 1.328708
```

The *raw* format downloads all tables that are created by the ARULE stored procedure of Netezza Analytics and stores them in a list of data.frames.

```
rules2 = nzArule(retail, "TID", "ITEM", format="raw")

names(rules2)
#[1] "group"   "item"    "itemset" "rule"    "model"

head(rules2$rule, n=3)
#  GID RHS_SID LHS_SID RHS_SIZE LHS_SIZE   SUPPORT CONFIDENCE      LIFT
#1   1       4       2        1        1 0.3305506  0.6916340 1.2032726
#2   1       4       3        1        1 0.1294662  0.7637337 1.3287082
#3   1       4       1        1        1 0.0959030  0.5574603 0.9698434
#  CONVICTION  AFFINITY     LEVERAGE
#1   1.378900 0.4577182  0.055840945
#2   1.799689 0.2105671  0.032028558
#3   0.960831 0.1473330 -0.002982039
```

Visualization is available for results from nzArule in *arules* format only. Loading the package *arulesViz* is required. For more information about the overloaded plot() function, see the help for *arulesViz.*

```
library(arulesViz)
plot(rules)
plot(rules, method="grouped")
```

Sufficient Statistics and Support for Two-step Processing

There is a large number of statistical algorithms that can be split into two steps. In the first step, data aggregates, which are called sufficient statistics, are calculated. The benefit of this calculation is that the size of the data aggregates is usually much smaller than the original data set. In the second step, the statistical routine uses the data of the aggregates rather than the original data set. With Netezza Analytics, the second step can be done by using R. This method opens up the calculations to the large variety of available R functions.

Internal stored procedures compute data aggregates in a parallel way. Data that is stored in parallel nodes, or multi-nodes, or both nodes of the database has the following advantages:

- There is no practical size limit for the accessible data set, as it uses the data storage capacity of the Netezza appliance
- Data aggregates are computed in a parallel way, which significantly reduces computation time

In some cases, the reduction in processing time might be linear with the number of processors, and processing speed might be linear with the number of cores.

The following algorithms are implemented in the Netezza Analytics Library for R package by using the sufficient statistic approach:

- Correspondence analysis
- Canonical analysis
- Simple regression
- Principal component analysis
- ANOVA
- ANCOVA

- Other linear models with interactions
- Ridge regression
- PCR
- Other algorithms

The following subsections explain how the nzTable() function works. A real-data example is also included. Analyses are made for categorical variables by using the ctable procedure.

The nzTable() Function

The nzTable() function leverages the functionality of the ctable() procedure in R. The operation of the nzTable() function can be described by the following algorithm:

1. In the first step, the *nz.data.frame* object and the model formula are parsed. A list of variables of interest is constructed. In the example, these variables are **EDUCATION** and **OCCUPATION** in the **ADULT** data table .
2. The model description for the given variables is built. This model describes the variables and the table to be analyzed, and where the outcomes should be stored.
3. A contingency table for the given model is derived in the NPS system. Therefore, the number of rows is not limited, and the entire data set does not have to be stored in R memory.
4. The calculated contingency table is read from the database to R through an ODBC connection.

Example Using ADULT Data

This section provides an example of the nzTable() function and its usage. Real-world examples are described in a later subsection.

To run this example, load the Netezza Analytics Library for R package by running the following command:

```
library(nza)
```

To use the data set that is stored in the database, a connection to this database must be established. To establish the connection, you can use the nzConnect() function.

```
nzConnect("user", "password", "TT4-R040", "mm")
```

It is assumed that the **ADULT** data set is available in the database. A pointer to the **adult** table is created by using the function nz.data.frame().

In this code snippet, the column names and number of rows in this table are printed out.

