

Chapter 2

Getting Started

New ideas are often most effectively understood and appreciated by actually doing something with them. So it is with data mining. Fundamentally, data mining is about practical application—application of the algorithms developed by researchers in artificial intelligence, machine learning, computer science, and statistics. This chapter is about getting started with data mining.

Our aim throughout this book is to provide hands-on practise in data mining, and to do so we need some computer software. There is a choice of software packages available for data mining. These include commercial closed source software (which is also often quite expensive) as well as free open source software. Open source software (whether freely available or commercially available) is *always* the best option, as it offers us the freedom to do whatever we like with it, as discussed in Chapter 1. This includes extending it, verifying it, tuning it to suit our needs, and even selling it. Such software is often of higher quality than commercial closed source software because of its open nature.

For our purposes, we need some good tools that are freely available to everyone and can be freely modified and extended by anyone. Therefore we use the open source and free data mining tool Rattle, which is built on the open source and free statistical software environment R. See Appendix A for instructions on obtaining the software. Now is a good time to install R. Much of what follows for the rest of the book, and specifically this chapter, relies on interacting with R and Rattle.

We can, quite quickly, begin our first data mining project, with Rattle's support. The aim is to build a model that captures the essence of the knowledge discovered from our data. Be careful though—there is a

lot of effort required in getting our data into shape. Once we have quality data, **Rattle** can build a model with just four mouse clicks, but the effort is in preparing the data and understanding and then fine-tuning the models.

In this chapter, we use **Rattle** to build our first data mining model—a simple decision tree model, which is one of the most common models in data mining. We cover starting up (and quitting from) R, an overview of how we interact with **Rattle**, and then how to load a dataset and build a model. Once the enthusiasm for building a model is satisfied, we then review the larger tasks of understanding the data and evaluating the model. Each element of **Rattle**’s user interface is then reviewed before we finish by introducing some basic concepts related to interacting directly with and writing instructions for R.

2.1 Starting R

R is a command line tool that is initiated either by typing the letter R (capital R—R is case-sensitive) into a command line window (e.g., a terminal in GNU/Linux) or by opening R from the desktop icon (e.g., in Microsoft Windows and Mac/OSX). This assumes that we have already installed R, as detailed in [Appendix A](#).

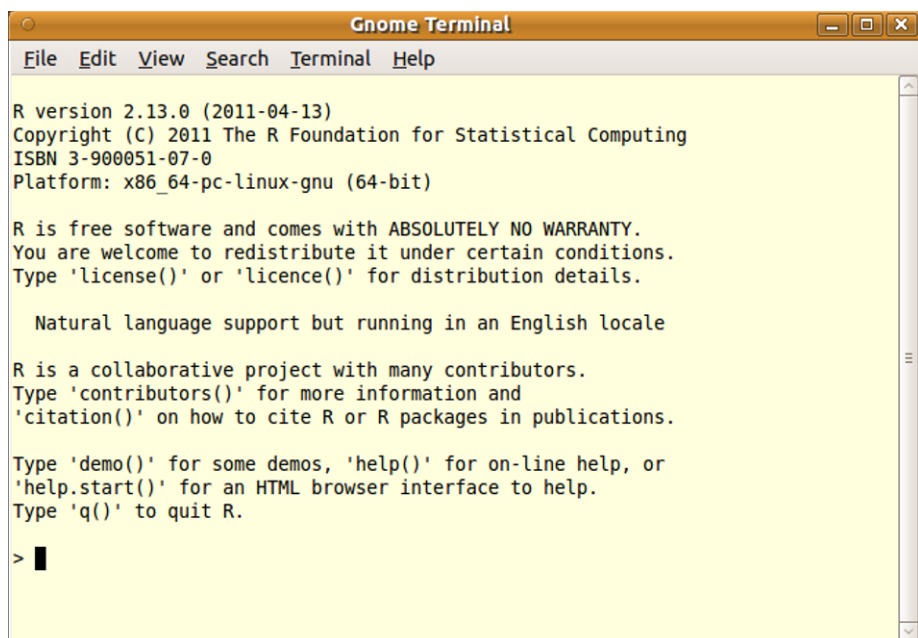
One way or another, we should see a window ([Figure 2.1](#)) displaying the R prompt (`>`), indicating that R is waiting for our commands. We will generally refer to this as the R Console.

The Microsoft Windows R Console provides additional menus specifically for working with R. These include options for working with script files, managing packages, and obtaining help.

We start **Rattle** by loading **rattle** into the R library using `library()`. We supply the name of the package to load as the argument to the command. The `rattle()` command is then entered with an empty argument list, as shown below. We will then see the **Rattle** GUI displayed, as in [Figure 2.2](#).

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

The **Rattle** user interface is a simple tab-based interface, with the idea being to work from the leftmost tab to the rightmost tab, mimicking the typical data mining process.



The screenshot shows a Gnome Terminal window titled "Gnome Terminal". The menu bar includes File, Edit, View, Search, Terminal, and Help. The terminal text displays the R version 2.13.0 (2011-04-13), copyright information for The R Foundation, and the platform x86_64-pc-linux-gnu (64-bit). It includes a disclaimer about no warranty and provides instructions on how to use R, including commands for license, contributors, and help. The prompt is a black character followed by a vertical bar.

```
R version 2.13.0 (2011-04-13)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-linux-gnu (64-bit)

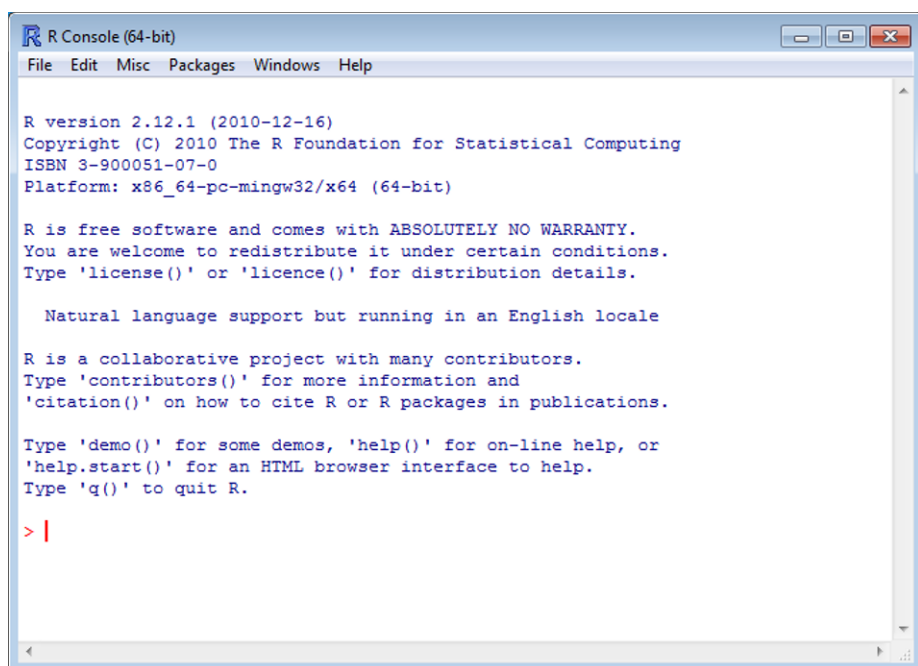
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> █
```



