TIBCO®

The Power of Now®

Big Data User's Guide for TIBCO Spotfire S+® 8.2

November 2010

TIBCO Software Inc.

IMPORTANT INFORMATION

software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

Reference

The correct bibliographic reference for this document is as follows:

Big Data User’s Guide for TIBCO Spotfire S+® 8.2, TIBCO Software Inc.

Technical Support

For technical support, please visit http://spotfire.tibco.com/support and register for a support account.

TIBCO SPOTFIRE S+ BOOKS

Note about Naming
Throughout the documentation, we have attempted to distinguish between the language (S-PLUS) and the product (Spotfire S+). • "S-PLUS" refers to the engine, the language, and its constituents (that is objects, functions, expressions, and so forth). • "Spotfire S+" refers to all and any parts of the product beyond the language, including the product user interfaces, libraries, and documentation, as well as general product and language behavior.

The TIBCO Spotfire S+® documentation includes books to address your focus and knowledge level. Review the following table to help you choose the Spotfire S+ book that meets your needs. These books are available in PDF format in the following locations:

- In your Spotfire S+ installation directory (**SHOME\help** on Windows, **SHOME/doc** on UNIX/Linux).
- In the Spotfire S+ Workbench, from the **Help ▶ Spotfire S+ Manuals** menu item.
- In Microsoft® Windows®, in the Spotfire S+ GUI, from the **Help ▶ Online Manuals** menu item.

Spotfire S+ documentation.

Information you need if you...	See the...
Must install or configure your current installation of Spotfire S+; review system requirements.	*Installtion and Administration Guide*
Want to review the third-party products included in Spotfire S+, along with their legal notices and licenses.	*Licenses*

Spotfire S+ documentation. (Continued)

Information you need if you...	**See the...**
Are new to the S language and the Spotfire S+ GUI, and you want an introduction to importing data, producing simple graphs, applying statistical models, and viewing data in Microsoft Excel®.	*Getting Started Guide*
Are a new Spotfire S+ user and need how to use Spotfire S+, primarily through the GUI.	*User's Guide*
Are familiar with the S language and Spotfire S+, and you want to use the Spotfire S+ plug-in, or customization, of the Eclipse Integrated Development Environment (IDE).	*Spotfire S+ Workbench User's Guide*
Have used the S language and Spotfire S+, and you want to know how to write, debug, and program functions from the **Commands** window.	*Programmer's Guide*
Are familiar with the S language and Spotfire S+, and you want to extend its functionality in your own application or within Spotfire S+.	*Application Developer's Guide*
Are familiar with the S language and Spotfire S+, and you are looking for information about creating or editing graphics, either from a **Commands** window or the Windows GUI, or using Spotfire S+ supported graphics devices.	*Guide to Graphics*
Are familiar with the S language and Spotfire S+, and you want to use the Big Data library to import and manipulate very large data sets.	*Big Data User's Guide*
Want to download or create Spotfire S+ packages for submission to the Comprehensive S-PLUS Archive Network (CSAN) site, and need to know the steps.	*Guide to Packages*

Spotfire S+ documentation. (Continued)

Information you need if you...	**See the...**
Are looking for categorized information about individual S-PLUS functions.	*Function Guide*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 1 includes information on specifying models in Spotfire S+, on probability, on estimation and inference, on regression and smoothing, and on analysis of variance.	*Guide to Statistics, Vol. 1*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 2 includes information on multivariate techniques, time series analysis, survival analysis, resampling techniques, and mathematical computing in Spotfire S+.	*Guide to Statistics, Vol. 2*

CONTENTS

INTRODUCTION TO THE BIG DATA LIBRARY

1

INTRODUCTION

In this chapter, we discuss the history of the S language and large data sets and describe improvements that the Big Data library presents. This chapter discusses data set size considerations, including when to use the Big Data library. The chapter also describes in further detail the Big Data library architecture: its data objects, classes, functions, and advanced operations.

To use the Big Data library, you must load it as you would any other library provided with Spotfire S+: that is, at the command prompt, type `library(bigdata)`.

- To ensure that the library is always loaded on startup, add `library(bigdata)` to your **SHOME/local/S.init** file.
- Alternatively, in the Spotfire S+ GUI for Microsoft Windows®, you can set this option in the **General Settings** dialog box.
- In the Spotfire S+ Workbench, you can set this option in the **Spotfire S+** section of the **Preferences** dialog box, available from the **Window** menu.

WORKING WITH A LARGE DATA SET

When it was first developed, the S programming language was designed to hold and manipulate data in memory. Historically, this design made sense; it provided faster and more efficient calculations and modeling by not requiring the user's program to access information stored on the hard drive. Data size has outstripped the rate at which RAM size increased; consequently, S program users could have encountered an error similar to the following:

```
Problem in read.table: Unable to obtain requested dynamic
memory.
```

This error occurs because Spotfire S+ requires the operating system to provide a block of memory large enough to contain the contents of the data file, and the operating system responds that not enough memory is available.

While Spotfire S+ can access data contained in virtual memory, the maximum size of data files depends on the amount of virtual memory available to Spotfire S+, which depends in turn on the user's hardware and operating system. In typical environments, virtual memory limits your data file size, and then it returns an out-of-memory error.

Finally, you can also encounter an out-of-memory error after successfully reading in a large data object, because many S functions require one or more temporary copies of the source data in RAM for certain manipulation or analysis functions.

Finding a Solution

S programmers with large data sets have historically dealt with memory limitations in a variety of ways. Some opted to use other applications, and some divided their data into "digestible" batches, and then recompile the results. For S programmers who like the flexibility and elegant syntax of the S language and the support provided to owners of a Spotfire S+ license, the option to analyze and model large data sets in S has been a long-awaited enhancement.

Out-of-Memory Processing

The Big Data library provides this enhancement by processing large data sets using scalable algorithms and data streaming. Instead of loading the contents of a large data file into memory, Spotfire S+ creates a special binary cache file of the data on the user's hard disk, and then refers to the cache file on disk. This out-of-memory design requires relatively small amounts of RAM, regardless of the total size of the data.

Scalable Algorithms

Although the large data set is stored on the hard drive, the scalable algorithms of the Big Data library are designed to optimize access to the data, reading from disk a minimum number of times. Many techniques require a single pass through the data, and the data is read from the disk in blocks, not randomly, to minimize disk access times. These scalable algorithms are described in more detail in the section The Big Data Library Architecture on page 8.

Data Streaming

Spotfire S+ operates on the data binary cache file directly, using "streaming" techniques, where data flows through the application rather than being processed all at once in memory. The cache file is processed on a row-by-row basis, meaning that only a small part of the data is stored in RAM at any one time. It is this out-of-memory data processing technique that enables Spotfire S+ to process data sets hundreds of megabytes, or even gigabytes, in size without requiring large quantities of RAM.

Data Type

Spotfire S+ provides the large data frame, an object of class `bdFrame`. A big data frame object is similar in function to standard S-PLUS data frames, except its data is stored in a cache file on disk, rather than in RAM. The `bdFrame` object is essentially a reference to that external file: While you can create a `bdFrame` object that represents an extremely large data set, the `bdFrame` object itself requires very little RAM.

For more information on `bdFrame`, see the section Data Frames on page 11.

Spotfire S+ also provides time date (`bdTimeDate`), time span (`bdTimeSpan`), and series (`bdSeries`, `bdSignalSeries`, and `bdTimeSeries`) support for large data sets. For more information, see the section Time Date Creation on page 175 in the Appendix.

Flexibility

The Big Data library provides reading, manipulating, and analyzing capability for large data sets using the familiar S programming language. Because most existing data frame methods work in the same way with `bdFrame` objects as they do with `data.frame` objects, the style of programming is familiar to Spotfire S+ programmers. Much existing code from previous versions of Spotfire S+ runs without modification in the Big Data library, and only minor modifications are needed to take advantage of the big-data capabilities of the pipeline engine.

Balancing Scalability with Performance

While accessing data on disk (rather than in RAM) allows for scalable statistical computing, some compromises are inevitable. The most obvious of these is computation speed. The Big Data library provides scalable algorithms that are designed to minimize disk access, and therefore provide optimal performance with out-of-memory data sets. This makes Spotfire S+ a reliable workhorse for processing very large amounts of data. When your data is small enough for traditional Spotfire S+, it's best to remember that in-memory processes are faster than out-of-memory processes.

If your data set size is not extremely large, all of the Spotfire S+ traditional in-memory algorithms remain available, so you need not compromise speed and flexibility for scalability when it's not needed.

Metadata

To optimize performance, Spotfire S+ stores certain calculated statistics as metadata with each column of a `bdFrame` object and updates the metadata every time the data changes. These statistics include the following:

- Column mean (for numeric columns).
- Column maximum and minimum (for numeric and date columns).
- Number of missing values in the column.
- Frequency counts for each level in a categorical column.

Requesting the value of any of these statistics (or a value derived from them) is essentially a free operation on a `bdFrame` object. Instead of processing the data set, Spotfire S+ just returns the precomputed statistic. As a result, calculations on columns of `bdFrame` objects such as the following examples are practically instantaneous, regardless of the data set size. For example:

- `mean(census$Income)`
- `range(census$Age)`

No 64-Bit Solution

Are out-of-memory data analysis techniques still necessary in the 64-bit age? While 64-bit operating systems allow access to greater amounts of *virtual* memory, it is the amount of *physical* memory that is the primary determinant of efficient operation on large data sets. For this reason, the out-of-memory techniques described above are still required to analyze truly large data sets.

64-bit systems increase the amount of memory that the system can address. This can help in-memory algorithms handle larger problems, provided that all of the data can be in physical memory. If the data and the algorithm require virtual memory, page-swapping (that is, accessing the data in virtual memory on the disk) can have a severe impact on performance.

With data sets now in the multiple gigabyte range, out-of-memory techniques are essential. Even on 64-bit systems, out-of-memory techniques can dramatically outperform in-memory techniques when the data set exceeds the available physical RAM.

SIZE CONSIDERATIONS

While the Big Data library imposes no predetermined limit for the number of rows allowed in a big data object or the number of elements in a big data vector, your computer's hard drive must contain enough space to hold the data set and create the data cache. Given sufficient disk space, the big data object can be created and processed by any scalable function.

The speed of most Big Data library operations is proportional to the number of rows in the data set: if the number of rows doubles, then the processing time also doubles.

The amount of RAM in a machine imposes a predetermined limit on the number of columns allowed in a big data object, because column information is stored in the data set's metadata. This limit is in the tens of thousands of columns. If you have a data set with a large number of columns, remember that some operations (especially statistical modeling functions) increase at a greater than linear rate as the number of columns increases. Doubling the number of columns can have a much greater effect than doubling the processing time. This is important to remember if processing time is an issue.

Summary

By bringing together flexible programming and big-data capability, Spotfire S+ is a data analysis environment that provides both rapid prototyping of analytic applications and a scalable production engine capable of handling datasets hundreds of megabytes, or even gigabytes, in size.

In the next section, we provide an overview to the Big Data library architecture, including data types, functions, and naming conventions.

THE BIG DATA LIBRARY ARCHITECTURE

The Big Data library is a separate library from the S-PLUS engine library. It is designed so that you can work with large data objects the same way you work with existing S-PLUS objects, such as data frames and vectors.

Block-based Computations

Data sets that are much larger than the system memory are manipulated by processing one "block" of data at a time. That is, if the data is too large to fit in RAM, then the data will be broken into multiple data sets and the function will be applied to each of the data sets. As an example, a 1,000,000 row by 10 column data set of double values is 76MB in size, so it could be handled as a single data set on a machine with 256MB RAM. If the data set was 10,000,000 rows by 100 columns, it would be 7.4GB in size and would have to be handled as multiple blocks.

Table 1.1 lists a few of the optional arguments for the function `bd.options` that you can use to set limits for caching and for warnings:

Table 1.1: *`bd.options` block-based computation arguments.*

`bd.option` **argument**	**Description**
`block.size`	The block size (in number of rows), the number of bytes in the cache to be converted to a `data.frame`.
`max.convert.bytes`	The maximum size (in bytes) of the big data cache that can be converted to a `data.frame`.
`max.block.mb`	The maximum number of megabytes used for block processing buffers. If the specified block size requires too much space, the number of rows is reduced so that the entire buffer is smaller than this size. This prevents unexpected out-of-memory errors when processing wide data with many columns. The default value is 10.

The function `bd.options` contains other optional arguments for controlling column string width, display parameters, factor level limits, and overflow warnings. See its help topic for more information.

The Big Data library also contains functions that you can use to control block-based computations. These include the functions in Table 1.2. For more information and examples showing how to use these functions, see their help topics.

Table 1.2: *Block-based computation functions.*

Function name	Description
`bd.aggregate`	Use `bd.aggregate` to divide a data object into blocks according to the values of one or more of its columns, and then apply aggregation functions to columns within each block. `bd.aggregate` takes two required arguments: data, which is the input data set, and `by.columns`, which identifies the names or numbers of columns defining how the input data is divided into blocks. Optional arguments include `columns`, which identifies the names or numbers of columns to be summarized, and `methods`, which is a vector of summary methods to be calculated for columns. See the help topic for `bd.aggregate` for a list of the summary methods you can specify for `methods`.
`bd.block.apply`	Run a S-PLUS script on blocks of data, with options for reading multiple input datasets and generating multiple output data sets, and processing blocks in different orders. See the help topic for `bd.block.apply` for a discussion on processing multiple data blocks.
`bd.by.group`	Apply the specified S-PLUS function to multiple data blocks within the input dataset.

Table 1.2: *Block-based computation functions. (Continued)*

Function name	Description
`bd.by.window`	Apply the specified S-PLUS function to multiple data blocks defined by a moving window over the input dataset. Each data block is converted to a `data.frame`, and passed to the specified function. If one of the data blocks is too large to fit in memory, an error occurs.
`bd.split.by.group`	Divide a dataset into multiple data blocks, and return a list of these data blocks.
`bd.split.by.window`	Divide a dataset into multiple data blocks defined by a moving window over the dataset, and return a list of these data blocks.

For a detailed discussion on advanced topics, such as block size issues and increasing efficiency, see Chapter 5, Advanced Programming Information.

Data Types

S-PLUS provides the following data types, described in more detail below:

Table 1.3: *Big Data types and data names for S-PLUS.*

Big Data class	Data type
`bdFrame`	Data frame
`bdVector`, `bdCharacter`, `bdFactor`, `bdLogical`, `bdNumeric`, `bdTimeDate`, `bdTimeSpan`	Vector
`bdLM`, `bdGLM`, `bdPrincomp`, `bdCluster`	Models
`bdSeries`, `bdTimeSeries`, `bdSignalSeries`	Series

Data Frames

The main object to contain your large data set is the big data frame, an object of class `bdFrame`. Most methods commonly used for a `data.frame` are also available for a `bdFrame`. Big data frame objects are similar to standard S-PLUS data frames, except in the following ways:

- A `bdFrame` object stores its data on disk, while a `data.frame` object stores its data in RAM. As a result, a `bdFrame` object has a much smaller memory footprint than a `data.frame` object.
- A `bdFrame` object does not have row labels, as a `data.frame` object does. While this means that you cannot refer to the rows of a `bdFrame` object using character row labels, this design reduces storage requirements and improves performance by eliminating the need to maintain unique row labels.
- A `bdFrame` object can contain columns of only types `double`, `character`, `factor`, `timeDate`, `timeSpan` or `logical`. No other column types (such as matrix objects or user-defined classes) are allowed. By limiting the allowed column types, Spotfire S+ ensures that the binary cache file representing the data is as compact as possible and can be efficiently accessed.

- The `print` function works differently on a `bdFrame` object than it does for a data frame. It displays only the first few rows and columns of data instead of the entire data set. This design prevents accidentally generating thousands of pages of output when you display a `bdFrame` object at the command line.

> **Note**
>
> You can specify the numbers of rows and columns to print using the `bd.options` function. See `bd.options` in the Spotfire S+ Language Reference for more information.

- The `summary` function works differently on a `bdFrame` object than it does for a data frame. It calculates an abbreviated set of summary statistics for numeric columns. This design is for efficiency reasons: `summary` displays only statistics that are precalculated for each column in the big data object, making `summary` an extremely fast function, even when called on a very large data set.

Vectors

The Spotfire S+ Big Data library also introduces `bdVector` and six subclasses, which represent new vector types to support very long vectors. Like a `bdFrame` object, the big vector object stores data out-of-memory as a cache file on disk, so you can create very long big vector objects without needing a lot of RAM.

You can extract an individual column from a `bdFrame` object (using the `$` operator) to create a large vector object. Alternatively, you can generate a large vector using the functions listed in Table A.3 in the Appendix. Like `bdFrame` objects, the actual data is stored out of memory as a cache file on disk, so you can create very long big vector objects without worrying about fitting them into RAM. You can use standard vector operations, such as selections and mathematical operations, on these data types. For example, you can create new columns in your data set, as follows:

```
census$adjusted.income <- log(census$income -
  census$tax)
```

Models

Spotfire S+ Big Data library provides scalable modeling algorithms to process big data objects using out-of-memory techniques. With these modeling algorithms, you can create and evaluate statistical models on very large data sets.

A model object is available for each of the following statistical analysis model types.

Table 1.4: *Big Data library model objects.*

Model Type	Model Object
Linear regression	`bdLm`
Generalized linear models	`bdGlm`
Clustering	`bdCluster`
Principal Components Analysis	`bdPrincomp`

When you perform statistical analysis on a large data set with the Big Data library, you can use familiar S-PLUS modeling functions and syntax, but you supply a `bdFrame` object as the data argument, instead of a data frame. This forces out-of-memory algorithms to be used, rather than the traditional in-memory algorithms.

When you apply the modeling function `lm` to a `bdFrame` object, it produces a model object of class `bdLm`. You can apply the standard `predict`, `summary`, `plot`, `residuals`, `coef`, `formula`, `anova`, and `fitted` methods to these new model objects.

For more information on statistical modeling, see Chapter 2, Census Data Example.

Series Objects

The standard S-PLUS library contains a series object, with two subclasses: `timeSeries` and `signalSeries`. The series object contain:

- A `data` component that is typically a data frame.
- A `positions` component that is a `timeDate` or `timeSequence` object (`timeSeries`), or a `bdNumeric` or `numericSeries` object (`signalSeries`).
- A `units` component that is a character vector with information on the units used in the data columns.

The Big Data library equivalent is a `bdSeries` object with two subclasses: `bdTimeSeries` and `bdSignalSeries`. They contain:

- A `data` component that is a `bdFrame`.
- A `positions` component that is a `bdTimeDate` object (`bdTimeSeries`), or `bdNumeric` object (`bdSignalSeries`).
- A `units` component that is a character vector.

For more information about using large time series objects and their classes, see the section Time Classes on page 17.

Classes

The Big Data library follows the same object-oriented design as the standard Spotfire S+ Sv4 design. For a review of object-oriented programming concepts, see Chapter 8, Object-Oriented Programming in Spotfire S+ in the *Programmer's Guide*.

Each object has a class that defines methods that act on the object. The library is extensible; you can add your own objects and classes, and you can write your own methods.

The following classes are defined in the Big Data library. For more information about each of these classes, see their individual help topics.

Table 1.5: *Big Data classes.*

Class(es)	**Description**
`bdFrame`	Big data frame
`bdLm`, `bdGlm`, `bdCluster`, `bdPrincomp`	Rich model objects
`bdVector`	Big data vector
`bdCharacter`, `bdFactor`, `bdLogical`, `bdNumeric`, `bdTimeDate`, `bdTimeSpan`	Vector type subclasses
`bdTimeSeries, bdSignalSeries`	Series objects

Functions

In addition to the standard S-PLUS functions that are available to call on large data sets, the Big Data library includes functions specific to big data objects. These functions include the following.

- Big vector generating functions
- Data exploration and manipulation functions.
- Traditional and Trellis graphics functions.
- Modeling functions.

The functions for these general tasks are listed in the Appendix.

Data Import and Export

Two of the most frequent tasks using Spotfire S+ are importing and exporting data. The functions are described in Table A.1 in Appendix. You can perform these tasks from the **Commands** window, from the Console view in the Spotfire S+ Workbench, or from the Spotfire S+ import and export dialog boxes in the Spotfire S+ GUI. For more information about importing large data sets, see the section Data Import on page 25 in Chapter 2, Census Data Example.

Big Vector Generation

To generate a vector for a large data set, call one of the S-PLUS functions described in Table A.3 in the Appendix. When you set the `bigdata` flag to `TRUE`, the standard S-PLUS functions generate a `bdVector` object of the specified type. For example:

```
# sample of size 2000000 with mean 10*0.5 = 5
rbinom(2000000, 10, 0.5, bigdata = T)
```

Data Exploration Functions

After you import your data into Spotfire S+ and create the appropriate objects, you can use the functions described in Table A.4 in the Appendix. to compare, correlate, crosstabulate, and examine univariate computations.

Data Manipulation Functions

After you import and examine your data in Spotfire S+, you can use the data manipulation functions to append, filter, and clean the data. For an overview of these functions, see Table A.5 in the Appendix. For a more in-depth discussion of these functions, see the section Data Manipulation on page 36 in Chapter 2, Census Data Example.

Graph Functions

The Big Data library supports graphing large data sets intelligently, using the following techniques to manage many thousands or millions of data points:

- Hexagonal binning. (That is, functions that create one point per observation in standard Spotfire S+ create a hexagonal binning plot when applied to a big data object.)
- Plot-specific summarizing. (That is, functions that are based on data summaries in standard Spotfire S+ compute the required summaries from a big data object.)
- Preprocessing data, using `table`, `tapply`, `loess`, or `aggregate`.
- Preprocessing using `interp` or `hist2d`.

> **Note**
>
> The Windows GUI editable graphics do not support big data objects. To use these graphics, create a data frame containing either all of the data or a sample of the data.

For a more detailed discussion of graph functions available in the Big Data library, see Chapter 4, Creating Graphical Displays of Large Data Sets.

Modeling Functions

Algorithms for large data sets are available for the following statistical modeling types:

- Linear regression.
- Generalized linear regression.
- Clustering.
- Principal components.

See the section Models on page 12 for more information about the modeling objects.

If the data argument for a modeling function is a big data object, then Spotfire S+ calls the corresponding big data modeling function. The modeling function returns an object with the appropriate class, such as `bdLm`.

See Table A.12 in the Appendix for a list of the modeling functions that return a model object.

See Tables A.10 through A.13 in the Appendix for lists of the functions available for large data set modeling. See the Spotfire S+ Language Reference for more information about these functions.

Formula operators

The Big Data library supports using the formula operators+, `-`, `*`, `:`, `%in%`, and `/`.

Time Classes

The following classes support time operations in the Big Data library. See the Appendix for more information.

Table 1.6: *Time classes.*

Class name	Comment
`bdSignalSeries`	A `bdSignalSeries` object from positions and data
`bdTimeDate`	A `bdVector` class
`bdTimeSeries`	See the section Time Series Operations for more information.
`bdTimeSpan`	A `bdVector` class

Time Series Operations

Time series operations are available through the `bdTimeSeries` class and its related functions. The `bdTimeSeries` class supports the same methods as the standard S-PLUS library's `timeSeries` class. See the Spotfire S+ Language Reference for more information about these classes.

Time and Date Operations

- When you create a time object using `timeSeq`, and you set the `bigdata` argument to `TRUE`, then a `bdTimeDate` object is created.
- When you create a time object using `timeDate` or `timeCalendar`, and any of the arguments are big data objects, then a `bdTimeDate` object is created.

See Table A.14 in the Appendix.

> **Note**
>
> `bdTimeDate` always assumes the time as Greenwich Mean Time (GMT); however, Spotfire S+ stores no time zone with an object. You can convert to a time zone with `timeZoneConvert`, or specify the zone in the `bdTimeDate` constructor.

Time Conversion Operations

To convert time and date values, apply the standard S-PLUS time conversion operations to the `bdTimeDate` object, as listed in Table A.14 in the Appendix.

Matrix Operations

The Big Data library does not contain separate equivalents to `matrix` and `data.frame`.

S-PLUS matrix operations are available for `bdFrame` objects:

- matrix algebra (`+`, `-`, `/`, `*`, `!`, `&`, `|`, `>`, `<`, `==`, `!=`, `<=`, `=>`, `%%`, `%/%`)
- matrix multiplication (`%*%`)
- Crossproduct (`crossprod`)

In algebraic operations, the operators require the big data objects to have appropriately-corresponding dimensions. Rows or columns are not automatically replicated.

Basic algebra

You can perform addition, subtraction, multiplication, division, logical (`!`, `&`, and `|`), and comparison (`>`, `<`, `=`, `!=`, `<=`, `>=`) operations between:

- A scalar and a `bdFrame`.
- Two `bdFrames` of the same dimension.
- A `bdFrame` and a single-row `bdFrame` with the same number of columns.
- A `bdFrame` and a single-column `bdFrame` with the same number of rows.

The library also offers support for element-wise `+`, `-`, `*`, `/`, and matrix multiplication (`%*%`).

Matrix multiplication is available for two `bdFrames` with the appropriate dimensions.

Cross Product Function

When applied against two `bdFrames`, the cross product function, `crossprod`, returns a `bdFrame` that is the cross product of the given `bdFrames`. That is, it returns the matrix product of the transpose of the first `bdFrame` with the second.

Summary

In this section, we've provided an overview to the Big Data library architecture, including the new data types, classes, and functions that support managing large data sets. For more detailed information and lists of functions that are included in the Big Data library, see the Appendix: Big Data Library Functions.

In the next chapter, we provide examples for working with data sets using the types, classes, and functions described in this chapter.

CENSUS DATA EXAMPLE

2

INTRODUCTION

Census data provides a rich context for exploratory data analysis and the application of both unsupervised (e.g., clustering) and supervised (e.g., regression) statistical learning models. Furthermore the data sets (in their unaggragated state) are quite large. The US Census 2000 estimates the total US population at over 281 million people. In its raw form, the data set (which includes demographic variables such as age, gender, location, income and education) is huge. For this example, we focus on a subset of the US Census data that allows us to demonstrate principles of working with large data on a data set that we have included in the product.

Problem Description

Census data has many uses. One of interest to the US government and many commercial enterprises is geographical distribution of sub populations and their characteristics. In this initial example, we look for distinct geographical groups based on age, gender and housing information (data that is easy to obtain in a survey), and then characterize them by modeling the group structure as a function of much harder-to-obtain demographics such as income, education, race, and family structure.