```
(adult = nz.data.frame("adult"))
#SELECT
#ID,AGE,WORKCLASS,FNLWGT,EDUCATION,EDUCATION_NUM,MARITAL_STATUS,OCCUPATION,RELA#T
IONSHIP,RACE,SEX,CAPITAL_GAIN,CAPITAL_LOSS,HOURS_PER_WEEK,INCOME FROM ADULT
nzQuery("select count(*) from adult")
#  COUNT
```

```
#1 32561
```

To build the contingency table, you can use the nzTable() function. The first parameter of this function is a model specification; the second parameter is a pointer to the data table in the database, that is, an object of class *nz.data.frame*.

In the following example, a contingency table for the **EDUCATION** variable and the **OCCUPATION** variable is built:

```
ntab = nzTable(~EDUCATION+OCCUPATION, adult)
```

The function nzTable() returns an object of class *table*.

Further Statistical Examination

The following examples show functions that use nzTable() with two-step processing.

Chisqare Test and Cochran-Mantel-Haenszel Test

The nzChisq.test() function and the nzMantelhaen.test() functions test the independence for two or three variables.

```
#chisq test
nzChisq.test(~EDUCATION+OCCUPATION,adult)
#
#        Pearson's Chi-squared test
#
#data:  out$mat
#X-squared = 12608.77, df = 195, p-value < 2.2e-16
#
# Cochran-Mantel-Haenszel test
nzMantelHenszel.test(~EDUCATION+OCCUPATION+INCOME,adult)
#
#        Cochran-Mantel-Haenszel test
#
#data:  out$mat
#Cochran-Mantel-Haenszel M^2 = 10574.19, df = 195, p-value < 2.2e-16
```

Correspondence Analysis

The nzCa() function does correspondence analysis.

Note: For clarity, the following example displays only a portion of the output.

```
# correspondence analysis
nzC = nzCa(~EDUCATION+OCCUPATION,adult)
print(nzC)
#
# Principal inertias (eigenvalues):
#           1        2        3        4
#Value      0.272529 0.069857 0.016438 0.011202
#Percentage 70.38%   18.04%   4.24%    2.89%
#
#
# Rows:
#             12th       9th      11th   1st-4th
#Mass     0.028654  0.036086  0.013298  0.005160
#ChiDist  0.657799  0.627431  0.568731  1.290176
#Inertia  0.012399  0.014206  0.004301  0.008588
#Dim. 1   0.954252  0.857823  0.828357  1.269082
#Dim. 2  -1.430711 -1.170233 -0.920234 -2.727149
#
#
# Columns:
#        Adm-clerical Armed-Forces Craft-repair
#Mass        0.115783     0.000276     0.125887
```

```
#ChiDist     0.431425     1.034305     0.482142
#Inertia     0.021550     0.000296     0.029264
#Dim. 1      0.245669     0.173363     0.866454
#Dim. 2      1.382571     0.188128    -0.015785
#        Tech-support Transport-moving
#Mass        0.028500         0.049046
#ChiDist     0.715867         0.609291
#Inertia     0.014605         0.018208
#Dim. 1     -0.490015         1.060725
#Dim. 2      1.816526        -0.594527
#
#(...)
plot(nzC)
```

The nzDotProduct() Function

The nzDotProduct() function leverages the functionality of the dotProduct() procedure in R. The operation of the nzDotProduct() function can be described by the following algorithm:

1. In the first step, the *nz.data.frame* object and the model formula are parsed. A list of variables of interest is constructed. Factor variables and interactions are identified.
2. The model description for the given variables is built. This model describes the variables and the table to be analyzed, and where the results should be stored.
3. The dot product for the given model is calculated in the NPS system. Therefore, the number of rows is not limited, and the entire data set does not have to be stored in R memory. The output matrix is stored in the row-column-value format.
4. The calculated dot product matrix is read from the database to R through an ODBC connection. Then it is transformed to an R matrix with the nzSparse2matrix() function.

Example Using IRIS Data

To run this example, you must have installed the Netezza Analytics Library for R package.

First, the package is loaded and the connection to the database is established.

```
library(nza)
nzConnect("user", "password", "TT4-R040", "mm")
```

It is assumed that the **DPTEST** table is available in the database. This data set is similar to the **IRIS** data set, but it includes an additional **WEIGHTS** column. A pointer to the **DPTEST** table is created with the function nz.data.frame().