The screenshot shows an R Console window titled "R Console (64-bit)". The menu bar includes File, Edit, Misc, Packages, Windows, and Help. The console text displays the R version 2.12.1 (2010-12-16), copyright information for The R Foundation, and the platform x86_64-pc-mingw32/x64 (64-bit). It includes a disclaimer about no warranty and provides instructions on how to use R, including commands for license, contributors, and help. The prompt is a red character followed by a vertical bar.

```
R version 2.12.1 (2010-12-16)
Copyright (C) 2010 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Figure 2.1: The R Console for GNU/Linux and Microsoft Windows. The prompt indicates that R is awaiting user commands.

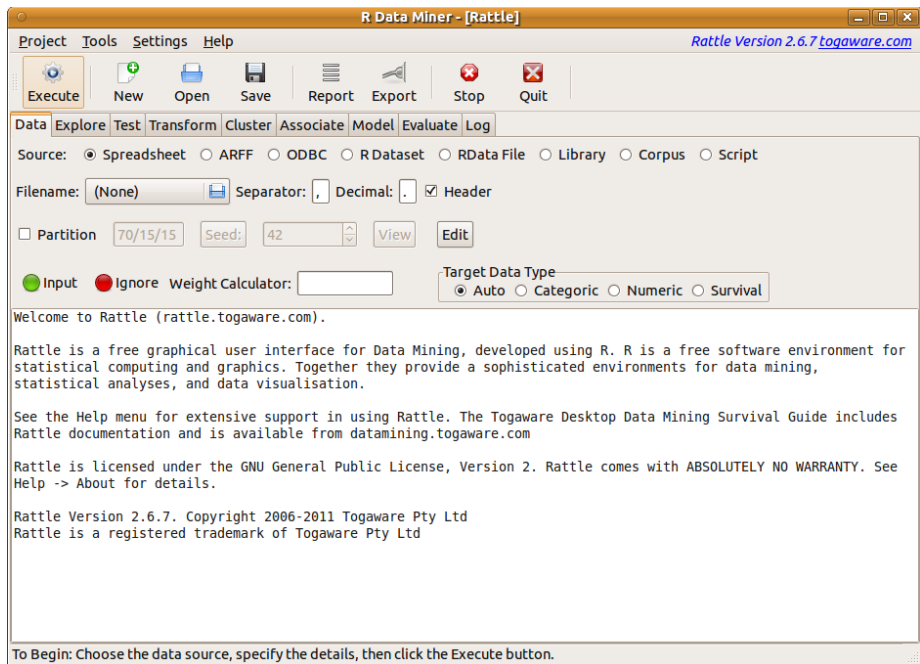


Figure 2.2: The initial Rattle window displays a welcome message and a little introduction to Rattle and R.

***Tip:** The key to using Rattle, as hinted at in the status bar on starting up Rattle, is to supply the appropriate information for a particular tab and to then **click the Execute button** to perform the action. Always make sure you have clicked the Execute button before proceeding to the next step.*

2.2 Quitting Rattle and R

A rather important piece of information, before we get into the details, is how to quit from the applications. To exit from Rattle, we simply click the Quit button. In general, this won't terminate the R Console. For R, the startup message (Figure 2.1) tells us to type `q()` to quit. We type this command into the R Console, including the parentheses so that the command is invoked rather than simply listing its definition. Pressing **Enter** will then ask R to quit:

```
> q()
Save workspace image? [y/n/c]:
```

We are prompted to save our workspace image. The workspace refers to all of the datasets and any other objects we have created in the current R session. We can save all of the objects currently available in a workspace between different invocations of R. We do so by choosing the `y` option. We might be in the middle of some complex analysis and wish to resume it at a later time, so this option is useful.

Many users generally answer `n` each time here, having already captured their analyses into script files. Script files allow us to automatically regenerate the results as required, and perhaps avoid saving and managing very large workspace files.

If we do not actually want to quit, we can answer `c` to cancel the operation and return to the R Console.

2.3 First Contact

In Chapter 1, we identified that a significant amount of effort within a data mining project is spent in processing our data into a form suitable for data mining. The amount of such effort should not be underestimated, but we do skip this step for now.

Once we have processed our data, we are ready to build a model—and with Rattle we can build the model with just a few mouse clicks. Using a sample dataset that someone else has already prepared for us, in Rattle we simply:

1. Click on the **Execute** button.

Rattle will notice that no dataset has been identified, so it will take action, as in the next step, to ensure we have some data. This is covered in detail in Section 2.4 and Chapter 4.

2. Click on **Yes** within the resulting popup.

The *weather* dataset is provided with Rattle as a small and simple dataset to explore the concepts of data mining. The dataset is described in detail in Chapter 3.

3. Click on the **Model** tab.

This will change the contents of Rattle's main window to display options and information related to the building of models. This is where we tell Rattle what kind of model we want to build and how it should be built. The **Model** tab is described in more detail in Section 2.5, and model building is discussed in considerable detail in Chapters 8 to 14.

4. Click on the **Execute** button.

Once we have specified what we want done, we ask **Rattle** to do it by clicking the **Execute** button. For simple model builders for small datasets, **Rattle** will only take a second or two before we see the results displayed in the text view window.

The resulting decision tree model, displayed textually in **Rattle**'s text view, is based on a sample dataset of historic daily weather observations (the curious can skip a few pages ahead to see the actual decision tree in [Figure 2.5](#) on page 30).

The data comes from a weather monitoring station located in Canberra, Australia, via the [Australian Bureau of Meteorology](#). Each observation is a summary of the weather conditions on a particular day. It has been processed to include a target variable that indicates whether it rained the day following the particular observation. Using this historic data, we have built a model to predict whether it will rain tomorrow. Weather data is commonly available, and you might be able to build a similar model based on data from your own region.

With only one or two more clicks, further models can be built. A few more clicks and we have an evaluation chart displaying the performance of the model. Then, with just a click or two more, we will have the model applied to a new dataset to generate scores for new observations.

Now to the details. We will continue to use **Rattle** and also the simple command line facility. The command line is not strictly necessary in using **Rattle**, but as we develop our data mining capability, it will become useful. We will load data into **Rattle** and explain the model that we have built. We will build a second model and compare their performances. We will then apply the model to a new dataset to provide scores for a collection of new observations (i.e., predictions of the likelihood of it raining tomorrow).

2.4 Loading a Dataset

With **Rattle** we can load a sample dataset in preparation for modelling, as we have just done. Now we want to illustrate loading any data (perhaps our own data) into **Rattle**.

If we have followed the four steps in [Section 2.3](#), then we will now need to reset **Rattle**. Simply click the **New** button within the toolbar. We are asked to confirm that we would like to clear the current project.

Alternatively, we might have exited Rattle and R, as described in Section 2.1, and need to restart everything, as also described in Section 2.1. Either way, we need to have a fresh Rattle ready so that we can follow the examples below.

On starting Rattle, we can, without any other action, click the Execute button in the toolbar. Rattle will notice that no CSV file (the default data format) has been specified (notice the “(None)” in the Filename: chooser) and will ask whether we wish to use one of the sample datasets supplied with the package. Click on Yes to do so, to see the data listed, as shown in Figure 2.3.

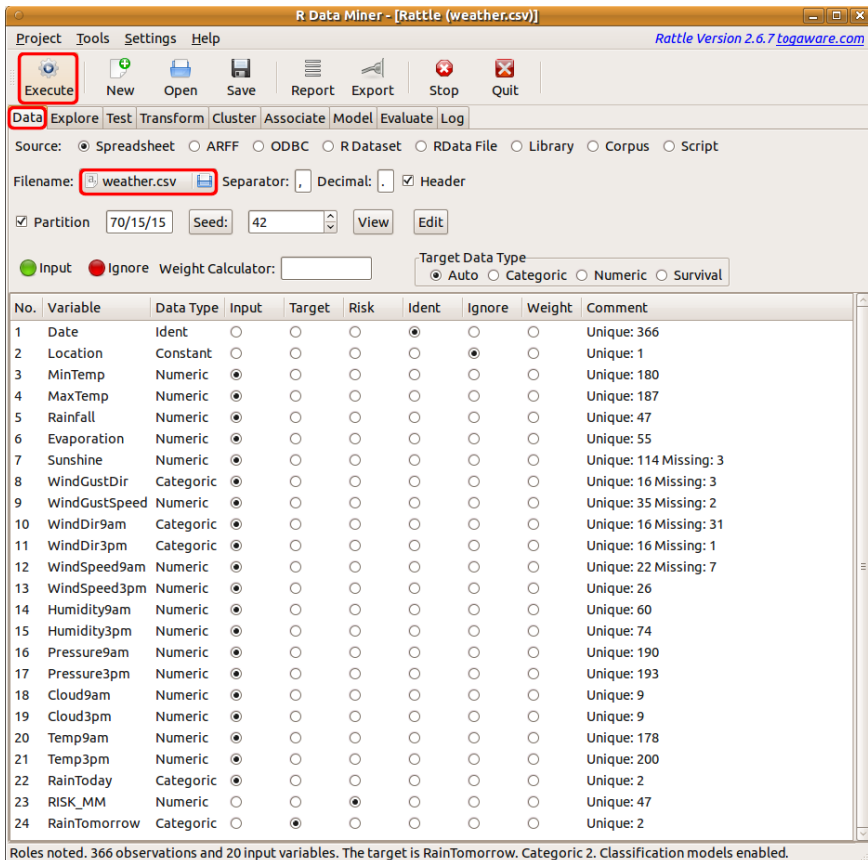


Figure 2.3: The sample `weather.csv` file has been loaded into memory as a dataset ready for mining. The dataset consists of 366 observations and 24 variables, as noted in the status bar. The first variable has a role other than the default Input role. Rattle uses heuristics to initialise the roles.

The file *weather.csv* will be loaded by default into Rattle as its dataset. Within R, a dataset is actually known as a *data frame*, and we will see this terminology frequently.

The dataset summary (Figure 2.3) provides a list of the variables, their data types, default roles, and other useful information. The types will generally be **Numeric** (if the data consists of numbers, like temperature, rainfall, and wind speed) or **Categoric** (if the data consists of characters from the alphabet, like the wind direction, which might be **N** or **S**, etc.), though we can also see an **Ident** (identifier). An **Ident** is often one of the variables (columns) in the data that uniquely identifies each observation (row) of the data. The **Comments** column includes general information like the number of unique (or distinct) values the variable has and how many observations have a missing value for a variable.

2.5 Building a Model

Using Rattle, we click the **Model** tab and are presented with the **Model** options (Figure 2.4). To build a decision tree model, one of the most common data mining models, click the **Execute** button (decision trees are the default). A textual representation of the model is shown in Figure 2.4.

The target variable (which stores the outcome we want to model or predict) is **RainTomorrow**, as we see in the **Data** tab window of Figure 2.3. Rattle automatically chose this variable as the target because it is the last variable in the data file and is a binary (i.e., two-valued) categoric. Using the *weather* dataset, our modelling task is to learn about the prospect of it raining tomorrow given what we know about today.

The textual presentation of the model in Figure 2.4 takes a little effort to understand and is further explained in Chapter 11. For now, we might click on the **Draw** button provided by Rattle to obtain the plot that we see in Figure 2.5. The plot provides a better idea of why it is called a decision **tree**. This is just a different way of representing the same model.

Clicking the **Rules** button will display a list of rules that are derived directly from the decision tree (we'll need to scroll the panel contained in the **Model** tab to see them). This is yet another way to represent the same model. The rules are listed here, and we explain them in detail next.

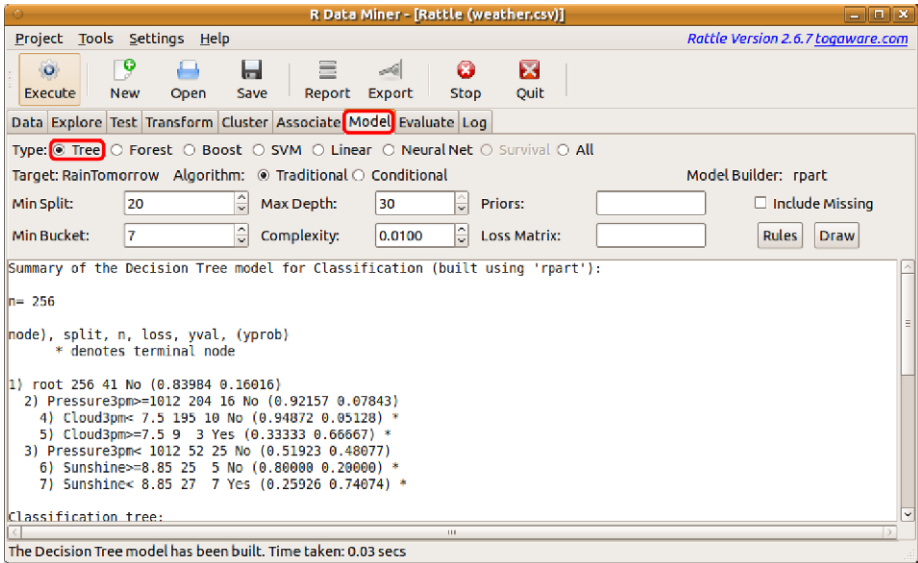


Figure 2.4: The *weather* dataset has been loaded, and a decision tree model has been built.