Data Description

The data for this example is included with Spotfire S+ and is part of the US Census 2000 Summary File 3 (SF3). SF3 consists of 813 detailed tables of Census 2000 social, economic, and housing characteristics compiled from a sample of approximately 19 million housing units (about 1 in 6 households) that received the Census 2000 *long-form* questionnaire. The levels of aggregation for SF3 data is depicted in Figure 2.1.

The data for this example is the summary table aggregated by Zip Code Tabulation Areas (`ZCTA5`) depicted as the left-most branch of the schematic in Figure 2.1.

The following site provides download access to many additional SF3 summary tables:

```
http://www.census.gov/Press-Release/www/2002/sumfile3.html
```

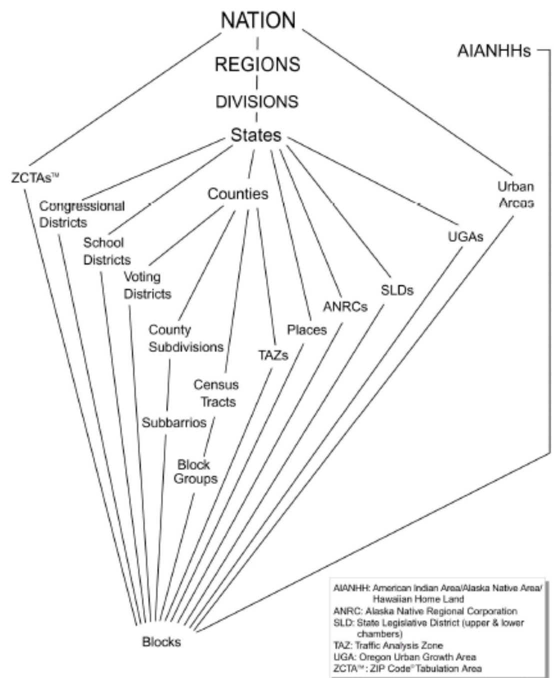

Figure 2.1: *US Census 2000 data grouping hierarchy schematic with implied aggregation levels. The data used in this example comes from the Zip Code Tabulation Area (ZCTA) depicted at the far left side of the schematic.*

The variables included in the census data set are listed in Table 2.1. They include the zip code, latitude and longitude for each zip code region, and population counts. Population counts include the total population for the region and a breakdown of the population by gender and age group: Counts of males and females for ages 0 - 5, 5 - 10, ..., 80 - 85, and 85 or older.

Table 2.1: *Variable descriptions for the census data example.*

Variable(s)	New Variable Name(s)	Description
`ZCAT5`	`zipcode`	five-number zip code
`INTPT.LAT`	`lat`	Interpolated latitude
`INTPT.LON`	`long`	Interpolated longitude
`P008001`	`popTotal`	Total population
`M.00 - M.85`	`male.00 - male.85`	Male population by age group: 0 - 4 years, 5 - 9 years, and so on.
`F.00 - F.85`	`female.00 - female.85`	Female population by age group: 0 - 4 years, 5 - 9 years, and so on.
`H007001`	`housingTotal`	Total housing units
`H007002`	`own`	Owner occupied
`H007003`	`rent`	Renter occupied

You can use the following command to find the script file that is included with Spotfire S+:

```
> system.file("samples/bigdata/census/censusDemo.ssc")
```

This script contains all the commands used in this chapter.

EXPLORATORY ANALYSIS

Data Import

The data is provided as a comma-separated text file (**.csv** format). The file is located in the **SHOME** location (by default your installation directory) in **/samples/bigdata/census/census.csv**.

Reading big data is identical to what you are familiar with in previous versions of Spotfire S+ with one exception: an additional argument to specify that the data object created is stored as a big data (`bd`) object.

```
> census <- importData(system.file
        ("samples/bigdata/census/census.csv"),
        stringsAsFactors=F, bigdata=T)
```

View the data with the **Data Viewer** as follows:

```
> bd.data.viewer(census)
```

The **Data Viewer** is an efficient interface to the data. It works on big out-of-memory data frames (such as `census`) and on in-memory data frames.

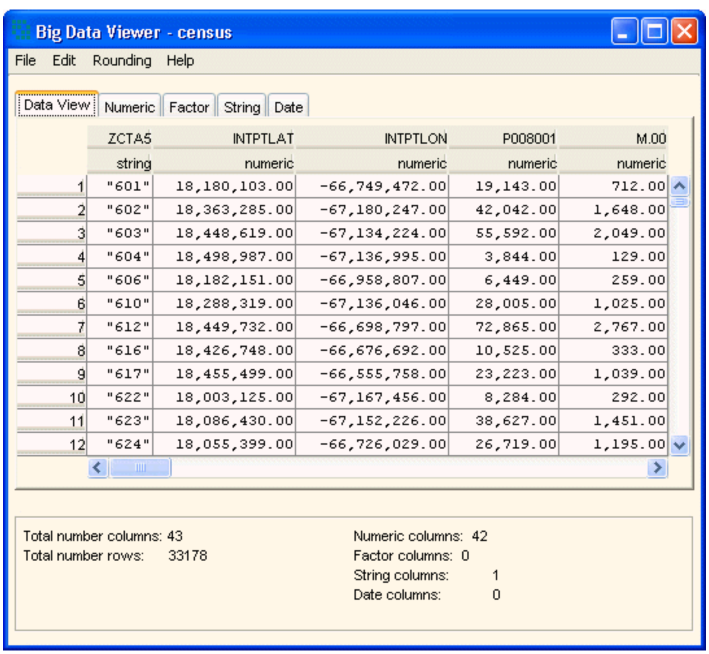

Figure 2.2: *Viewing big data objects is done with the Data Viewer.*

The **Data View** page (Figure 2.2) of the **Data Viewer** lists all rows and all variables in a scrollable window plus summary information at the bottom, including the number of rows, the number of columns, and a count of the number of different types of variables (for example, a numeric, factor). From the summary information, we see that `census` has 33,178 rows.

In addition to the **Data View** page, the **Data Viewer** contains tabs with summary information for numeric, factor, character, and date variables. These summary tabs provide quick access to minimums, maximums, means, standard deviations, and missing value counts for numeric variables and levels, level counts, and missing value counts for factor variables.

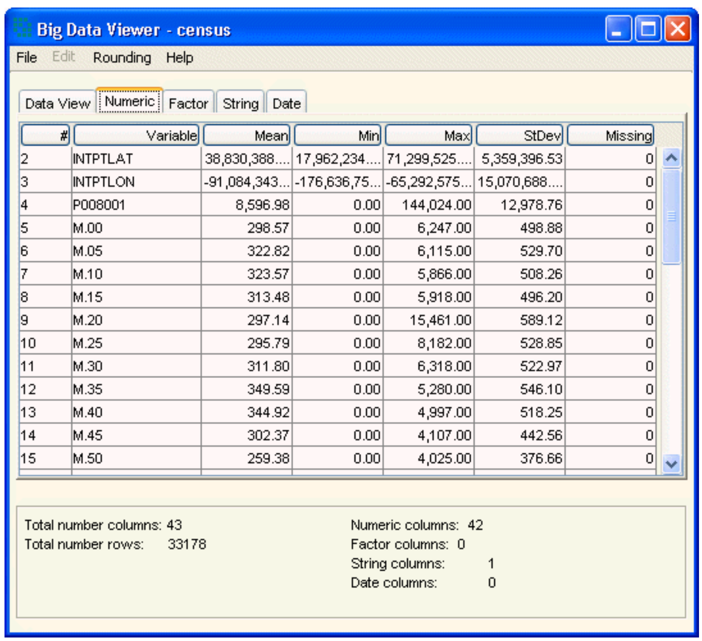

Figure 2.3: *The **Numeric** summary page of the **Data Viewer** provides quick access to minimum, maximum, mean, standard deviation, and missing value count for numeric data.*

Data Preparation

Before beginning any data preparation, start by making the names more intuitive using the `names` assignment expression:

```
> names(census) <- c("zipcode", "lat", "long", "popTotal",
```

```
paste("male", seq(0, 85, by = 5), sep = "."),
paste("female", seq(0, 85, by = 5), sep = "."),
"housingTotal", "own", "rent")
```

The row names are shown in Table 2.1, along with the original names.

Note
The S-PLUS expression `paste("male", seq(0, 85, by = 5), sep = ".")` creates a sequence of 18 variable names starting with `male.0` and ending with `male.85`. The call to `seq` generates a sequence of integers from 0 to 85 incremented by 5, and the call to `paste` pastes together the string "`male`" with the sequence of integers separated with a period (.).

A summary of the data now is:

```
> summary(census)
    zipcode             lat               long
 Length:    33178   Min.:17962234    Min.:-176636755
  Class:            Mean:38830389    Mean: -91084343
   Mode:character   Max.:71299525    Max.: -65292575

  popTotal            male.0            male.5
 Min.:      0.000   Min.:    0.0000   Min.:    0.000
 Mean:   8596.977   Mean: 298.5727    Mean: 322.822
 Max.:144024.000    Max.:6247.0000    Max.:6115.000
  .
  .
  .
```

From summary of the census data, you might notice a couple of problems:

1. The population total (`popTotal`) has some zero values, implying that some zip codes regions contain no population.
2. The zip codes are stored as character strings which is odd because they are defined as five-digit numbers.

To remove the zero-population zip codes you can use the syntax that you would typically use when working with data frames:

```
> census <- census[census[, "popTotal"] > 0, ]
```

However, there is a more efficient way. Notice that the example above (finding rows with non-zero population counts) implies two passes through the data. The first pass extracts the `popTotal` column and compares it (row by row) with the value of zero. The second pass removes the bad `popTotal` rows. If your data is very large, using subscripting and nested function calls can result in a prohibitively lengthy execution time.

A more efficient "big data" way to remove rows with no population is to use the `bd.filter.rows` function available in the Big Data library in S-PLUS. `bd.filter.rows` has two required arguments:

1. `data`: the big data object to be filtered.
2. `expr`: an expression to evaluate. By default, the expression must be valid, based on the rules of the row-oriented Expression Language. For more details on the expression language, see the help file for `ExpressionLanguage`.

Note

If you are familiar with the S-PLUS language, the Excel formula language, or another programming language, you will find the row-oriented Expression Language natural and easy to use. An expression is a combination of constants, operators, function calls, and references to columns that returns a single value when evaluated

For our example, the expression is simply `popTotal > 0`, which you pass as a character string to `bd.filter.rows`. The more efficient way to filter the rows is:

```
> census <- bd.filter.rows(census, expr= "popTotal >  0")
```

Using the row-oriented Expression Language with `bd.filter.rows` results in only one pass through the data, so the computation time will usually be reduced to about half the execution time of the previously-described S-PLUS expression. Table 2.2 displays additional examples of row-oriented expressions.

Table 2.2: *Some examples of the row-oriented Expression Language.*

Expression	Description
`age > 40 & gender == “F”`	All rows with females greater than 40 years of age.
`Test != “Failed”`	All rows where `Test` is not equal to `“Failed”`.
`Date > 6/30/04`	All rows with `Date` later than 6/30/04.
`voter == “Dem” | voter == “Ind”`	All rows where `voter` is either democrat or independent.

Now, remove the cases with bad zip codes by using the regular expression function, `regexpr`, to find the row indices of zip codes that have only numeric characters:

```
> census <- bd.filter.rows(census,
                "regexpr('^[0-9]+$', zipcode)>0",
                 row.language=F)
```

Notes

- The call to the `regexpr` function finds all zip codes that have only integer characters in them. The regular expression “`^[0-9]+$`” produces a search for strings that contain only the characters `0, 1, 2, ..., 9`. The `^` character indicates starting at the beginning of the string, the `$` character indicates continuing to the end of the string and the + symbol implies any number of characters from the set {0, 1, 2,..., 9}.
- The call to `bd.filter.rows` specified the optional argument, `row.language=F`. This argument produces the effect of using the standard S-PLUS expression language, rather than the row-oriented Expression Language designed for row operations on big data.

Tabular Summaries

Generate the basic tabular summary of variables in the census data set with a call to the `summary` function, the same as for in-memory data frames. The call to `summary` is quite fast, even for very large data sets, because the summary information is computed and stored internally at the time the object is created.

```
> summary(census)
    zipcode              lat              long
 Length:    32165   Min.:17964529   Min.:-176636755
  Class:            Mean:38847016   Mean: -91103295
   Mode:character   Max.:71299525   Max.: -65292575

  popTotal            male.0            male.5
 Min.:      1.000   Min.:    0.0000   Min.:    0.0000
 Mean:   8867.729   Mean:  307.9759   Mean:  332.9889
 Max.:144024.000    Max.:6247.0000    Max.:6115.0000
  .
  .
  .
 female.85          housingTotal          own
 Min.:    0.00000   Min.:     0.000    Min.:     0.000
 Mean:   92.77398   Mean:  3318.558    Mean:  2199.168
 Max.:2906.00000    Max.:61541.000     Max.:35446.000

    rent
 Min.:     0.000
 Mean:  1119.391
 Max.:40424.000
```

To check the class of objects contained in a big data data frame (class `bdFrame`), call `sapply`, which applies a specified function to all the columns of the `bdFrame`.

```
> sapply(census, class)
      zipcode         lat         long     popTotal
 "bdCharacter" "bdNumeric" "bdNumeric" "bdNumeric"

     male.0       male.5      male.10      male.15
 "bdNumeric" "bdNumeric" "bdNumeric" "bdNumeric"
  .
  .
  .
```

Generate age distribution tables with the same operations you use for in-memory data. Multiply column means by 100 to convert to a percentage scale and round the output to one significant digit:

```
> ageDist <-
     colMeans(census[, 5:40] / census[, "popTotal"]) * 100
> round(matrix(ageDist,
               nrow = 2,
               byrow = T,
               dimnames = list(c("Male", "Female"),
                               seq(0, 85, by=5))), 1)
numeric matrix: 2 rows, 18 columns.
         0   5  10  15  20  25  30  35  40  45  50  55
  Male 3.2 3.6 3.8 3.8 2.9 2.9 3.2 3.9 4.1 3.8 3.3 2.7
Female 3.0 3.4 3.6 3.4 2.7 2.8 3.2 3.9 4.0 3.7 3.3 2.7

        60  65  70  75  80  85
  Male 2.3 2.0 1.7 1.3 0.8 0.5
Female 2.3 2.1 2.0 1.7 1.2 1.1
```

Graphics

You can plot the columns of a `bdFrame` in the same manner as you do for regular (in-memory) data frames:

```
> hist(census$popTotal)
```

will produce a histogram of total population counts for all zip codes. Figure 2.4 displays the result.

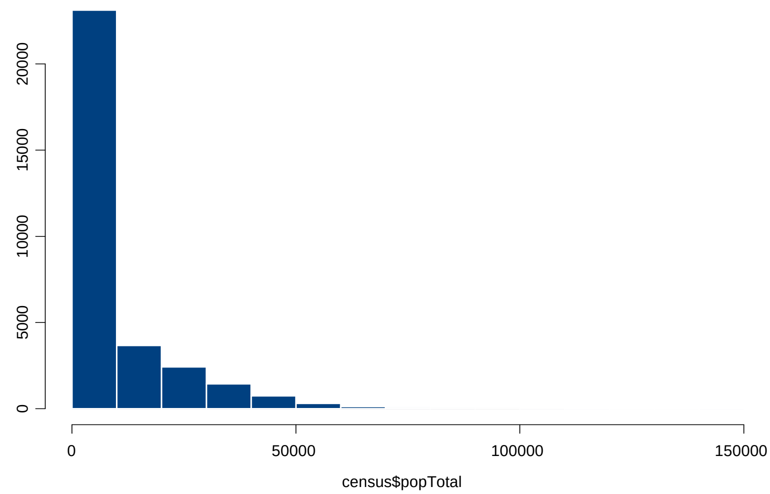

Figure 2.4: *Histogram of total population counts for all zip codes.*

You can get fancier. In fact, in general, the Trellis graphics in Spotfire S+ work on big data. For example, the median number of rental units over all zip codes is 193:

```
> median(census$rent)
[1] 193
```

You would expect that, if the number of rental units is high (typical of cities), the population would likewise be high. We can check this expectation with a simple Trellis boxplot:

```
> bwplot(rent > 193 ~ log(popTotal), data = census)
```

Figure 2.5 displays the resulting graph.

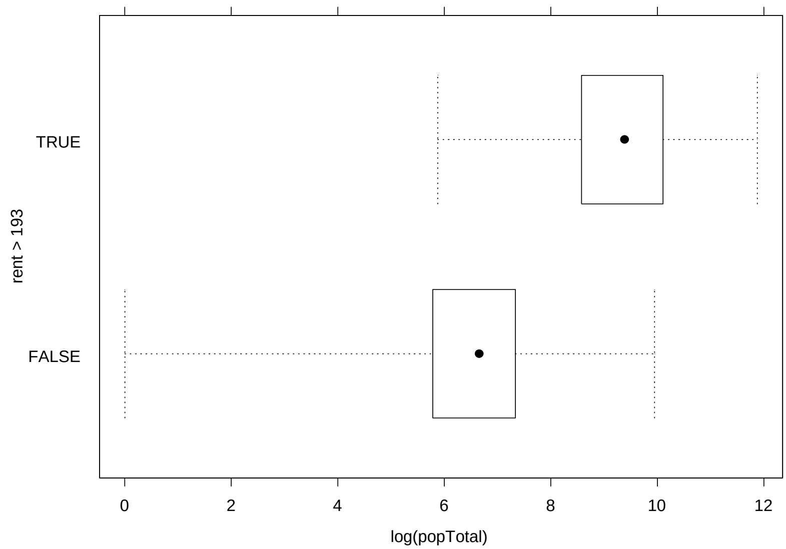

Figure 2.5: *Boxplots of the log of popTotal for the number of rental units above and below the median, showing higher populations in areas with more rental units.*

You can address the question of population size relative to the number of rental units in a more general way by examining a scatterplot of `popTotal` vs. `rent`. Call the Trellis function `xyplot` for this. Take logs (after adding 0.5 to eliminate zeros) of each of the variables to rescale the data so the relationship is more exposed:

```
> xyplot(log(popTotal) ~ log(rent + 0.5), data = census)
```

The resulting plot is displayed in Figure 2.6.

Note

The default scatterplot for big data is a *hexbin* scatterplot. The color shading of the hexagonal “points” indicate the number of observations in that region of the graph. For the darkest shaded hexagon in the center of the graph, over 800 zip codes are represented, as indicated by the legend on the right side of the graph.

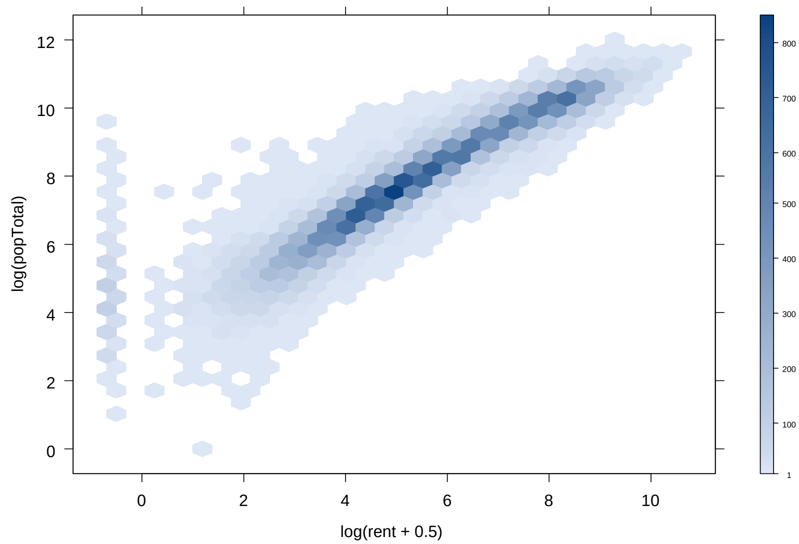

Figure 2.6: *This hexbin scatterplot of `log(popTotal)` vs. `log(rent+0.5)` shows population sizes increasing with the increasing number of rental units.*

The result displayed in Figure 2.6 is not surprising; however, it demonstrates the straightforward use of known functions on big data objects. This example continues with Trellis graphics with conditioning in the following sections.

The age distribution table created in the section Tabular Summaries on page 30 produces the plot shown in Figure 2.7:

```
> bars <- barplot(rbind(ageDist[1:18], -ageDist [19:36]),
                horiz=T)
> mtext(c("Female", "Male"), side = 1, line = 3, cex = 1.5,
         at = c(-2, 2))
> axis(2, at = bars, labels = seq(0, 85, by = 5),
        ticks =F)
```

Note

In creating this plot, the example starts with big out-of-memory data (`census`) and ends with small in-memory summary data (`ageDist`) without having to do anything special to transition between the two. Spotfire S+ takes care of the data management.

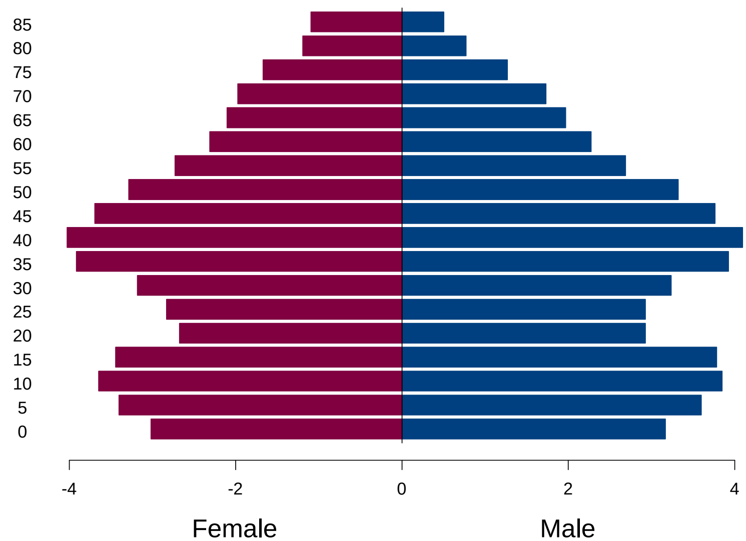

Figure 2.7: *Age distribution by gender estimated by US Census 2000.*

DATA MANIPULATION

The `census` data contains raw population counts by gender and age; however, the counts for different genders and ages are in different columns. To compare them more easily, stack the columns end to end and create factors for gender and age. Start with the stacking operation.

Stacking

The `bd.stack` function provides the needed stacking operation. Stack all the population counts for males and females for all ages with one call to `bd.stack`:

```
> censusStack <- bd.stack(census,
                     columns = 5:40,
                     replicate = c(1:4, 41:43),
                     stack.column.name = "pop",
                     group.column.name = "sexAge")
```

Table 2.3 lists the arguments to `bd.stack`.

Table 2.3: *Arguments to* `bd.stack`.

Argument Name	Description
`data`	Input data set, a `bdFrame` or `data.frame`.
`columns`	Names or numbers of columns to be stacked.
`replicate`	Names or numbers of columns to be replicated.
`stack.column.name`	Name of new stacked column.
`group.column.name`	Name of an additional group column to be created in the output data set. In each output row, the group column contains the name of the original column that contained the data value in the new stacked column.

The first few rows of the resulting data are listed below. Notice the values for the `sexAge` variable are the names of the columns that were stacked.

```
> censusStack
** bdFrame: 1150236 rows, 9 columns **
  zipcode      lat      long popTotal housingTotal   own rent
1     601 18180103 -66749472    19143         5895  4232 1663
2     602 18363285 -67180247    42042        13520 10903 2617
3     603 18448619 -67134224    55592        19182 12631 6551
4     604 18498987 -67136995     3844         1089   719  370
5     606 18182151 -66958807     6449         2013  1463  550

   pop sexAge
1  712 male.0
2 1648 male.0
3 2049 male.0
4  129 male.0
5  259 male.0
   ... 1150231 more rows ...
```

Notice that the census data started with a little over 33,000 rows. Now, after stacking, there are over 1.15 million rows.

Variable Creation

Now create the `sex` and `age` factors. There are several ways to do this, but the most computationally efficient way for large data is to use the `bd.create.columns` function, along with the row-oriented expression language. Before starting, notice that the column names for the stacked columns (`male.0`, `male.5`, ..., `female.80`, `female.85`) can be separated into male and female groups simply by the number of characters in their names. All male names have seven or fewer characters and all female names have eight or more characters. Therefore, by checking the number of characters in the string, you can determine whether the value should be “`male`” or “`female`”. Here is an example of the row-oriented Expression Language:

```
" ifelse(nchar(sexAge) > 7, 'female', 'male' "
```

Notice the use of a single quote, ‘, to embed a quote within a quote.

To create the `age` variable is a little harder. You must subset the string differently, depending on whether the value of `sexAge` corresponds to a male or female.

1. For males, extract from the sixth character to the end, and for females, extract from the eighth character to the end. The row-oriented expression language follows:

```
" ifelse(nchar(sexAge) > 7,
   substring(sexAge, 8, nchar(sexAge)),
   substring(sexAge, 6, nchar(sexAge))) "
```

2. Create an additional variable that is a measure of the population size for each age and gender group relative to the population size for the entire zip code area. Because each row contains gender and age specific population estimates *and* the total population estimate for that zip code area, the relative population size for each gender and age group is simply

   ```
   "pop/popTotal"
   ```

3. Create all three new variables in a *single* call to `bd.create.columns` (which requires only a single pass through the data) by including all three of the above expressions in the call.

```
> censusStack <- bd.create.columns(censusStack,
    exprs = c("ifelse(nchar(sexAge) > 7, 'female', 'male')",
            "ifelse(nchar(sexAge) > 7,
                  substring(sexAge, 8, nchar(sexAge)),
                  substring(sexAge, 6, nchar(sexAge)))" ,
            "pop/popTotal"),
    names. = c("sex", "age", "popProp"),
    types = c("factor", "character", "numeric"))
```

In this example, `bd.create.columns` arguments include the following:

- `exprs` takes a character vector of strings; each string is the expression that creates a different column.
- `names` supplies the names for the newly-created columns.
- `types` specifies the type of data in the resulting column.

For more information on `bd.create.columns`, see its help file by typing `help(bd.create.columns)`, or by typing `?bd.create.columns` in Spotfire S+.

Note

The `age` column in the call to `bd.create.columns` is stored as a character column so we have more control when creating an `age` factor. A discussion of this is included in the next section Factors.