```
# create dptest locally and upload it
dptest = data.frame(iris, weights=rnorm(150))
dptest = as.nz.data.frame(dptest, 'dptest')
#SELECT  Sepal_Length, Sepal_Width, Petal_Length, Petpal_Width, Species,
#  weights FROM  dptest
nzQuery("select count(*) from dptest")
#  COUNT
#1   150
```

The function nzDotProduct() calculates the dot product matrix. The first parameter of this function is a model specification. The second parameter is a pointer to a table in the database, that is, the object of the class *nz.data.frame*. An additional **weight** argument can be specified, which results in the weighted dot product being calculated.

In the following example, a dot product for a set of variables is built. Note there are categorical variables in the data set, which are coded with dummy variables.

```
ntab = nzDotProduct(Sepal_Length~factor(Species) + Sepal_Width +
Sepal_Width:Sepal_Length + Sepal_Width:factor(Species), weights=weights, dptest)
```

The nzDotProduct() function returns a matrix. The PCA is performed by eigen decomposition of the matrix ntab, where ntab is a dot product of the matrix with scaled and centered variables. The linear regression is performed by solving the equation $\beta X^X = y^X$, where ntab is the dot product of $[y \cdot X]$. In the example above y is the Sepal_Length while X is a design matrix that is defined by the formula factor(Species) + Sepal_Width + Sepal_Width:Sepal_Length+ Sepal_Width:factor(Species).

Further Statistical Examination

This section shows an overview of further statistical functions that are based on nzDotProduct(). There are three functions in this example that use nzDotProduct() with two-step processing.

Linear regression

The nzLm() function fits a linear regression model and measures line test statistics, **logLikelihood**, **R2**, **AIC,** and **BIC** criteria. To create dummy variables, you must wrap the categorical variables names with the factor() function.

```
nzAdult = nz.data.frame("adult")
#linear regression
nzLm(AGE~EDUCATION_NUM+factor(INCOME)+factor(SEX), nzAdult)
#
#Coefficients:
#                Estimate  Std.Error    t.value      p.value
#EDUCATION_NUM -0.2369251 0.03033361  -7.810647 5.884182e-15
#INCOME.small  -7.6866075 0.18691249 -41.124099 0.000000e+00
#SEX.Male       1.0825294 0.16004813   6.763774 1.367129e-11
#Intercept     46.0811774 0.41372080 111.382307 0.000000e+00
#
#Residual standard error:  5706327  on  32557  degrees of freedom
#
#Log-likelihood:  -130310.6
#AIC:  260629.3
#BIC:  260631.7
```

Ridge Regression

The Ridge regression is an extension of the linear regression. An additional **lambda** parameter sets the model penalty coefficient.

```
# ridge regression
nzRidge(AGE~EDUCATION_NUM+factor(INCOME)+factor(SEX), nzAdult, lambda=10)
#
#Coefficients:
#                Estimate  Std.Error t.value p.value
#EDUCATION_NUM -0.2364995 0.03033229      NA      NA
#INCOME.large   3.8395680 2.96193391      NA      NA
#INCOME.small  -3.8395680 2.96193391      NA      NA
#SEX.Female    -0.5416080 2.96154118      NA      NA
#SEX.Male       0.5416080 2.96154118      NA      NA
#
#Residual standard error:  5706930  on  32556  degrees of freedom
#
#Log-likelihood:  -130311.5
#AIC:  260633
BIC:  260633.4
```

PCA

The nzPCA() function calculates the transformation matrix.