```

Rule number: 7 [RainTomorrow=Yes cover=27 (11%) prob=0.74]
  Pressure3pm< 1012
  Sunshine< 8.85

Rule number: 5 [RainTomorrow=Yes cover=9 (4%) prob=0.67]
  Pressure3pm>=1012
  Cloud3pm>=7.5

Rule number: 6 [RainTomorrow=No cover=25 (10%) prob=0.20]
  Pressure3pm< 1012
  Sunshine>=8.85

Rule number: 4 [RainTomorrow=No cover=195 (76%) prob=0.05]
  Pressure3pm>=1012
  Cloud3pm< 7.5

```

A well-recognised advantage of the decision tree representation for a model is that the paths through the decision tree can be interpreted as a collection of rules, as above. The rules are perhaps the more readable representation of the model. They are listed in the order of the prob-

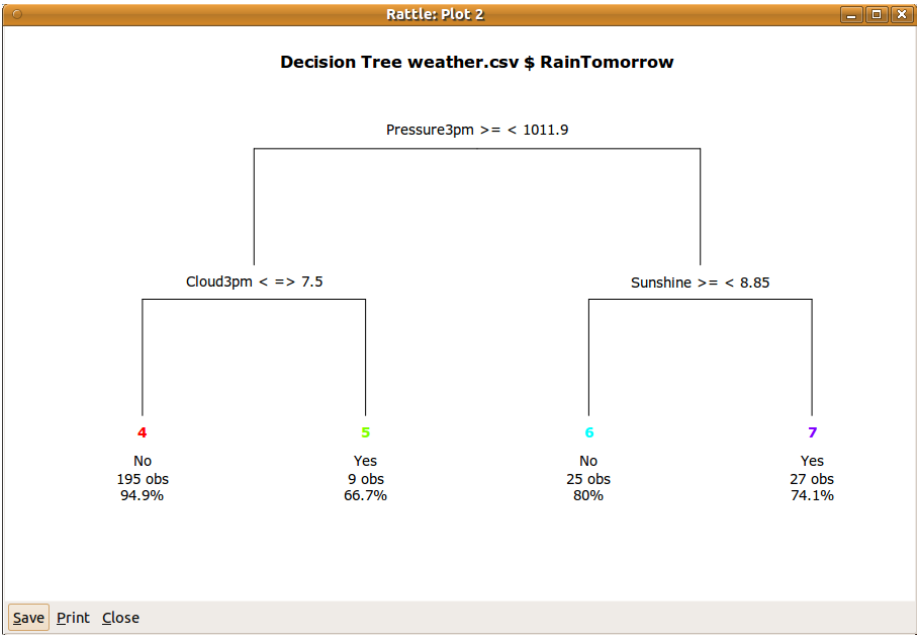


Figure 2.5: The decision tree built “out of the box” with Rattle. We traverse the tree by following the branches corresponding to the tests at each node. The $> \leq$ notation on the root (top) node indicates that we travel left if `Pressure3pm` is greater than 1011.9 and down the right branch if it is less than or equal to 1011.9. The $< = >$ is similar, but reversed. The leaf nodes include a node number for reference, a decision of No or Yes to indicate whether it will `RainTomorrow`, the number of training observations, and the strength or confidence of the decision.

ability (`prob`) that we see listed with each rule. The interpretation of the probability will be explained in more detail in Chapter 11, but we provide an intuitive reading here.

Rule number 7 (which also corresponds to the “7”) in Figure 2.4 and leaf node number 7 in Figure 2.5) is the strongest rule predicting rain (having the highest probability for a **Yes**). We can read it as saying that if the atmospheric pressure (reduced to mean sea level) at 3 pm was less than 1012 hectopascals and the amount of sunshine today was less than 8.85 hours, then it seems there is a 74% chance of rain tomorrow (*yval* = *yes* and *prob* = 0.74). That is to say that on most days when we have previously seen these conditions (as represented in the data) it has rained the following day.

Progressing down to the other end of the list of rules, we find the conditions under which it appears much less likely that there will be rain the following day. Rule number 4 has two conditions: the atmospheric pressure at 3 pm greater than or equal to 1012 hectopascals and cloud cover at 3 pm less than 7.5. When these conditions hold, the historic data tells us that it is unlikely to be raining tomorrow. In this particular case, it suggests only a 5% probability (`prob=0.05`) of rain tomorrow.

We now have our first model. We have data-mined our historic observations of weather to help provide some insight about the likelihood of it raining tomorrow.

2.6 Understanding Our Data

We have reviewed the modelling part of data mining above with very little attention to the data. A realistic data mining project, though, will precede modelling with quite an extensive exploration of data, in addition to understanding the business, understanding what data is available, and transforming such data into a form suitable for modelling. There is a lot more involved than just building a model. We look now at exploring our data to better understand it and to identify what we might want to do with it.

Rattle's Explore tab provides access to some common plots as well as extensive data exploration possibilities through **latticist** (Andrews, 2010) and **rggobi** (Lang et al., 2011). We will cover exploratory data analysis in detail in Chapters 5 and 6. We present here an initial flavour of exploratory data analysis.

One of the first things we might want to know is how the values of the target variable (`RainTomorrow`) are distributed. A histogram might be useful for this. The simplest way to create one is to go to the **Data** tab, click on the **Input** role for `RainTomorrow`, and click the **Execute** button. Then go to the **Explore** tab, choose the **Distributions** option, and then select **Bar Plot** for `RainTomorrow`. The plot of Figure 2.6 will be shown.

We can see from Figure 2.6 that the target variable is highly skewed. More than 80% of the days have no rain. This is typical of data mining, where even greater skewness is not uncommon. We need to be aware of the skewness, for example, in evaluating any models we build—a model that simply predicts that it never rains is going to be over 80% accurate, but pretty useless.

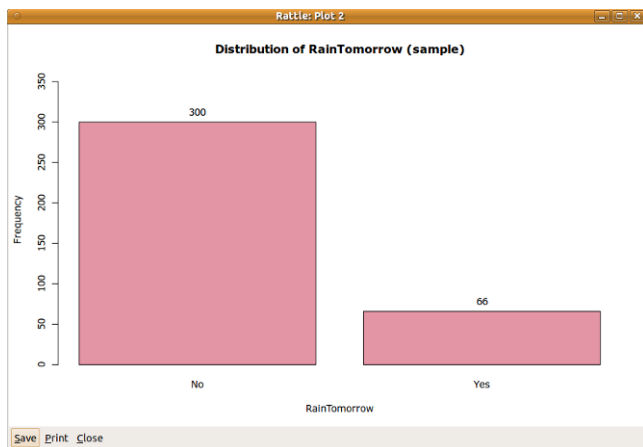


Figure 2.6: The target variable, `RainTomorrow`, is skewed, with `Yes` being quite underrepresented.

We can display other simple plots from the **Explore** tab by selecting the **Distributions** option. Under both the **Box Plot** and **Histogram** columns, select **MaxTemp** and **Sunshine** (as in Figure 2.7). Then click on **Execute** to display the plots in Figure 2.8. The plots begin to tell a story about the data. We sketch the story here, leaving the details to Chapter 5.

The top two plots are known as box-and-whisker plots. The top left plot tells us that the maximum temperature is generally higher the day before it rains (the plot above the x-axis label **Yes**) than before the days when it does not rain (above the **No**).

The top right plot suggests an even more dramatic skew for the amount of sunshine the day prior to the prediction. Generally we see that if there is less sunshine the day before, then the chance of rain (**Yes**) seems to be increased.

Both box plots also give another clue about the distribution of the values of the target variable. The width of the boxes in a box plot provides a visual indication of this distribution.

Each bottom plot overlays three separate plots that give further insight into the distribution of the observations. The three plots within each figure are a histogram (bars), a density plot (lines), and a rug plot (short spikes on the x-axis), each of which we now briefly describe.

The histogram has partitioned the numeric data into segments of equal width, showing the frequency for each segment. We see again that

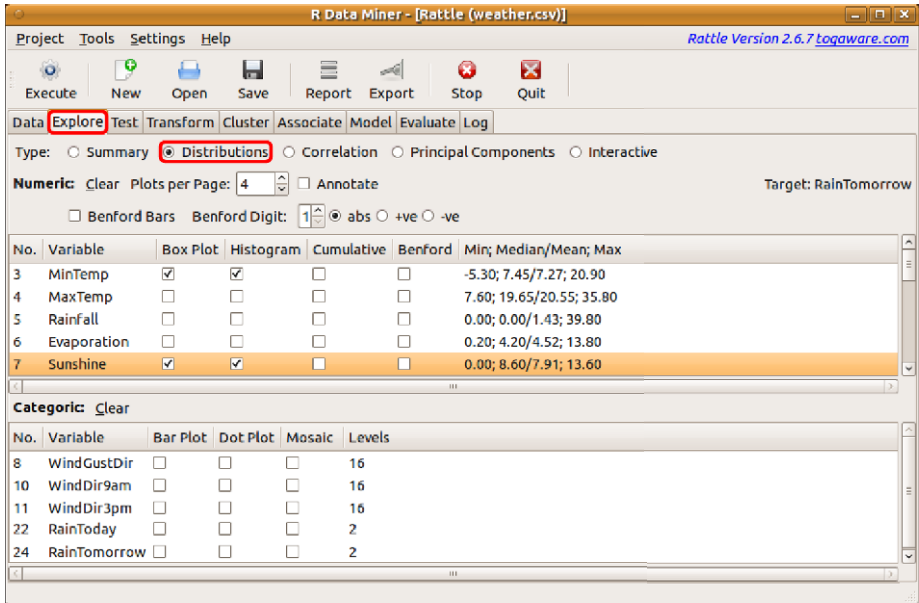


Figure 2.7: The *weather* dataset has been loaded and a decision tree model has been built.

sunshine (the bottom right) is quite skewed compared with the maximum temperature.

The density plots tend to convey a more accurate picture of the distribution of the data. Because the density plot is a simple line, we can also display the density plots for each of the target classes (**Yes** and **No**).

Along the x-axis is the rug plot. The short vertical lines represent actual observations. This can give us an idea of where any extreme values are, and the dense parts show where more of the observations lie.

