Chapter 4

Loading Data

Data can come in many different formats from many different sources. By using R's extensive capabilities, Rattle provides direct access to such data. Indeed, we are fortunate with the R system in that it is an open system and therefore is strong on sharing and cooperating with other applications. R supports importing data in many formats.

One of the most common formats for data exchange between applications is the comma-separated value (CSV) file. Such files typically have a `csv` filename extension. This is a simple text file format that is oriented around rows and columns, using a comma to separate the columns in the file. Such files can be used to transfer data through export and import between spreadsheets, databases, weather monitoring stations, and many other applications. A variation on the idea is to separate the columns with other markers, such as a tab character, which is often associated with files having a `txt` filename extension.

These simple data files (the CSV and TXT files) contain no explicit metadata information—that is, there is no data to describe the structure of the data contained in the file. That information often needs to be guessed at by the software reading the data.

Other types of data sources do provide information about the data so that our software does not need to make guesses about what it is reading. Attribute-Relation File Format files (Section 4.2) have an `arff` filename extension and add metadata to the CSV format.

Extracting data directly from a database often delivers the metadata along with the data itself. The Open Database Connectivity (ODBC) standard provides an open access method for accessing data stored in a variety of databases and is supported by R. This allows direct connection

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_4, © Springer Science+Business Media, LLC 2011

to a vast collection of data sources, including Microsoft Excel, Microsoft Access, SQL Server, Oracle, MySQL, Postgres, and SQLite. Section 4.3 covers the package **RODBC**.

The full variety of R's capability for loading data is necessarily not available directly within Rattle. However, we can use the underlying R commands to load data and then access it within Rattle, as in Section 4.4.

R packages themselves also provide an extensive collection of sample datasets. Whilst many datasets will be irrelevant to our specific tasks, they can be used to experiment with data mining using R. A list of datasets contained in the R Library is available through the Rattle interface by choosing Library as the Source on the Data tab. We cover this further in Section 4.6.

Having loaded our data into Rattle through some mechanism, we need to decide on the role played by each of the variables in the dataset. We also need to decide how the observations in the dataset are going to be used in the mining. We record these decisions through the Rattle interface, with Rattle itself providing useful defaults.

Once a dataset source has been identified and the Data tab executed, an overview of the data will be displayed in the text view. Figure 4.1 displays the Rattle application after loading the `weather.csv` file, which is supplied as a sample dataset with the Rattle package. We get here by starting up R and then loading **rattle**, starting up Rattle, and then clicking the Execute button for an offer to load the *weather* dataset:

```
> library(rattle)
> rattle()
```

In this chapter, we review the different source data formats and discuss how to load them for data mining. We then review the options that Rattle provides for identifying how the data is to be used for data mining.

4.1 CSV Data

One of the simplest and most common ways of sharing data today is via the comma-separated values (CSV) format. CSV has become a standard file format used to exchange data between many different applications. CSV files, which usually have a `csv` extension, can be exported and imported by spreadsheets and databases, including LibreOffice Calc, Gnumeric, Microsoft Excel, SAS/Enterprise Miner, Teradata, Netezza, and

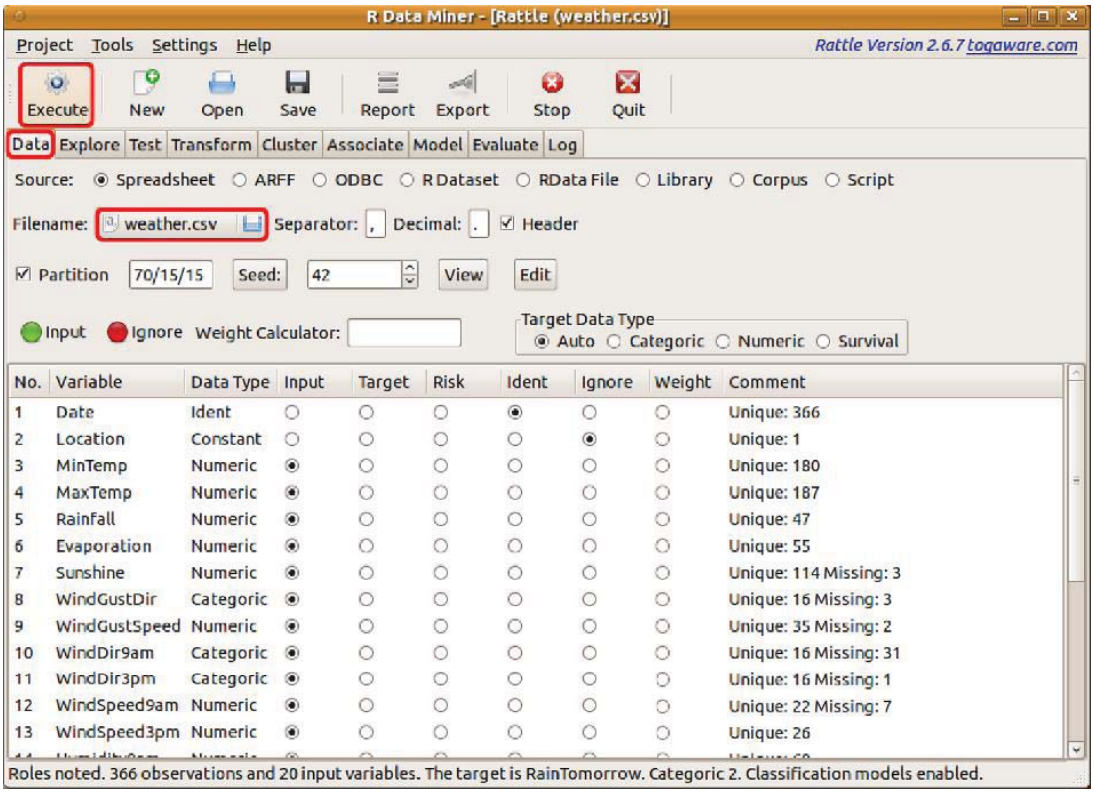


Figure 4.1: Loading the `weather.csv` dataset.

very many other applications. For these reasons, CSV is a good option for importing data into Rattle. The downside is that a CSV file does not contain explicit metadata (i.e., data about the data—including whether the data is numeric or categoric). Without this metadata, R sometimes determines the wrong data type for a particular column. This is not usually fatal, and we can help R along when loading data using R.