```
nzPCA(nzAdult[,c(2,4,6)], scale=FALSE)
#Call:
#nzPCA(nzAdult[,c(2,4,6)], scale=FALSE)
#
#Standard deviations:
#      Comp.1       Comp.2       Comp.3
#3.627444e+14 6.022818e+06 2.148609e+05
#
# 0  variables and  32561 observations.

nzPCA(nzAdult[,c(2,4,6)], scale=TRUE)
#Call:
#nzPCA(nzAdult[,c(2,4,6)], scale=TRUE)
#
#Standard deviations:
#  Comp.1   Comp.2   Comp.3
#36028.10 31600.33 30051.57
#
# 0  variables and  32561 observations.
loadings(nzPCA(nzAdult[,c(2,4,6)], scale=TRUE))
#                  Comp.1     Comp.2     Comp.3
#AGE            0.6162540 -0.3926090 0.68270730
#FNLWGT        -0.6332958  0.2682674 0.72592635
#EDUCATION_NUM  0.4681533  0.8797106 0.08331679
```

List of R Supplementary Functions

Parsing an R Formula to the NPS Input Format

Many Netezza Analytics stored procedures take arguments as a list of parameters. Because formulas are typically used to create a model description in R, the nzParseRFormula() function is used to parse the R formula to an object with a structure that is usable by R wrappers for Netezza Analytics functions.

In formulas, symbols such as “-1” or “.” are supported. For some procedures, such as regression, some implicit type casting is needed for categorical variables.

```
nzdf = nz.data.frame("adult")
form = ~EDUCATION+OCCUPATION
cf = nzParseRFormula(form, nzdf)
str(cf)

form = ~AGE+factor(OCCUPATION)
cf = nzParseRFormula(form, nzdf)
str(cf)
```

Conversion of Row - Column - Value Format into a R Matrix

Typically, matrices are stored in the NPS system tables in the row-column-value format. Therefore, there is a row in the table with values for every matrix cell. Tensors with an order higher than two —multivariate contingency tables—are stored in a similar way. For a dimensional matrix, the first column in the table denotes cell coordinates, while the last column denotes the corresponding value.

The nzSparse2matrix() function takes an *nz.data.frame* or *data.frame* object as input and returns a transformed matrix.

```
mat = as.data.frame(matrix(c(1,1,2,1,3,2,2,4,1),3,3))
nzSparse2matrix(mat)
```

CHAPTER 5
Netezza Matrix Package

The Netezza Matrix Library package consists of R functions and objects that enable working with large in-database matrices directly from R clients such as the R GUI.

Matrix Catalog Management

All the database matrix objects are stored in the Matrix Catalog that you can access through the SQL-based interface. The Netezza Matrix Library provides R wrappers to the interface.

The following table lists arguments that are taken by the R functions that are described in this topic.

Table 2: Arguments taken by R functions

Argument	Value
x	nz.matrix object
name	name of database matrix object

nzMatrixEngineInitialization

This function initializes or re-initializes the Matrix Engine in the currently connected database. This function does not delete any preexisting matrices. To delete all matrices, use the nzDeleteAllMatrices() function.

Usage

```
nzMatrixEngineInitialization()
```

nzDeleteAllMatrices

This function deletes all matrices from the currently connected database.

Usage

```
nzDeleteAllMatrices()
```

nzDeleteMatrix

This function removes the database matrix object specified by **x** from the matrix catalog and database.

Usage

```
nzDeleteMatrix(x)
```

nzDeleteMatrixByName

This function removes the database matrix object specified by **name** from the matrix catalog and database.

Usage

```
nzDeleteMatrixByName(name)
```

nzExistMatrix

This function checks if the database matrix object specified by **x** exists in the catalog.

Usage

```
nzExistMatrix(x)
```

nzExistMatrixByName

This function queries the database matrix catalog to check if the matrix object specified by **name** exists and is accessible by the user.

Usage

```
nzExistMatrixByName(name)
```

Functions for Matrix Creation

The following table shows typical argument names for matrix functions.

Argument	Value
x	Object of class nz.matrix
name	Matrix name. The name is optional. If a name is not specified, the function chooses a name that does not conflict with existing matrices.
size	Matrix dimension for square matrices. The default size is 1 x 1 .
nrows	Number of rows
ncols	Number of columns
z	R object

The value that is returned by all the functions is an object of class nz.matrix that stores a reference to the corresponding database matrix object in the database.

as.nz.matrix

This function creates a database matrix object that mirrors a given R matrix. The cell values of the matrix specified by **z** are transmitted and stored in the database.

Usage