These plots are useful in understanding the distribution of the numeric data. Rattle similarly provides a number of simple standard plots for categorical variables. A selection are shown in Figure 2.9. All three plots show a different view of the one variable, *WindDir9am*, as we now describe.

The top plot of Figure 2.9 shows a very simple bar chart, with bars corresponding to each of the levels (or values) of the categorical variable of interest (*WindDir9am*). The bar chart has been sorted from the overall most frequent to the overall least frequent categorical value. We note that each value of the variable (e.g., the value “SE,” representing a wind direc-

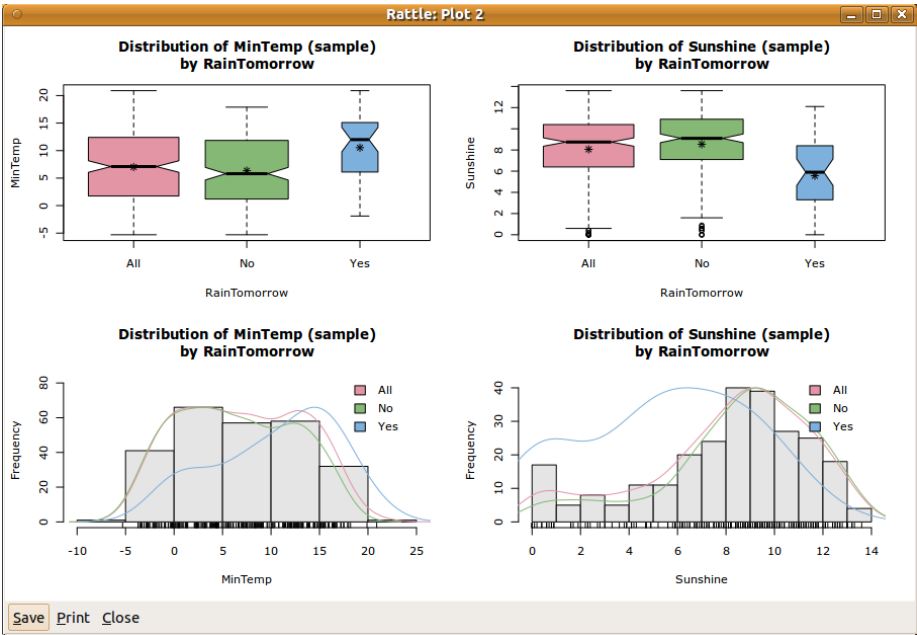


Figure 2.8: A sample of distribution plots for two variables.

tion of southeast) has three bars. The first bar is the overall frequency (i.e., the number of days) for which the wind direction at 9 am was from the southeast. The second and third bars show the breakdown for the values across the respective values of the categoric target variable (i.e., for **No** and **Yes**). We can see that the distribution within each wind direction differs between the three groups, some more than others. Recall that the three groups correspond to all observations (**All**), observations where it did not rain on the following day (**No**), and observations where it did (**Yes**).

The lower two plots show essentially the same information, in different forms. The bottom left plot is a dot plot. It is similar to the bar chart, on its side, and with dots representing the “top” of the bars. The breakdown into the levels of the target variable is compactly shown as dots within the same row.

The bottom right plot is a mosaic plot, with all bars having the same height. The relative frequencies between the values of **WindDir9am** are now indicated by the widths of the bars. Thus, **SE** is the widest bar, and **WSW** is the thinnest. The proportion between **No** and **Yes** within each bar

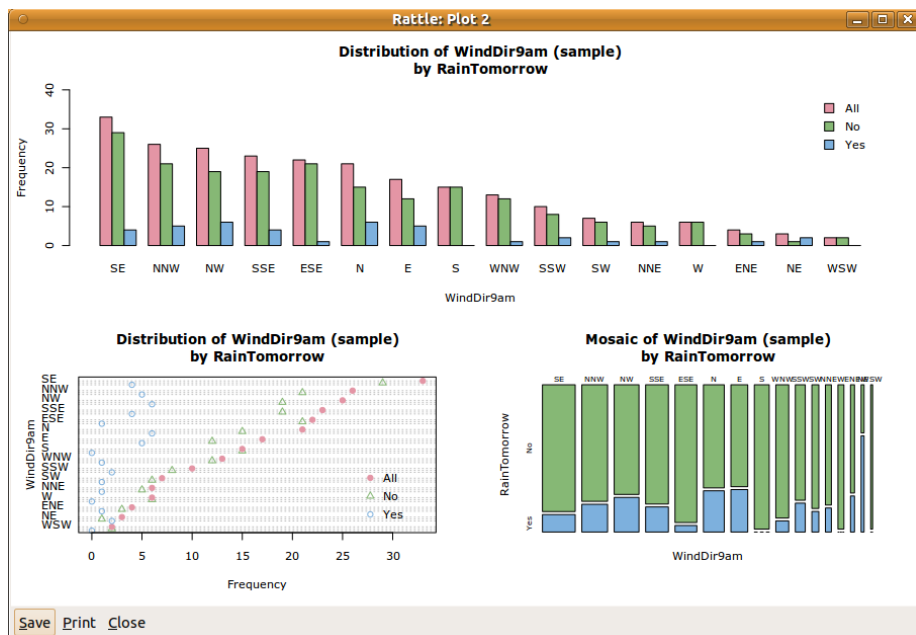


Figure 2.9: A sample of the three distribution plots for the one categorical variable.

is clearly shown.

A mosaic plot allows us to easily identify levels that have very different proportions associated with the levels of the target variable. We can see that a north wind direction has a higher proportion of observations where it rains the following day. That is, if there is a northerly wind today, then the chance of rain tomorrow seems to be increased.

These examples demonstrate that data visualisation (or exploratory data analysis) is a powerful tool for understanding our data—a picture is worth a thousand words. We actually learn quite a lot about our data even before we start to specifically model it. Many data miners begin to deliver significant benefits to their clients simply by providing such insights. We delve further into exploring data in Chapter 5.

2.7 Evaluating the Model: Confusion Matrix

We often begin a data mining project by exploring the data to gain our initial insights. In all likelihood, we then also transform and clean up

our data in various ways. We have illustrated above how to then build our first model. It is now time to evaluate the performance or quality of the model.

Evaluation is a critical step in any data mining process, and one that is often left underdone. For the sake of getting started, we will look at a simple evaluation tool. The **confusion matrix** (also referred to as the *error matrix*) is a common mechanism for evaluating model performance.

In building our model we used a 70% subset of all of the available data. Figure 2.3 (page 27) shows the default sampling strategy of 70/15/15. We call the 70% sample the *training dataset*. The remainder is split equally into a *validation dataset* (15%) and a *testing dataset* (15%).

The validation dataset is used to test different parameter settings or different choices of variables whilst we are data mining. It is important to note that this dataset should not be used to provide any error estimations of the final results from data mining since it has been used as part of the process of building the model.

The testing dataset is *only* to be used to predict the unbiased error of the final results. It is important not to use this testing dataset in any way in building or even fine-tuning the models that we build. Otherwise, it no longer provides an unbiased estimate of the model performance.

The testing dataset and, whilst we are building models, the validation dataset, are used to test the performance of the models we build. This often involves calculating the model error rate. A confusion matrix simply compares the decisions made by the model with the actual decisions. This will provide us with an understanding of the level of accuracy of the model in terms of how well the model will perform on new, previously unseen, data.

Figure 2.10 shows the Evaluate tab with the Error Matrix (confusion matrix) using the Testing dataset for the Tree model that we have previously seen in Figures 2.4 and 2.5. Two tables are presented. The first lists the actual counts of observations and the second the percentages. We can observe that for 62% of the predictions the model correctly predicts that it won't rain (called the *true negatives*). That is, 35 days out of the 56 days are correctly predicted as not raining. Similarly, we see the model correctly predicts rain (called the *true positives*) on 18% of the days.

In terms of how correct the model is, we observe that it correctly predicts rain for 10 days out of the 15 days on which it actually does rain. This is a 67% accuracy in predicting rain. We call this the *true*

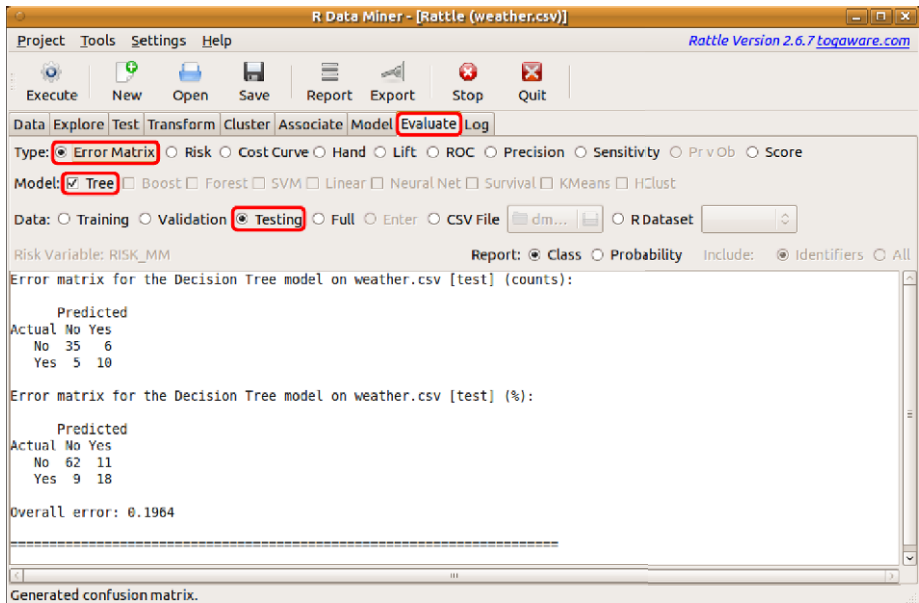


Figure 2.10: A confusion matrix applying the model to the testing dataset is displayed.

positive rate, but it is also known as the *recall* and the *sensitivity* of the model. Similarly, the *true negative rate* (also called the *specificity* of the model) is 85%.

We also see six days when we are expecting rain and none occurs (called the *false positives*). If we were using this model to help us decide whether to take an umbrella or raincoat with us on our travels tomorrow, then it is probably not a serious loss in this circumstance—we had to carry an umbrella without needing to use it. Perhaps more serious though is that there are five days when our model tells us there will be no rain yet it rains (called the *false negatives*). We might get inconveniently wet without our umbrella. The concepts of true and false positives and negatives will be further covered in Chapter 15.

The performance measure here tells us that we are going to get wet more often than we would like. This is an important issue—the fact that the different types of errors have different consequences for us. We'll also see more about this in Chapter 15.

It is useful to compare the performance as measured using the validation and testing datasets with the performance as measured using