Locating and Loading Data

Using the Spreadsheet option of Rattle's Data tab, we can load data directly from a CSV file. Click the Filename button (Figure 4.2) to display the file chooser dialogue (Figure 4.3). We can browse to the CSV file we wish to load, highlight it, and click the Open button.

We now need to actually load the data into Rattle from the file. As always, we do this with a click on the Execute button (or a press of the F2 key). This will load the contents of the file from the hard disk into the computer's memory for processing by Rattle as a dataset.

Rattle supplies a number of sample CSV files and in particular pro-

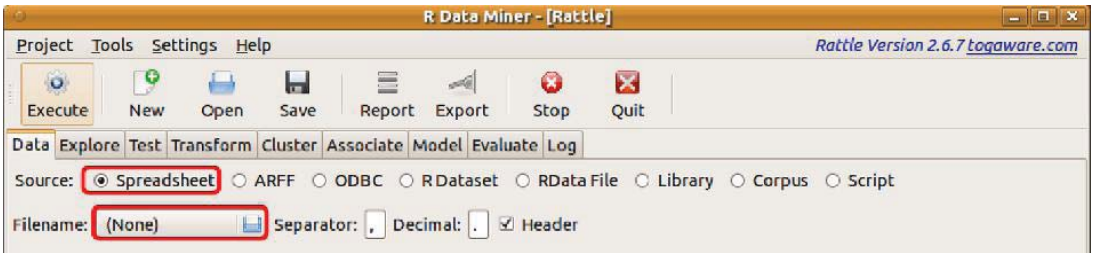


Figure 4.2: The Spreadsheet option of the Data tab, highlighting the Filename button. Click this button to open up the file chooser.

vides the `weather.csv` data file. The data file will have been installed when **rattle** was installed. We can ask R to tell us the actual location of the file using `system.file()`, which we can type into the R Console:

```
> system.file("csv", "weather.csv", package="rattle")

[1] "/usr/local/lib/R/site-library/rattle/csv/weather.csv"
```

The location reported will depend on your particular installation and operating system. Here the location is relative to a standard installation of a Ubuntu GNU/Linux system.

Tip: *We can also load this file into a new instance of Rattle with just two mouse clicks (Execute and Yes). We can then click the Filename button (displaying `weather.csv`) to open up a file browser showing the file path at the top of the window.*

We can review the contents of the file using `file.show()`. This will pop up a window displaying the contents of the file:

```
> fn <- system.file("csv", "weather.csv", package="rattle")
> file.show(fn)
```

The file contents can be directly viewed outside of R and Rattle with any simple text editor. If you aren't familiar with CSV files, it is instructional to become so. We will see that the top of the file begins:

```
Date,Location,MinTemp,MaxTemp,Rainfall,Evaporation...
2007-11-01,Canberra,8,24.3,0,3.4,6.3,NW,30,SW,NW...
2007-11-02,Canberra,14,26.9,3.6,4.4,9.7,ENE,39,E,W...
2007-11-03,Canberra,13.7,23.4,3.6,5.8,3.3,NW,85,N,NNE...
```

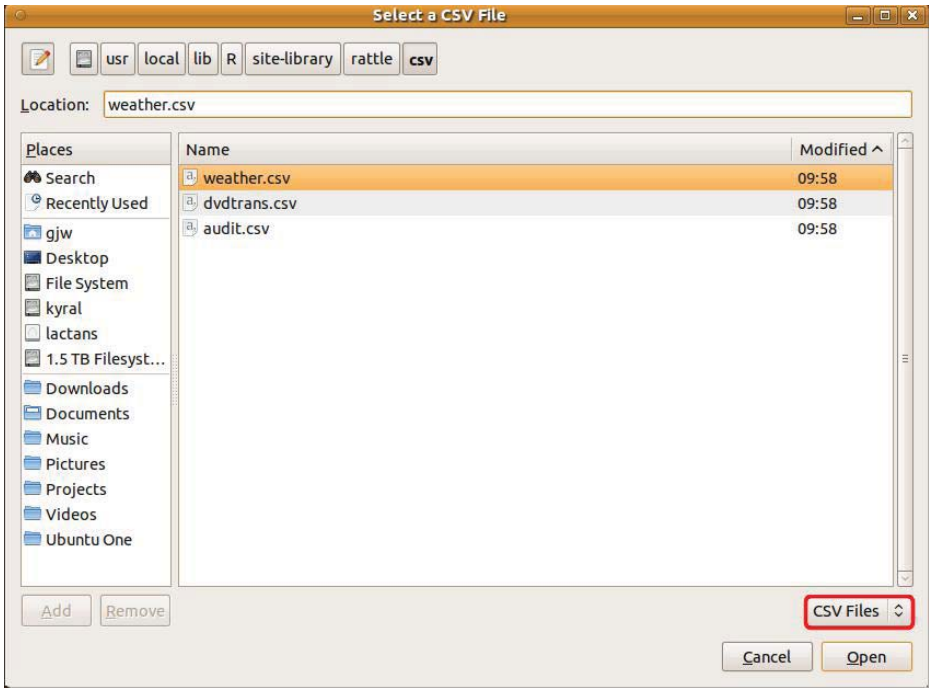


Figure 4.3: The CSV file chooser showing just those files with a .csv extension in the folder. We can also select to display just the .txt files (e.g., the extension often used for tab-delimited files) or else all files by selecting from the drop-down menu at the bottom right.

A CSV file is just a normal text file that commonly begins with a header line listing the names of the variables, each separated by a comma. The remainder of the file after the header row is expected to consist of rows of data that record the observations. For each observation, the fields are separated by commas, delimiting the actual observation of each of the variables.

Loading data into Rattle from a CSV file uses `read.csv()`. We can see this by reviewing the contents of the Log tab. From the Log tab we will see something like the following:

```
> crs$dataset <- read.csv("file:.../weather.csv",
                          na.strings=c(".", "NA", "", "?"),
                          strip.white=TRUE)
```

The full path to the `weather.csv` file is truncated here for brevity, so the command above won't succeed with a copy-and-paste. Instead, copy the corresponding line from the Log tab into the R Console. The result

of executing this function is that the dataset itself is loaded into memory and referenced using the name `crs\$dataset`.

The second argument in the function call above (`na.strings=`) lists the four strings that, if found as the value of a variable, will be translated into R's representation for missing values (`NA`). The list of strings used here captures the most common approaches to representing missing values. SAS, for example, uses the dot (".") to denote missing values, and R uses the special string "NA". Other applications simply use the empty string, whilst yet others (including machine learning applications like C4.5) use the question mark ("?").