```
as.nz.matrix(z, name = NULL)
```

nzIdentityMatrix

This function creates an identity matrix as a database matrix object of the specified size and with the specified name. An identity matrix is a matrix with all off–diagonal cells equal to 0 and all diagonal cells equal to 1.

Usage

```
nzIdentityMatrix(size=1, name = NULL)
```

nzNormalMatrix

This function creates a database matrix object of the specified size and with the specified name by using random values that are drawn from the normal distribution (*N*(0,1)).

Usage

```
nzNormalMatrix(nrows = 1, ncols = 1, name = NULL)
```

nzRandomMatrix

This function creates a database matrix object of the specified size and with the specified name by using random values drawn from the uniform distribution (*U*[0, 1]).

Usage

```
nzRandomMatrix(nrows = 1, ncols = 1, name = NULL)
```

nzOnesMatrix

This function creates a database matrix object of the specified size and with the specified name, where all cell values are equal to 1.

Usage

```
nzOnesMatrix(nrows = 1, ncols = 1, name = NULL)
```

nzVecToDiag

This function creates a database matrix object that contains a square matrix that is created from the specified column matrix. The specified column matrix must be of size *n* x 1. The resulting matrix size is *n* x *n* with off–diagonal elements equal to 0 and diagonal part equal to the given column matrix, where cell [*i*, *i*] of the resulting matrix corresponds to the cell [*i*, 1] of specified matrix.

Usage

```
nzVecToDiag(x)
```

Examples

```
# Environment is already initialized, DB connection established
mat <- matrix(0,2,2)
nzmat <- as.nz.matrix(mat)
nzmat2 = nzRandomMatrix(10, 10)
```

Scalar Operations

Scalar operations are part of the class of functions that are applied to each cell of a matrix. Scalar operations provide the transformation of a given database matrix object into a newly created one. Note that the values of the specified matrix are not changed. Each cell value is transformed independently of other cells. The return value of each scalar operation is an object of class *nz.matrix* containing a reference to the database matrix object containing the result of the operation.

The following table shows typical argument names for scalar operation functions.

Table 3: Argument names for scalar operation functions

Argument	Value
x	nz.matrix object
dvr	numeric
num	numeric

abs

This function does an absolute value transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**:

$$x[i,j] \mapsto |x[i,j]|$$

Usage

```
abs(x)
```

exp

This function does an exponential transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**:

$$x[i,j] \mapsto \exp\{x[i,j]\}$$

Usage

```
exp(x)
```

ln

This function does a logarithmic transformation of all positive elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**:

$$x[i,j] \mapsto \log_e x[i,j]$$

If one or more cell values are non-positive, an error is returned.

Usage

```
ln(x)
```

log10

This function does logarithmic transformation of all positive elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**:

$$x[i,j] \mapsto \log_{10} x[i,j]$$

If one or more cell value is non-positive, an error is reported.

Usage

```
log10(x)
```

pow

This function does a power transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x** with the exponent specified by **num**:

$$x[i,j] \mapsto (x[i,j])^{\text{num}}$$

In case of an undefined result, that is, a negative non-integer number raised to the power of non-integer exponent, an error is reported.

Usage

```
pow(x, num=2)
```

sqrt

This function does the square root transformation of all non–negative elements of the database matrix object referred to by the nz.matrix.object specified by **x**:

$$x[i,j] \mapsto \sqrt{x[i,j]}$$

If one or more cell value is non-positive, an error is reported.

Usage

```
sqrt(x)
```

rounding

This function does a rounding transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**. Each cell value is rounded, that is, transformed to the nearest integer.

Usage

```
rounding(x, digits=1)
```

trunc

This function does a truncating transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**. Each cell value is rounded towards zero, that is, it is transformed to the nearest integer number, which is equivalent to discarding any fractional part of the value, regardless of the sign of the number.

Usage