Factors

In the previous section, we created `age` as a character vector, because when `bd.create.columns` creates factors, it establishes levels as the set of *alphabetically* sorted unique values in the column. The levels are not arranged numerically. In the example output below, notice the placement of the "`5`" between "`45`" and "`50`".

```
> ageFactor <- bdFactor(censusStack[,"age"])
> levels(ageFactor)
 [1] "0"  "10" "15" "20" "25" "30" "35" "40" "45" "5"  "50"
[12] "55" "60" "65" "70" "75" "80" "85"
```

When Spotfire S+ creates tables or graphics that use the levels as labels, the order is as the levels are listed, rather than in numerical order.

To control the order of the levels of a factor, call the `bdFactor` function directly and state explicitly the order for the levels. For example, using the census data:

```
> censusStack[, "age"] <- bdFactor(censusStack[, "age"],
          levels = c("0", "5", "10", "15", "20", "25",
                     "30", "35", "40", "45", "50", "55",
                     "60", "65", "70", "75", "80", "85"))
```

MORE GRAPHICS

The data is now prepared to allow more interesting graphics. For example, create an age distribution plot conditional on gender (Figure 2.8) with the following call to `bwplot`, a Trellis graphic function:

```
> bwplot(age ~ log(popProp + 0.00001) | sex,
         data = censusStack)
```

Note

0.00001 is added to the population proportions to avoid taking the log of zero.

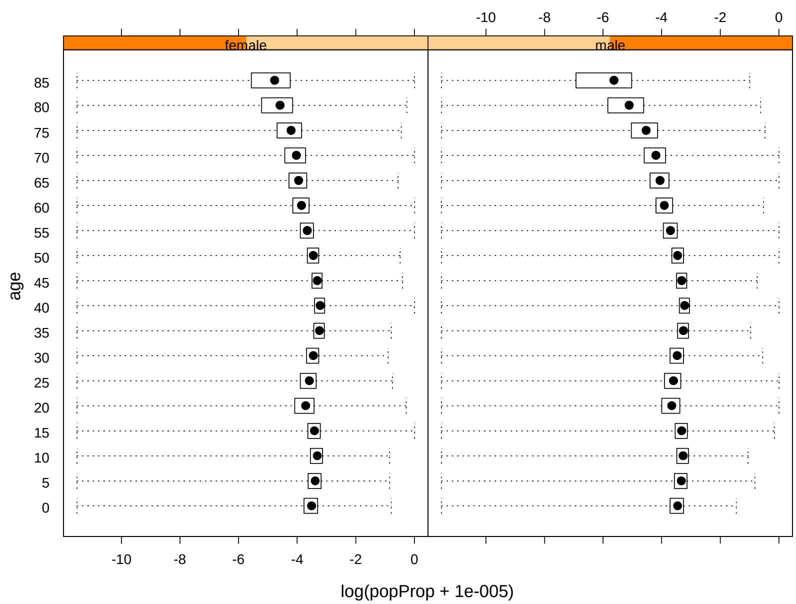

Figure 2.8: *Boxplots of logged relative population numbers by* `age` *and* `sex`*.*

The following call to `bwplot` creates a plot (Figure 2.9) of logged relative population numbers by `age` and whether the zip code area contains more than the median number of rental units:

```
> bwplot(age ~ log(popProp + 0.00001) | rent > 193,
         data = censusStack)
```

Note the span of the boxes for 80 and older when there are fewer than the median number of rental units, implying that the population numbers for this group drops dramatically in some areas where there few rental units.

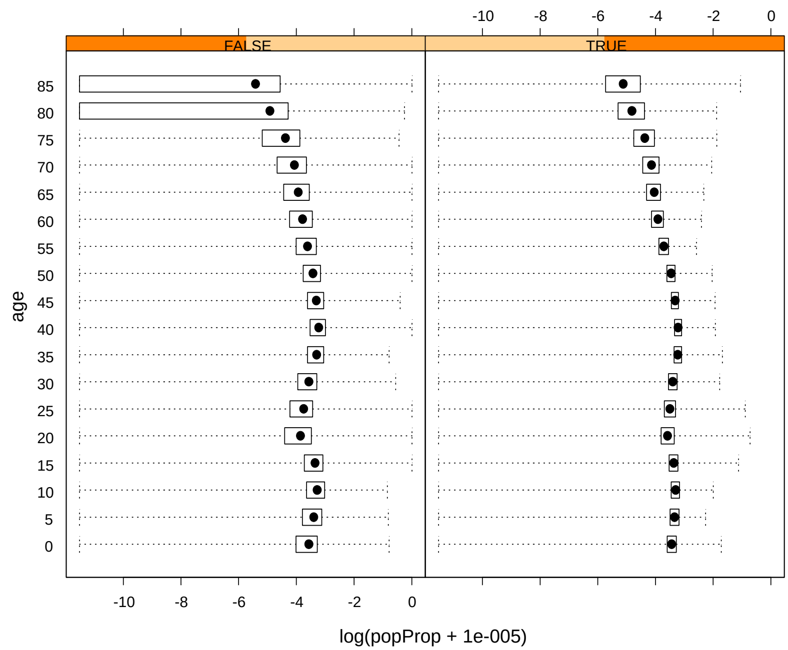

Figure 2.9: *Boxplots of logged relative population numbers by* `age` *and* `rent>193`*.*

Another interesting plot is of the zip code area centers in units of latitude and longitude. Highly populated areas show a higher density of zip code numbers; therefore, they show greater density in the hexbin scatterplot. First, however, notice that the scale of `lat` and `long` is off by a factor of 1,000,000. The `lat` variable should be in the range of 20 to 70 and `long` should be in the range of -60 to -180. So first rescale these variables by a call to `bd.create.columns`.

```
> summary(census[, c("lat", "long")])
     lat                long
  Min.:17964529    Min.:-176636755
  Mean:38851462    Mean: -91044543
  Max.:71299525    Max.: -65292575
```

Even more efficient, requiring no passes through the data:

```
> summary(census)[, c("lat", "long")]
```

Because the summary is stored in metadata, it does not have to be computed. The first form creates a two-column big data object, and then gets the summary from that object.

To rescale `lat` and `long` simultaneously, use the following expressions:

```
"lat/1e6", "long/1e6"
```

Use the original data set `census`, rather than `censusStack`, because `census` has just one row per zip code.

```
> census <- bd.create.columns(census,
        exprs=c("lat/1.e6", "long/1.e6"),
        names=c("lat","long"))
```

The values of `lat` and `long` are now scaled appropriately:

```
> summary(census[, c("lat", "long")])
     lat                long
 Min.:17.96453    Min.:-176.63675
 Mean:38.85146    Mean: -91.04454
 Max.:71.29953    Max.: -65.29257
```

Or, more efficiently:

```
> summary(census)[, c("lat", "long")]
```

Now produce the plot with a simple call to `xyplot`.

```
> xyplot(lat ~ long, data = census)
```

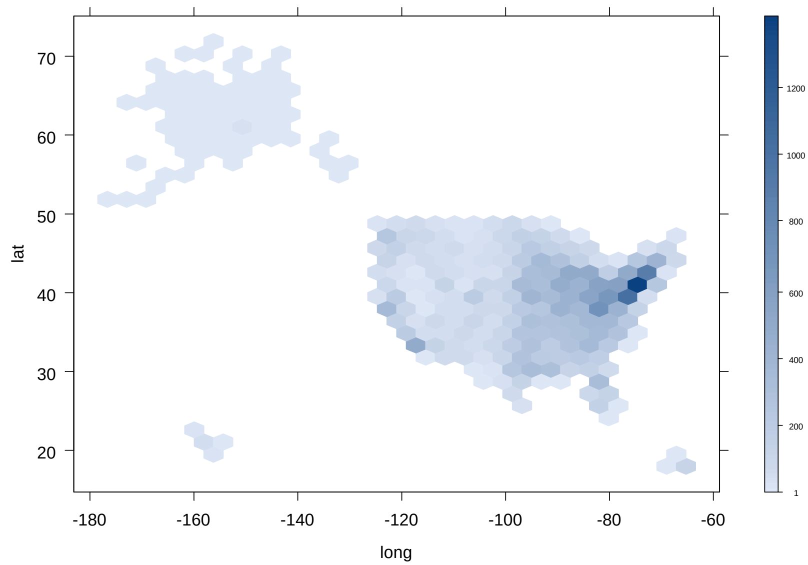

Figure 2.10: *Hexbin scatterplot of latitudes and longitudes. Zip codes are denser where populations are denser, so this plot displays relative population densities.*

CLUSTERING

This section applies clustering techniques to the census data to find sub populations (collections of zip code areas) with similar age distributions. The section Modeling Group Membership develops models that characterize the subgroups we find by clustering.

Data Preparation

The section Tabular Summaries computed the *average* age distribution across all zip code areas by age and gender, depicted in Figure 2.7. Next, group zip-code areas by age distribution characteristics, paying close attention to those that deviate from the national average. For example, age distributions in areas with military bases, typically dominated by young adult single males without children, should stand out from the national average.

Unusual populations are most noticeable if the population proportions (previously computed as `pop/popTotal` by age and gender) are normalized by the national average. One way to normalize is to divide population proportions in each age and gender group by the national average for each age and gender group. The (odds) ratio represents how similar (or dissimilar) a zip-code population is from the national average. For example, a ratio of 2 for females 85 years or older indicates that the proportion of women 85 and older is twice that of the national average.

To prepare the population proportions, recall that the national averages are produced with the `colMeans` function:

```
> ageDist <-
    colMeans(census[, 5:40] / census[, "popTotal"])
```

Also recall that, in Spotfire S+, if you multiply (or divide) a matrix by a vector, the elements of each column are multiplied by the corresponding element of the vector (assuming the length of the vector is equivalent to the number of rows of the matrix). We want to divide each element of a column by the mean of that column. In-memory computation might proceed as follows:

```
popPropN <- t(t(census[, 5:40]) / ageDist)
```

That is, transpose the data matrix, divide by a vector as long as each column of the transposed matrix, and then transpose the matrix back.

The above operation is inefficient for large data. It requires multiple passes through the data. A more efficient way to compute the normalized population proportions is to create a series of row-oriented expressions:

```
"male.0/ageDist[1]"
```

and process them with `bd.create.columns`.

Here is how to do this:

1. Create the proportions matrix:

```
> popProp <- census[, 5:40] / census[, "popTotal"]
```

2. Create the expression vector:

```
> norm.exprs <- paste(names(popProp),
      paste("/ageDist[", 1:36, "]",sep=""), sep="")
```

3. Normalize the population proportions:

```
> popPropN <- bd.create.columns(popProp,
                    exprs = norm.exprs,
                    names. = names(popProp),
                    row.language = F)
```

4. Join the normalized population proportions with the rest of the census data:

```
> censusN <- bd.join(list(census[, c(1:4, 41:43)],
                    popPropN))
```

Notes

- In step 3, `row.language = F` is specified because the expressions use Spotfire S+ syntax to do subscripting.
- In step 4, there are no key variables specified in the join operation, which results in a join by row number.

K-Means Clustering

You are now ready to do the clustering. The big data version of k-means clustering is `bdCluster`. The important arguments are:

- The `data` (a `bdFrame` in this example).
- The `columns` to cluster (if all columns of the `bdFrame` are not included in the clustering operation).

- The number of clusters, `k`.

Typically, determining a reasonable value for `k` requires some effort. Usually, this involves clustering repeatedly for a sequence of `k` values and choosing the `k` that gives interpretable results. For this example, after a little experimentation, we set `k = 40`.

```
> set.seed(22)
```

```
> clusterCensusN <- bdCluster(censusN,
        columns=names(popPropN),k=40)
```

Notes

To match the results presented here, we set the random seed to 22 by typing `set.seed(22)` at the prompt before calling `bdCluster`.

This example focuses on only the age x gender distributions, so `columns` is set to just those columns with population counts.

The `bdCluster` function has a `predict` method, so you can extract group membership identifiers for each observation and append them onto the normalized data, as follows:

```
> censusNPred <- cbind(censusN, predict(clusterCensusN))
```

Analyzing the Results

In this section, examine the results of applying k-means clustering to the census data. To get a sense of how big the clusters are and what they look like, start by combining cluster means and counts.

1. To compute cluster means, call `bd.aggregate` as follows:

```
> clusterMeans <- bd.aggregate(censusNPred,
                     columns = names(popProp),
                     by.columns="PREDICT.membership",
                     methods="mean")
```

2. To compute cluster group sizes, call `bd.aggregate` again with “`count`” as the method:

```
> clusterCounts <- bd.aggregate(censusNPred,
                      columns=1,
                      by.columns="PREDICT.membership",
                      methods="count")
```

3. Coerce `clusterMeans` and `clusterCounts` from a `bdFrame` to a `data.frame`:

```
> clusterMeans <- bd.coerce(clusterMeans)
> clusterCounts <- bd.coerce(clusterCounts)
```

4. Merge the two aggregates:

```
> clusterMeansCounts <- merge(clusterCounts, clusterMeans)
```

The call to `merge` without a `key.variables` argument matches on the common columns names, by default.

5. Set the order of `clusterMeansCounts` by the number of members within each cluster:

```
> clusterMeansCounts <- bd.sort(clusterMeansCounts,
                           by = "zipcode.count",
                           ascending = F)
```

The `clusterMeansCounts` object contains mean population estimates for each zip code area, age and gender. The first 24 groups (ordered by the number of zip code regions that comprise them) are plotted in Figure 2.11. The upper left panel corresponds to the group with the most zip codes and the lower right panel has the fewest. The graphs that appear top-heavy reflect more older people. Notice the panel in the third row down, first position on the left. It is very heavily weighted on the top. These are retirement communities. Also, notice the second panel from the left in the bottom row. The population is dominated by young adult males. These are primarily military bases.

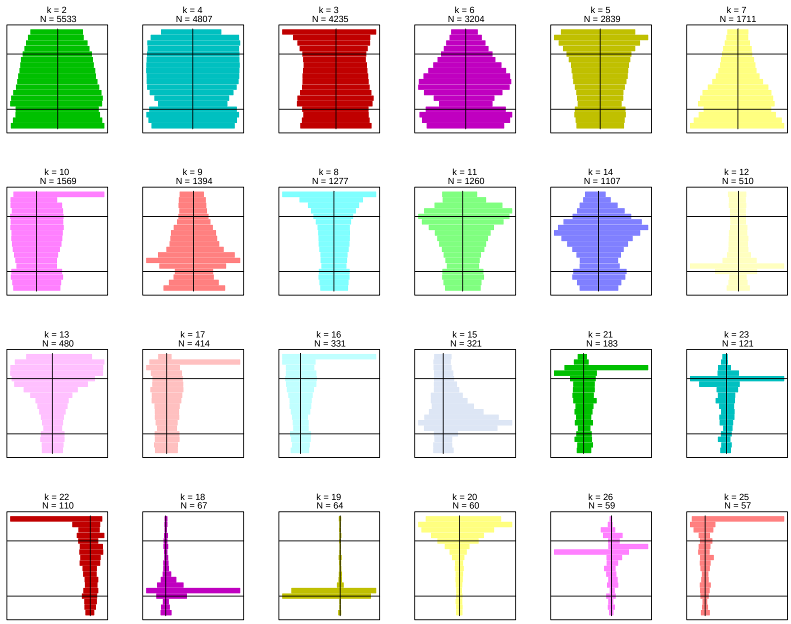

Figure 2.11: *Age distribution barplots for the first 24 groups resulting from k-means clustering with 40 groups specified. The horizontal lines in each panel correspond to 20 (the lower one) and 70 years of age. Females are to the left of the vertical and males are to the right.*

To produce Figure 2.11, run the following:

```
> source(system.file(
      "samples/bigdata/census/my.vbar.q"))
> index16 <- rep(1:16, length = 24)
> par(mfrow=c(4,6))
> for(k in 1:24) {
   my.vbar(bd.coerce(clusterMeansCounts), k=k,
              plotcols=3:38,
              Nreport.col=2,
              col=1+index16[k])
              }
```

An interesting graphic that dramatizes group membership displays each zip code as a single black point for the center of the zip code region, and then overlays points for any given cluster group in another color. Technically, this plot is more interesting, because it uses a new function, `bd.block.apply`, to process the data a block at a time.

The `bd.block.apply` function takes two primary arguments:

- The data, usually a `bdFrame`, `census` in this case.
- a function for processing the data a block at a time.

Note

The `bd.block.apply` argument `FUN` is a S-PLUS function called to process a data frame. This function itself cannot perform big data operations, or an error is generated. (This is true for `bd.by.group` and `bd.by.window`, as well.)

Define the block processing function as follows:

```
f <- function(SP){
        par(plt = c(.1, 1, .1, 1))
        if(SP$in1.pos == 1){
            plot(SP$in1[,"long"], SP$in1[, "lat"],
                 pch = 1, cex = 0.15,
                 xlim=c(-125,-70), ylim=c(25, 50),
                 xlab="", ylab="", axes = F)
            axis(1, cex = 0.5)
            axis(2, cex = 0.5)
           title(xlab = "Longitude", ylab = "Latitude")
        } else {
           points(SP$in1[, "long"],
           SP$in1[, "lat"], cex = 0.2)
    }
}
```

This function processes a list object, which contains one block of the `census` `bdFrame`. `SP$in1` corresponds to the data, and `SP$in1.pos` corresponds to the starting row position of each block of the `bdFrame` that is passed to the function. The test `if(SP$in1.pos == 1)` checks if the first block is being processed. If the first block is processed, a call to `plot` is made; if the first block is not processed, a call to `points` is made. The call to `bd.block.apply` is:

```
bd.block.apply(census, FUN = f)
```

This call makes this new graph select only those rows that belong to the cluster group of interest, and then coerce it to a data frame to demonstrate the simplicity of using both `bdFrame` and a `data.frame` objects in the same function. Start by keeping only those variables that are useful for displaying the cluster group locations.

```
> censusNPsub <- bd.filter.columns(censusNPred,
                 keep = c("lat","long","PREDICT.membership"))
```

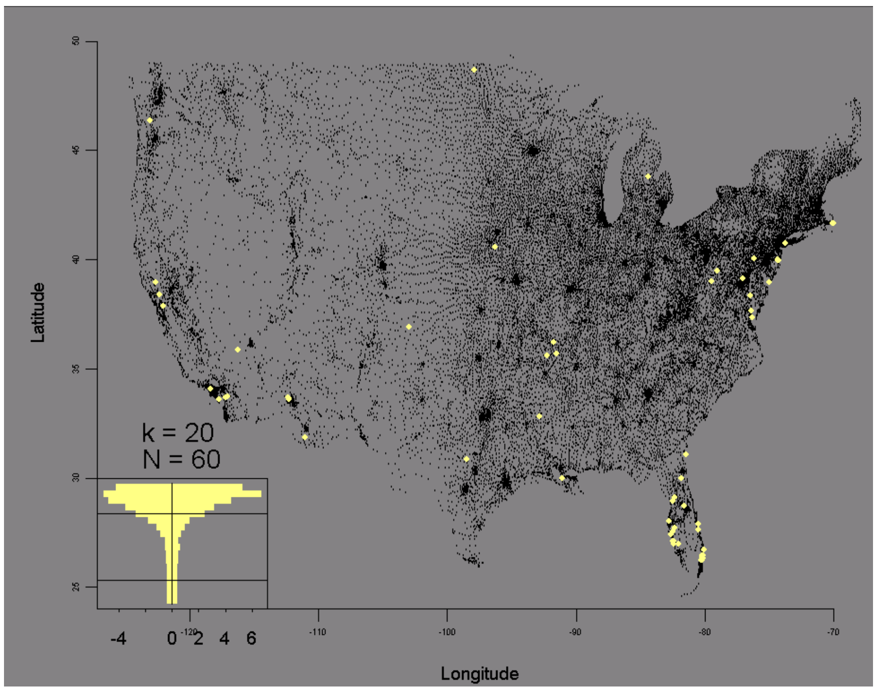

Figure 2.12: *Plot of all zip code region centers with cluster group 20 overlaid in another color. The double histogram in the bottom left corner displays the age distributions for females to the left and males to the right for cluster group 20. The horizontal lines in the histogram are at 20 and 70 years of age.*

To generate graphs for the first 22 cluster groups, it is slightly more work:

```
> dev.start()
> par(err=-1)
> pred <- clusterMeansCounts[, "PREDICT.membership"]
```

```
> for(k in 1:22) {
  setk <- bd.coerce(bd.filter.rows(censusNPsub,
                expr = "PREDICT.membership == pred[k]",
                columns = c("lat", "long"),
                row.language = F))
        par(plt=c(.1, 1, .1, 1))
        bd.block.apply(census, FUN = f)
        points(setk[, "long"], setk[, "lat"],
               col=1+index16[k],
                cex=0.6, pch=16)
        par(new=T)
        par(plt=c(.1, .3, .1, .3))
        my.vbar(clusterMeansCounts, k=k, plotcols=3:38,
              Nreport.col=2, col=1+index16[k])
        box()
}
```

Notes

1. `setk` is created as a regular data frame using `bd.coerce`, assuming that once a given cluster group is selected the data is small enough to process it entirely in memory.
2. `bd.block.apply` is used to plot all the zip code region centers, which requires processing the entire `bdFrame`.
3. `setk` contains the latitude and longitude locations for zip code centers for the selected group, `pred[k]`
4. `setk` was created to demonstrate the use of both `bdFrame` objects and `data.frame` objects in a single function. Placing the cluster group points on the graph could also be accomplished in the function passed to `bd.block.apply`.

MODELING GROUP MEMBERSHIP

The age distributions in Figure 2.11 are intriguing, but we know little about why the ages are distributed the way they are. Except for obvious deductions like retirement communities and military bases, we do not have much more information in the current data set. Another data set, `censusDemogr`, provides additional demographics variables such as household income, education and marital status.

The `censusDemogr` data can be loaded with

```
> data.restore(system.file(
              "samples/bigdata/census/censusDemogr.sdd"))
```

By modeling group membership as a function of an assortment of explanatory variables, we can characterize the groups relative to those variables. The data in `censusDemogr` contains the variables listed in Table 2.4. Note that all the variables except `housingTotal` and the cluster group variables at the end contain the proportion of households (hh) with the characteristic stated in the description column.

Table 2.4: *Variables contained in* `censusDemogr`*, a* `bdFrame` *object. All variables, except* `housingTotal`*, contain the proportion of households (hh) in the zip code area with the stated characteristic.*

Variable	Description
`housingTotal`	Total number of housing units.
`own`	Own residence.
`onePlusPersonHouse`	Two or more family members in hh.
`nonFamily`	Two or more non-family members in hh.
`Plus65InHouse`	65 or older in family hh.
`Plus65InNonFamily`	65 or older in non-family hh.
`Plus65InGroup`	65 or older in group quarters.

Table 2.4: *Variables contained in* `censusDemogr`*, a* `bdFrame` *object. All variables, except* `housingTotal`*, contain the proportion of households (hh) in the zip code area with the stated characteristic.*

Variable	Description
`marriedChildren`	Married-couple families with children.
`marriedNoChildren`	Married-couple families without children.
`maleChildren`	Male householder with children.
`maleNoChildren`	Male householder without children.
`femaleChildren`	Female householder with children.
`femaleNoChildren`	Female householder without children.
`maleSingle`	Single male.
`femaleSingle`	Single female.
`maleMarried`	Married male.
`femaleMarried`	Married female.
`maleWidow`	Male widower.
`femaleWidow`	Female widow.
`maleDiv`	Male divorced.
`femaleDiv`	Female divorced.
`english5to17`	5 - 17 year olds speak only English.
`english18to65`	18 - 65 year olds speak only English.
`englishOver65`	Over 65 year olds speak only English.

Table 2.4: *Variables contained in* `censusDemogr`, *a* `bdFrame` *object. All variables, except* `housingTotal`, *contain the proportion of households (hh) in the zip code area with the stated characteristic.*

Variable	**Description**
`native`	Born in US.
`entryToUS95to00`	Entry to US from 1995 to 2000.
`entryToUS90to94`	Entry to US from 1990 to 1994.
`entryToUS85to89`	Entry to US from 1985 to 1989.
`entryToUS80to84`	Entry to US from 1980 to 1984.
`entryToUS75to79`	Entry to US from 1975 to 1979.
`entryToUS70to74`	Entry to US from 1970 to 1974.
`entryToUS65to69`	Entry to US from 1965 to 1969.
`entryToUSBefore65`	Entry to US before 1965.
`changedHouseSince95`	Changed residence since 1995.
`maleLoEd`	Male head of household with low education.
`femaleLoEd`	Female head of hh with low education.
`maleHS`	Male head of hh with HS education.
`femaleHS`	Female head of hh with HS education.
`maleCollege`	Male head of hh with college education.
`femaleCollege`	Female head of hh with college education.
`maleBA`	Male head of hh with bachelor's degree.

Table 2.4: *Variables contained in* `censusDemogr`*, a* `bdFrame` *object. All variables, except* `housingTotal`*, contain the proportion of households (hh) in the zip code area with the stated characteristic.*

Variable	Description
`femaleBA`	Female head of hh with bachelor's degree.
`maleAdvDeg`	Male head of hh with advanced degree.
`femaleAdvDeg`	Female head of hh with advanced degree.
`maleWorked99`	Male head of hh worked in 1999.
`femaleWorked99`	Female head of hh worked in 1999.
`maleBlueCollar`	Male head of hh blue-collar worker.
`femaleBlueCollar`	Female head of hh blue-collar worker.
`maleWhiteCollar`	Male head of hh white-collar worker.
`femaleWhiteCollar`	Female head of hh white-collar worker.
`houseUnder30K`	hh income under $30K.
`house30to60K`	hh income $30K - $60K.
`house60to200K`	hh income $60K - $200K.
`houseOver200K`	hh income over $200K.
`houseWithSalary`	hh with salary income.
`houseSelfEmpl`	hh with self-employment income.
`houseInterestEtc`	hh with interest and other investment income.
`houseSS`	hh with social security income.

Table 2.4: *Variables contained in* `censusDemogr`*, a* `bdFrame` *object. All variables, except* `housingTotal`*, contain the proportion of households (hh) in the zip code area with the stated characteristic.*

Variable	Description
`housePubAssist`	hh with public assistance income.
`houseRetired`	Head of hh retired.
`houseNotVacant`	House not vacant.
`houseOwnerOccupied`	House owner occupied.
`group18`	Cluster group18.