We also use the `strip.white=` argument, setting it to TRUE, which has the effect of stripping white space (i.e., spaces and/or tabs). This allows the source CSV file to have the commas aligned for easier human viewing and still support missing values appropriately.

The `read.csv()` function need not be quite so complex. If we have a CSV file to load into R (again substituting the "..." with the actual path to the file), we can usually simply type the following command:

```
> ds <- read.csv(".../weather.csv")
```

We can also load data directly from the Internet. For example, the weather dataset is available from `togaware.com`:

```
> ds <- read.csv("http://rattle.togaware.com/weather.csv")
```

As we saw in Chapter 2 Rattle will offer to load the supplied sample data file (`weather.csv`) if no other data file is specified through the Filename button. This is the simplest way to load sample data into Rattle, and is useful for learning the Rattle interface.

After identifying the file to load, we do need to remember to click the Execute button to actually load the dataset into Rattle. The main text panel of the Data tab then changes to list the variables, together with their types and roles and some other useful information, as can be seen in Figure 4.1.

After loading the data from the file into Rattle, thereby creating a dataset, we can begin to explore it. The top of the file can be viewed in the R Console, as we saw in Chapter 2. Here we limit the display to just the first five variables and request just six observations:

```
> head(crs$dataset[1:5], 6)

        Date Location MinTemp MaxTemp Rainfall
1 2007-11-01 Canberra     8.0    24.3      0.0
2 2007-11-02 Canberra    14.0    26.9      3.6
3 2007-11-03 Canberra    13.7    23.4      3.6
4 2007-11-04 Canberra    13.3    15.5     39.8
5 2007-11-05 Canberra     7.6    16.1      2.8
6 2007-11-06 Canberra     6.2    16.9      0.0
```

As we described earlier (Section 2.9, page 50), Rattle stores the `dataset` within an environment called `crs`, so we can reference it directly in R as `crs$dataset`.

Through the Rattle interface, once we have loaded the dataset, we can also view it as a spreadsheet by clicking the View button, which uses `dfedit()` from **RGtk2Extras** (Taverner et al., 2010).

Data Variations

The Rattle interface provides options for tuning how we read the data from a CSV file. As we can see in Figure 4.2, the options include the Separator and Header.

We can choose the field delimiter through the Separator entry. A comma is the default. To load a TXT file, which uses a tab as the field separator, we replace the comma with the special code `\\t` (that is, two slashes followed by a `t`) to represent a tab. We can also leave the entry empty and any white space (i.e., any number of spaces and/or tabs) will be used as the separator.

From the `read.csv()` viewpoint, the effect of the separator entry is to include the appropriate argument (using `sep=`) in the call to the function. In this example, if we happen to have a file named "`mydata.txt`" that contained tab-delimited data, then we would include the `sep=`:

```
> ds <- read.csv("mydata.txt", sep="\t")
```

Tip: *Note that when specifying the tab as the separator directly within R we use a single slash rather than the double slashes through the Rattle interface.*

Another option of interest when loading a dataset is the Header check box. Generally, a CSV file will have as its first row a list of column names.

These names will be used by R and Rattle as the names of the variables. However, not all CSV files include headers. For such files, uncheck the Header check box. On loading a CSV file that does not contain headers, R will generate variable names for the columns. The check box translates to the `header=` argument in the call to `read.csv()`. Setting the value of `header=` to `FALSE` will result in the first line being read as data rather than as a header line. If we had such a file, perhaps called "`mydata.csv`", then the call to `read.csv()` would be:

```
> ds <- read.csv("mydata.csv", header=FALSE)
```

Tip: *The data can contain different numbers of columns in different rows, with missing columns at the end of a row being filled with* `NA`*s. This is handled using the* `fill=` *argument of* `read.csv()`*, which is* `TRUE` *by default.*

Basic Data Summary

Once a dataset has been loaded into Rattle, we can start to obtain an idea of the shape of the data from the simple summary that is displayed. In Figure 4.1, for example, the first variable, `Date`, is recognised as a unique identifier for each observation. It has 366 unique values, which is the same as the number of observations.

The variable `Location` has only a single value across all observations in the dataset. Consequently, it is identified as a constant and plays no role in the modelling. It is ignored.

The next five variables in Figure 4.1 are all tagged as numeric, followed by the categoric `WindGustDir`, and so on. The Comment column identifies the unique number of values and the number of missing observations for each variable. `Sunshine`, for example, has 114 unique values and 3 missing values. How to deal with missing values is covered in Chapter 7.

4.2 ARFF Data

The Attribute-Relation File Format (ARFF) is a text file format that is essentially a CSV file with a number of rows at the top of the file that contain metadata. The ARFF format was developed for use in the

Weka (Witten and Frank, 2005) machine learning software, and there are many datasets available in this format. We can load an ARFF dataset into Rattle through the ARFF option (Figure 4.4), specifying the filename from which the data is loaded.

Rattle provides sample ARFF datasets. To access them, after starting up Rattle and loading the sample *weather* dataset (Section 2.4), choose the ARFF option and then click the Filename chooser. Browse to the parent folder and then into the `arff` folder to choose a dataset to load.

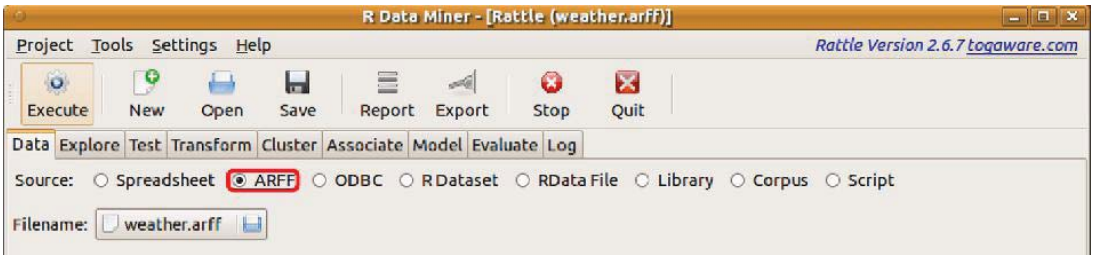


Figure 4.4: Choosing the ARFF radio button to load an ARFF file.