```
trunc(x)
```

ceiling

This function does a rounding transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**. Each cell value is rounded towards positive infinity, that is, the nearest equal or greater integer.

Usage

```
ceiling(x)
```

floor

This function does a rounding transformation on all elements of the database matrix object that is referred to by the nz.matrix.object specified by **x**. Each cell value is rounded towards negative infinity, that is, the nearest equal or smaller integer.

Usage

```
floor(x)
```

mod

This function maps each element of the database matrix object that is referred to by the nz.matrix.object specified by **x** to its truncated modulo division remainder resulting from the divisor specified by **dvr,** where the trunc function is defined accordingly to the respective scalar operator.

$$x[i,j] \mapsto x[i,j] - \mathrm{dvr} \cdot \mathrm{trunc}\left(\frac{x[i,j]}{\mathrm{dvr}}\right)$$

Usage

```
mod(x, dvr)
```

add

This function does a transformation of the database matrix object that is referred to by the nz.matrix.object specified by **x** by adding the value specified by **num** to each of the cells:

$$x[i,j] \mapsto x[i,j] + \mathrm{num}$$

Usage

```
add(x, num)
```

subt

This function does a transformation of the database matrix object that is referred to by the nz.matrix.object specified by **x** by subtracting the value specified by **num** from each of the cells:

$$x[i,j] \mapsto x[i,j] - \mathrm{num}$$

Usage

```
subt(x, num)
```

mult

This function does a transformation of the database matrix object that is referred to by the nz.matrix.object specified by **x** by multiplying each of the cells by the value specified by **num**:

$$x[i, j] \mapsto x[i, j] \cdot \mathrm{num}$$

Usage

```
mult(x, num)
```

div

This function does a transformation of the database matrix object that is referred to by the nz.matrix.object specified by **x** by dividing each of the cells by the value specified by **num**. The value of **num** must be non-zero.

$$x[i, j] \mapsto \frac{x[i, j]}{\mathrm{num}}$$

Usage

```
div(x, num)
```

Reduction Operators

Reduction operators is the class of functions that reduce all cells of a given matrix to one numeric variable. The output variable depends only on cell values.

The argument for the reduction operation functions is as follows:

Argument	Value
x	nz.matrix object

nzAll

This function checks if all values of the elements of the database nz.matrix object that is specified by **x** are not equal to 0, returning **TRUE** when the condition is met, and **FALSE** otherwise.

Usage

```
nzAll(x)
```

nzAny

This function checks if any value of the elements of the database nz.matrix object that is specified by **x** is not equal to 0, returning **TRUE** when the condition is met, and **FALSE** otherwise.

Usage

```
nzAny(x)
```

nzMax

This function returns the maximum value element of the database nz.matrix object that is specified by **x**.

Usage

```
nzMax(x)
```

nzMin

This function returns the minimum element value of the database nz.matrix object that is specified by **x**.

Usage

```
nzMin(x)
```

nzSsq

This function returns the sum of squared values of all elements of the database nz.matrix object that is specified by **x**.

Usage

```
nzSsq(x)
```

nzSum

This function returns the sum of values of all elements of the database nz.matrix object that is specified by **x**.

Usage

```
nzSum(x)
```

nzTr

This function calculates and returns the value of the trace of a square database nz.matrix object that is specified by **x**. A trace is the sum of the diagonal part elements.

Usage

```
nzTr(x)
```

Matrix Inquiry Functions

Matrix inquiry functions provide simple information about database matrix objects.

The argument for the reduction operation functions is as follows:

Argument	Value
x	nz.matrix object

dim

This function returns a numerical vector of length 2 that contains information about the dimensions of the given database matrix object. The first element of the vector specifies the number of rows; the second element specifies the number of columns.

ncol

This function returns the number of columns of a given database matrix object.

nrow

This function returns the number of rows of a given database matrix object.

is.nz.matrix

This function checks if the R object specified by **x** is an object of class *nz.matrix*, returning **TRUE** when the condition is met, and **FALSE** otherwise.

Usage