Building a Model

The cluster group membership variables are binary with "`yes`" or "`no`", indicating group membership for each zip code area. To get a sense of group membership characteristics, you can create a logistic model for each group of interest using `glm`, which has been extended to handle `bdFrame` objects. The syntax is identical to that of `glm` with regular data frames.The model specification is as follows:

```
> group18Fit <- glm(group18 ~ ., data = censusDemogr,
                    family = binomial)
```

And the output is similar:

```
> group18Fit
Call:
bdGlm(formula = group18 ~ ., family = binomial, data
   = censusDemogr)

Coefficients:
 (Intercept) housingTotal           own
   -51.49204 0.0002713171 -0.0005471851

 onePlusPersonHouse nonFamily Plus65InHouse
           3.560468  10.21905      18.44271
.
.
```

```
.
Degrees of freedom: 31951 total; 31888 residual
Residual Deviance: 5445.941
```

Note

The `glm` function call is the same as for regular in-memory data frames; however, the extended version of `glm` in the bigdata library applies appropriate methods to `bdFrame` data by initiating a call to `bdGlm`. The `call` expression shows the actual call went to `bdGlm`.

Summarizing the Fit

You can apply the usual operations (for example, `summary`, `coef`, `plot`) to the resulting fit object. The plots are displayed as hexbin scatterplots because of the volume of data.

```
> plot(group18Fit)
```

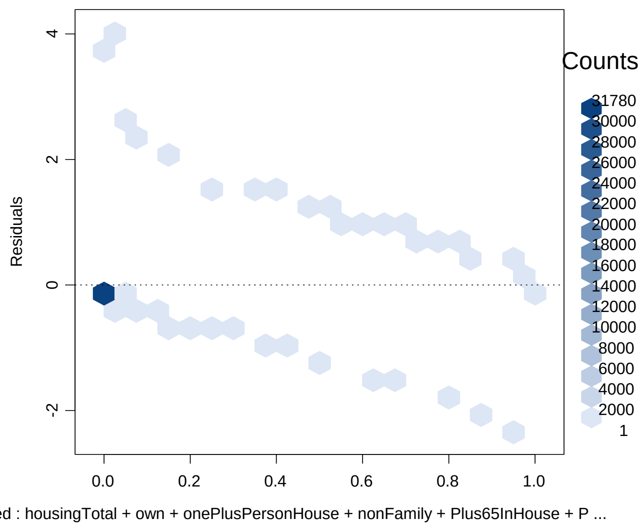

Figure 2.13: *Residuals vs. fitted values resulting from modeling cluster group 18 membership as a function of census demographics.*

Characterizing the Group

To characterize the group, examine the significant coefficients as follows:

```
> group18Coeff <- summary(group18Fit)[["coef"]]
> group18Coeff[abs(group18Coeff[, "t value"]) >
     qnorm(0.975), ]
                          Value Std. Error   t value
        (Intercept) -51.492043  13.866083 -3.713525
          nonFamily  10.219051   4.079199  2.505161
       Plus65InHouse 18.442709   6.172655  2.987808
    Plus65InNonFamily 19.186751   5.953835  3.222587
          maleSingle 39.541568   9.123876  4.333857
         femaleWidow 23.710092  10.332282  2.294759
             maleDiv 23.374178   8.807237  2.653974
  changedHouseSince95  6.253725   2.492780  2.508735
          femaleLoEd -12.132175   2.986016 -4.062997
         maleCollege  5.820187   2.897105  2.008966
            femaleBA  -9.518559   3.518594 -2.705217
          maleAdvDeg  10.536835   3.553861  2.964898
        femaleAdvDeg  -7.932499   3.568260 -2.223072
        maleWorked99   6.598822   2.787717  2.367107
      femaleWorked99   7.200051   3.244321  2.219278
```

To interpret the above table, note that positive coefficients predict group 18 membership and negative coefficients predict non-group membership. With that understanding, group 18 members are more likely:

- In non-family households that have changed location in the last 5 years.
- Single or divorced males or widowed females.
- Males with some college education and frequently with advanced degrees who worked the previous year.

Cluster group 18 corresponds to zip code regions dominated by young adult males, typical of military bases and penal institutions.

ANALYZING LARGE DATASETS FOR ASSOCIATION RULES

3

INTRODUCTION

Association rules specify how likely certain items occur together with other items in a set of transactions. The classic example used to describe association rules is the "market basket" analogy, where each transaction contains the set of items brought on one shopping trip. The store manager might want to ask questions, such as “if a shopper buys chips, does the shopper usually also buy dip?” Using a market basket analysis, the store manager can discover association rules for these items, so he knows whether he should plan on stocking chips and dip amounts accordingly and place the items near each other in the store.

When you encounter an association rule, you might see it notated as **X <- Y**, where item **X** is the *consequent* and item **Y** is the *antecedent*. For example, examine the following rule:

```
chips <- dip
```

Your analysis would show the relationship between chips (the consequent) and dip (the antecedent).

For the Big Data library’s implementation of association rules, only one consequent is allowed; however the rule can have multiple antecedents. To the above example, you might also add beer:

```
chips <- dip beer
```

A collection of items is sometimes referred to as an itemset. You are interested in the significance of items in an itemset and the likelihood of them occurring with other items (that is, chips and dip, in the example above). In association rule algorithms, these two measures (the significance and the occurrence) are referred to as *support* and *confidence*, respectively. A third measure, *lift*, is the ratio of the confidence to that expected by chance. These three measures determine if a rule is interesting. They are discussed more thoroughly later.

The Apriori Algorithm

You can use the Big Data library function `bd.assoc.rules` to generate association rules from a set of transactions that have a specified minimum support and confidence. This function uses the

Apriori algorithm, which is the best-known algorithm to mine association rules. It uses a breadth-first search strategy to counting the support of itemsets and rules.

Downward closure property

The apriori characteristic *support*, described in the section Support on page 64, possesses the *downward closure property*, indicating that all subsets of a frequent set also are frequent. This property, which specifies that no superset of an infrequent set can be frequent, is used in the apriori algorithm to prune the search space. Usually, the search space is represented as a lattice or tree of itemsets with increasing size.

Note

Using the apriori algorithm with support introduces the disadvantage of the *rare item problem.* Items that occur infrequently in the data set are pruned; although they could produce interesting and potentially valuable rules. The rare item problem is important for transaction data that usually have a very uneven distribution of support for the individual items (few items are used all the time and most items are used rarely).

A solution to the rare item problem is to pre-filter your dataset. For example, if you were interested in the occurrence of certain furniture items in transactions in a department store, you might filter out sales of women's clothing, where sales might far outpace furniture sales.

BIG DATA ASSOCIATION RULES IMPLEMENTATION

The Big Data library defines three association rules functions:

- `bd.assoc.rules`
- `bd.assoc.rules.get.item.counts`
- `bd.assoc.rules.graph`

bd.assoc.rules

The Big Data library defines the function `bd.assoc.rules`, which reads input transactions from a `bdFrame` or `data.frame`, and then generates association rules using the apriori algorithm. The input data can be very large, with millions of transactions. The input transactions can be expressed in several different input formats, which are described in Table 3.1. `bd.assoc.rules` provides control over the output format of the generated rules and associated measures..

Note

The apriori algorithm was originally developed by Argawal (1994). The Big Data library uses a version of the apriori algorithm implemented by Christian Borgelt (2002). The original source code and the modified source code provided by the Big Data functions are included in the ***SHOME*/library/bigdata/apriori** directory (where ***SHOME*** is your Spotfire S+ installation directory).

bd.assoc.rules arguments

The Help files for `bd.assoc.rules` provide detailed information about each of its arguments. This section provides a high-level discussion of some of the options.

The argument `input.format`, along with several others, specify how the transaction items are read from the input data. For more detailed information about the recognized input formats, see Table 3.1.

Other arguments specify which elelents (rule strings, measures, and so on) are output by the function.

Other arguments, such as `min.support`, `min.confidence`, `min.rule.items`, and `max.rule.items`, control how the algorithm is applied to give meaningful results. `min.rule.items` and `max.rule.items` determine how many antecedents your rule can have. (Remember: you can have one and only one consequent.) For example, if you set `min.rule.items` to `1`, then your results can return rules with just the consequent and no antecedents. (The default is `2`, which allows for one consequent and at least one antecedent.) The default of `max.rule.items` is `5`, which allows for `1` consequent and up to `4` antecedents.

The argument `rule.support.both` indicates whether to include both the consequent and the antecedent when calculating the support. For more information on this argument, see the section Support on page 64.

Definitions

This section contains definitions of some of the key terms for using the S-PLUS function `bd.assoc.rules`. To help describe these terms, we use a small dataset called `marketdata2`. In this dataset, each row represents a transaction. The `TransID` column contains a unique identifier for each transaction. The other columns (`Milk`, `Bread`, `Cheese`, `Apple`) represent products of interest. The presence or absence of each item in a particular transaction is represented by a `1` or a `0`, respectively, in the appropriate column. (You can find this sample in the file ***SHOME*/samples/bigdata/assocrules/ marketdata2.txt**.) While this dataset is too small to provide any real meaningful output, it helps to demonstrate the terms and their formulas.)

TransId	Milk	Bread	Cheese	Apple
1	1	1	1	1
2	1	0	0	1
3	0	1	0	1
4	0	1	1	1
5	0	1	0	1
6	1	1	0	0
7	1	0	1	1

We can pass this dataset to the `bd.assoc.rules` functions, as follows:

```
bd.assoc.rules(marketdata2,
   item.columns=c(2:5),
   input.format="column.flag")
```

This function returns the following data:

```
       rule                        support  confidence      lift
1 Cheese <- Apple Bread Milk 0.1428571  1.0       2.3333333
```

```
2 Apple <- Bread Cheese      0.2857143  1.0      1.1666667
3 Apple <- Bread Cheese Milk 0.1428571  1.0      1.1666667
4 Apple <- Cheese            0.4285714  1.0      1.1666667
5 Apple <- Cheese Milk       0.2857143  1.0      1.1666667
6 Apple <- Bread             0.5714286  0.8      0.9333333
```

Support, confidence, and lift are the measures that determine whether a rule is interesting. The following sections describe the results displayed in the columns `support`, `confidence`, and `lift`..

Note

The following formula explanations use the raw count column names, which are output by `bd.assoc.rules` when `output.counts=TRUE`:

- `antCount`: Number of input transactions containing the rule antecedents.
- `conCount`: Number of input transactions containing the rule consequent.
- `ruleCount`: Number of input transactions containing both the rule consequent and antecedents.
- `itemCount`: Number of items used for creating rules.
- `transCount`: Total number of transactions in the input set.

The `transCount` and `itemCount` values are the same for every rule

Support

The input of an itemset is defined as the proportion of transactions containing all of the items in the itemset. The support of a rule can be defined in different ways

By default, in `bd.assoc.rules`, support is measured as follows:

`ruleCount` / `transCount`

or < the # of transactions containing the rule consequent and antecedent> /
<the total number of transactions>

Support measures significance (that is, the importance) of a rule. The user determines the minimum support threshold; that is, the minimum rule support for generated rules. The default value for the minimum rule support is `0.1`. Any rule with a support below the minimum is disregarded.

Using our `marketdata2` data, above, we see the following rule:

```
     rule               support   confidence      lift
6 Apple <- Bread       0.5714286    0.8         0.9333333
```

Support for this rule (consequent `Apple`, the antecedent `Bread`) is `0.5714286`

support = `ruleCount` / `transCount`
= <# transactions with `Apple` and `Bread`> / <total # of transactions>
= 4 / 7
= `0.5714286`

> **Note**
>
> `bd.assoc.rules` also provides the argument `rule.support.both`, which is set to `T` by default. If you set this flag to `F`, then only the antecedent is included in the support calculation. That is, for the rule `Apple` and `Bread`:
>
> support = `antCount` / `transCount`
> = <# transactions w `Bread`> / <total # transactions>
> = 5 / 7
> = `0.7142857`
>
> As you can see, calculating support using this argument provides very different results.

Next, try these calculations for a rule that contains multiple antecedents:

```
     rule                        support   confidence      lift
1 Cheese <- Apple Bread Milk 0.1428571   1.0        2.3333333
```

The standard rule support for `Cheese <- Apple Bread Milk` is as follows:

support = `ruleCount` / `transCount`
= <# transactions w rule consequent and antecedents> / <total # transactions>
= <# transactions w `Cheese Apple Bread Milk` > / <total # transactions>
= 1 / 7
= `0.1428571`

The alternative rule support (setting `rule.support.both` to `F`) for `Cheese <- Apple Bread Milk` is the same for this rule:

support = `antCount` / `transCount`
= <# transactions w rule antecedents>
/ <total # transactions>
= <# transactions w `Apple Bread Milk`>
/ <total # transactions>
= 1 / 7
= `0.1428571`

Confidence

Also called strength. Confidence can be interpreted as an estimate of the probability of finding the antecedent of the rule under the condition that a transaction also contains the consequent. In our `marketdata2` example, we see that the confidence for the rule `Apple <- Bread` is `0.8`:

```
            rule      support  confidence        lift
  6 Apple <- Bread   0.5714286         0.8   0.9333333
```

confidence = `ruleCount` / `antCount`
= <# transactions w rule consequent and antecedents>
/ <# transactions w rule antecedents>
= <# transactions w `Apple` and `Bread`>
/ <# transactions w `Bread`>
= 4 / 5
= `0.8`

`bd.assoc.rules` sets the minimum confidence as `0.8` by default. Any rule with a confidence below the minimum is disregarded.

Next, try these calculations for a rule that contains multiple antecedents:

```
        rule                     support  confidence      lift
  1 Cheese <- Apple Bread Milk 0.1428571  1.0        2.3333333
```

confidence = `ruleCount` / `antCount`
= <# transactions w rule consequent and antecedents>
/ <# transactions w rule antecedents>
= <# transactions w `Cheese Apple Bread Milk` >

/ <# transactions w `Apple Bread Milk` >
= 1 / 1
= `1.0`

Lift

Often, `bd.assoc.rules` returns too many rules, given the `min.support` and `min.confidence` constraints. If this is the case, you might want to apply another measure to rank your results. *Lift* is such a measure. Greater lift values indicate stronger associations. (Hahsler et al, 2008).

In our `marketdata2` example, we see the following:

```
        rule               support   confidence      lift
 6 Apple <- Bread        0.5714286     0.8         0.9333333
```

Lift is defined as the ratio of the observed confidence to that expected by chance. That is, lift for `Apple <- Bread` is `0.9333333`:

lift = (`ruleCount` / `antCount`) / (`conCount` / `transCount`)
= (<# transactions w rule consequent and antecedents> /
<# transactions w rule antecedents>) /
(<# transactions w rule consequent> /
<total # transactions>)
= (<# transactions w `Apple` and `Bread`>
/ <# transactions w `Bread`>) /
(<# transactions w `Apple`> / <total # transactions>)

= (4 / 5) / (6 / 7)
= `0.9333333`

The lift looks to be lower than what we might find interesting. Examining the data, we see that an `Apple` purchase appears in six of our seven transactions, suggesting that nearly everyone buys `Apple`. Knowing that everyone buys `Apple` might be interesting on its own, but it is not that interesting for our association rules. To get meaningful lift results, you might consider filtering lower results (less than `1`). Note that in small databases, lift can be subject to a lot of noise; it is most useful for analyzing larger databases.

Try these calculations for a rule that contains multiple antecedents:

```
        rule                        support   confidence     lift
 1 Cheese <- Apple Bread Milk 0.1428571   1.0        2.3333333
```

lift = (`ruleCount` / `antCount`) / (`conCount` / `transCount`)
= (<# transactions w rule consequent and antecedents>
/ <# transactions w rule antecedents>) /
(<# transactions w rule consequent> / <total # transactions>)
= (<# transactions w `Cheese Apple Bread Milk` >
/ <# transactions w `Apple Bread Milk` >) /
(<# transactions w `Cheese`> / <total # transactions>)
= (1 / 1) / (3 / 7)
= `2.333333`

bd.assoc.rules.get.item.counts

Market analysis databases can be very large, so you need tools to manage memory use for your analysis. The Big Data library function `bd.assoc.rules.get.item.counts` is a function used along with, and sometimes by, `bd.assoc.rules` to count the occurance of items within a set of transactions without storing all of the different items in memory. That is, you can use this function to avoid memory problems generating association rules when you have a large number of different possible items.

This function is used in two ways:

- It is called by `bd.assoc.rules` if the argument `prescan.items=T` so all of the unique items are not stored in memory.
- It is called by the user to generate the list of items and filter the resulting list to produce a vector of interesting items. The user then can pass this vector of items as the `bd.assoc.rules` argument `init.items`.

The arguments for `bd.assoc.rules.get.item.counts` are a subset of those for `bd.assoc.rules`.

The following shows a call to `bd.assoc.rules.get.item.counts` on our `marketdata2` data:

```
bd.assoc.rules.get.item.counts(marketdata2,
item.columns=2:5, input.format="column.flag")

      item count totalTransactions
1    Apple     6                 7
2    Bread     5                 7
3   Cheese     3                 7
4     Milk     4                 7
```

bd.assoc.rules.graph

Plotting your association rules can give you a rough sense of which consequent and antecedent items appear most often in the rules with high column values. The function `bd.assoc.rules.graph` creates a plot of a set of association rules. It takes one required argument, `rules`, which is the rules produced by your call to `bd.assoc.rules`. Optionally, you can limit the number of rules displayed to those columns within a specified range using the arguments `column.min` and `column.max`.

To create an association rules graph

1. Create a `data.frame` or `bdFrame` using `bd.assoc.rules`:

```
x<-bd.assoc.rules(marketdata2, item.columns=2:5,
input.format="column.flag")
```

2. Graph the results:

```
bd.assoc.rules.graph(x)
```

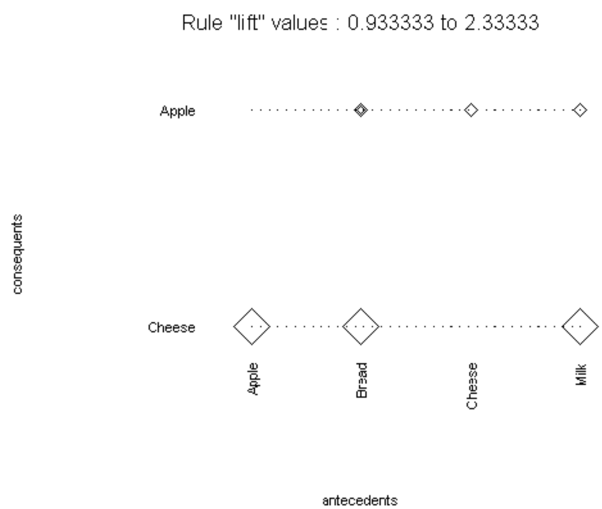

Figure 3.1: *Plot of* `marketdata2`.

This plot processes the association rules, collecting a list of all items that appear as consequents in any rules, and a list of all items that appear as antecedents in any rules. Each of these lists is sorted alphabetically and displayed in the graph, with consequent items displayed in a vertical list along the left side, and the antecedent items displayed in a list along the bottom side. For each rule, a symbol is displayed at the intersection of the rule's consequent item and each of its antecedent items. The symbol is an unfilled diamond, whose size is proportional to the column value for the rule. Because the diamond is not filled, multiple diamonds can be plotted in the same location and still be visible, if they represent rules with different column values.

You can use this plot to get a rough idea of which consequent and antecedent items appear most often in the rules with high column values. Because information from multiple rules can be plotted over each other, it is not possible to read individual rules from this graph. (To view individual rules, examine the rules data directly.)

Data Input Types

The AssocRules functions `bd.assoc.rules` and `bd.assoc.rules.get.item.counts` handle input data formatted in the four ways described below. In each input format, the input data contains a series of transactions, where each transaction contains a set of items.

Table 3.1: *Association Rules Data Input Types*

<table>
<tr><th>Input Format</th><th>Description</th></tr>
<tr><td><code>item.list</code></td><td>Each input row contains one transaction. The transaction items are all non-NA, non-empty strings in the item columns. There must be enough columns to handle the maximum number of items in a single transaction.

For example, the file SHOME/samples/bigdata/assocrules/groceries.il.txt starts with the following column names and first two rows:

<code>"i1",   "i2",   "i3",   "i4",   "i5",   "i6"</code>
<code>"milk", "cheese", "bread"   ,   ,   ,</code>
<code>"meat", "bread"   ,   ,   ,   ,</code>

The first transaction contains items <code>"milk"</code>, <code>"cheese"</code>, and <code>"bread"</code>, and the second transaction contains items <code>"meat"</code> and <code>"bread"</code>.</td></tr>
</table>

Table 3.1: *Association Rules Data Input Types (Continued)*

Input Format	Description

`column.flag`	Each input row contains one transaction. The column names are the item names, and each column's item is included in the transaction if the column's value is "flagged." More specifically, if an item column is numeric, it is flagged if its value is anything other than `0.0` or `NA`. If the column is a string or factor, the item is flagged if the value is anything other than `"0"`, `NA`, or an empty string.
	
	For example, the file ***SHOME*/samples/bigdata/assocrules/ groceries.cf.txt** starts with the following two transactions, encoding the same transactions as the example above:
	
	`"bread","meat","cheese","milk","cereal","chips","dip"`
	`      1,     0,       1,     1,       0,      0,    0`
	`      1,     1,       0,     0,       0,      0,    0`
	
	This format is not suitable for data where there are a large number of possible items, such as a retail market basket analysis with thousands of SKUs, because it requires so many columns.

`transaction.id`	One or more rows specify each transaction. Each row has a transaction.id column, specifying which transaction contains the items. This is a very efficient format when individual transactions can have a large number of items, and when there are many possible distinct items.
	
	For example, the file ***SHOME*/samples/bigdata/assocrules/ groceries.ti.txt** starts with the following two transactions, encoding the same transactions as the example above:
	
	`"id","item"`
	`10001,"bread"`
	`10001,"cheese"`
	`10001,"milk"`
	`10002,"meat"`
	`10002,"bread"`

Table 3.1: *Association Rules Data Input Types (Continued)*

Input Format	Description

`column.value`	Each input row contains one transaction. Items are created by combining column names and column values to produce strings of the form `"<col>=<val>"`. This is useful for applying association rules to surveys where the results are encoded into a set of factor values.
	
	This format is not suitable for the groceries example described for the three other input types. The file ***SHOME*/samples/bigdata/assocrules/fuel.cv.txt** starts with the following four transactions:
	
	`"Weight",  "Mileage",  "Fuel"`
	`"medium",  "high",     "low"`
	`"medium",  "high",     "low"`
	`"low",     "high",     "low"`
	`"medium",  "high",     "low"`
	
	The first, second, and fourth transactions contain the items `"Weight=medium"`, `"Mileage=high"`, and `"Fuel=low"`. The third transaction contains the items `"Weight=low"`, `"Mileage=high"`, and `"Fuel=low"`.

ASSOCIATION RULE SAMPLE

The directory ***SHOME*/samples/bigdata/assocrules/** (where ***SHOME*** is your Spotfire S+ installation) contains the following example datasets in different input formats.

- **groceries.il.txt**
- **groceries.cf.txt**
- **groceries.ti.txt**
- **fuel.cv.txt**

The first three datasets encode the same set of transactions. The data was generated randomly, and then modified to produce some interesting associations. **fuel.cv.txt** was derived from the standard **fuel.frame** dataset.

These datasets are small enough that they can be read as **data.frame** objects; however, **bd.assoc.rules** can handle very large input datasets represented as **bdFrame** objects with millions of rows.

To load the library and import association rules examples

1. Load the `bigdata` library, which contains the S-PLUS association rules functions.

```
library(bigdata)
```

2. Read in the data files, as follows:

```
groceries.il <-
    importData(file.path(getenv("SHOME"),
    "samples/bigdata/assocrules/groceries.il.txt",
       sep=""),
    colNameRow=1,stringsAsFactors=F)

groceries.cf <-
    importData(file.path(getenv("SHOME"),
    "samples/bigdata/assocrules/groceries.cf.txt",
       sep=""),
    colNameRow=1,stringsAsFactors=F)

groceries.ti <-
    importData(file.path(getenv("SHOME"),
```

```
        "samples/bigdata/assocrules/groceries.ti.txt",
           sep=""),
        colNameRow=1,stringsAsFactors=F)

fuel.cv <-
    importData(file.path(getenv("SHOME"),
    "samples/bigdata/assocrules/fuel.cv.txt", sep=""),
    colNameRow=1,stringsAsFactors=F)
```

The following example demonstrates processing the dataset `groceries.cf` with `bd.assoc.rules`.

To work through association rules examples

1. By default, the output is sorted so the rules with the highest lift are listed first.

```
bd.assoc.rules(groceries.cf,
   input.format="column.flag")
```

```
                          rule support confidence      lift
 1                dip <- chips   0.180  0.9183673 3.6156195
 2           dip <- chips milk   0.162  0.9101124 3.5831195
 3        bread <- cheese meat   0.120  0.8955224 1.5821950
 4   bread <- cheese meat milk   0.110  0.8870968 1.5673088
 5         milk <- bread chips   0.100  0.9433962 1.0165908
 6           milk <- bread dip   0.126  0.9264706 0.9983519
 7         milk <- cheese meat   0.124  0.9253731 0.9971693
 8          milk <- bread meat   0.196  0.9245283 0.9962589
 9               milk <- bread   0.522  0.9222615 0.9938163
10        milk <- bread cereal   0.250  0.9191176 0.9904285
11   milk <- bread cheese meat   0.110  0.9166667 0.9877874
12                milk <- meat   0.276  0.9139073 0.9848139
13                 milk <- dip   0.232  0.9133858 0.9842520
14              milk <- cheese   0.372  0.9117647 0.9825051
15              milk <- cereal   0.454  0.9116466 0.9823778
16               milk <- chips   0.178  0.9081633 0.9786242
17        milk <- bread cheese   0.240  0.9022556 0.9722582
18           milk <- chips dip   0.162  0.9000000 0.9698276
19          milk <- cereal dip   0.118  0.8939394 0.9632968
20       milk <- cereal cheese   0.168  0.8936170 0.9629494
21         milk <- cereal meat   0.134  0.8933333 0.9626437
22 milk <- bread cereal cheese   0.100  0.8928571 0.9621305
```

The first observation from the results is that many of the rules contain `milk` because almost all of the original transactions contain milk, as shown in the item counts:

```
bd.coerce(bd.assoc.rules.get.item.counts(groceries.cf,
 input.format="column.flag"))
```

```
     item count totalTransactions
1   bread   283               500
2  cereal   249               500
3  cheese   204               500
4   chips    98               500
5     dip   127               500
6    meat   151               500
7    milk   464               500
```

You can see the same item counts by using `colSums` on `groceries.cf`:

```
colSums(groceries.cf)
```

```
bread meat cheese milk cereal chips dip
  283  151    204  464    249    98 127
```

In this case, we probably are not interested in associations involving `milk`, because it is so frequent. We can ignore the item `milk` by listing the other items as follows:

```
bd.assoc.rules(groceries.cf,
    input.format="column.flag",
    init.items=c("bread", "meat", "cheese",
                 "cereal", "chips", "dip"))
```

```
                    rule support confidence     lift
1         dip <- chips    0.18  0.9183673 3.615619
2 bread <- cheese meat    0.12  0.8955224 1.582195
```

Without the `milk` item, we have only a few rules. These rules also appeared in the larger list, above.