The key difference between CSV and ARFF is in the top part of the file, which contains information about each of the variables in the data—this is the data description section. An example of the ARFF format for our *weather* dataset is shown below. Note that ARFF refers to variables as *attributes*.

```
@relation weather
@attribute Date date
@attribute Location {Adelaide, Albany, ...}
@attribute MinTemp numeric
@attribute MaxTemp numeric
...
@attribute RainTomrrow {No, Yes}
@data
2010-11-01,Canberra,8,24.3,0,...,Yes
2010-11-02,Canberra,14,26.9,3.6,...,Yes
2010-11-03,Canberra,?,23.4,3.6,...,Yes
...
```

The data description section is straightforward, beginning with the name of the dataset (or the name of the *relation* in ARFF terminology). Each of the variables used to describe each observation is then identified together with its data type. Each variable definition appears on a

single line (we have truncated the lines in the example above). Numeric variables are identified as `numeric`, `real`, or `integer`. For categoric variables, the possible values are listed.

Two other data types recognised by ARFF are `string` and `date`. A `string` data type indicates that the variable can have a string (a sequence of characters) as its value. The `date` data type can also optionally specify the format in which the date is recorded. The default for dates is the ISO-8601 standard format, which is `"yyyy-MM-dd'T'HH:mm:ss"`.

Following the metadata specification, the actual observations are then listed, each on a single line, with fields separated by commas, exactly as with a CSV file.

A significant advantage of the ARFF data file over the CSV data file is the metadata information. This is particularly useful in Rattle, where for categoric data the possible values are determined from the data when reading in a CSV file. Any possible values of a categoric variable that are not present in the data will, of course, not be known. When reading the data from an ARFF file, the metadata will list all possible values of a categoric variable, even if one of the values might not be used in the actual data. We will come across this as an issue, particularly when we build and deploy random forest models, as covered in Chapter 12.

Comments can also be included in an ARFF file with a "`%`" at the beginning of the comment line. Including comments in the data file allows us to record extra information about the dataset, including how it was derived, where it came from, and how it might be cited.

Missing values in an ARFF data file are identified using the question mark "`?`". These are identified by R's `read.arff()`, and we see them as the usual `NA`s in Rattle.

Overall, the ARFF format, whilst simple, is quite an advance over a CSV file. Nonetheless, CSV still remains the more common data file.

4.3 ODBC Sourced Data

Much data is stored within databases and data warehouses. The Open Database Connectivity (ODBC) standard has been developed as a common approach for accessing data from databases (and hence data warehouses). The technology is based on the Structured Query Language (SQL) used to query relational databases. We discuss here how to access data directly from such databases.

Rattle can obtain a dataset from any database accessible through ODBC by using Rattle's ODBC option (Figure 4.5). Underneath the GUI, **RODBC** (Ripley and Lapsley, 2010) provides the actual interface to the ODBC data source.

Figure 4.5: Loading data through an ODBC database connection.

The key to accessing data via ODBC is to identify the data source through a so-called data source name (or DSN). Different operating systems provide different mechanisms for setting up DSNs. Under the GNU/Linux operating system, for example, using the `unixodbc` application, the system DSNs are often defined in `/etc/odbcinst.ini` and `/etc/odbc.ini`. Under Microsoft Windows, the control panel provides access to the ODBC Data Sources tool.

Using Rattle, we identify a configured DSN by typing its name into the DSN text entry (Figure 4.5). Once a DSN is specified, Rattle will attempt to make a connection. Many ODBC drivers will prompt for a username and password before establishing the connection. Figure 4.6 illustrates a typical popup for entering such data, in this case for connecting to a Netezza data warehouse.

To establish a connection using R directly, we use `odbcConnect()` from **RODBC**. This function establishes what we might think of as a channel connecting to the remote data source:

```
> library(RODBC)
> channel <- odbcConnect("myDWH", uid="kayon", pwd="toga")
```

After establishing a connection to a data source, Rattle will query the database for the names of the available tables and provide access to that list through the Table combo box of Figure 4.5. We need to select the specific table to load.

A limited number of options available in R are exposed through Rattle for fine-tuning the ODBC connection. One option allows us to limit the

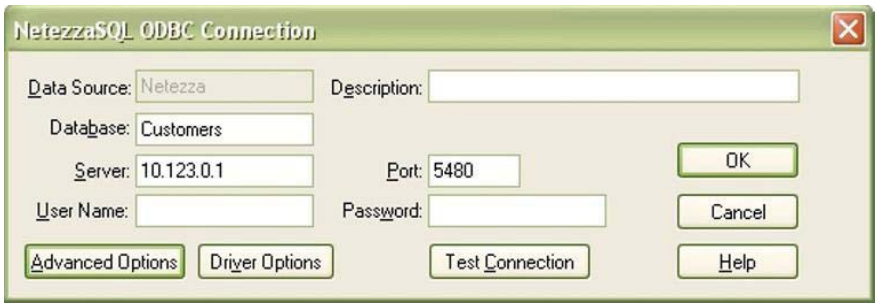


Figure 4.6: Netezza ODBC connection

number of rows retrieved from the chosen table. If the row limit is set to 0, then all of the rows from the table are retrieved. Unfortunately, there is no SQL standard for limiting the number of rows returned from a query. For some database systems (e.g., Teradata and Netezza), the SQL keyword is `LIMIT`, and this is what is used by Rattle.

A variety of R functions, provided by **RODBC**, are available to interact with the database. For example, the list of available tables is obtained using `sqlTables()`. We pass to it the channel that we created above to communicate with the database:

```
> tables <- sqlTables(channel)
```

If there is a table in the connected database called, for example, *clients*, we can obtain a list of column names using `sqlColumns()`:

```
> columns <- sqlColumns(channel, "clients")
```

Often, we are interested in loading only a specific subset of a table from the database. We can directly formulate an SQL query to retrieve just the data we want. For example:

```
> query <- "SELECT * FROM clients WHERE cost > 2500"
> myds <- sqlQuery(channel, query)
```

Using R directly provides a lot more scope for carefully identifying the data we wish to load. Any SQL query can be substituted for the simple `SELECT` statement used above. For those with skills in writing SQL queries, this provides quite a powerful mechanism for refining the data to be loaded, before it is loaded.

Loading data by directly sending an SQL query to the channel as above will store the data in R as a dataset, which we can reference as

`myds` (as defined above). This dataset can be accessed in Rattle with the R Dataset option, which we now introduce.