the training dataset. To do so, we can select the **Validation** and then the **Training** options (and for completeness the **Full** option) from the **Data** line of the **Evaluate** tab and then **Execute** each. The resulting performance will be reported. We reproduce all four here for comparison, including the count and the percentages.

Evaluation Using the Training Dataset:

Count		Predict		Percentage		Predict	
		No	Yes			No	Yes
Actual	No	205	10	Actual	No	80	4
	Yes	15	26		Yes	6	10

Evaluation Using the Validation Dataset:

Count		Predict		Percentage		Predict	
		No	Yes			No	Yes
Actual	No	39	5	Actual	No	72	9
	Yes	5	5		Yes	9	9

Evaluation Using the Testing Dataset:

Count		Predict		Percentage		Predict	
		No	Yes			No	Yes
Actual	No	35	6	Actual	No	62	11
	Yes	5	10		Yes	9	18

Evaluation Using the Full Dataset:

Count		Predict		Percentage		Predict	
		No	Yes			No	Yes
Actual	No	279	21	Actual	No	76	6
	Yes	25	41		Yes	7	11

We can see that there are fewer errors in the training dataset than in either the validation or testing datasets. That is not surprising since the tree was built using the training dataset, and so it should be more accurate on what it has already seen. This provides a hint as to why we do not validate our model on the training dataset—the evaluation will provide optimistic estimates of the performance of the model. By applying the model to the validation and testing datasets (which the

model has not previously seen), we expect to obtain a more realistic estimate of the performance of the model on new data.

Notice that the overall accuracy from the training dataset is 90% (i.e., adding the diagonal percentages, 80% plus 10%), which is excellent. For the validation and testing datasets, it is around 80%. This is more likely how accurate the model will be longer-term as we apply it to new observations.

2.8 Interacting with Rattle

We have now stepped through some of the process of data mining. We have loaded some data, explored it, cleaned and transformed it, built a model, and evaluated the model. The model is now ready to be deployed. Of course, there is a lot more to what we have just done than what we have covered here. The remainder of the book provides much of these details. Before proceeding to the details, though, we might review how we interact with Rattle and R.

We have seen the Rattle interface throughout this chapter and we now introduce it more systematically. The interface is based on a set of tabs through which we progress as we work our way through a data mining project. For any tab, once we have set up the required information, we will click the **Execute** button to perform the actions. Take a moment to explore the interface a little. Notice the **Help** menu and that the help layout mimics the tab layout.

The Rattle interface is designed as a simple interface to a powerful suite of underlying tools for data mining. The general process is to step through each tab, left to right, performing the corresponding actions. For any tab, we configure the options and then click the **Execute** button (or **F2**) to perform the appropriate tasks. It is important to note that the tasks are **not** performed until the **Execute** button (or **F2** or the **Execute** menu item under **Tools**) is clicked.

The **Status Bar** at the base of the window will indicate when the action is completed. Messages from R (e.g., error messages) may appear in the **R Console** from which Rattle was started. Since Rattle is a simple graphical interface sitting on top of R itself, it is important to remember that some errors encountered by R on loading the data (and in fact during any operation performed by Rattle) may be displayed in the **R Console**.

The R code that **Rattle** passes on to R to execute underneath the interface is recorded in the **Log** tab. This allows us to review the R commands that perform the corresponding data mining tasks. The R code snippets can be copied as text from the **Log** tab and pasted into the R Console from which **Rattle** is running, to be directly executed. This allows us to deploy **Rattle** for basic tasks yet still gives us the full power of R to be deployed as needed, perhaps through using more command options than are exposed through the **Rattle** interface. This also allows us the opportunity to export the whole session as an R script file.

The log serves as a record of the actions taken and allows those actions to be repeated directly and automatically through R itself at a later time. Simply select (to display) the **Log** tab and click on the **Export** button. This will export the log to a file that will have an R extension. We can choose to include or exclude the extensive comments provided in the log and to rename the internal **Rattle** variables (from “**crs\$**” to a string of our own choosing).

We now traverse the main elements of the **Rattle** user interface, specifically the toolbar and menus. We begin with a basic concept—a project.

Projects

A project is a packaging of a dataset, variable selections, explorations, and models built from the data. **Rattle** allows projects to be saved for later resumption of the work or for sharing the data mining project with other users.

A project is typically saved to a file with a **rattle** extension. In fact, the file is a standard binary **RData** file used by R to store objects in a more compact binary form. Any R system can load such a file and hence have access to these objects, even without running **Rattle**.

Loading a **rattle** file into **Rattle** (using the **Open** button) will load that project into **Rattle**, restoring the data, models, and other displayed information related to the project, including the log and summary information. We can then resume our data mining from that point.

From a file system point of view, we can rename the files (as well as the filename extension, though that is not recommended) without impacting the project file itself—that is, the filename has no formal bearing on the contents, so use it to be descriptive. It is best to avoid spaces and unusual characters in the filenames.

Projects are opened and saved using the appropriate buttons on the toolbar or from the **Project** menu.

Toolbar

The most important button on the **Toolbar** (Figure 2.11) is the **Execute** button. All action is initiated with an **Execute**, often with a click of the **Execute** button. A keyboard shortcut for **Execute** is the F2 function key. A menu item for **Execute** is also available. It is worth repeating that the user interface paradigm used within **Rattle** is to set up the parameters on a tab and then **Execute** the tab.



Figure 2.11: The Rattle menu and toolbar.

The next few buttons on the **Toolbar** relate to the concept of a project within **Rattle**. Projects were discussed above.