We created the grocery data by selecting random items (with differing probabilities), and then we changed the data by:

- Increasing the probability of including `dip` for transactions containing `chips`.

- Increasing the probability of including `bread` for transactions containing both `cheese` and `meat`.

The second and fourth rules detect both of these changes.

We could produce the same sets of rules with the other grocery datasets, because they encode the same sets of transactions:

```
bd.assoc.rules(groceries.il,
   input.format="item.list")

bd.assoc.rules(groceries.ti,
   input.format="transaction.id",
   item.columns="item",
   id.columns="id")
```

Also, we could derive rules from the `fuel.cv` dataset:

```
bd.assoc.rules(fuel.cv,
    input.format="column.value",
    min.support=0.3)

  rule                         support    confidence     lift
1 Fuel=high <- Weight=high 0.3833333  0.8260870 2.155009
2 Weight=high <- Fuel=high 0.3833333  0.8260870 2.155009
3 Weight=medium <- Fuel=medium 0.4333333  0.8461538
                                                  1.450549
4 Weight=medium <- Fuel=medium Mileage=medium 0.4333333
                                        0.8461538 1.450549
5 Mileage=medium <- Fuel=medium 0.4333333  1.0000000
                                                  1.363636
6 Mileage=medium <- Fuel=medium Weight=medium 0.3666667
                                        1.0000000 1.363636
7 Mileage=medium <- Weight=medium 0.5833333  0.8000000
                                                  1.090909
```

In this case, we specify `min.support=0.3` to reduce the number of rules generated to those with the given minimum support. The most interesting rules are those indicating that `Fuel=high` is associated with `Weight=high`, which is what one would expect from this data.

MORE INFORMATION

Many valuable sources of information on Association Rules and the Apriori algorithm exist. Additionally, the Spotfire S+ Big Data library functions for association rules is similar to the arules package available on the CRAN Web site.

For more information on Association Rules, we suggest the following sources:

http://cran.org/ (Package arules)

http://www.borgelt.net/doc/apriori/apriori.html

http://michael.hahsler.net/research/association_rules/

CREATING GRAPHICAL DISPLAYS OF LARGE DATA SETS

4

INTRODUCTION

This chapter includes information on the following:

- An overview of the graph functions available in the Big Data Library, listed according to whether they take a big data object directly, or require a preprocessing function to produce a chart.
- Procedures for creating plots, traditional graphs, and Trellis graphs.

Note
In Microsoft Windows, editable graphs in the graphical user interface (GUI) do not support big data objects. To use these graphs, create a S-PLUS `data.frame` containing either all of the data or a sample of the data.

OVERVIEW OF GRAPH FUNCTIONS

The Big Data Library supports most (but not all) of the traditional and Trellis graph functions available in the S-PLUS library. The design of graph support for big data can be attributed to practical application. For example, if you had a data set of a million rows or tens of thousands of columns, a cloud chart would produce an illegible plot.

Functions Supporting Graphs

This section lists the functions that produce graphs for big data objects. If you are unfamiliar with plotting and graph functions in Spotfire S+, review the *Guide to Graphics*.

Implementing plotting and graph functions to support large data sets requires an intelligent way to handle thousands of data points. To address this need, the graph functions to support big data are designed in the following categories:

- Functions to plot big data objects without preprocessing, including:
    - Functions to plot big data objects by hexagonal binning.
    - Functions to plot big data objects by summarizing data in a plot-specific manner.
- Functions providing the preprocessing support for plotting big data objects.
- Functions requiring preprocessing support to plot big data objects.

The following sections list the functions, organized into these categories. For an alphabetical list of graph functions supporting big data objects, see the Appendix.

Using `cloud` or `parallel` results in an error message. Instead, sample or aggregate the data to create a `data.frame` that can be plotted using these functions.

Graph Functions using Hexagonal Binning

The following functions can plot a large data set (that is, can accept a big data object without preprocessing) by plotting large amounts of data using hexagonal binning.

Table 4.1: *Functions for plotting big data using hexagonal binning.*

Function	Comment
`pairs`	Can accept a `bdFrame` object.
`plot`	Can accept a `hexbin`, a single `bdVector`, two `bdVectors`, or a `bdFrame` object.
`splom`	Creates a Trellis graphic object of a scatterplot matrix.
`xyplot`	Creates a Trellis graphic object, which graphs one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

Functions Adding Reference Lines to Plots

The following functions add reference lines to hexbin plots.

Table 4.2: *Functions that add reference lines to hexbin plots.*

Function	Type of line
`abline(lsfit())`	Regression line.
`lines(loess.smooth())`	Loess smoother.
`lines(smooth.spline())`	Smoothing spline.
`panel.lmline`	Adds a least squares line to an `xyplot` in a Trellis graph.

Table 4.2: *Functions that add reference lines to hexbin plots. (Continued)*

Function	Type of line
`panel.loess`	Adds a loess smoother to an `xyplot` in a Trellis graph.
`qqline()`	QQ-plot reference line.
`xyplot(lmline=T)`	Adds a least squares line to an `xyplot` in a Trellis graph.

Graph Functions Summarizing Data

The following functions summarize data in a plot-specific manner to plot big data objects.

Table 4.3: *Functions that summarize in plot-specific manner.*

Function	Description
`boxplot`	Produces side by side boxplots from a number of vectors. The boxplots can be made to display the variability of the median, and can have variable widths to represent differences in sample size.
`bwplot`	Produces a box and whisker Trellis graph, which you can use to compare the distributions of several data sets.
`plot(density)`	`density` returns x and y coordinates of a non-parametric estimate of the probability density of the data.
`densityplot`	Produces a Trellis graph demonstrating the distribution of a single set of data.
`hist`	Creates a histogram.
`histogram`	Creates a histogram in a Trellis graph.
`qq`	Creates a Trellis graphic object comparing the distributions of two sets of data

Table 4.3: *Functions that summarize in plot-specific manner. (Continued)*

Function	Description
`qqmath`	Creates normal probability plot for only one data object in a Trellis graph. `qqmath` can also make probability plots for other distributions. It has an argument distribution whose input is any function that computes quantiles.
`qqnorm`	Creates normal probability plot in a Trellis graph. `qqnorm` can accept a single `bdVector` object.
`qqplot`	Creates normal probability plot in a Trellis graph. Can accept two `bdVector` objects. In `qqplot`, each vector or `bdVector` is taken as a sample, for the x- and y-axis values of an empirical probability plot.
`stripplot`	Creates a Trellis graphic object similar to a box plot in layout; however, it displays the density of the datapoints as shaded boxes.

Functions Providing Support to Preprocess Data for Graphing

The following functions are used to preprocess large data sets for graphing:

Table 4.4: *Functions used for preprocessing large data sets.*

Function	Description
`aggregate`	Splits up data by time period or other factors and computes summary for each subset.
`hexbin`	Creates an object of class hexbin. Its basic components are a cell identifier and a count of the points falling into each occupied cell.
`hist2d`	Returns a structure for a 2-dimensional histogram which can be given to a graphics function such as `image` or `persp`.
`interp`	Interpolates the value of the third variable onto an evenly spaced grid of the first two variables.

Table 4.4: *Functions used for preprocessing large data sets. (Continued)*

Function	Description
`loess`	Fits a local regression model.
`loess.smooth`	Returns a list of values at which the loess curve is evaluated.
`lsfit`	Fits a (weighted) least squares multivariate regression.
`smooth.spline`	Fits a cubic B-spline smooth to the input data.
`table`	Returns a contingency table (array) with the same number of dimensions as arguments given.
`tapply`	Partitions a vector according to one or more categorical indices.

Functions Requiring Preprocessing Support for Graphing

The following functions do not accept a big data object directly to create a graph; rather, they require one of the specified preprocessing functions.

Table 4.5: *Functions requiring preprocessors for graphing large data sets.*

Function	Preprocessors	Description
`barchart`	`table`, `tapply`, `aggregate`	Creates a bar chart in a Trellis graph.
`barplot`	`table`, `tapply`, `aggregate`	Creates a bar graph.
`contour`	`interp`, `hist2d`	Make a contour plot and possibly return coordinates of contour lines.
`contourplot`	`loess`	Displays contour plots and level plots in a Trellis graph.

Table 4.5: *Functions requiring preprocessors for graphing large data sets. (Continued)*

Function	Preprocessors	Description
`dotchart`	`table`, `tapply`, `aggregate`	Plots a dot chart from a vector.
`dotplot`	`table`, `tapply`, `aggregate`	Creates a Trellis graph, displaying dots and labels.
`image`	`interp`, `hist2d`	Creates an image, under some graphics devices, of shades of gray or colors that represent a third dimension.
`levelplot`	`loess`	Displays a level plot in a Trellis graph.
`persp`	`interp`, `hist2d`	Creates a perspective plot, given a matrix that represents heights on an evenly spaced grid.
`pie`	`table`, `tapply`, `aggregate`	Creates a pie chart from a vector of data.
`piechart`	`table`, `tapply`, `aggregate`	Creates a pie chart in a Trellis graph
`wireframe`	`loess`	Displays a three-dimensional wireframe plot in a Trellis graph.

EXAMPLE GRAPHS

The examples in this chapter require that you have the Big Data Library loaded. The examples are not large data sets; rather, they are small data objects that you convert to big data objects to demonstrate using the Big Data Library graphing functions.

Plotting Using Hexagonal Binning

Hexagonal binning plots are available for:

- Single plot (`plot`)
- Matrix of plots (`pairs`)
- Conditioned single or matrix plots (`xyplot`)

Functions that evaluate data over a grid in standard Spotfire S+ aggregate the data over the grid (such as binning the data and taking the mean in each grid cell, and then plot the aggregated values) when applied to a big data object.

Hexagonal binning is a data grouping or reduction method typically used on large data sets to clarify a spatial display structure in two dimensions. Think of it as partitioning a scatter plot into larger units to reduce dimensionality, while maintaining a measure of data clarity. Each unit of data is displayed with a hexagon and represents a bin of points in the plot. Hexagons are used instead of squares or rectangles to avoid misleading structure that occurs when edges of the rectangles line up exactly.

Plotting using hexagonal binning is the standard technique used when a plotting function that currently plots one point per row is applied to a big data object.

Plotting using hexagonal bins is available for a single plot, a matrix of plots, and conditioned single or matrix plots.

The Census example introduced in Chapter 2 demonstrates plotting using hexagonal binning (see Figure 2.6). When you create a plot showing a distribution of zip codes by latitude and longitude, the following simple plot is displayed:

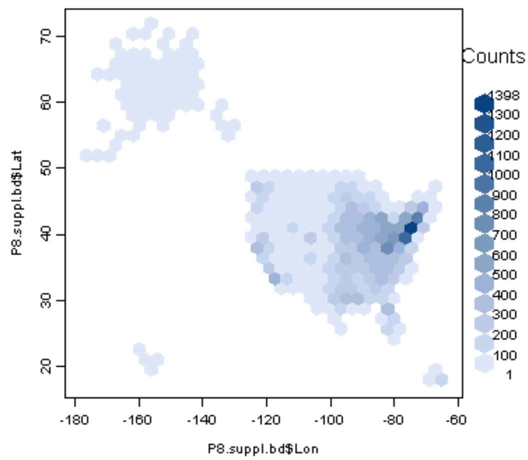

Figure 4.1: *Example of graph showing hexagonal binning.*

The functions listed in Table 4.1 support big data objects by using hexagonal binning. This section shows examples of how to call these functions for a big data object.

Create a Pair-wise Scatter Plot

The `pairs` function creates a figure that contains a scatter plot for each pair of variables in a `bdFrame` object.

To create a sample pair-wise scatter plot for the `fuel.frame` `bdFrame` object, in the **Commands** window, type the following:

```
pairs(as.bdFrame(fuel.frame))
```

The pair-wise scatter plot appears as follows:

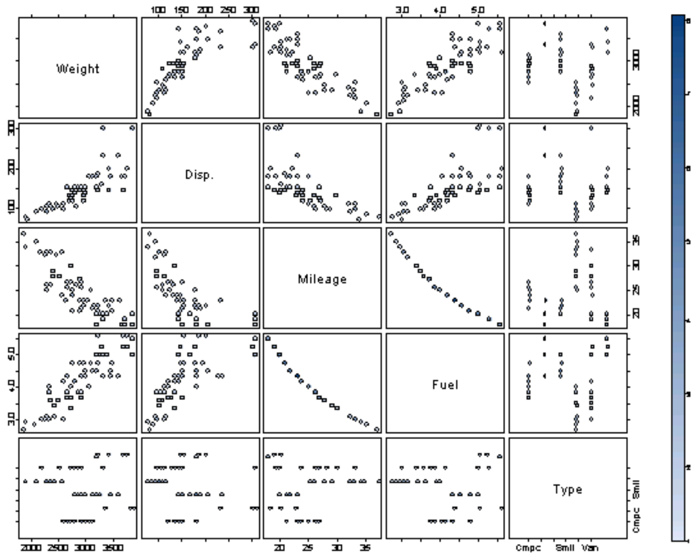

Figure 4.2: *Graph using* `pairs` *for a* `bdFrame`.

This scatter plot looks similar to the one created by calling `pairs(fuel.frame)`; however, close examination shows that the plot is composed of hexagons.

Create a Single Plot

The `plot` function can accept a `hexbin` object, a single `bdVector`, two `bdVectors`, or a `bdFrame` object. The following example plots a simple hexbin plot using the weight and mileage vectors of the `fuel.bd` object.

To create a sample single plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
plot(hexbin(fuel.bd$Weight, fuel.bd$Mileage))
```

The hexbin plot is displayed as follows:

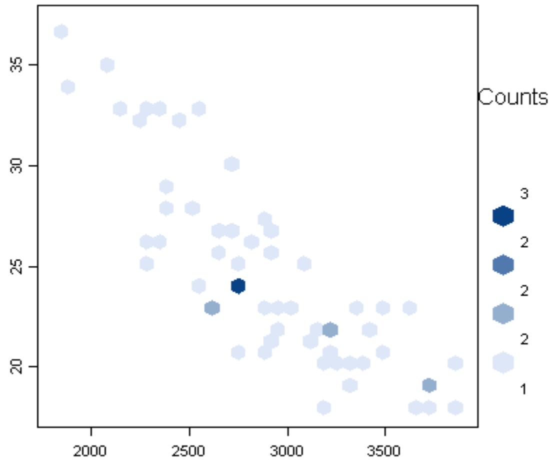

Figure 4.3: *Graph using single hexbin plot for* `fuel.bd`.

Create a Multi-Panel Scatterplot Matrix

The function `splom` creates a Trellis graph of a scatterplot matrix. The scatterplot matrix is a good tool for displaying measurements of three or more variables.

To create a sample multi-panel scatterplot matrix, where you create a hexbin plot of the columns in `fuel.bd` against each other, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
splom(~., data=fuel.bd)
```

Note
Trellis functions in the Big Data Library require the `data` argument. You cannot use formulas that refer to `bdVectors` that are not in a specified `bdFrame`.

Notice that the '.' is interpreted as all columns in the data set specified by `data`.

The `splom` plot is displayed as follows:

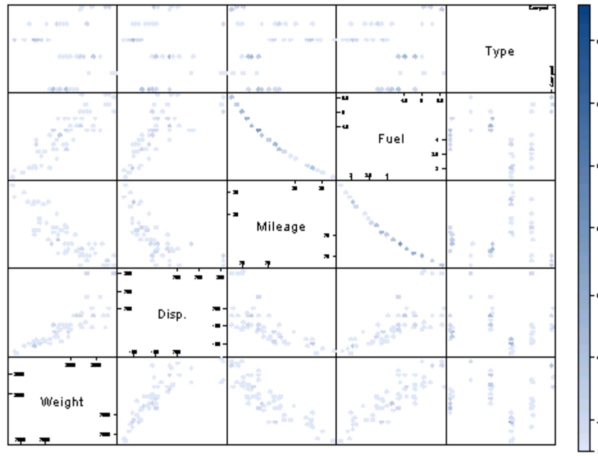

Figure 4.4: *Graph using* `splom` *for* `fuel.bd`.

To remove a column, use `-`*`term`*. To add a column, use `+`*`term`*. For example, the following code replaces the column `Disp.` with its log.

```
fuel.bd <- as.bdFrame(fuel.frame)
splom(~.-Disp.+log(Disp.), data=fuel.bd)
```

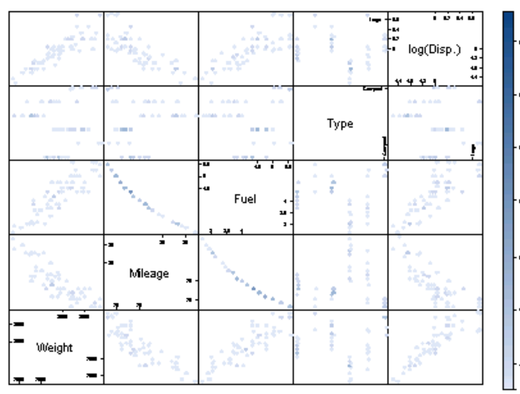

Figure 4.5: *Graph using* `splom` *to designate a formula for* `fuel.bd`

For more information about `splom`, see its help topic.

Create a Conditioning Plot or Scatter Plot

The function `xyplot` creates a Trellis graph, which graphs one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

To create a sample conditioning plot, in the **Commands** window, type the following:

```
xyplot(data=as.bdFrame(air),
  ozone~radiation|temperature,
  shingle.args=list(n=4), lmline=T)
```

The variable on the left of the ~ goes on the vertical (or y) axis, and the variable on the right goes on the horizontal (or x) axis.

The function `xyplot` contains the default argument `lmline=T` to add the approximate least squares line to a panel quickly. This argument performs the same action as `panel.lmline` in standard Spotfire S+.

The `xyplot` plot is displayed as follows:

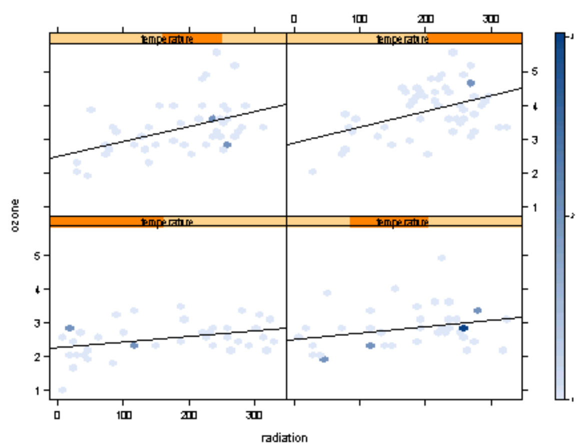

Figure 4.6: *Graph using `xyplot` with `lmline=T`.*

Trellis functions in the Big Data Library handle continuous "given" variables differently than standard data Trellis functions: they are sent through `equal.count`, rather than `factor`.

Adding Reference Lines

You can add a regression line or scatterplot smoother to hexbin plots. The regression line or smoother is a weighted fit, based on the binned values.

The following functions add the following types of reference lines to hexbin plots:

- A regression line with `abline`
- A Loess smoother with `loess.smooth`
- A smooth spline with `smooth.spline`
- A line to a `qqplot` with `qqline`
- A least squares line to an `xyplot` in a Trellis graph.

For `smooth.spline` and `loess.smooth`, when the data consists of `bdVectors`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and then the mean for `x` and `y` is computed in each bin. A weighted smooth is then computed on the bin means, weighted based on the bin counts. This computation results in values that differ somewhat from those where the smoother is applied to the unaggregated data. The values are usually close enough to be indistinguishable when used in a plot, but the difference could be important when the smoother is used for prediction or optimization.

Add a Regression Line

When you create a scatterplot from your large data set, and you notice a linear association between the y-axis variable and the x-axis variable, you might want to display a straight line that has been fit to the data. Call `lsfit` to perform a least squares regression, and then use that regression to plot a regression line.

The following example draws an `abline` on the chart that plots `fuel.bd` weight and mileage data. First, create a `hexbin` object and plot it, and then add the abline to the plot.

To add a regression line to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
# use add.to.hexbin to keep the abline within the
# hexbin area. If you just call abline, then the
# line might draw outside of the hexbin and interfere
# with the label.
add.to.hexbin(hexbin.out, abline(lsfit(fuel.bd$Weight,
  fuel.bd$Mileage)))
```

The resulting chart is displayed as follows:

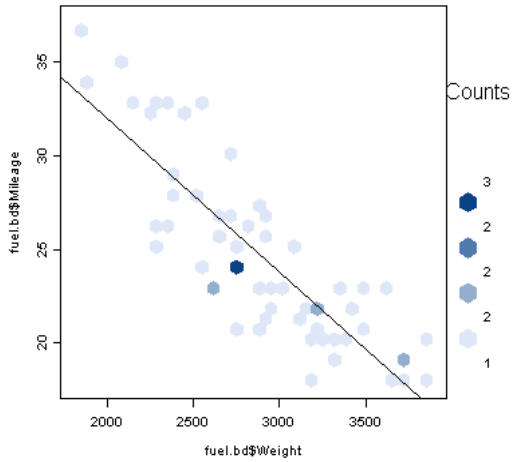

Figure 4.7: *Graph drawing an* `abline` *in a hexbin plot.*

Add a Loess Smoother

Use `lines(loess.smooth)` to add a smooth curved line to a scatter plot.

To add a `loess` smoother to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
add.to.hexbin(hexbin.out,
  lines(loess.smooth(fuel.bd$Weight,
  fuel.bd$Mileage), lty=2))
```

The resulting chart is displayed as follows:

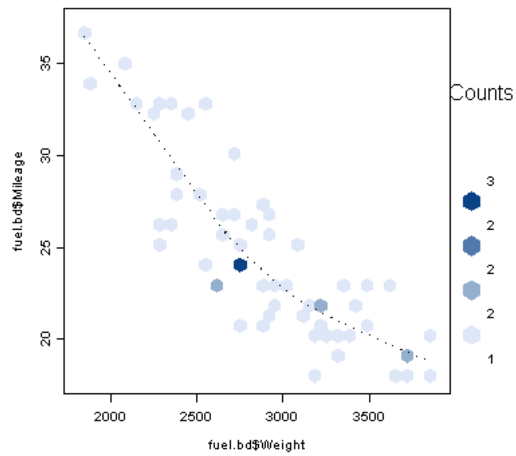

Figure 4.8: *Graph using* `loess.smooth` *in a hexbin plot.*

Add a Smoothing Spline

Use `lines(smooth.spline)` to add a smoothing spline to a scatter plot.

To add a smoothing spline to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
add.to.hexbin(hexbin.out,
  lines(smooth.spline(fuel.bd$Weight,
     fuel.bd$Mileage),lty=3))
```

The resulting chart is displayed as follows:

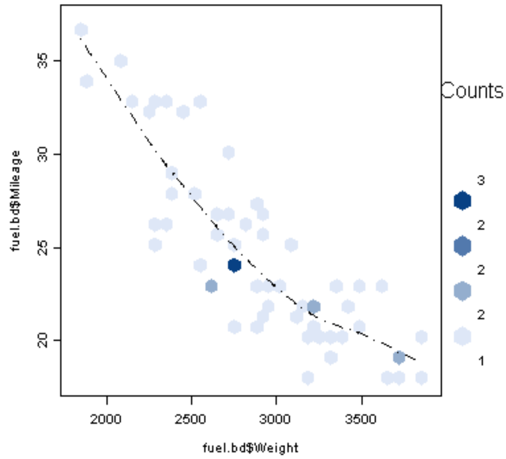

Figure 4.9: *Graph using* `smooth.spline` *in a hexbin plot.*

Add a Least Squares Line to an `xyplot`

To add a reference line to an `xyplot`, set `lmline=T`. Alternatively, you can call `panel.lmline` or `panel.loess`. See the section Create a Conditioning Plot or Scatter Plot on page 92 for an example.

Add a `qqplot` Reference Line

The function `qqline` fits and plots a line through a normal `qqplot`.

To add a `qqline` reference line to a sample `qqplot`, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqnorm(fuel.bd$Mileage)
qqline(fuel.bd$Mileage)
```

The `qqline` chart is displayed as follows:

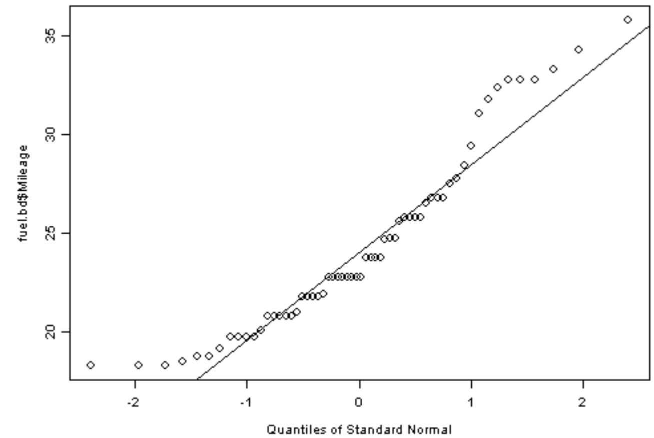

Figure 4.10: *Graph using* `qqline` *in a* `qqplot` *chart.*

Plotting by Summarizing Data

The following examples demonstrate functions that summarize data in a plot-specific manner to plot big data objects. These functions do not use hexagonal binning. Because the plots for these functions are always monotonically increasing, hexagonal binning would obscure the results. Rather, summarizing provides the appropriate information.

Create a Box Plot

The following example creates a simple box plot from `fuel.bd`. To create a Trellis box and whisker plot, see the following section.