4.4 R Dataset—Other Data Sources

Data can be loaded from any source, one way or another, into R. We have covered loading data from a data file (as in loading a CSV or TXT file) or directly from a database. However, R supports many more options for importing data from a variety of sources.

Rattle can use any dataset (technically, any data frame) that has been loaded into R as a dataset to be mined. When choosing the R Dataset option of the Data tab (Figure 4.7), the Data Name box will list each of the available data frames that can be brought into Rattle as a dataset.

Using **foreign** (DebRoy and Bivand, 2011), for example, R can be used to read SPSS datasets (`read.spss()`), SAS XPORT format datasets (`read.xport()`), and DBF database files (`read.dbf()`). One notable exception, though, is the proprietary SAS dataset format, which cannot be loaded unless we have a licensed copy of SAS to read the data for us.

Loading SPSS Datasets

As an example, suppose we have an SPSS data file saved or exported from SPSS. We can read that into R using `read.spss()`:

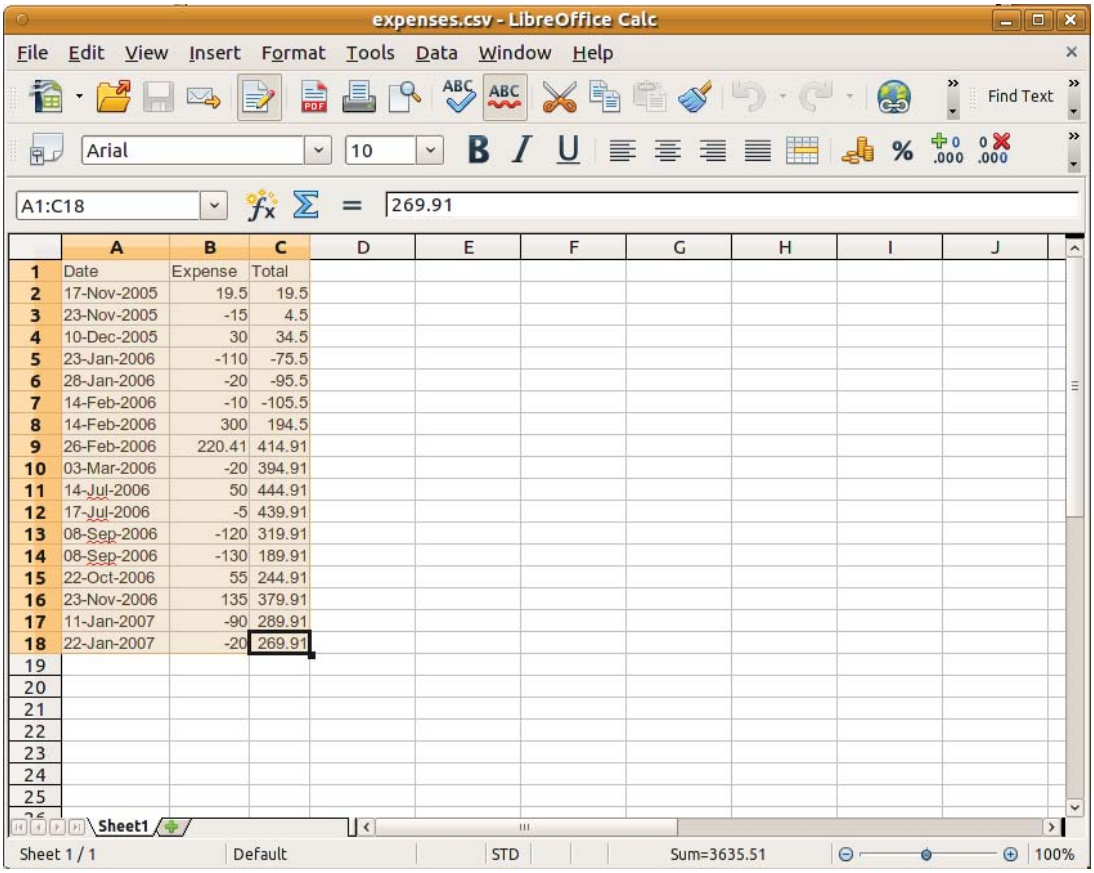
```
> library(foreign)
> mydataset <- read.spss(file="mydataset.sav")
```

Then, as in Figure 4.7, we can find the data frame name, `mydataset`, listed as an available R Dataset:

Figure 4.7: Loading an already defined R data frame as a dataset for use in Rattle.

The datasets that we wish to use with Rattle need to be constructed or loaded into the same R session that is running Rattle (i.e., the same R Console in which we loaded the Rattle package).

Reading Data from the Clipboard



	A	B	C
1	Date	Expense	Total
2	17-Nov-2005	19.5	19.5
3	23-Nov-2005	-15	4.5
4	10-Dec-2005	30	34.5
5	23-Jan-2006	-110	-75.5
6	28-Jan-2006	-20	-95.5
7	14-Feb-2006	-10	-105.5
8	14-Feb-2006	300	194.5
9	26-Feb-2006	220.41	414.91
10	03-Mar-2006	-20	394.91
11	14-Jul-2006	50	444.91
12	17-Jul-2006	-5	439.91
13	08-Sep-2006	-120	319.91
14	08-Sep-2006	-130	189.91
15	22-Oct-2006	55	244.91
16	23-Nov-2006	135	379.91
17	11-Jan-2007	-90	289.91
18	22-Jan-2007	-20	269.91

Figure 4.8: Selected region of a spreadsheet copied to the clipboard.

An interesting variation that may at times be quite convenient is the ability to directly copy and paste a selection via the system clipboard. Through this mechanism, we can "copy" (as in "copy-and-paste") data from a spreadsheet into the clipboard. Then, within R we can "paste" the data into a dataset using `read.table()`.

Suppose we have opened a spreadsheet with the data we see in Figure 4.8. If we select the 16 rows, including the header, in the usual way, we can very simply load the data using R:

```
> expenses <- read.table(file("clipboard"), header=TRUE)
```

The `expenses` data frame is then available to Rattle.