Clicking on the **New** button will restore **Rattle** to its pristine startup state with no dataset loaded. This can be useful when a source dataset has been externally modified (external to **Rattle** and **R**). We might, for example, have manipulated our data in a spreadsheet or database program and re-exported the data to a CSV file. To reload this file into **Rattle**, if we have previously loaded it into the current **Rattle** session, we need to clear **Rattle** as with a click of the **New** button. We can then specify the filename and reload it.

The **Report** button will generate a formatted report based on the current tab. A number of report templates are provided with **Rattle** and will generate a document in the open standard **ODT** format, for the open source and open standards supporting LibreOffice. Whilst support for user-generated reports is limited, the log provides the necessary commands used to generate the **ODT** file. We can thus create our own **ODT** templates and apply them within the context of the current **Rattle** session.

The **Export** button is available to export various objects and entities from **Rattle**. Details are available together with the specific sections in the following chapters. The nature of the export depends on which tab is active and within the tab, which option is active. For example, if

the **Model** tab is on display then **Export** will save the current model as PMML (the Predictive Modelling Markup Language—see Chapter 16). The **Export** button is not available for all tabs and options.

Menus

The menus (Figure 2.11) provide alternative access to many of the functions of the interface. A key point in introducing menus is that they can be navigated from the keyboard and contain keyboard shortcuts so that we can navigate more easily through Rattle using the keyboard.

The **Project** menu provides access to the **Open** and **Save** options for loading and saving projects from or to files. The **Tools** menu provides access to some of the other toolbar functions as well as access to specific tabs. The **Settings** menu allows us to control a number of optional characteristics of Rattle. This includes tooltips and the use of the more modern Cairo graphics device.

Extensive help is available through the **Help** menu. The structure of the menu follows that of the tabs of the main interface. On selecting a help topic, a brief text popup will display some basic information. Many of the popups then have the option of displaying further information, which will be displayed within a Web browser. This additional documentation comes directly from the documentation provided by R or the relevant R package.

Interacting with Plots

It is useful to know how we interact with plots in Rattle. Often we will generate plots and want to include them in our own reports. Plots are generated from various places within the Rattle interface.

Rattle optionally uses the Cairo device, which is a vector graphics engine for displaying high-quality graphic plots. If the Cairo device is not available within your installation, then Rattle resorts to the default window device for the operating system (`x11()` for GNU/Linux and `window()` for Microsoft Windows). The **Settings** menu also allows control of the choice of graphics device (allowing us to use the default by disabling support for Cairo). The Cairo device has a number of advantages, one being that it can be encapsulated within other windows, as is done with Rattle. This allows Rattle to provide some operating-system-independent functionality and a common interface. If we choose not to

use the Cairo device, we will have the default devices, which still work just fine, but with less obvious functionality.

Figure 2.8 (page 34) shows a typical **Rattle** plot window. At the bottom of the window, we see a series of buttons that allow us to **Save** the plot to a file, to **Print** it, and **Close** it.

The **Save** button allows us to save the graphics to a file in one of the supported formats. The supported formats include **pdf** (for high-resolution pictures), **png** (for vector images and text), **jpg** (for colourful images), **svg** (for general scalable vector graphics), and, in Microsoft Windows, **wmf** (for Windows Metafile, Microsoft Windows-specific vector graphics). A popup will request the filename to save to. The default is to save in PDF format, saving to a file with the filename extension of **.pdf**. You can choose to save in the other formats simply by specifying the appropriate filename extension.

The **Print** button will send the plot to a printer. This requires the underlying R application to have been set up properly to access the required printer. This should be the case by default.

Once we are finished with the plot, we can click the **Close** button to shut down that particular plot window.

Keyboard Navigation

Keyboard navigation of the menus is usually initiated with the **F10** function key. The keyboard arrow keys can then be used to navigate. Pressing the keyboard's **Enter** key will then select the highlighted menu item.

Judicious use of the keyboard (in particular, the arrow keys, the **Tab** and **Shift-Tab** keys, and the **Enter** key, together with **F2** and **F10**) allows us to completely control **Rattle** from the keyboard if desired or required.

2.9 Interacting with R

R is a command line tool. We saw in Section 2.1 how to interact with R to start up **Rattle**. Essentially, R displays a prompt to indicate that it is waiting for us to issue a command. Two such commands are **library()** and **rattle()**. In this section, we introduce some basic concepts and commands for interacting with R directly.

Basic Functionality

Generally we instruct R to evaluate **functions**—a technical term used to describe mathematical objects that return a result. All functions in R return a result, and that result can be passed to other functions to do other things. This simple idea is actually a very powerful concept, allowing functions to do well what they are designed to do (like building a model) and pass on their output to other functions to do something with it (like formatting it for easy reading).

We saw in Section 2.1 two function calls, which we repeat below. The first was a call to the function `library()`, where we asked R to load **rattle**. We then started up Rattle with a call to the `rattle()` function:

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

Irrespective of the purpose of the function, for each function call we usually supply arguments that refine the behaviour of the function. We did that above in the call to `library()`, where the argument was **rattle**. Another simple example is to call `dim()` (dimensions) with the argument **weather**.