To create a sample box plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
boxplot(split(fuel.bd$Fuel, fuel.bd$Type), style.bxp="att")
```

The box plot is displayed as follows:

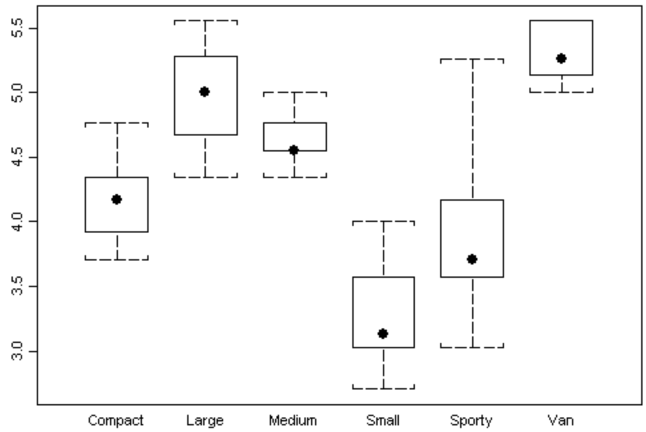

Figure 4.11: *Graph using* `boxplot`.

Create a Trellis Box and Whisker Plot

The box and whisker plot provides graphical representation showing the center and spread of a distribution.

To create a sample box and whisker plot in a Trellis graph, in the **Commands** window, type the following:

```
bwplot(Type~Fuel, data=(as.bdFrame(fuel.frame)))
```

The box and whisker plot is displayed as follows:

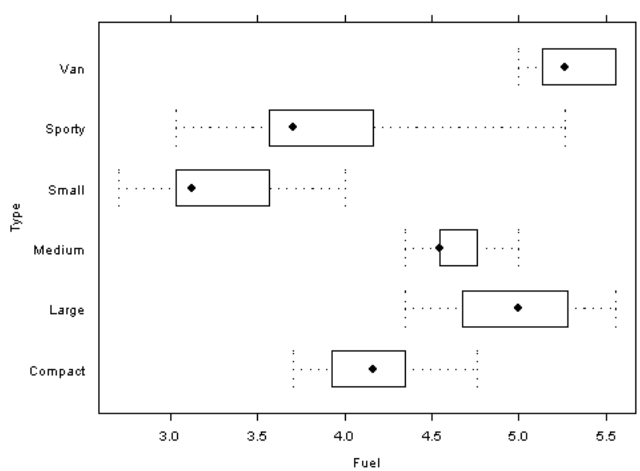

Figure 4.12: *Graph using* `bwplot`.

For more information about `bwplot`, see Chapter 3, Traditional Trellis Graphics, in the *Guide to Graphics.*

Create a Density Plot

The `density` function returns x and y coordinates of a non-parametric estimate of the probability density of the data. Options include the choice of the window to use and the number of points at which to estimate the density. Weights may also be supplied.

Density estimation is essentially a smoothing operation. Inevitably there is a trade-off between bias in the estimate and the estimate's variability: wide windows produce smooth estimates that may hide local features of the density.

Density summarizes data. That is, when the data is a `bdVector`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and the mean for `x` is computed in each bin. A weighted density estimate is then computed on the bin means, weighted based on the bin counts. This calculation gives values that differ somewhat from those when `density` is applied to the unaggregated data. The values are usually close enough to be indistinguishable when used in a plot, but the difference could be important when `density` is used for prediction or optimization.

To plot density, use the `plot` function.

To create a sample density plot from `fuel.bd`, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
plot(density(fuel.bd$Weight), type="l")
```

The density plot is displayed as follows:

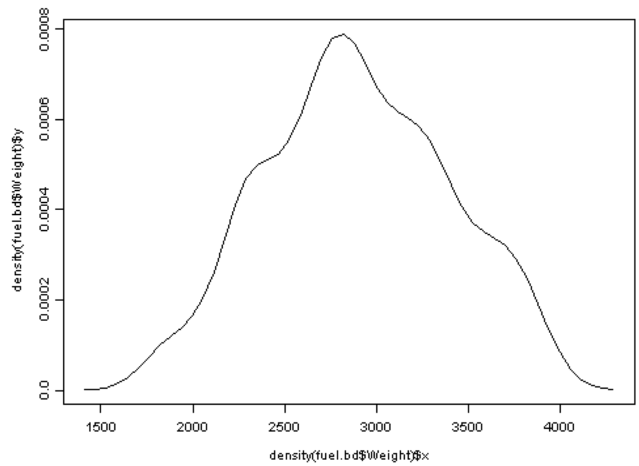

Figure 4.13: *Graph using* `density`

Create a Trellis Density Plot

The following example creates a Trellis graph of a density plot, which displays the shape of a distribution. You can use the Trellis density plot for analyzing a one-dimensional data distribution. A density plot displays an estimate of the underlying probability density function for a data set, allowing you to approximate the probability that your data fall in any interval.

To create a sample Trellis density plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
densityplot( ~ height | voice.part, data = singer.bd,
  layout = c(2, 4), aspect= 1, xlab = "Height (inches)",
  width = 5)
```

The Trellis density plot is displayed as follows:

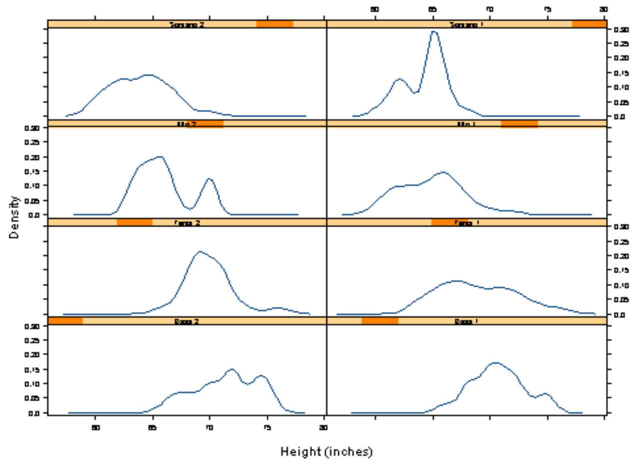

Figure 4.14: *Graph using* `densityplot`.

For more information about Trellis density plots, see Chapter 3, Traditional Trellis Graphics, in the *Guide to Graphics.*

Create a Simple Histogram

A histogram displays the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis. For this reason, density plots are often superposed with (scaled) histograms.

To create a sample `hist` chart of a full dataset for a numeric vector, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hist(fuel.bd$Weight)
```

The numeric `hist` chart is displayed as follows:

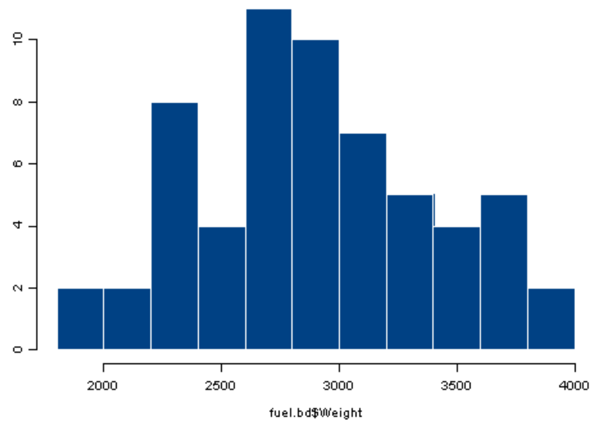

Figure 4.15: *Graph using* `hist` *for numeric data.*

To create a sample `hist` chart of a full dataset for a factor column, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hist(fuel.bd$Type)
```

The factor `hist` chart is displayed as follows:

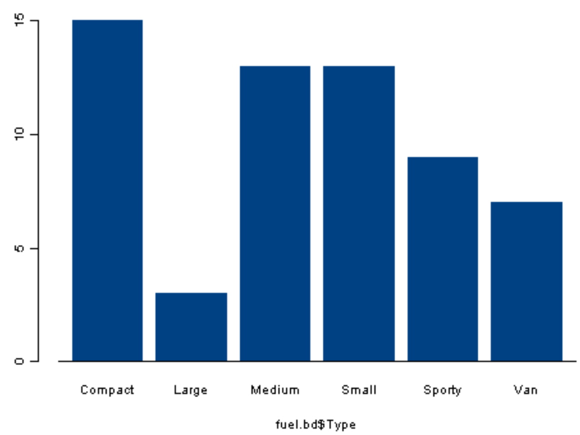

Figure 4.16: *Graph using* `hist` *for factor data.*

Create a Trellis Histogram

The `histogram` function for a Trellis graph is histogram.

To create a sample Trellis histogram, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
histogram( ~ height | voice.part, data = singer.bd,
  nint = 17, endpoints = c(59.5, 76.5), layout = c(2,4),
  aspect = 1, xlab = "Height (inches)")
```

The Trellis histogram chart is displayed as follows:

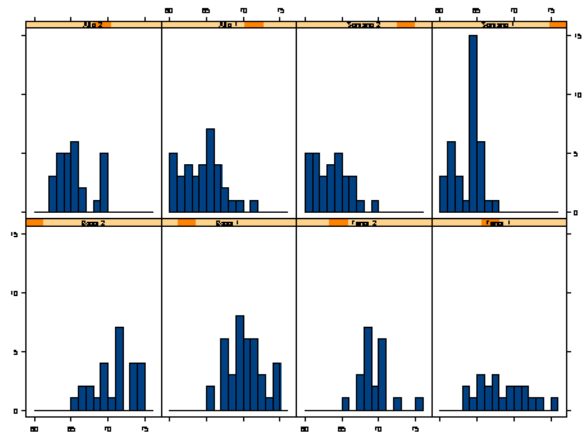

Figure 4.17: *Graph using* `histogram`*.*

For more information about Trellis histograms, see Chapter 3, Traditional Trellis Graphics, in the *Guide to Graphics*.

Create a Quantile-Quantile (QQ) Plot for Comparing Multiple Distributions

The functions `qq`, `qqmath`, `qqnorm`, and `qqplot` create an ordinary x-y plot of 500 evenly-spaced quantiles of data.

The function `qq` creates a Trellis graph comparing the distributions of two sets of data. Quantiles of one dataset are graphed against corresponding quantiles of the other data set.

To create a sample `qq` plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qq((Type=="Compact")~Mileage, data = fuel.bd)
```

The factor on the left side of the ~ must have exactly two levels (`fuel.bd$Compact` has five levels).

The `qq` plot is displayed as follows:

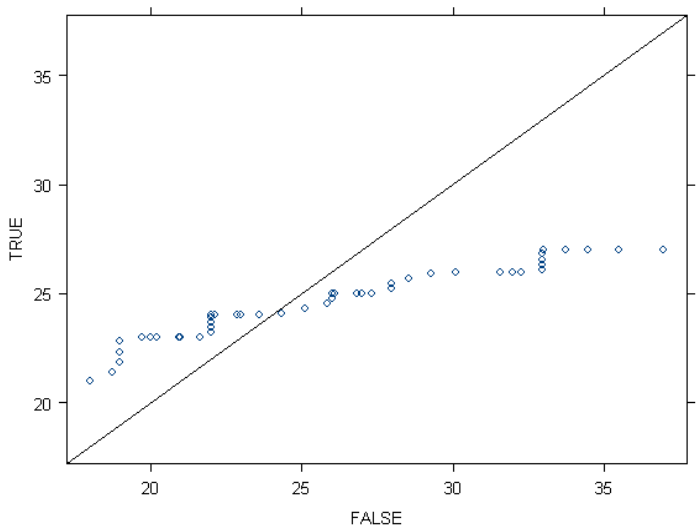

Figure 4.18: *Graph using* `qq`.

(Note that in this example, by setting `Type` to the logical `Compact`, the labels are set to `FALSE` and `TRUE` on the x and y axis, respectively.)

Create a QQ Plot Using a Theoretical or Empirical Distribution

The function `qqmath` creates normal probability plot in a Trellis graph. that is, the ordered data are graphed against quantiles of the standard normal distribution.

`qqmath` can also make probability plots for other distributions. It has an argument `distribution`, whose input is any function that computes quantiles. The default for `distribution` is `qnorm`. If you set `distribution = qexp`, the result is an exponential probability plot.

To create a sample `qqmath` plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
qqmath( ~ height | voice.part, data = singer.bd,
  layout = c(2, 4), aspect = 1,
  xlab = "Unit Normal Quantile",
  ylab = "Height (inches)")
```

The `qqmath` plot is displayed as follows:

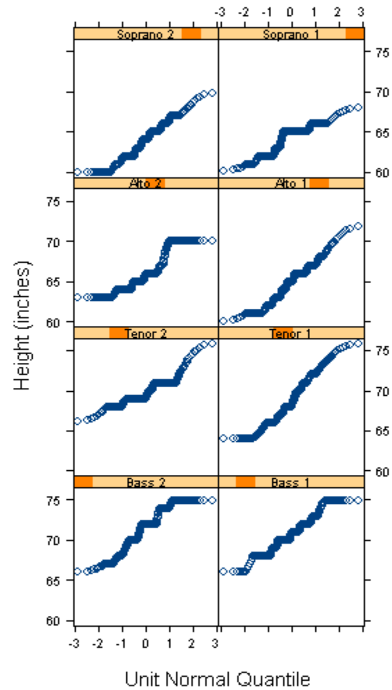

Figure 4.19: *Graph using* `qqmath`.

Create a Single Vector QQ Plot

The function `qqnorm` creates a plot using a single `bdVector` object. The following example creates a plot from the mileage vector of the `fuel.bd` object.

To create a sample `qqnorm` plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqnorm(fuel.bd$Mileage)
```

The `qqnorm` plot is displayed as follows:

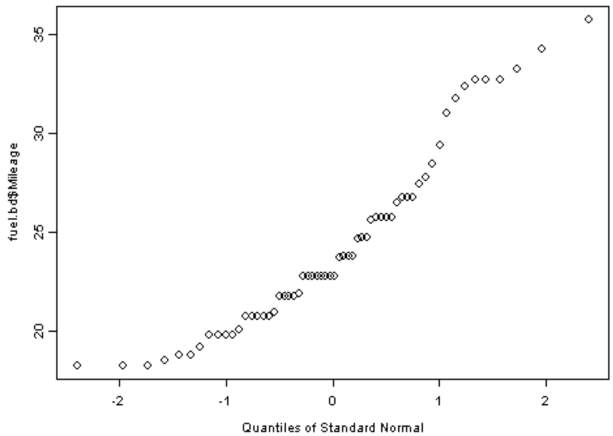

Figure 4.20: *Graph using* `qqnorm`*.*

Create a Two Vector QQ Plot

The function `qqplot` creates a hexbin plot using two `bdVectors`. The quantile-quantile plot is a good tool for determining a good approximation to a data set's distribution. In a `qqplot`, the ordered data are graphed against quantiles of a known theoretical distribution.

To create a sample two-vector `qqplot`, In the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqplot(fuel.bd$Mileage, runif(length(fuel.bd$Mileage),
  bigdata=T))
```

Note that in this example, the required `y` argument for `qqplot` is `runif(length(fuel.bd$Mileage)`: the random generation for the uniform distribution for the vector `fuel.bd$Mileage`. Also note that using `runif` with a big data object requires that you set the `runif` argument `bigdata=T`.

The `qqplot` plot is displayed as follows:

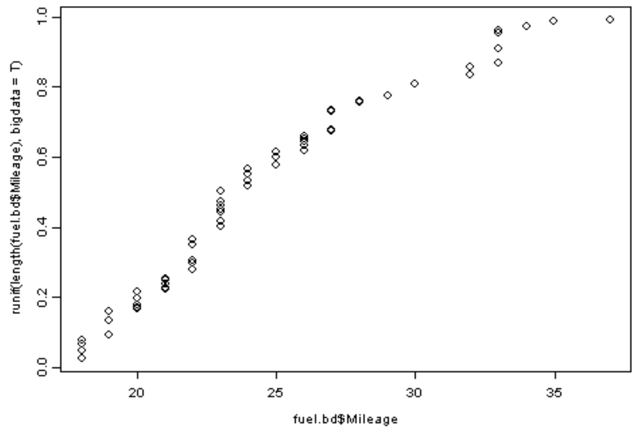

Figure 4.21: *Graph using* `qqplot`.

Create a One-Dimensional Scatter Plot

The function `stripplot` creates a Trellis graph similar to a box plot in layout; however, the individual data points are shown instead of the box plot summary.

To create sample one-dimensional scatter plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
stripplot(voice.part ~ jitter(height),
   data = singer.bd, aspect = 1,
   xlab = "Height (inches)")
```

The `stripplot` plot is displayed as follows:

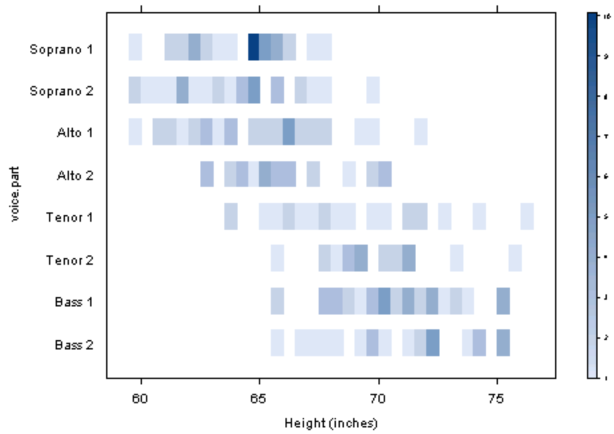

Figure 4.22: *Graph using* `stripplot` *for* `singer.bd`.

Creating Graphs with Preprocessing Functions

The functions discussed in this section do not accept a big data object directly to create a graph; rather, they require a preprocessing function such as those listed in the section Functions Providing Support to Preprocess Data for Graphing on page 84.

Create a Bar Chart

Calling `barchart` directly on a large data set produces a large number of bars, which results in an illegible plot.

- If your data contains a small number of cases, convert the data to a standard `data.frame` before calling `barchart`.
- If your data contains a large number of cases, first use `aggregate`, and then use `bd.coerce` to create the appropriate small data set.

In the following example, sum the yields over sites to get the total yearly yield for each variety.

To create a sample bar chart, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year,
  variety=barley.bd$variety), sum))
barchart(variety ~ x | year, data = temp.df,
  aspect = 0.4,xlab = "Barley Yield (bushels/acre)")
```

The resulting bar chart appears as follows:

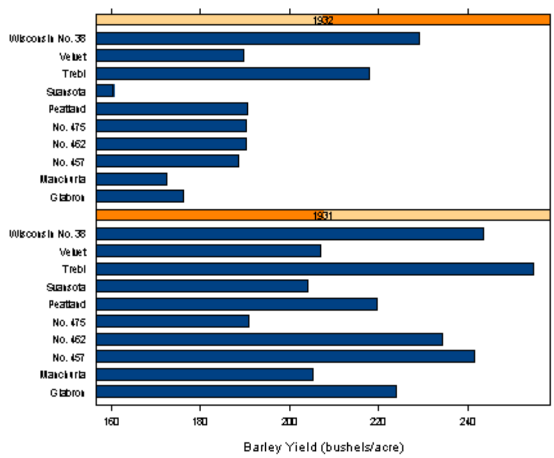

Figure 4.23: *Graph using* `barchart`.

Create a Bar Plot

The following example creates a simple bar plot from `fuel.bd`, using `table` to preprocess data.

To create a sample bar plot using `table` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
barplot(table(fuel.bd$Type), names=levels(fuel.bd$Type),
   ylab="Count")
```

The bar plot is displayed as follows:

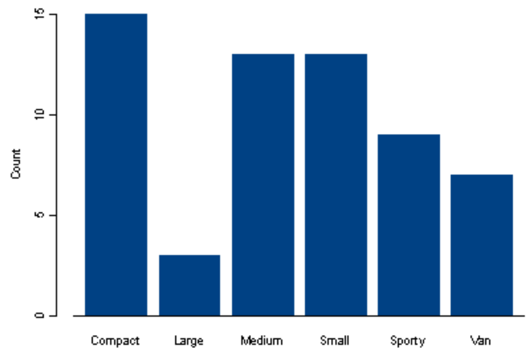

Figure 4.24: *Graph using* `barplot`.

To create a sample bar plot using `tapply` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
barplot(tapply(fuel.bd$Mileage, fuel.bd$Type, mean),
  names=levels(fuel.bd$Type), ylab="Average Mileage")
```

The bar plot is displayed as follows:

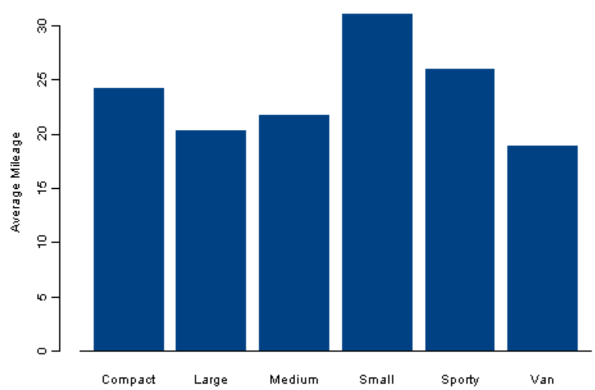

Figure 4.25: *Graph using* `tapply` *to create a bar plot.*

Create a Contour Plot

A contour plot is a representation of three-dimensional data in a flat, two-dimensional plane. Each contour line represents a height in the *z* direction from the corresponding three-dimensional surface. A level plot is essentially identical to a contour plot, but it has default options that allow you to view a particular surface differently.

The following example creates a contour plot from `fuel.bd`, using `interp` to preprocess data. For more information about `interp`, see the section Visualizing Three-Dimensional Data in the *Application Developer's Guide.*

Like `density`, `interp` and `loess` summarize the data. That is, when the data is a `bdVector`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and the mean for `x` computed in each bin. See the section Create a Density Plot on page 99 for more information.

To create a sample contour plot using `interp` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
contour(interp(fuel.bd$Weight, fuel.bd$Disp.,
  fuel.bd$Mileage))
```

The contour plot is displayed as follows:

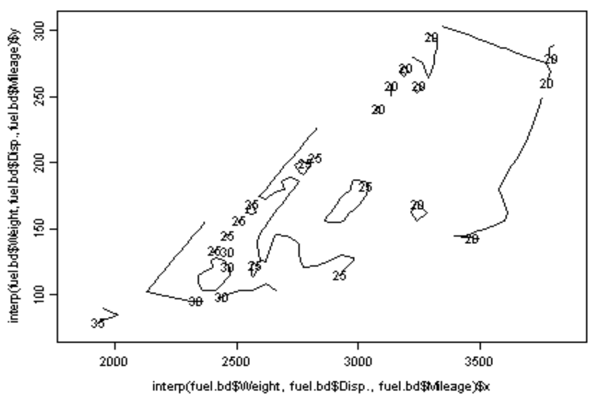

Figure 4.26: *Graph using* `interp` *to create a contour plot.*

Create a Trellis Contour Plot

The function `contourplot` creates a Trellis contour plot. The `contourplot` function creates a Trellis graph of a contour plot. For big data sets, `contourplot` requires a preprocessing function such as `loess`.

The following example creates a contour plot of predictions from `loess`.

To create a sample Trellis contour plot using `loess` to preprocess data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
     wind * temperature * radiation,data = environ.bd,
     parametric = c("radiation", "wind"),
     span = 1, degree = 2)
  w.marginal <- seq(min(environ.bd$wind),
     max(environ.bd$wind), length = 50)
  t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
  r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
  wtr.marginal <- list(wind = w.marginal,
     temperature = t.marginal, radiation = r.marginal)
  grid <- expand.grid(wtr.marginal)
  grid[, "fit"] <- c(predict(ozo.m, grid))
  print(contourplot(fit ~ wind * temperature | radiation,
     data = grid, xlab = "Wind Speed (mph)",
     ylab = "Temperature (F)",
     main = "Cube Root Ozone (cube root ppb)"))
  }
```

The Trellis contour plot is displayed as follows:

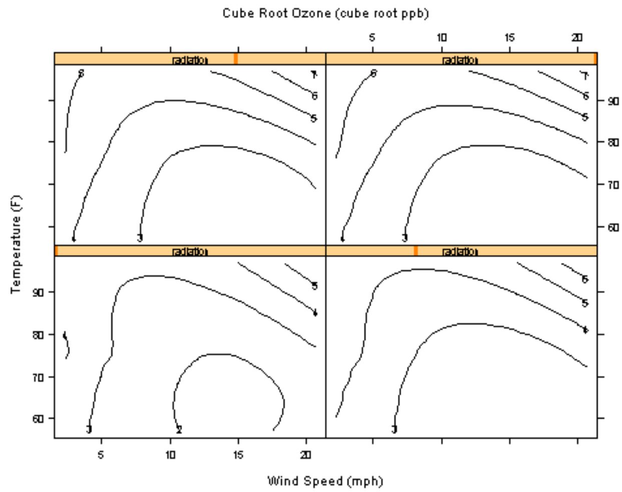

Figure 4.27: *Graph using* `loess` *to create a Trellis contour plot.*

Create a Dot Chart

When you create a dot chart, you can use a grouping variable and group summary, along with other options. The function `dotchart` can be preprocessed using either `table` or `tapply`.

To create a sample dot chart using `table` to preprocess data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
dotchart(table(fuel.bd$Type), labels=levels(fuel.bd$Type),
  xlab="Count")
```

The dot chart is displayed as follows:

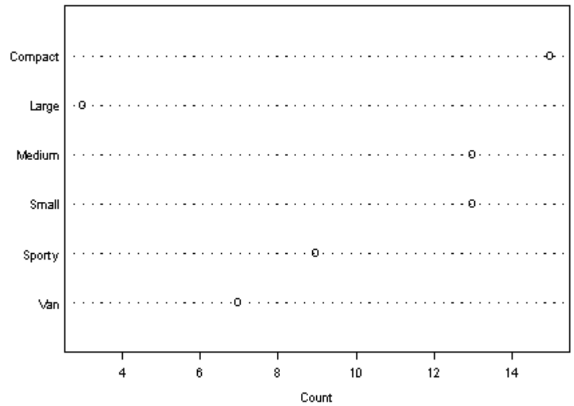

Figure 4.28: *Graph using* `table` *to create a dot chart.*

To create a sample dot chart using `tapply` to preprocess data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
dotchart(tapply(fuel.bd$Mileage, fuel.bd$Type, median),
  labels=levels(fuel.bd$Type), xlab="Median Mileage")
```

The dot chart is displayed as follows:

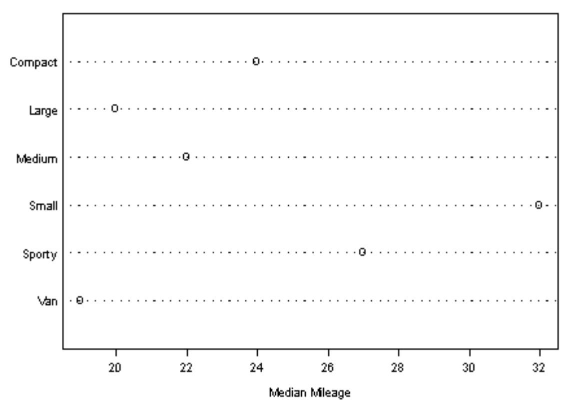

Figure 4.29: *Graph using* `tapply` *to create a dot chart.*

Create a Dot Plot

The function `dotplot` creates a Trellis graph that displays that displays dots and gridlines to mark the data values in dot plots. The dot plot reduces most data comparisons to straightforward length comparisons on a common scale.