Converting Dates

By default, the `Date` variable in the example above is loaded as categoric. We can convert it into a date type, as below, before we load it into Rattle, as in Figure 4.9:

```
> expenses$Date <- as.Date(expenses$Date,
                           format="%d-%b-%Y")
> head(expenses)
        Date Expense  Total
1 2005-11-17    19.5   19.5
2 2005-11-23   -15.0    4.5
3 2005-12-10    30.0   34.5
4 2006-01-23  -110.0  -75.5
5 2006-01-28   -20.0  -95.5
6 2006-02-14   -10.0 -105.5
```

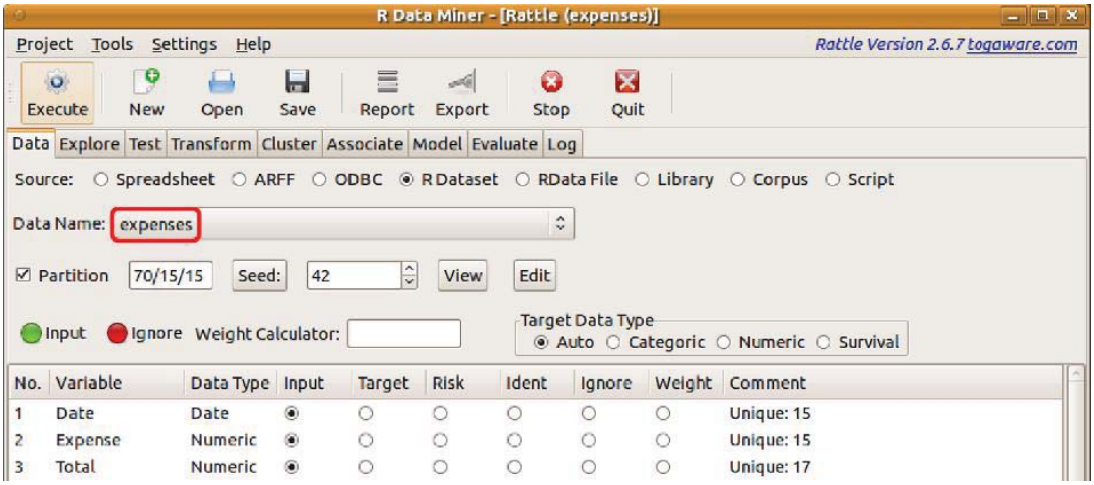


Figure 4.9: Loading an R data frame that was obtained from a copy-and-paste, via the clipboard, from a spreadsheet.

Reading Data from the World Wide Web

A lot of data today is available in HTML format on the World Wide Web. **XML** (Lang, 2011) provides functions to read such data directly into R and so make that data available for analysis in Rattle (and, of course,

R). As an example, we can read data from Google's list of most visited web sites, converting it to a data frame and thus making it available to Rattle. We begin this by loading **XML** and setting up some locations:

```
> library(XML)
> google <- "http://www.google.com/"
> path <- "adplanner/static/top1000/"
> top1000urls <- paste(google, path, sep="")
```

Now we can read in the data using `readHTMLTable()`, extracting the relevant table and setting up the column names:

```
> tables <- readHTMLTable(top1000urls)
> top1000 <- tables[[2]]
> colnames(top1000) <- c('Rank', 'Site', 'Category',
                         'Users', 'Reach', 'Views',
                         'Advertising')
```

The top few rows of data from the table can be viewed using `head()`:

```
> head(top1000)

  Rank          Site                     Category
1    1  facebook.com              Social Networks
2    2   youtube.com                 Online Video
3    3     yahoo.com                  Web Portals
4    4      live.com                Search Engines
5    5 wikipedia.org Dictionaries & Encyclopedias
6    6       msn.com                  Web Portals
       Users Reach           Views Advertising
1 880,000,000 47.2% 910,000,000,000         Yes
2 800,000,000 42.7% 100,000,000,000         Yes
3 660,000,000 35.3%  77,000,000,000         Yes
4 550,000,000 29.3%  36,000,000,000         Yes
5 490,000,000 26.2%   7,000,000,000          No
6 450,000,000   24%  15,000,000,000         Yes
```

4.5 R Data

Using the RData File option (Figure 4.10), data can be loaded directly from a native binary R data file (usually with the `RData` filename exten-

sion). Such files may contain multiple datasets (usually in a compressed format) and will have been saved from R sometime previously (using `save()`).

RData can be loaded by first identifying the file containing the data. The data will be loaded once the file is identified, and we will be given an option to choose just one of the available data frames to be loaded as Rattle's dataset. We specify this through the Data Name combo box and then click Execute to make the dataset available within Rattle.

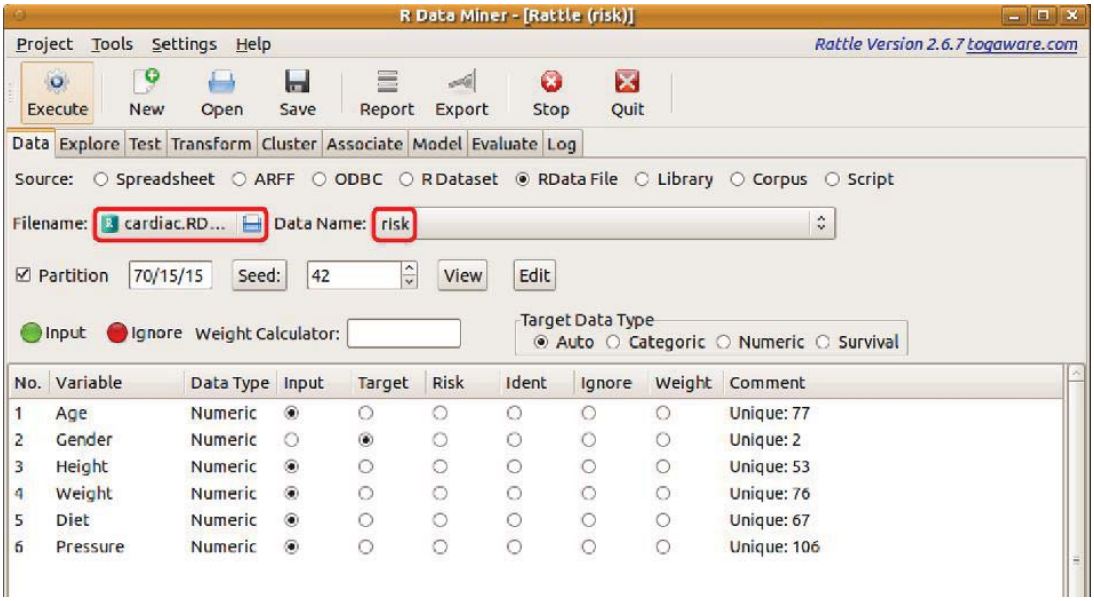


Figure 4.10: Loading a dataset from a binary R data file.