```
is.nz.matrix(x)
```

Matrix Manipulation Operations

The arguments for matrix manipulation operation functions are:

Argument	Value
x	nz.matrix object
y	nz.matrix object

nzCBind

This function combines the two database matrix objects that are referenced by the arguments by appending the columns of the object specified by **y** to the columns of the object specified by **x**. The result is stored in newly created database matrix object. This function returns the reference to the resulting nz.matrix object.

Usage

```
nzCBind(x,y)
```

nzRBind

This function combines two database matrix objects that are referenced by the arguments by appending the rows of the second argument (**y**) to the rows of the first (**x**). The result is stored in newly created database matrix object. This function returns the reference to the resulting nz.matrix object .

Usage

```
nzRBind(x,y)
```

Transposition operator

This function does the transposition of the database matrix object that is referenced by the argument and stores the result in the newly created database matrix object. This function returns the reference to the resulting nz.matrix object.

Usage

```
t(x)
```

Linear Algebra Operations

The arguments for linear algebra operation functions are:

Argument	Value
x	nz.matrix object
y	nz.matrix object

Eigenvalues and Eigenvectors

This function computes the eigenvalues and eigenvectors of the symmetric database matrix object that is referred to by the nz.matrix.object specified in argument **x**. As a result, two new database matrix objects are created. One object stores eigenvalues (denoted by W) in the form of a one-column matrix of dimensions n x 1, the other object stores the eigenvectors column-wise (denoted by Z), where $W[k,1]$ is an eigenvalue for each eigenvector $Z[,k]$. This function returns a list of two nz.matrix objects that contain the references to matrix W (named w) and matrix Z (named z) in that order.

Usage

```
nzEigen(x)
```

Inversion

This function computes the inversion (based on LU factorization) of the square database matrix object that is referenced by the nz.matrix.object specified in argument **x**. If the matrix is singular, the pseudo-inversion is computed. The result is stored as the database matrix object. This function returns the reference nz.matrix class object to the resulting database matrix object.

Usage

```
nzInv(x)
```

Solve

This function solves the matrix equation concerning two database matrix objects that are referred to by the provided arguments (nz.matrix objects x and y) in the following form for the matrix:

$$x \cdot b = y$$

The result is stored as the database matrix object. This function returns the reference object (object of nz.matrix class) to the resulting database matrix object.

Usage

```
nzSolve(x, y)
```

nzSolveLLS

This function finds the linear least squares solution to the matrix equation concerning two database matrix objects referred to by the provided arguments (nz.matrix objects x and y) of the following form for the matrix:

$$x \cdot b = y$$

The result is stored as the database matrix object. This function returns the reference object (object of nz.matrix class) to the resulting database matrix object.

Usage

```
nzSolveLLS(x,y)
```

Singular Value Decomposition (nzSVD)

This function does a singular value decomposition (SVD) of the database matrix object that is referred to by the provided argument (object of nz.matrix class) and stores the results that consist of four matrices as database matrix objects. The function returns a list of four nz.matrix objects in the order u, s, vt, and v, where u and v are the SVD transformation matrices, vt is the transpose of v, and s is a one-column matrix containing the singular values. To create a diagonal matrix containing the singular values, you can use nzVecToDiag(s) t.

Usage