When using `dotplot` on a big data object, call `dotplot` after using `aggregate` to reduce size of data.

In the following example, sum the barley yields over sites to get the total yearly yield for each variety.

To create a sample dot plot, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year, variety=barley.bd$variety),
  sum))
(dotplot(variety ~ x | year, data = temp.df,
  aspect = 0.4, xlab = "Barley Yield (bushels/acre)"))
```

The resulting Trellis dot plot appears as follows:

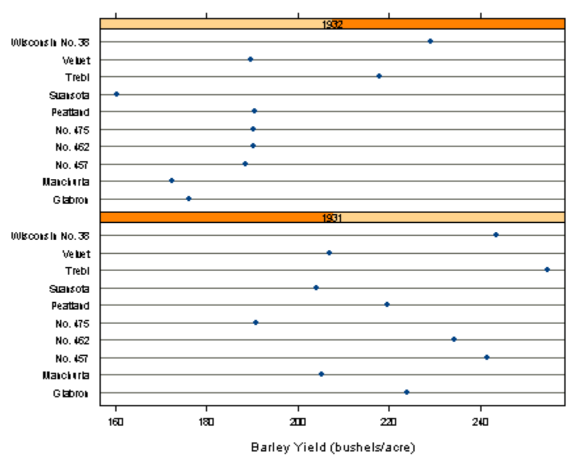

Figure 4.30: *Graph using* `aggregate` *to create a dot chart.*

Create an Image Graph Using `hist2d`

The following example creates an `image` graph using `hist2d` to preprocess data. The function `image` creates an image, under some graphics devices, of shades of gray or colors that represent a third dimension.

To create a sample image plot using `hist2d` preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
image(hist2d(fuel.bd$Weight, fuel.bd$Mileage, nx=9, ny=9))
```

The `image` plot is displayed as follows:

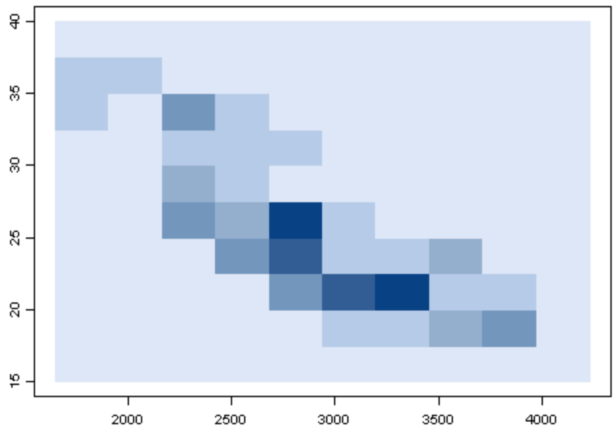

Figure 4.31: *Graph using* `hist2d` *to create an image plot.*

Create a Trellis Level Plot

The `levelplot` function creates a Trellis graph of a level plot. For big data sets, `levelplot` requires a preprocessing function such as `loess`.

A level plot is essentially identical to a contour plot, but it has default options so you can view a particular surface differently. Like contour plots, level plots are representations of three-dimensional data in flat, two-dimensional planes. Instead of using contour lines to indicate heights in the *z* direction, level plots use colors. The following example produces a level plot of predictions from `loess`.

To create a sample Trellis level plot using `loess` to preprocess the data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
      wind * temperature * radiation, data = environ.bd,
      parametric = c("radiation", "wind"),
      span = 1, degree = 2)
```

```
w.marginal <- seq(min(environ.bd$wind),
    max(environ.bd$wind), length = 50)
t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
wtr.marginal <- list(wind = w.marginal,
    temperature = t.marginal, radiation = r.marginal)
grid <- expand.grid(wtr.marginal)
grid[, "fit"] <- c(predict(ozo.m, grid))
print(levelplot(fit ~ wind * temperature | radiation,
     data = grid, xlab = "Wind Speed (mph)",
    ylab = "Temperature (F)",
    main = "Cube Root Ozone (cube root ppb)"))
}
```

The level plot is displayed as follows:

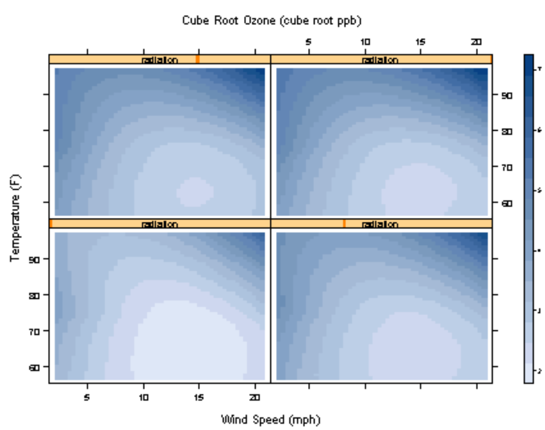

Figure 4.32: *Graph using* `loess` *to create a level plot.*

Create a `persp` Graph Using `hist2d`

The `persp` function creates a perspective plot given a matrix that represents heights on an evenly spaced grid. For more information about persp, see the section Perspective Plots in the *Application Developer's Guide*.

To create a sample `persp` graph using `hist2d` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
persp(hist2d(fuel.bd$Weight, fuel.bd$Mileage))
```

The `persp` graph is displayed as follows:

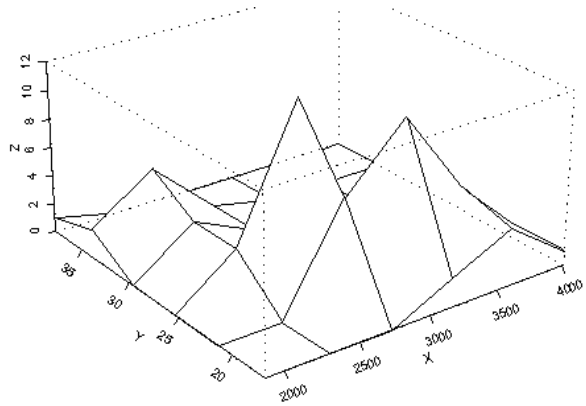

Figure 4.33: *Graph using* `hist2d` *to create a perspective plot*

Hint
Using `persp` of `interp` might produce a more attractive graph.

Create a Pie Chart

A pie chart shows the share of individual values in a variable, relative to the sum total of all the values. Pie charts display the same information as bar charts and dot plots, but can be more difficult to interpret. This is because the size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages.

Calling `pie` directly on a big data object can result in a pie with thousands of wedges; therefore, preprocess the data using `table` to reduce the number of wedges.

To create a sample pie chart using `table` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
pie(table(fuel.bd$Type), names=levels(fuel.bd$Type),
  sub="Count")
```

The pie chart appears as follows:

Figure 4.34: *Graph using* `table` *to create a pie chart.*

Create a Trellis Pie Chart

The function `piechart` creates a pie chart in a Trellis graph.

- If your data contains a small number of cases, convert the data to a standard `data.frame` before calling `piechart`.
- If your data contains a large number of cases, first use `aggregate`, and then use `bd.coerce` to create the appropriate small data set.

To create a sample Trellis pie chart using `aggregate` to preprocess the data, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year, variety=barley.bd$variety),
  sum))
piechart(variety ~ x | year, data = temp.df,
  xlab = "Barley Yield (bushels/acre)")
```

The Trellis pie chart appears as follows:

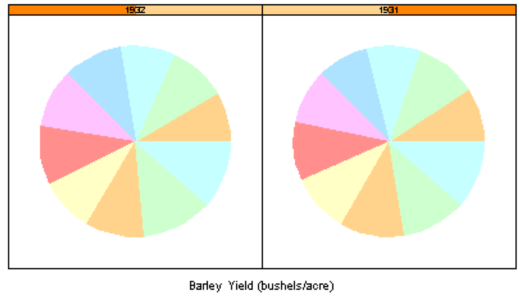

Figure 4.35: *Graph using* `aggregate` *to create a Trellis pie chart.*

Create a Trellis `Wireframe` Plot

A surface plot is an approximation to the shape of a three-dimensional data set. Surface plots are used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot the surface. The Trellis function that displays surface plots is `wireframe`.

For big data sets, `wireframe` requires a preprocessing function such as `loess`.

To create a sample Trellis surface plot using `loess` to preprocess the data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
     wind * temperature * radiation, data = environ.bd,
     parametric = c("radiation", "wind"),
     span = 1, degree = 2)
  w.marginal <- seq(min(environ.bd$wind),
     max(environ.bd$wind), length = 50)
  t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
  r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
  wtr.marginal <- list(wind = w.marginal,
     temperature = t.marginal, radiation = r.marginal)
  grid <- expand.grid(wtr.marginal)
  grid[, "fit"] <- c(predict(ozo.m, grid))
```

```
print(wireframe(fit ~ wind * temperature | radiation,
   data = grid, xlab = "Wind Speed (mph)",
   ylab = "Temperature (F)",
   main = "Cube Root Ozone (cube root ppb)"))
}
```

The surface plot is displayed as follows:

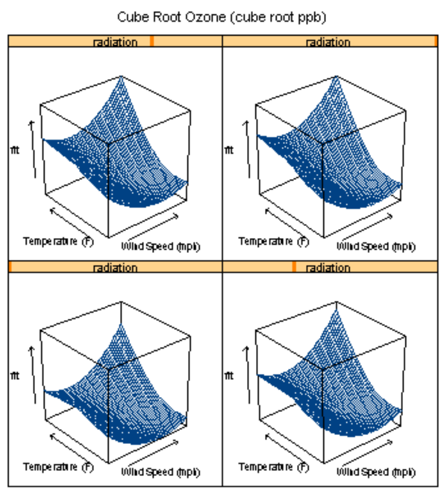

Figure 4.36: *Graph using* `loess` *to create a surface plot.*

Unsupported Functions

Using the functions that add to a plot, such as `points` and `lines`, results in an error message.

ADVANCED PROGRAMMING INFORMATION 5

INTRODUCTION

As a Spotfire S+ Big Data library user, you might encounter unexpected or unusual behavior when you manipulate blocks of data or work with strings and factors.

This section includes warnings and advice about such behavior, and provides examples and further information for handling these unusual situations.

Alternatively, you might need to implement your own big-data algorithms using out-of-memory techniques.

BIG DATA BLOCK SIZE ISSUES

Big data objects represent very large amounts of data by storing the data in external files. When a big data object is processed, pieces of this data are read into memory and processed as data "blocks." For most operations, this happens automatically. This section describes situations where you might need to understand the processing of individual blocks.

Block Size Options

When processing big data, the system must decide how much data to read and process in each block. Each block should be as big as possible, because it is more efficient to process a few large blocks, rather than many small blocks. However, the available memory limits the block size. If space is allocated for a block that is larger than the physical memory on the computer, either it uses virtual memory to store the block (which slows all operations), or the memory allocation operation fails.

The size of the blocks used is controlled by two options:

- `bd.options("block.size")`
  The option `"block.size"` specifies the maximum number of rows to be processed at a time, when executing big data operations. The default value is `1e9`; however, the actual number of rows processed is determined by this value, adjusted downwards to fit within the value specified by the option `"max.block.mb"`.

- `bd.options("max.block.mb")`
  The option `"max.block.mb"` places a limit on the maximum size of the block in megabytes. The default value is 10.

When Spotfire S+ reads a given `bdFrame`, it sets the block size initially to the value passed in `"block.size"`, and then adjusts downward until the block size is no greater than `"max.block.mb"`. Because the default for `"block.size"` is set so high, this effectively ensures that the size of the block is around the given number of megabytes.

The resulting number of rows in a block depends on the types and numbers of columns in the data. Given the default `"max.block.mb"` of 10 megabytes, reading a `bdFrame` with a single numeric column could be read in blocks of 1,250,000 rows. A `bdFrame` with 200 numeric columns could be read in blocks of 6,250 rows. The column types also enter into the determination of the number of rows in a block.

Changing Block Size Options

There is rarely a reason to change `bd.options("block.size")` or `bd.options("max.block.mb")`. The default values work well in almost all situations. In this section, we examine possible reasons for changing these values.

A bad reason for changing the block size options is to guarantee a particular block size. For example, one might set `bd.options("block.size")` to `50` before calling `bd.block.apply` with its `FUN` argument set to a function that depends on receiving blocks of exactly 50 rows. Writing functions that depend on a specific number of rows is strongly discouraged, because there are so many situations where this function might fail, including:

- If the whole dataset is not a multiple of 50 rows, then the last block will have fewer than 50 rows.
- If the dataset being processed has a large number of columns, then the actual rows in each block will be less than 50 (if `bd.options("max.block.mb")` is too small), or an out of memory error might occur when allocating the block (if `bd.options("max.block.mb")` is too high). If it is necessary to guarantee 50-row blocks, it would be better to call `bd.by.window` with `window=50`, `offset=0`, and `drop.incomplete=T`.

A good reason for changing `bd.options("block.size")` is if you are developing and debugging new code for processing big data.

Consider developing code that calls `bd.block.apply` to processes very large data in a series of chunks. To test whether this code works when the data is broken into multiple blocks, set `"block.size"` to a very small value, such as `bd.options(block.size=10)`. Test it with several small values of `bd.options("block.size")` to ensure that it does not depend on the block size. Using this technique, you can test processing multiple blocks quickly with very small data sets.

One situation where it might be necessary to increase `bd.options("max.block.mb")` is when you use `bd.by.group` or `bd.by.window`. These functions call a S-PLUS function on each data block defined by the group columns or the window size, and it will generate an error if a data block is larger than `bd.options("max.block.mb")`.

You can work around this problem by increasing `bd.options("max.block.mb")`, but you run the risk of an out of memory error. If the number of groups is not large, it would be better to call `bd.split.by.group` or `bd.split.by.window` to divide the dataset into separate datasets for each group, and then process them individually. The section Group or Window Blocks on page 128 contains an example.

A common reason for increasing `bd.options("block.size")` or `bd.options("max.block.mb")` is to attempt to improve performance. Most of the time this is not effective. While it is often faster to process a few large blocks than many small blocks, this does not mean that the best way to improve performance is to set the block size as high as possible.

With very small block sizes, a lot of time can go into the overhead of reading and writing and managing the individual blocks. As the block sizes get larger, this overhead gets lower relative to the other processing. Eventually, increasing the block size will not make much difference. This is shown in Figure 5.1, where the time for calling `bd.block.apply` on a large data set is measured for different values of `bd.options("max.block.mb")`.

`bd.options("block.size")` is set to the default of `1e9` in all cases, so the actual block size used is determined by `bd.options("max.block.mb")`. The different symbols show measurements with four different `FUN` functions. All of the symbols show the same trend: Increasing the block size improves the performance for a while, but eventually the improvement levels out.

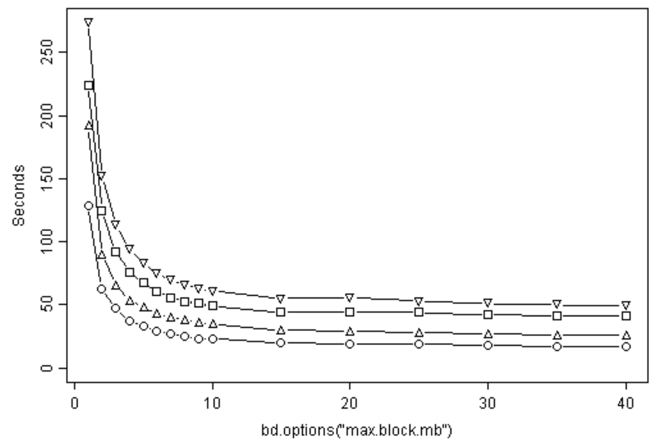

Figure 5.1: *Efficiency of setting* `bd.options("max.block.mb")`.

If you suspect that increasing the block size could help the performance of a particular computation, the best strategy is to measure the performance of the computation with `bd.options("max.block.mb")` set to the default of `10`, and then measure it again with `bd.options("max.block.mb")` set to `20`. If this test shows no significant performance improvement, it probably will not help to increase the block size further, but could lead only to out of memory problems. Using large block sizes can actually lead to worse performance, if it causes virtual memory page swapping.

Group or Window Blocks

Note that the "block" size determined by these options and the data is distinct from the "blocks" defined in the functions `bd.by.group`, `bd.by.window`, `bd.split.by.group`, and `bd.split.by.window`. These functions divide their input data into subsets to process as determined by the values in certain columns or a moving window. Spotfire S+ imposes a limit on the size of the data that can be processed in each block by `bd.by.group` and `bd.by.window`: if the number of rows in a block is larger than the block size determined by `bd.options("block.size")` and `bd.options("max.block.mb")`, an error is displayed. This limitation does not apply to the functions `bd.split.by.group` and `bd.split.by.window`.

To demonstrate this restriction, consider the code below. The variable `BIG.GROUPS` contains a 1,000-row `data.frame` with a column `GENDER` with factor values `MALE` and `FEMALE`, split evenly between the rows. If the block size is large enough, we can use `bd.by.group` to process each of the `GENDER` groups of 500 rows:

```
BIG.GROUPS <-
  data.frame(GENDER=rep(c("MALE","FEMALE"),
  length=1000), NUM=rnorm(1000))

bd.options(block.size=5000)

bd.by.group(BIG.GROUPS, by.columns="GENDER",
  FUN=function(df)
  data.frame(GENDER=df$GENDER[1],
  NROW=nrow(df)))

GENDER      NROW
1 FEMALE    500
2 MALE      500
```

If the block size is set below the size of the groups, this same operation will generate an error:

```
bd.options(block.size=10)

bd.by.group(BIG.GROUPS, by.columns="GENDER",
  FUN=function(df)
  data.frame(GENDER=df$GENDER[1],
  NROW=nrow(df)))

Problem in bd.internal.exec.node(engine.class = :
BDLManager$BDLSplusScriptEngineNode (0): Problem in
bd.internal.by.group.script(IM, function(..: can't process
block with 500 rows for group [FEMALE]: can only process 10
rows at a time (check bd.options() values for block.size
and max.block.mb)
Use traceback() to see the call stack
```

In this case, `bd.split.by.group` could be called to divide the data into a list of multiple `bdFrame` objects and process them individually:

```
BIG.GROUPS.LIST <- bd.split.by.group(BIG.GROUPS,
  by.columns="GENDER")

data.frame(GENDER=names(BIG.GROUPS.LIST),
  NROW=sapply(BIG.GROUPS.LIST, nrow, simplify=T),
  row.names=NULL)

GENDER      NROW
1 FEMALE    500
2 MALE      500
```

BIG DATA STRING AND FACTOR ISSUES

Big data columns of types `character` and `factor` have limitations that are not present for regular `data.frame` objects. Most of the time, these limitations do not cause problems, but in some situations, warning messages can appear, indicating that long strings have been truncated, or factors with too many levels had some values changed to `NA`. This section explains why these warnings may appear, and how to deal with them.

String Column Widths

When a `bdFrame` character column is initially defined, before any data is stored in it, the maximum number of characters (or string width) that can appear in the column must be specified. This restriction is necessary for rapid access to the cache file. Once this is specified, an attempt to store a longer string in the column causes the string to be truncated and generate a warning. It is important to specify this maximum string width correctly. All of the big data operations attempt to estimate this width, but there are situations where this estimated value is incorrect. In these cases, it is possible to explicitly specify the column string width.

To retrieve the actual column string widths used in a particular `bdFrame`, call the function `bd.string.column.width`.

Unless the column string width is explicitly specified in other ways, the default string width for newly-created columns is set with the following option. The default value is 32.

```
bd.options("string.column.width")
```

When you convert a `data.frame` with a character column to a `bdFrame`, the maximum string width in the column data is used to set the `bdFrame` column string width, so there is no possibility of string truncation.

String Widths and `importData`

When you import a big data object using `importData` for file types other than ASCII text, Spotfire S+ determines the maximum number of characters in each string column and uses this value to set the `bdFrame` column string width.

When you import ASCII text files, Spotfire S+ measures the maximum number of characters in each column while scanning the file to determine the column types. The number of lines scanned is controlled by the argument `scanLines`. If this is too small, and the scan stops before some very long strings, it is possible for the estimated column width to be too low. For example, the following code generates a file with steadily-longer strings.

```
f <- tempfile()
cat("strsize,str\n",file=f)
for(x in 1:30) {
   str <- paste(rep("abcd:",x),collapse="")
   cat(nchar(str), ",", str, "\n", sep="",
   append=T, file=f)
 }
```

Importing this file with the default `scanLines` value (256) detects that the maximum string has 150 characters, and sets this column string length correctly.

```
dat <- importData(f, type="ASCII", stringsAsFactors=F,
   bigdata=T)
dat
```

```
**bdFrame: 30 rows, 2 columns**
strsize      str
1  5        abcd:
2  10     abcd:abcd:
3  15    abcd:abcd:abcd:
4  20   abcd:abcd:abcd:abcd:
5  25  abcd:abcd:abcd:abcd:abcd:
   ... 25 more rows ...
```

```
bd.string.column.width(dat)
```

```
     strsize       str
     -1           150
```

(In the above output, the `strsize` value of `-1` represents the value for non-character columns.)

If you import this file with the `scanLines` argument set to scan only the first few lines, the column string width is set too low. In this case, the column string width is set to 45 characters, so longer strings are truncated, and a warning is generated:

```
dat <- importData(f, type="ASCII", stringsAsFactors=F,
   bigdata=T, scanLines=10)
```

```
Warning messages:
"ReadTextFileEngineNode (0): output column str has 21
string values truncated because they were longer than the
column string width of 45 characters -- maximum string size
before truncation was 150 characters" in:
bd.internal.exec.node(engine.class = engine.class, ...
```

You can read this data correctly without scanning the entire file by explicitly setting `bd.options("default.string.column.width")` before the call to `importData`:

```
bd.options("default.string.column.width"=200)
dat <- importData(f, type="ASCII", stringsAsFactors=F,
   bigdata=T, scanLines=10)
bd.string.column.width(dat)

strsize           str
 -1               200
```

This string truncation does not occur when Spotfire S+ reads long strings as factors, because there is no limit on factor-level string length.

One more point to remember when you import strings: the low-level `importData` and `exportData` code truncates any strings (either character strings or factor levels) that have more than 254 characters. Spotfire S+ generates a warning in `importData` if `bigdata=T` if it encounters such strings.

String Widths and `bd.create.columns`

You can use one of the following techniques for setting string column widths explicitly:

- To set the default width (if it is not determined some other way), use `bd.options("string.column.width")`.
- To override the default column string widths, in `bd.block.apply`, specify the `out1.column.string.widths` list element when `IM$test==T`, or when outputting the first non-`NULL` output block.
- To set the width for new output columns, use the `string.column.width` argument to `bd.create.columns`. When you use `bd.create.columns` to create a new character

column, you must set the column string width. You can set this width explicitly with the `string.column.width` argument. If you set it smaller than the maximum string generated, then this will generate a warning:

```
bd.create.columns(as.bdFrame(fuel.frame),
    "Type+Type", "t2", "character",
    string.column.width=6)
```

```
Warning in bd.internal.exec.node(engine.class = engi..:
"CreateColumnsEngineNode (0): output column t2 has 53
string values truncated because they were longer than the
column string width of 6 characters -- maximum string size
before truncation was 14 characters"

**bdFrame: 60 rows, 6 columns**
  Weight Disp. Mileage  Fuel   Type  t2
1  2560    97    33    3.030303 Small SmallS
2  2345   114    33    3.030303 Small SmallS
3  1845    81    37    2.702703 Small SmallS
4  2260    91    32    3.125000 Small SmallS
5  2440   113    32    3.125000 Small SmallS
... 55 more rows ...
```

If the character column width is not set with the `string.column.width` argument, the value is estimated differently, depending on whether the `call.splus` argument is true or false. If `row.language=T`, the expression is analyzed to determine the maximum length string that could possibly be generated. This estimate is not perfect, but it works well enough most of the time.

If `row.language=F`, the first time that the S-PLUS expression is evaluated, the string widths are measured, and the new column's string width is set from this value. If future evaluations produce longer strings, they are truncated, and a warning is generated.

Whether `row.language=T` or `F`, the estimated string widths will never be less than the value of `bd.options("default.string.column.width")`.

Factor Column Levels

Because of the way that `bdFrame` factor columns are represented, a factor cannot have an unlimited number of levels. The number of levels is restricted to the value of the option. (The default is 500.)

```
bd.options("max.levels")
```

If you attempt to create a factor with more than this many levels, a warning is generated. For example:

```
dat <- bd.create.columns(data.frame(num=1:2000),
    "'x'+num", "f", "factor")
```

```
Warning messages:
"CreateColumnsEngineNode (0): output column f has 1500 NA
values due to categorical level overflow (more than 500
levels) -- you may want to change this column type from
categorical to string" in: bd.internal.ex\
ec.node(engine.class = engine.class, node.props =
node.props, ....
```

```
summary(dat)
```

```
num           f
Min.:  1.0      x99: 1
1st Qu.: 500.8    x98: 1
Median: 1001.0    x97: 1
Mean: 1001.0     x96: 1
3rd Qu.: 1500.0    x95: 1
Max.: 2000.0 (Other): 495
     NA's:1500
```

You can increase the `"max.levels"` option up to 65,534, but factors with so many levels should probably be represented as character strings instead.

Note
Strings are used for identifiers (such as street addresses or social security numbers), while factors are used when you have a limited number of categories (such as state names or product types) that are used to group rows for tables, models, or graphs.

String Truncation and Level Overflow Errors

Normally, if strings are truncated or factor levels overflow, Spotfire S+ displays a warning with detailed information on the number of altered values after the operation is completed. You can set the following options to make an error occur immediately when a string truncation or level overflow occurs.