Figure 4.10 illustrates the selection of an RData file. The file is called `cardiac.RData`. Having identified the file, Rattle will populate the Data Name combo box with the names of each of the data frames found in the file. We can choose the *risk* dataset, from within the data file, to be loaded into Rattle.

4.6 Library

Almost every R package provides a sample dataset that is used to illustrate the functionality of the package. Rattle, as we have seen, provides the *weather*, *weatherAUS*, and *audit* datasets. We can explore the wealth of datasets that are available to us through the packages that are contained in our installed R library.

The Library option of the Data tab provides access to this vast collection of sample datasets. Clicking the radio button will generate the list of available datasets, which can then be accessed from the Data Name combo box. The dataset name, the package that provides that dataset, and a short description of the dataset will be included in the list. Note that the list can be quite long, and its contents will depend on the packages that are installed. We can see a sample of the list here, illustrating the R code that Rattle uses to generate the list:

```
> da <- data(package=.packages(all.available=TRUE))
> sort(paste(da$results[, "Item"], " : ",
             da$results[, "Package"], " : ",
             da$results[, "Title"], sep=""))

...
[10] "Adult : arules : Adult Data Set"
...
[12] "Affairs : AER : Fair's Extramarital Affairs Data"
...
[14] "Aids2 : MASS : Australian AIDS Survival Data"
...
[19] "airmay : robustbase : Air Quality Data"
...
[23] "ais : DAAG : Australian athletes data set"
...
[66] "audit : rattle : Sample dataset for data mining"
...
[74] "Baseball : vcd : Baseball Data"
...
[1082] "weather : rattle : Sample dataset for ..."
...
```

To access a dataset provided by a particular package, the actual package will first need to be loaded using `library()` (Rattle will do so automatically). For many packages (specifically those that declare the datasets as being *lazy loaded*—that is, loaded when they are referenced), the dataset will then be available from the R Console simply by typing the dataset name. Otherwise, `data()` needs to be run before the dataset can be accessed. We need to provide `data()` with the name of the dataset to be made available. Rattle takes care of this for us to ensure the appropriate action is taken to have the dataset available.

4.7 Data Options

All of Rattle's data load options that we have described above share a common set of further options that relate to the dataset once it has been loaded. The additional options relate to sampling the data as well as deciding on the role played by each of the variables. We review these options in the context of data mining.

Partitioning Data

As we first saw in Section 2.7, the Partition option allows us to partition our dataset into a training dataset, a validation dataset, and a testing dataset. The concept of partitioning a dataset was further defined in Section 3.1. The concepts are primarily oriented towards predictive data mining. Generally we will build a model using the *training dataset*.

To evaluate (Chapter 15) the performance of the model, we might then apply it to the *evaluation dataset*. This dataset has not been used to build the model and so provides an estimate of how well the model will perform when presented with new observations. Depending on the performance, we may tune the model-building parameters to seek an improvement in model performance.

Once we have a model that appears to perform well, or as well as possible with respect to the validation dataset, we might then evaluate its performance on the third partition, the *testing dataset*. The model has not previously been exposed to the observations contained in the testing dataset. Thus, the performance of the model on this dataset is probably a very good indication of how well the model will perform on new observations as they become available.

The concept of partitioning or sampling, though, is more general than simply a mechanism for partitioning for predictive data mining purposes. Statisticians have developed an understanding of sampling as a mechanism for analysing a small dataset to make conclusions about the whole population. Thus there is much literature from the statistics community on ensuring a good understanding of the uncertainty surrounding any conclusions we might make from analyses performed on any data. Such an understanding is important, though often underplayed in the data mining context.

Rattle creates a random partition/sample using `sample()`. A random sample will generally have a good chance of reflecting the distributions of

the whole population. Thus, exploring the data, as we do in Chapter 5, will be made easier when very large datasets are sampled down into much smaller ones. Exploring 10,000 observations is often a more interactive and practical proposition than exploring 1,000,000 observations. Other advantages of sampling include allowing analyses or plots to be repeated over different samples to gauge the stability and statistical accuracy of results. Model building, as we will see particularly when building random forests (Chapter 12), can take advantage of this.

The use of sampling in this way will also be necessary in data mining when the datasets available to model are so large that model building may take a considerable amount of time (hours or days). Sampling down to small proportions of a dataset will allow us to experiment more interactively with building a model. Once we are sure of how the data needs to be cleaned and transformed from our initial interactions, we can start experimenting with models. After we have the basic model parameters in place, we might be in a position to clean, transform, and model over a much larger portion of the data. We can leave the model building to complete over the possibly many hours that are often needed.

The downside of sampling, particularly in the data mining context, is that observations that correspond to rare events might disappear from a sample. Cases of rare diseases, or of the few instances of fraud from amongst millions of electronic funds transfers, may well be lost, even though they are the items that are of most interest to us in many data mining projects. This problem is often referred to as the class imbalance problem.

Rattle provides a default random partitioning of a dataset with 70% of the data going into a training dataset, 15% into a validation dataset, and 15% into a testing dataset (see Figure 4.11). We can override these choices, depending on our needs. A very small sampling may be required to perform some explorations of otherwise very large datasets. Smaller samples may also be required to build models using some of the more computationally expensive algorithms (like support vector machines).

Random numbers are used to select samples. Any sequence of random numbers must start with a so-called seed. If we use the same seed each time we will get the same sequence of random numbers. Thus the process is repeatable. By changing the seed we can select different random samples. This is often useful when we wish to explore the sensitivity of our models to different data samples.