```
nzSVD(x)
```

Matrix Operators

Matrix operators are the class of functions transforming database matrix objects, usually of the same dimensions, provided as arguments to a single database matrix object, usually of the same dimensions as the arguments.

Elementwise Operators

Most of the introduced operators work in an element-wise manner: the cell of the resulting matrix, which is denoted by z, depends only on the values of corresponding cells in both arguments, which are denoted by x and y:

Operator	Value of $z[i,j]$
x+y	$x[i,j]+y[i,j]$
x-y	$x[i,j]-y[i,j]$
x*y	$x[i,j]\cdot y[i,j]$
x/y	$\frac{x[i,j]}{y[i,j]}$
nzPMin(x,y)	$\min(x[i,j],y[i,j])$
nzPMax(x,y)	$\max(x[i,j],y[i,j])$

Element-wise Comparison Operators

A special case of element-wise operators are comparison operators that test the given logical condition for each corresponding cell:

Operator	Value of $z[i,j]$
x<y	$\#\{x[i,j]<y[i,j]\}$
x>y	$\#\{x[i,j]>y[i,j]\}$
x<=y	$\#\{x[i,j]\le y[i,j]\}$
x>=y	$\#\{x[i,j]\ge y[i,j]\}$
x==y	$\#\{x[i,j]=y[i,j]\}$
x!=y	$\#\{x[i,j]\ne y[i,j]\}$

Subscripting Operator

This operator creates a rectangular submatrix of a given database matrix object and stores it as a database matrix object. The reference to the resulting object is returned.

Usage

```
x[1:5,5:10]
```

nzPowerMatrix

This operator raises the given database matrix object to the power of the positive integer that is specified by the **num** argument and stores it as a database matrix object. The reference to the resulting object is returned.

Usage

```
nzPowerMatrix(x,num)
```

Matrix Multiplication Operator

This operator does the multiplication of two specified database matrix objects and stores it as a database matrix object. The reference to the resulting object is returned.

Usage

```
x %*% y
```

Kronecker Product (nzKronecker)

This operator calculates the Kronecker product of two database matrix objects and stores it as a database matrix object. The reference to the resulting object is returned.

Usage

```
nzKronecker(x,y)
```

or

```
x %x% y
```

CHAPTER 6

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.

Open Source Notifications

PostgreSQL

Portions of this publication were derived from PostgreSQL documentation. For those portions of the documentation that were derived originally from PostgreSQL documentation, and only for those portions, the following applies:

PostgreSQL is copyright © 1996-2001 by the PostgreSQL global development group and is distributed under the terms of the license of the University of California below.

Postgres95 is copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

In no event shall the University of California be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation, even if the University of California has been advised of the possibility of such damage.

The University of California specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The documentation provided hereunder is on an "as-is" basis, and the University of California has no obligations to provide maintenance, support, updates, enhancements, or modifications.

ICU Library

The Netezza implementation of the ICU library is an adaptation of an open source library Copyright (c) 1995-2003 International Business Machines Corporation and others.

ICU License - ICU 1.8.1 and later
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2003 International Business Machines Corporation and others. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,

INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

ODBC Driver

The Netezza implementation of the ODBC driver is an adaptation of an open source driver, Copyright © 2000, 2001, Great Bridge LLC. The source code for this driver and the object code of any Netezza software that links with it are available upon request to sourcerequest@netezza.com

Botan License

Copyright (C) 1999-2008 Jack Lloyd

2001 Peter J Jones
2004-2007 Justin Karneges
2005 Matthew Gregan
2005-2006 Matt Johnston
2006 Luca Piccarreta
2007 Yves Jerschow
2007-2008 FlexSecure GmbH
2007-2008 Technische Universitat Darmstadt
2007-2008 Falko Strenzke
2007-2008 Martin Doering
2007 Manuel Hartl
2007 Christoph Ludwig
2007 Patrick Sona

All rights reserved.

Redistribution and use in source and binary forms, for any use, with or without modification, of Botan (http://botan.randombit.net/license.html) is permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR(S) "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY

AND FITNESS FOR A PARTICULAR PURPOSE, ARE DISCLAIMED.

IN NO EVENT SHALL THE AUTHOR(S) OR CONTRIBUTOR(S) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITYOF SUCH DAMAGE.

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

Attention: This product is not intended to be connected directly or indirectly by any means whatsoever to interfaces of public telecommunications networks, neither to be used in a Public Services Network.

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.

WEEE

Netezza Corporation is committed to meeting the requirements of the European Union (EU) Waste Electrical and Electronic Equipment (WEEE) Directive. This Directive requires producers of electrical and electronic equipment to finance the takeback, for reuse or recycling, of their products placed on the EU market after August 13, 2005.

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。