```
bd.options("error.on.string.truncation"=T)
bd.options("error.on.level.overflow"=T)
```

The default for both options is `F`. If one of these is set to `T`, an error occurs, with a short error message. Because all of the data has not been processed, it is impossible to determine how many values might be effected.

These options are useful in situations where you are performing a lengthy operation, such as importing a huge data set, and you want to terminate it immediately if there is a possible problem.

STORING AND RETRIEVING LARGE S OBJECTS

When you work with very large data, you might encounter a situation where an object or collection of objects is too large to fit into available memory. The Big Data library offers two functions to manage storing and retrieving large data objects:

- `bd.pack.object`
- `bd.unpack.object`

This topic contains examples of using these functions.

Managing Large Amounts of Data

Suppose you want to create a list containing thousands of model objects, and a single list containing all of the models is too large to fit in your available memory. By using the function `bd.pack.object`, you can store each model in an external cache, and create a list of the smaller "packed" models. You can then use `bd.unpack.object` to restore the models to manipulate them.

Creating a Packed Object with `bd.pack.object`

In the following example, use the data object `fuel.frame` to create 1000 linear models. The resulting object takes about 6MB.

In the **Commands** window, type the following:

```
#Create the linear models:
many.models <- lapply(1:1000, function(x)
  lm(Fuel ~ Weight + Disp., sample(fuel.frame, size=30)))

#Get the size of the object:
object.size(many.models)

[1] 6210981
```

You can make a smaller object by packing each model. While this exercise takes longer, the resulting object is smaller than 2MB.

In the **Commands** window, type the following:

```
#Create the packed linear models:
many.models.packed <- lapply(1:1000,
  function(x) bd.pack.object(
  lm(Fuel ~ Weight + Disp., sample(fuel.frame, size=30))))
```

```
#Get the size of the packed object:
object.size(many.models.packed)

[1] 1880041
```

Restoring a Packed Object with `bd.unpack.object`

Remember if you use `bd.pack.object`, you must unpack the object to use it again. The following example code unpacks some of the models within `many.models.packed` object and displays them in a plot.

In the **Commands** window, type the following:

```
for(x in 1:5)
  plot(
  bd.unpack.object(many.models.packed[[x]]),
  which.plots=3)
```

Summary

The above example shows a space difference of only a few MB, (6MB to 2MB), which is probably not a large enough saving to take the time to pack the object. However, if each of the model objects were very large, and the whole list were too large to represent, the packed version would be useful.

INCREASING EFFICIENCY

The Big Data library offers several alternatives to standard S-PLUS functions, to provide greater efficiency when you work with a large data set. Key efficiency functions include:

Table E.1: *Efficient Big Data library functions.*

Function name	Description
`bd.select.rows`	Use to extract specific columns and a block of contiguous rows.
`bd.filter.rows`	Use to keep all rows for which a condition is `TRUE`.
`bd.create.columns`	Use to add columns to a data set.

The following section provides comparisons between these Big Data library functions and their standard S-PLUS function equivalents

bd.select.rows

Using `bd.select.rows` to extract a block of rows is much more efficient than using standard subscripting. Some standard subscripting and `bd.select.rows` equivalents include the following:.

Table E.2: *`bd.select.rows` efficiency equivalents.*

Standard S-PLUS subscripting function	`bd.select.rows` **equivalent**
`x[, "Weight"]`	`bd.select.rows(x, columns="Weight")`
`x[1:1000, c(1,3)]`	`bd.select.rows(x, from=1, to=1000, columns=c(1,3))`

bd.filter.rows

Using `bd.filter.rows` is equivalent to subscripting rows with a logical vector. By default, `bd.filter.rows` uses an “expression language” that provides quick evaluation of row-oriented expressions. Alternatively, you can use the full range of S-PLUS row functions by setting the `bd.filter.rows` argument `row.language=F`, but the computation is less efficient. Some standard subscripting and `bd.filter.rows` equivalents include the following:.

Table E.3: *`bd.filter.rows` efficiency equivalents.*

Standard S-PLUS subscripting function	`bd.filter.rows` **equivalent**
`x[x$Weight > 100, ]`	`bd.filter.rows(x, "Weight > 100")`
`x[pnorm(x$stat) > 0.5 ,]`	`bd.filter.rows(x, "pnorm(stat) > 0.5", row.language=F)`

`bd.create.columns`

Like `bd.filter.rows`, `bd.create.columns` offers you a choice of using the more efficient expression language or the more flexible general S-PLUS functions. Some standard subscripting and `bd.create.columns` equivalents include the following:

Table E.4: *`bd.create.columns` efficiency equivalents.*

Standard S-PLUS subscripting function	`bd.create.columns` **equivalent**
`x$d <- (x$a+x$b)/x$c`	`x <- bd.create.columns(x, "(a+b)/c", "d")`
`x$pval <- pnorm(x$stat)`	`x <- bd.create.columns(x, "pnorm(stat)", "pval", row.language=F)`
`y <- (x$a+x$b)/x$c`	`y <- bd.create.columns(x, "(a+b)/c", "d", copy=F)`

Note that in the last function, above, specifying `copy=F` creates a new column without copying the old columns.

APPENDIX: BIG DATA LIBRARY FUNCTIONS

INTRODUCTION

The Big Data library is supported by many standard S-PLUS functions, such as basic statistical and mathematical functions, properties functions, densities and quantiles functions, and so on. For more information about these functions, see their individual help topics. (To display a function's help topic, in the **Commands** window, type `help(`*`functionname`*`)`.)

The Big Data library also contains functions specific to big data objects. These functions include the following.

- Import and export functions.
- Object creation functions
- Big vector generating functions.
- Data exploration and manipulation functions.
- Traditional and Trellis graphics functions.
- Modeling functions.

These functions are described further in the following section.

BIG DATA LIBRARY FUNCTIONS

The following tables list the functions that are implemented in the Big Data library.

Data Import and Export

For more information and usage examples, see the functions' individual help topics.

Table A.1: *Import and export functions.*

Function name	**Description**
`data.dump`	Creates a file containing an ASCII representation of the objects that are named.
`data.restore`	Puts data objects that had previously been put into a file with `data.dump` into the specified database.
`exportData`	Exports a `bdFrame` to the specified file or database format. Not all standard S-PLUS arguments are available when you import a large data set. See `exportData` in the Spotfire S+ Language Reference for more information.
`importData`	When you set the `bigdata` flag to `TRUE`, imports data from a file or database into a `bdFrame`. Not all standard S-PLUS arguments are available when you import a large data set. See `importData` in the Spotfire S+ Language Reference for more information.

Object Creation

The following methods create an object of the specified type. For more information and usage examples, see the functions' individual help topics.

Table A.2: *Big Data library object creation functions*

Function
bdCharacter
bdCluster
bdFactor
bdFrame
bdGlm
bdLm
bdLogical
bdNumeric
bdPrincomp
bdSignalSeries
bdTimeDate
bdTimeSeries
bdTimeSpan

Big Vector Generation

For the following methods, set the bigdata argument to TRUE to generate a bdVector. This instruction applies to all functions in this table. For more information and usage examples, see the functions' individual help topics.

Table A.3: *Vector generation methods for large data sets.*

Method name
rbeta
rbinom
rcauchy
rchisq
rep
rexp
rf
rgamma
rgeom
rhyper
rlnorm
rlogis
rmvnorm
rnbinom
rnorm

Table A.3: *Vector generation methods for large data sets. (Continued)*

Method name
rnrange
rpois
rstab
rt
runif
rweibull
rwilcox

Big Data Library Functions

The Big Data library introduces a new set of "bd" functions designed to work efficiently on large data. For best performance, it is important that you write code minimizing the number of passes through the data. The Big Data library functions minimize the number of passes made through the data. Use these functions for the best performance. For more information and usage examples, see the functions' individual help topics.

Data Exploration Functions

Table A.4: *Data exploration functions.*

Function name	Description
`bd.cor`	Computes correlation or covariances for a data set. In addition, computes correlations or covariances between a single column and all other columns, rather than computing the full correlation/covariance matrix.
`bd.crosstabs`	Produces a series of tables containing counts for all combinations of the levels in categorical variables.
`bd.data.viewer`	Displays the data viewer window, which displays the input data in a scrollable window, as well as information about the data columns (names, types, means, and so on).
`bd.univariate`	Computes a wide variety of univariate statistics. It computes most of the statistics returned by `PROC UNIVARIATE` in SAS.

Data Manipulation Functions

Table A.5: *Data manipulation functions.*

Function name	Description
`bd.aggregate`	Divides a data object into blocks according to the values of one or more columns, and then applies aggregation functions to columns within each block.
`bd.append`	Appends one data set to a second data set.
`bd.bin`	Creates new categorical variables from continuous variables by splitting the numeric values into a number of bins. For example, it can be used to include a continuous age column as ranges (<18, 18-24, 25-35, and so on).
`bd.block.apply`	Executes a S-PLUS script on blocks of data, with options for reading multiple input datasets and generating multiple output data sets, and processing blocks in different orders.
`bd.by.group`	Apply an arbitrary S-PLUS function to multiple data blocks within the input dataset.
`bd.by.window`	Apply an arbitrary S-PLUS function to multiple data blocks defined by a moving window over the input dataset.
`bd.coerce`	Converts an object from a standard data frame to a `bdFrame`, or vice versa.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.create.columns`	Creates columns based on expressions.
`bd.duplicated`	Determine which rows in a dataset are unique.
`bd.filter.columns`	Removes one or more columns from a data set.
`bd.filter.rows`	Filters rows that satisfy the specified expression.
`bd.join`	Creates a composite data set from two or more data sets. For each data set, specify a set of key columns that defines the rows to combine in the output. Also, for each data set, specify whether to output unmatched rows.
`bd.modify.columns`	Changes column names or types. Can also be used to drop columns.
`bd.normalize`	Centers and scales continuous variables. Typically, variables are normalized so that they follow a standard Gaussian distribution (means of 0 and standard deviations of 1). To do this, `bd.normalize` subtracts the mean or median, and then divides by either the range or standard deviation.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.partition`	Randomly samples the rows of your data set to partition it into three subsets for training, testing, and validating your models.
`bd.relational.difference`	Get differing rows from two input data sets.
`bd.relational.divide`	Given a `Value` column and a `Group` column, determine which values belong to a given Membership as defined by a set of `Group` values.
`bd.relational.intersection`	Join two input data sets, ignoring all unmatched columns, with the common columns acting as key columns.
`bd.relational.join`	Join two input data sets with the common columns acting as key columns.
`bd.relational.product`	Join two input data sets, ignoring all matched columns, by performing the cross product of each row.
`bd.relational.project`	Remove one or more columns from a data set.
`bd.relational.restrict`	Select the rows that satisfy an expression. Determines whether each row should be selected by evaluating the restriction. The result should be a logical value.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.relational.union`	Retrieve the relational union of two data sets. Takes two inputs (`bdFrame` or `data.frame`). The output contains the common columns and includes the rows from both inputs, with duplicate rows eliminated.
`bd.remove.missing`	Drops rows with missing values, or replaces missing values with the column mean, a constant, or values generated from an empirical distribution, based on the observed values.
`bd.reorder.columns`	Changes the order of the columns in the data set.
`bd.sample`	Samples rows from a dataset, using one of several methods.
`bd.select.rows`	Extracts a block of data, as specified by a set of columns, start row, and end row.
`bd.shuffle`	Randomly shuffles the rows of your data set, reordering the values in each of the columns as a result
`bd.sort`	Sorts the data set rows, according to the values of one or more columns.
`bd.split`	Splits a data set into two data sets according to whether each row satisfies an expression.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.sql`	Specifies data manipulation operations using SQL syntax. • The `Select`, `Insert`, `Delete`, and `Update` statements are supported. • The column identifiers are case sensitive. • SQL interprets periods in names as indicating fields within tables; therefore, column names should not contain periods if you plan to use `bd.sql`. • Mathematical functions are allowed for aggregation (avg, min, max, sum, count, stdev, var). The following functionality is not implemented: • distinct • mathematical functions in set or select, such as `abs`, `round`, `floor`, and so on. • natural join • union • merge • between • subqueries You can use the `WHERE` clause only on the first referenced data table in a SQL statement.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.stack`	Combines or stacks separate columns of a data set into a single column, replicating values in other columns as necessary.
`bd.string.column.width`	Returns the maximum number of characters that can be stored in a big data string column.
`bd.transpose`	Turns a set of columns into a set of rows.
`bd.unique`	Remove all duplicated rows from the dataset so that each row is guaranteed to be unique.
`bd.unstack`	Separates one column into a number of columns based on a grouping column.

Programming

Table A.6: *Programming functions.*

Function name	Description
`bd.cache.cleanup`	Cleans up cache files that have not been deleted by the garbage collection system. (This is most likely to occur if the entire system crashes.)
`bd.cache.info`	Analyzes a directory containing big data cache files and returns information about cache files, references counts, and unknown files.
`bd.options`	Controls S-PLUS options used when processing big data objects.
`bd.pack.object`	Packs any object into an external cache.
`bd.split.by.group`	Divide a dataset into multiple data blocks, and return a list of these data blocks.
`bd.split.by.window`	Divide a dataset into multiple data blocks, defined by a moving window over the dataset, and return a list of these data blocks.
`bd.unpack.object`	Unpacks a `bdPackedObject` object that was previously stored in the cache using `bd.pack.object`.

Data Frame and Vector Functions

The following table lists the functions for both data frames (`bdFrame`) and vectors (`bdVector`). The the cross-hatch (#) indicates that the function is implemented for the corresponding object type. The Comment column provides information about the function, or indicates which `bdVector`-derived class(es) the function applies to. For more information and usage examples, see the functions' individual help topics.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`.*

Function Name	bdVector	bdFrame	Optional Comment
-	#	#	
!=	#	#	
$		#	
$<-		#	
[	#	#	
[[	#	#	
[[<-	#	#	
[<-	#	#	
abs	#		
aggregate	#	#	
all	#	#	
all.equal	#	#	
any	#	#	
anyMissing	#	#	
append	#		
apply		#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`Arith`	#	#	
`as.bdCharacter`	#		
`as.bdFactor`	#		
`as.bdFrame`	#	#	
`as.bdLogical`	#		Handles all `bdVector`-derived object types.
`as.bdVector`	#	#	
`attr`	#	#	
`attr<-`	#	#	
`attributes`	#	#	
`attributes<-`	#	#	
`bdFrame`	#	#	Constructor. Inputs can be `bdVectors`, `bdFrames`, or ordinary objects.
`boxplot`	#	#	Handles `bdNumeric`.
`by`		#	
`casefold`	#		
`ceiling`	#		
`coerce`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
colIds		#	
colIds<-		#	
colMaxs	#	#	
colMeans	#	#	
colMins	#	#	
colRanges	#	#	
colSums	#	#	
colVars	#	#	
concat.two	#	#	
cor	#	#	
cut	#		
dbeta	#		Density, cumulative distribution (CDF), and quantile function.
dbinom	#		Density, CDF, and quantile function.
dcauchy	#		Density, CDF, and quantile function.
dchisq	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
`density`	#		
`densityplot`		#	
`dexp`	#		Density, CDF, and quantile function.
`df`	#		Density, CDF, and quantile function.
`dgamma`	#		Density, CDF, and quantile function.
`dgeom`	#		Density, CDF, and quantile function.
`dhyper`	#		Density, CDF, and quantile function.
`diff`	#	#	
`digamma`	#		
`dim`		#	
`dimnames`		#	a `bdFrame` has no row names.
`dimnames<-`		#	a `bdFrame` has no row names.
`dlnorm`	#		Density, CDF, and quantile function.
`dlogis`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
dmvnorm		#	Density and CDF function.
dnbinom	#		Density, CDF, and quantile function.
dnorm	#		Density, CDF, and quantile function.
dnrange	#		Density, CDF, and quantile function.
dpois	#		Density, CDF, and quantile function.
dt	#		Density, CDF, and quantile function.
dunif	#		Density, CDF, and quantile function.
duplicated	#	#	Density, CDF, and quantile function.
durbinWatson	#		Density, CDF, and quantile function.
dweibull	#		Density, CDF, and quantile function.
dwilcox	#		Density, CDF, and quantile function.
floor	#	#	
format	#	#	

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`. *(Continued)*

Function Name	bdVector	bdFrame	Optional Comment
formula		#	
grep	#		
hist	#		
hist2d	#		
histogram		#	
html.table	#	#	
intersect	#		
is.all.white	#		
is.element	#		
is.finite	#	#	
is.infinite	#	#	
is.na	#	#	
is.nan	#	#	
is.number	#	#	
is.rectangular	#	#	
kurtosis	#		Handles `bdNumeric`.
length	#	#	

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
`levels`	#		Handles `bdFactor`.
`levels<-`	#		Handles `bdFactor`.
`mad`	#		
`match`	#	#	
`Math`	#	#	Operand function.
`Math2`	#	#	Operand function.
`matrix`	#	#	
`mean`	#	#	
`median`	#		
`merge`	#	#	
`na.exclude`	#	#	
`na.omit`	#	#	
`names`	#	#	`bdVector` cannot have names.
`names<-`	#	#	`bdVector` cannot have names.
`nchar`	#		Handles `bdCharacter`, not `bdFactor`.
`ncol`		#	

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
notSorted	#		
nrow		#	
numberMissing	#	#	
Ops	#	#	
pairs		#	
pbeta	#		Density, CDF, and quantile function.
pbinom	#		Density, CDF, and quantile function.
pcauchy	#		Density, CDF, and quantile function.
pchisq	#		Density, CDF, and quantile function.
pexp	#		Density, CDF, and quantile function.
pf	#		Density, CDF, and quantile function.
pgamma	#		Density, CDF, and quantile function.
pgeom	#		Density, CDF, and quantile function.
phyper	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`. *(Continued)*

Function Name	bdVector	bdFrame	Optional Comment
`plnorm`	#		Density, CDF, and quantile function.
`plogis`	#		Density, CDF, and quantile function.
`plot`	#	#	
`pmatch`	#		
`pmvnorm`		#	Density and CDF function.
`pnbinom`	#		Density, CDF, and quantile function.
`pnorm`	#		Density, CDF, and quantile function.
`pnrange`	#		Density, CDF, and quantile function.
`ppois`	#		Density, CDF, and quantile function.
`print`	#	#	
`pt`	#		Density, CDF, and quantile function.
`punif`	#		Density, CDF, and quantile function.
`pweibull`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`pwilcox`	#		Density, CDF, and quantile function.
`qbeta`	#		Density, CDF, and quantile function.
`qbinom`	#		Density, CDF, and quantile function.
`qcauchy`	#		Density, CDF, and quantile function.
`qchisq`	#		Density, CDF, and quantile function.
`qexp`	#		Density, CDF, and quantile function.
`qf`	#		Density, CDF, and quantile function.
`qgamma`	#		Density, CDF, and quantile function.
`qgeom`	#		Density, CDF, and quantile function.
`qhyper`	#		Density, CDF, and quantile function.
`qlnorm`	#		Density, CDF, and quantile function.
`qlogis`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
qnbinom	#		Density, CDF, and quantile function.
qnorm	#		Density, CDF, and quantile function.
qnrange	#		Density, CDF, and quantile function.
qpois	#		Density, CDF, and quantile function.
qq		#	
qqmath		#	
qqnorm	#		
qqplot	#		
qt	#		Density, CDF, and quantile function.
quantile	#		
qunif	#		Density, CDF, and quantile function.
qweibull	#		Density, CDF, and quantile function.
qwilcox	#		Density, CDF, and quantile function.
range	#		

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
rank	#		
replace	#		
rev	#	#	
rle	#		
row.names		#	Always NULL.
row.names<-		#	Does nothing.
rowIds		#	Always NULL.
rowIds<-		#	Does nothing.
rowMaxs		#	
rowMeans		#	
rowMins		#	
rowRanges		#	
rowSums		#	
rowVars		#	
runif	#		
sample	#	#	
scale		#	

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`. *(Continued)*

Function Name	bdVector	bdFrame	Optional Comment
setdiff	#		
shiftPositions	#		
show	#	#	
skewness	#		Handles `bdNumeric`.
sort	#		
split		#	
stdev	#		Handles `bdCharacter`.
sub	#	#	
sub<-		#	
substring	#		
substring<-	#		
Summary	#	#	Operand function.
summary	#	#	
sweep		#	
t		#	
tabulate	#		Handles `bdNumeric`.
tapply	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
trigamma	#		
union	#		
unique	#	#	
var	#	#	
which.infinite	#	#	
which.na	#	#	
which.nan	#	#	
xy2cell	#		
xyCall	#		
xyplot		#	

Graph Functions

For more information and examples for using the traditional graph functions, see their individual help topics, or see the section Functions Supporting Graphs on page 81.

Table A.8: *Traditional graph functions.*

Function name
barplot
boxplot
contour
dotchart

Table A.8: *Traditional graph functions. (Continued)*

Function name
hexbin
hist
hist2d
image
interp
pairs
persp
pie
plot
qqnorm
qqplot

For more information about using the Trellis graph functions, see their individual help topics, or see the section Functions Supporting Graphs on page 81.

Table A.9: *Trellis graph functions.*

Function name
barchart
contourplot
densityplot
dotplot

Table A.9: *Trellis graph functions. (Continued)*

Function name
`histogram`
`levelplot`
`piechart`
`qq`

Note
The `cloud` and `parallel` graphics functions are not implemented for `bdFrames`.

Data Modeling

For more information and usage examples, see the functions' individual help topics.

Table A.10: *Fitting functions*

Function name
`bdCluster`
`bdGlm`
`bdLm`
`bdPrincomp`

Table A.11: *Other modeling utilities.*

Function name
bd.model.frame.and.matrix
bs
ns
spline.des
C
contrasts
contrasts<-

Model Methods

The following table identifies functions implemented for generalized linear modeling, linear regression, principal components modeling, and clustering. The cross-hatch (#) indicates the function is implemented for the corresponding modeling type.

Table A.12: *Modeling and Clustering Functions.*

Function name	Generalized linear modeling (bdGlm)	Linear Regression (bdLm)	principal components (bdPrincomp)	bdCluster
AIC		#		
all.equal		#		
anova	#	#		
BIC		#		
coef	#	#		
deviance	#	#		

Table A.12: *Modeling and Clustering Functions. (Continued)*

Function name	**Generalized linear modeling** (bdGlm)	**Linear Regression** (bdLm)	**principal components** (bdPrincomp)	bdCluster
durbinWatson		#		
effects		#		
family	#	#		
fitted	#	#	#	#
formula	#	#		
kappa		#		
labels		#		
loadings			#	
logLik		#		
model.frame		#		
model.matrix		#		
plot		#	#	
predict	#	#	#	#
print	#	#	#	#
print.summary	#	#	#	
qqnorm	#	#		
residuals	#	#		

Table A.12: *Modeling and Clustering Functions. (Continued)*

Function name	Generalized linear modeling (bdGlm)	Linear Regression (bdLm)	principal components (bdPrincomp)	bdCluster
screeplot			#	
step	#	#		
summary	#	#	#	

Predict from Small Data Models

This table lists the small data models that support the `predict` function. For more information and usage examples, see the functions' individual help topics.

Table A.13: *Predicting from small data models.*

Small data model using `predict` function
arima.mle
bs
censorReg
coxph
coxph.penal
discrim
factanal
gam
glm

Table A.13: *Predicting from small data models. (Continued)*

Small data model using `predict` **function**
`gls`
`gnls`
`lm`
`lme`
`lmList`
`lmRobMM`
`loess`
`loess.smooth`
`mlm`
`nlme`
`nls`
`ns`
`princomp`
`safe.predict.gam`
`smooth.spline`
`smooth.spline.fit`
`survreg`

Table A.13: *Predicting from small data models. (Continued)*

Small data model using `predict` **function**
`survReg`
`survReg.penal`
`tree`

Time Date and Series Functions

The following tables include time date creation functions and functions for manipulating time and date, time span, time series, and signal series objects.

Time Date Creation

Table A.14: *Time date creation functions.*

Function name	**Description**
`bdTimeDate`	The object constructor. Note that when you call the `timeDate` function with any big data arguments, then a `bdTimeDate` object is created.
`timeCalendar`	Standard S-PLUS function. When you call the `timeCalendar` function with any big data arguments, then a `bdTimeDate` object is created
`timeSeq`	Standard S-PLUS function; to use with a large data set, set the `bigdata` argument to `TRUE`.

In the following table, the cross-hatch (#) indicates that the function is implemented for the corresponding class. If the table cell is blank, the function is not implemented for the class. This list includes `bdVector` objects (`bdTimeDate` and `bdTimeSpan`) and `bdSeries` classes (`bdSignalSeries`, `bdTimeSeries`).

Table A.15: *Time Date and Series Functions.*

Function	`bdTimeDate`	`bdTimeSpan`	`bdSignalSeries`	`bdTimeSeries`
`-`	#	#		
`[`		#	#	#
`[<-`		#		
`+`	#	#		
`align`			#	#
`all.equal`	#	#		
`Arith`	#	#		
`as.bdFrame`	#	#		#
`as.bdLogical`	#	#		
`bd.coerce`	#	#	#	#
`ceiling`	#	#		
`coerce/as`	#	#	#	#
`cor`	#	#	#	#
`cumsum`		#		
`cut`	#	#		

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSignalSeries	bdTimeSeries
data.frameAux	#	#		#
days	#			
deltat			#	#
diff			#	#
end			#	#
floor	#	#		
hms	#			
hours	#			
match	#	#		
Math	#	#	#	#
Math2	#	#	#	#
max	#	#		
mdy	#			
mean	#	#	#	#
median	#	#	#	#
min	#	#		
minutes	#			

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSignalSeries	bdTimeSeries
months	#			
plot	#	#	#	#
quantile	#	#	#	#
quarters	#			
range	#	#		
seconds	#			
seriesLag			#	#
shiftPositions	#	#		
show	#	#	#	#
sort	#	#	#	#
sort.list	#	#	#	#
split	#	#		
start			#	#
substring<-	#	#	#	#
sum		#		
Summary	#	#	#	#
summary	#	#	#	#

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSignalSeries	bdTimeSeries
timeConvert	#			
trunc	#	#		
var	#	#	#	#
wdydy	#			
weekdays	#			
yeardays	#			
years	#			

INDEX

O

P

Q

R

S

T

U

V

W

X

Y