```
> dim(weather)
[1] 366 24
```

Here, **weather** is an *object* name. We can think of it simply as a reference to some object (something that contains data). The object in this case is the weather dataset as used in this chapter. It is organised as rows and columns. The `dim()` function reports the number of rows and columns.

If we type a name (e.g., either **weather** or **dim**) at the R prompt, R will respond by showing us the object. Typing **weather** (followed by pressing the Enter key) will result in the actual data. We will see all 366 rows of data scrolled on the screen. If we type **dim** and press Enter, we will see the definition of the function (which in this case is a primitive function coded into the core of R):

```
> dim
function (x) .Primitive("dim")
```


A common mistake made by new users is to type a function name by itself (without arguments) and end up a little confused about the resulting output. To actually invoke the function, we need to supply the argument list, which may be an empty list. Thus, at a minimum, we add `()` to the function call on the command line:

```
> dim()
Error in dim: 0 arguments passed to 'dim' which requires 1
```

As we see, executing this function will generate an error message. We note that `dim()` actually needs one argument, and no arguments were passed to it. Some functions can be invoked with no arguments, as is the case for `rattle()`.

The examples above illustrate how we will show our interaction with R. The “> ” is R’s prompt, and when we see that we know that R is waiting for commands. We type the string of characters `dim(weather)` as the command—in this case a call to the `dim` function. We then press the Enter key to send the command to R. R responds with the result from the function. In the case above, it returned the result `[1] 366 24`.

Technically, `dim()` returns a *vector* (a sequence of *elements* or values) of length 2. The `[1]` simply tells us that the first number we see from the vector (the 366) is the first element of the vector. The second element is 24.

The two numbers listed by R in the example above (i.e., the vector returned by `dim()`) are the number of rows and columns, respectively, in the *weather* dataset—that is, its dimensions.

For very long vectors, the list of the elements of the vector will be wrapped to fit across the screen, and each line will start with a number within square brackets to indicate what element of the vector we are up to. We can illustrate this with `seq()`, which generates a sequence of numbers:

```
> seq(1, 50)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

We saw above that we can view the actual data stored in an object by typing the name of the object (`weather`) at the command prompt.

Generally this will print too many lines (although only 366 in the case of the *weather* dataset). A useful pair of functions for inspecting our data are `head()` and `tail()`. These will list just the top and bottom six observations (or rows of data), by default, from the data frame, based on the order in which they appear there. Here we request, through the arguments to the function, to list the top two observations (and we also use indexing, described shortly, to list only the first nine variables):

```
> head(weather[1:9], 2)
```

	Date	Location	MinTemp	MaxTemp	Rainfall
1	2007-11-01	Canberra	8	24.3	0.0
2	2007-11-02	Canberra	14	26.9	3.6
	Evaporation	Sunshine	WindGustDir	WindGustSpeed	
1	3.4	6.3	NW	30	
2	4.4	9.7	ENE	39	

Similarly, we can request the bottom three rows of the dataset.

```
> tail(weather[1:9], 3)
```

	Date	Location	MinTemp	MaxTemp	Rainfall
364	2008-10-29	Canberra	12.5	19.9	0
365	2008-10-30	Canberra	12.5	26.9	0
366	2008-10-31	Canberra	12.3	30.2	0
	Evaporation	Sunshine	WindGustDir	WindGustSpeed	
364	8.4	5.3	ESE	43	
365	5.0	7.1	NW	46	
366	6.0	12.6	NW	78	

The *weather* dataset is more complex than the simple vectors we have seen above. In fact, it is a special kind of list called a *data frame*, which is one of the most common data structures in R for storing our datasets. A data frame is essentially a list of columns. The *weather* dataset has 24 columns. For a data frame, each column is a vector, each of the same length.

If we only want to review certain rows or columns of the data frame, we can *index* the dataset name. Indexing simply uses square brackets to list the row numbers and column numbers that are of interest to us:

```
> weather[4:8, 2:4]
  Location MinTemp MaxTemp
4 Canberra   13.3   15.5
5 Canberra    7.6   16.1
6 Canberra    6.2   16.9
7 Canberra    6.1   18.2
8 Canberra    8.3   17.0
```

Notice the notation for a sequence of numbers. The string `4:8` is actually equivalent to a call to `seq()` with two arguments, 4 and 8. The function returns a vector containing the integers from 4 to 8. It's the same as listing them all and combining them using `c()`:

```
> 4:8
[1] 4 5 6 7 8
> seq(4, 8)
[1] 4 5 6 7 8
> c(4, 5, 6, 7, 8)
[1] 4 5 6 7 8
```

Getting Help

It is important to know how we can learn more about using R. From the command line, we obtain help on commands by calling `help()`:

```
> help(dim)
```

A shorthand is to precede the argument with a `?` as in: `?dim`. This is automatically converted into a call to `help()`.

The `help.search()` function will search the documentation to list functions that may be of relevance to the topic we supply as an argument:

```
> help.search("dimensions")
```

The shorthand here is to precede the string with two question marks as in `??dimensions`.

A third command for searching for help on a topic is `RSiteSearch()`. This will submit a query to the R project's search engine on the Internet:

```
> RSiteSearch("dimensions")
```

Quitting R

Recall that to exit from R, as we saw in Section 2.1, we issue `q()`:

```
> q()
```

Our first session with R is now complete. The command line, as we have introduced here, is where we access the full power of R. But not everyone wants to learn and remember commands, so Rattle will get us started quite quickly into data mining, with only our minimal knowledge of the command line.

R and Rattle Interactions

Rattle generates R commands that are passed on through to R at various times during our interactions with Rattle. In particular, whenever the Execute button is clicked, Rattle constructs the appropriate R commands and then sends them off to R and awaits R's response.

We can also interact with R itself directly, and even interleave our interactions with Rattle and R. In Section 2.5, for example, we saw a decision tree model represented textually within Rattle's text view. The same can also be viewed in the R Console using `print()`. We can replicate that here once we have built the decision tree model as described in Section 2.5.

The R Console window is where we can enter R commands directly. We first need to make the window active, usually by clicking the mouse within that window. For the example below, we assume we have run Rattle on the *weather* dataset to build a decision tree as described in Section 2.5.

We can then type the `print()` command at the prompt. We see this in the code box below. The command itself consists of the name of an R function we wish to call on (`print()` in this case), followed by a list of *arguments* we pass to the function. The arguments provide information about what we want the function to do. The reference we see here, `crs$rpart`, identifies where the model itself has been saved internally by Rattle. The parameter `digits=` specifies the precision of the printed numbers. In this case we are choosing a single digit.

After typing the full command (including the function name and arguments) we then press the **Enter** key. This has the effect of passing the command to R. R will respond with the text exactly as shown below. The text starts with an indication of the number of observations (256). This is followed by the same textual presentation of the model we saw in Section 2.5.

```
> print(crs$rpart, digits=1)
n= 256

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 256 40 No (0.84 0.16)
  2) Pressure3pm>=1e+03 204 20 No (0.92 0.08)
    4) Cloud3pm< 8 195 10 No (0.95 0.05) *
    5) Cloud3pm>=8 9 3 Yes (0.33 0.67) *
  3) Pressure3pm< 1e+03 52 20 No (0.52 0.48)
    6) Sunshine>=9 25 5 No (0.80 0.20) *
    7) Sunshine< 9 27 7 Yes (0.26 0.74) *
```

Commands versus Functions

We have referred above to the R command line, where we enter commands to be executed. We also talked about functions that we type on the command line that make up the command to be executed. In this book, we will adopt a particular terminology around functions and commands, which we describe here.