Within Rattle a default seed is always used. This ensures, for example,

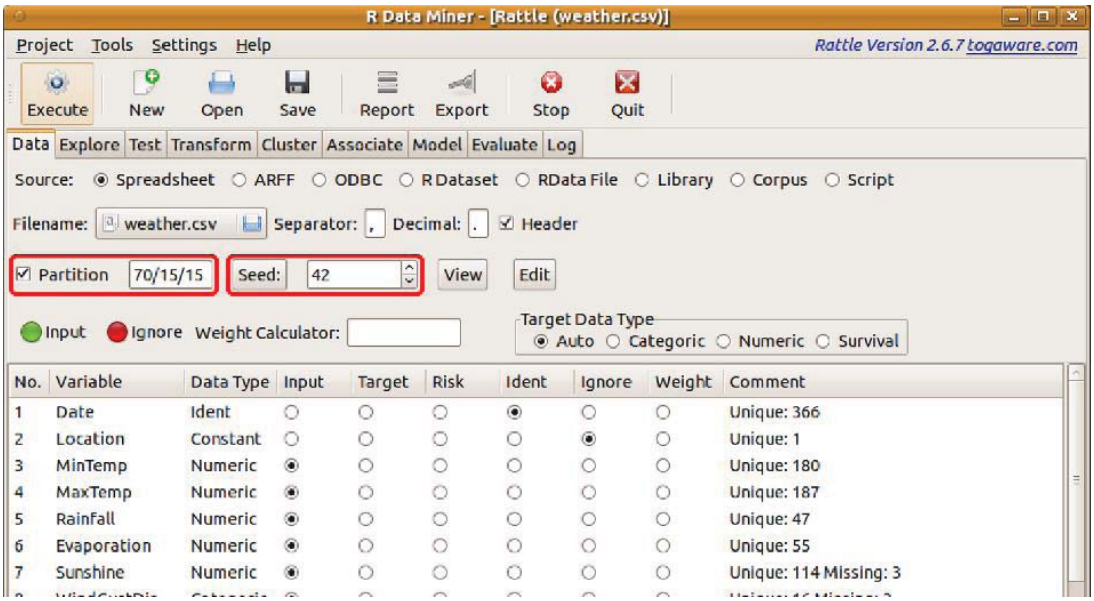


Figure 4.11: Sampling the *weather* dataset.

repeatable modelling. The seed is passed to the R function `set.seed()` to set a seed for the next generated sequence of random numbers. Thus, by setting the seed to the same number each time we can be assured of obtaining the same sample.

Conversely, we may like to set the seed to a different number in a series of model building exercises, and to then compare the performance of each model. Each model will have been built from a different random sample of the dataset. If we see significant variation between the different models, we may be concerned about the robustness of the approach we are taking. We discuss this further in Chapter 15.

Variable Roles

When building a model each variable will play a specific role. Most variables will be inputs to the model, and one variable is often identified as the target which we are modelling.

A variable can also be identified as a so-called *risk variable*. A risk variable might not be used for modelling as such. Generally it will record some magnitude associated with the risk or outcome. In the *audit* dataset, for example, it records the dollar amount of an adjustment that results from an audit—this is a measure of the size of the risk associated with that case. In the *weather* dataset the risk variable is the amount of

rain recorded for the following day—the amount of rain can be thought of as the size of the risk. See Section 15.4 for an example of using risk variables within Rattle, specifically for model evaluation.

Finally, we might also identify some variables to be ignored in the modelling altogether.

In loading data into Rattle we need to ensure our variables have their correct role for modelling. The default role for most variables is that of an Input variable. Generally, these are the variables that will be used to predict the value of a Target variable.

A *target variable*, if there is one associated with the dataset, is generally the variable of interest, from a predictive modelling point of view. That is, it is a variable that records the outcome from the historic data. In the case of the *weather* dataset this is `RainTomrrow`, whilst for the *audit* dataset the target is `Adjusted`.

Rattle uses simple heuristics to guess at a variable having a Target role. The primary heuristic is that a variable with a small number of distinct values (e.g., less than 5) is considered as a candidate target variable. The last variable in the dataset is usually considered as a candidate for being the target variable, because in many public datasets the last variable often is the target variable. If it has more than 5 distinct values Rattle will proceed from the first variable until it finds one with less than 5, if there are any. Only one variable can be tagged as a Target.

In a similar vain, integer variables that have a unique value for each observation are often automatically identified as an Ident (an identifier). Any number of variables can be tagged as being an Ident. All Ident variables are ignored when modelling, but are used after scoring a dataset, when it is being written to a score file, so that the observations that are scored can be identified.

Not all variables in our dataset might be wanted for the particular modelling task at hand. Such variables can be ignored, using the Ignore radio button.

When loading data into Rattle certain special strings are used to identify variable roles. For example, if the variable name starts with `ID` then the variable is automatically marked as having a role as an Ident. The user can override this.

Similarly, a variable with a name beginning with `IGNORE` will have the default role of Ignore. And so with `RISK` and `TARGET`.

At any one time a target is either treated as categoric or numeric. For a numeric variable chosen as the target, if it has 10 or fewer unique values then Rattle will automatically treat it as a categoric variable (by default). For modelling purposes, the consequence is that only classification type predictive models will be available. To build regression type predictive models we need to override the heuristic by selecting the Numeric radio button of the Data tab.

Weights Calculator and Role

The final data configuration option of the Data tab is the Weight Calculator and the associated Weight role. A single variable can be identified as representing some weight associated with each observation. The Weight Calculator allows us to provide a formula that could involve multiple variables as well as some scaling to give a weight for each observation. For example, with the audit dataset, we might enter a formula that uses the adjustment amount, and this will give more weight to those observations with a larger adjustment.

4.8 Command Summary

This chapter has referenced the following R packages, commands, functions, and datasets:

archetypes	package	Archetypal analysis.
audit	dataset	Sample dataset from **rattle**.
clients	dataset	A fictitious dataset.
`data()`	command	Make a dataset available to R.
`dfedit()`	command	Edit a data frame in a spreadsheet.
`file.show()`	command	Display a data frame.
foreign	package	Access multiple data formats.
`library()`	command	Load a package into the R library.
`file.show()`	command	Display the contents of a file.
`odbcConnect()`	function	Connect to a database.
`paste()`	function	Combine strings into one string.
rattle	package	Provides sample datasets.

`read.arff()`	function	Read an ARFF data file.
`read.csv()`	function	Read a comma-separated data file.
`read.dbf()`	function	Read data from a DBF database file.
`read.delim()`	function	Read a tab-delimited data file.
`read.spss()`	function	Read data from an SPSS data file.
`read.table()`	function	Read data from a text file.
`read.xport()`	function	Read data from a SAS Export data file.
`readHTMLTable()`	function	Read data from the World Wide Web.
risk	dataset	A fictitious dataset.
RODBC	package	Provides database connectivity.
`sample()`	function	Take a random sample of a dataset.
`save()`	command	Save R objects to a binary file.
`set.seed()`	command	Reset the random number sequence.
skel	dataset	Dataset from **archetypes** package.
`sqlColumns()`	function	List columns of a database table.
`sqlTables()`	function	List tables available from a database.
`system.file()`	function	Locate R or package file.
weather	dataset	Sample dataset from **rattle**.
XML	package	Access and generate XML like HTML.