In its true mathematical sense, a function is some operation that consumes some data and returns some result. Functions like `dim()`, `seq()`, and `head()`, as we have seen, do this. Functions might also have what we often call *side effects*—that is, they might do more than simply returning some result. In fact, the purpose of some functions is actually to perform some other action without necessarily returning a result. Such functions we will tend to call commands. The function `rattle()`, for example, does not return any result to the command line as such. Instead, its purpose is to start up the GUI and allow us to start data mining. Whilst `rattle()` is still a function, we will usually refer to it as a command rather than a function. The two terms can be used interchangeably.

Programming Styles for R

R is a programming language supporting different programming styles. We can use R to write programs that analyse data—we program the data analyses. Note that if we are only using **Rattle**, then we will not need to program directly. Nonetheless, for the programs we might write, we can take advantage of the numerous programming styles offered by R to develop code that analyses data in a consistent, simple, reusable, transparent, and error-free way.

Mistakenly, we are often trained to think that writing sentences in a programming language is primarily for the benefit of having a computer perform some activity for us. Instead, we should think of the task as really writing sentences that convey to other humans a story—a story about analysing our data. Coincidentally, we also want a computer to perform some activity.

Keeping this simple message in mind, whenever writing in R, helps to ensure we write in such a way that others can easily understand what we are doing and that we can also understand what we have done when we come back to it after six months or more.

Environments as Containers in R

For a particular project, we will usually analyse a collection of data, possibly transforming it and storing different bits of information about it. It is convenient to package all of our data and what we learn about it into some container, which we might save as a binary R object and reload more efficiently at a later time. We will use R's concept of an *environment* for this.

As a programming style, we can create a storage space and give it a name (i.e., it will look like a programming language variable) to act as a container. The container is an R environment and is initialised using `new.env()` (new environment). Here, we create a new environment and give it the name `en`:

```
> en <- new.env()
```

The object `en` now acts as a single container into which we can place all the relevant information associated with the dataset and that can also be shared amongst several models. We will store and access the relevant information from this container.

Data is placed into the container using the `$` notation and the assignment operator, as we see in the following example:

```
> en$obs <- 4:8
> en$obs
[1] 4 5 6 7 8
> en$vars <- 2:4
> en$vars
[1] 2 3 4
```

The variables `obs` and `vars` are now contained within the environment referenced as `en`.

We can operate on variables within an environment without using the `$` notation (which can become quite cumbersome) by wrapping the commands within `evalq()`:

```
> evalq(
  {
    nobs <- length(obs)
    nvars <- length(vars)
  }, en)
> en$nobs
[1] 5
> en$nvars
[1] 3
```

The use of `evalq()` becomes most convenient when we have more than a couple of statements to write.

At any time, we can list the contents of the container using `ls()`:

```
> ls(en)
[1] "nobs" "nvars" "obs" "vars"
```

Another useful function, provided by **gdata** ([Warnes, 2011](#)), is `ll()`, which provides a little more information:

```
> library(gdata)
> ll(en)

      Class KB
nobs integer 0
nvars integer 0
obs   integer 0
vars  integer 0
```

We can also convert the environment to a list using `as.list()`:

```
> as.list(en)

$nvars
[1] 3

$nobs
[1] 5

$vars
[1] 2 3 4

$obs
[1] 4 5 6 7 8
```

By keeping all the data related to a project together, we can save and load the project through this one object. We also avoid “polluting” the global environment with lots of objects and losing track of what they all related to, possibly confusing ourselves and others.

We can now also quite easily use the same variable names, but within different containers. Then, when we write scripts to build models, for example, often we will be able to use exactly the same scripts, changing only the name of the container. This encourages the reuse of our code and promotes efficiencies.

This approach is also sympathetic to the concept of object-oriented programming. The container is a basic “object” in the object-oriented programming context.

We will use this approach of encapsulating all of our data and information within a container when we start building models. The following provides the basic template:


```

> library(rpart)
> weatherDS <- new.env()
> evalq({
  data <- weather
  nobs <- nrow(data)
  vars <- c(2:22, 24)
  form <- formula(RainTomorrow ~ .)
  target <- all.vars(form)[1]
  train <- sample(nobs, 0.7*nobs)
}, weatherDS)
> weatherRPART <- new.env(parent=weatherDS)
> evalq({
  model <- rpart(formula=form, data=data[train, vars])
  predictions <- predict(model, data[-train, vars])
}, weatherRPART)

```

Here we have created two containers, one for the data and the other for the model. The model container (`weatherRPART`) has as its parent the data container (`weatherDS`), which is achieved by specifying the `parent=` argument. This makes the variables defined in the data container available within the model container.

To save a container to a file for use at a later time, or to document stages within the data mining project, use `save()`:

```

> save(weatherDS, file="weatherDS.Rdata")

```

It can later be loaded using `load()`:

```

> load("weatherDS.Rdata")

```

It can at times become tiresome to be wrapping our code up within a container. Whilst we retain the discipline of using containers we can also quickly interact with the variables in a container without having to specify the container each time. WE use `attach` and `detach` to add a container into the so called search path used by R to find variables. Thus we could do something like the following:

```

> attach(weatherRPART)
> print(model)
> detach(weatherRPART)

```

However, creating new variables to store within the environment will not work in the same way. Thus:

```
> attach(weatherRPART)
> new.model <- model
> detach(weatherRPART)
```

does not place the variable `new.model` into the `weatherRPART` environment. Instead it goes into the global environment.

A convenient feature, particularly with the layout used within the `evalq()` examples above and generally throughout the book, is that we could ignore the string that starts a block of code (which is the line containing “`evalq({`”) and the string that ends a block of code (which is the line containing “`}, weatherDS)`”) and simply copy-and-paste the other commands directly into the R console. The variables (`data`, `nobs`, etc.) are then created in the global environment, and nothing special is needed to access them. This is useful for quickly testing out ideas, for example, and is provided as a choice if you prefer not to use the container concept yourself. Containers do, however, provide useful benefits.

Rattle uses containers internally to collect together the data it needs. The Rattle container is called `crs` (the current rattle store). Once a dataset is loaded into Rattle, for example, it is stored as `crs$dataset`. We saw `crs$rpart` above as referring to the decision tree we built above.

2.10 Summary

In this chapter, we have become familiar with the Rattle interface for data mining with R. We have also built our first data mining model, albeit using an already prepared dataset. We have also introduced some of the basics of interacting with the R language.

We are now ready to delve into the details of data mining. Each of the following chapters will cover a specific aspect of the data mining process and illustrate how this is accomplished within Rattle and then further extended with direct coding in R.

Before proceeding, it is advisable to review Chapter 1 as an introduction to the overall data mining process if you have not already done so.

2.11 Command Summary

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

<code><-</code>	function	Assign a value into a named reference.
<code>c()</code>	function	Concatenate values into a vector.
<code>dim()</code>	function	Return the dimensions of a dataset.
<code>evalq()</code>	function	Access the environment for storing data.
<code>head()</code>	function	Return the first few rows of a dataset.
<code>help()</code>	command	Display help for a specific function.
<code>help.search()</code>	command	Search for help on a specific topic.
lattice	package	Interactive visualisation of data.
<code>library()</code>	command	Load a package into the R library.
<code>ll()</code>	function	Longer list of an environment.
<code>load()</code>	command	Load R objects from a file.
<code>ls()</code>	function	List the contents of an environment.
<code>new.env()</code>	function	Create a new object to store data.
<code>nrow()</code>	function	Number of rows in a dataset.
<code>print()</code>	command	Display representation of an R object.
<code>q()</code>	command	Quit from R.
R	shell	Start up the R statistical environment.
<code>rattle()</code>	command	Start the Rattle GUI.
rggobi	package	Interactive visualisation of data.
<code>rpart()</code>	function	Build a decision tree predictive model.
rpart	package	Provides decision tree functions.
<code>RSiteSearch()</code>	command	Search the R Web site for help.
<code>sample()</code>	function	Random selection of its first argument.
<code>save()</code>	command	Save R objects into a file.
<code>seq()</code>	function	Return a sequence of numbers.
<code>table()</code>	function	Make a table from some variables.
<code>tail()</code>	function	Return the last few rows of a dataset.
<i>weather</i>	dataset	Sample dataset from rattle .
<code>window()</code>	command	Open a new plot in Microsoft Windows.
<code>x11()</code>	command	Open a new plot in Unix/Linux.

<http://www.springer.com/978-1-4419-9889-7>

Data Mining with Rattle and R

The Art of Excavating Data for Knowledge Discovery

Williams, G.

2011, XX, 374 p. 95 illus., 80 illus. in color., Softcover

ISBN: 978-1-4419-9889-7