Appendix B

Sample Datasets

The following sections introduce the datasets that we use throughout the book to demonstrate data mining. R provides quite a collection of datasets. Each of the datasets we introduce here is available through R packages and may also be available from other sources.

In addition to introducing the datasets themselves, we also illustrate how they were derived. This entails presenting many new concepts from the R language. We purposefully do so within a real context of manipulating data to generate data suitable for data mining. A detailed understanding of many of these R programming constructs is not a necessity for understanding the material in this book. You are encouraged, though, to work through these programming examples at some time, as a data miner will need sophisticated tools for manipulating data.

Using the datasets supplied, as we do to demonstrate Rattle, oversimplifies the situation. This data will generally already be in a form suitable for mining, and in reality this is not what we find in a data mining project. In practise the data we are confronted with will come from a multitude of sources and will be in need of a variety of processing steps to clean it up and merge it into a single dataset. We have a lot of work in front of us in transforming a multitude of data into a form suitable for data mining. As we have reinforced throughout this is a major task.

The sizes of the datasets that we use throughout this book (and provided by **rattle**) are summarised below using `dim()`. The *weather* dataset, for example, has 366 observations and 24 variables:

```
> library(rattle)
> dim(weather)

[1] 366  24

> dim(weatherAUS)

[1] 39996    24

> dim(audit)

[1] 2000   13
```

B.1 Weather

We have seen the *weather* dataset in Chapter 2, where we identified it as being obtained[1] from the Australian Bureau of Meteorology's Web site. This is quite a small dataset, allowing the concepts we cover to be presented concisely. On the surface, it is not too complex, and most of us can relate to the issue of whether it might rain tomorrow! Keep in mind that real-world datasets tend not to be so small, and large datasets present many of the challenges we face in data mining.

The data used here (i.e., the actual *weather* dataset from **rattle**) comes from a weather monitoring station located in Canberra, Australia. The Bureau makes available 13 months of daily weather observations from many locations across Australia. The data is available as CSV (comma-separated value) files. The full list is available from http://www.bom.gov.au/climate/dwo/. Similar data is available from other government authorities around the world, as well as from many personal weather stations, which are now readily available.

B.1.1 Obtaining Data

The weather data for a specific month can be downloaded within R by using `read.csv()`. The process of doing so illustrates our interaction with R.

[1]Permission to use the dataset for this book and associated package (**rattle**) was obtained 17 December 2008 from mailto:webclim@bom.gov.au, and the Australian Bureau of Meteorology is acknowledged as the source of the data.

First, we need to determine the date for which we want to download data. For this, `Sys.Date()`, which returns the current date, will be useful:

```
> Sys.Date()

[1] "2011-06-13"
```

The current date can be passed to `format()`:

```
> today <- format(Sys.Date(), format="%Y%m")
```

This will generate a string consisting of the year and month, as specified by the `format=` argument to the function. The result of `format()` is stored into memory with a reference name of `today`. This is achieved using the assignment operator, `<-`, which itself is a function in R. No output is produced from the command above—the value is saved into the specified memory. We can inspect its contents simply by typing its name:

```
> today

[1] "201106"
```

Now we build up the actual Web address (the URL from which we can download the data) using `paste()`. This function takes a collection of strings and pastes them together into one string. Here we override the default behaviour of `paste()` using the `sep=` (i.e., separator) argument so that there will be no spaces between the strings that we are pasting together. We will save the result of the pasting into a memory reference named `bom`:

```
> bom <- paste("http://www.bom.gov.au/climate/dwo/", today,
               "/text/IDCJDW2801.", today, ".csv", sep="")
```

The string referenced by the name `bom` is then the URL[2] to extract the current month's data for the Canberra weather station (identified as IDCJDW2801):

```
> bom

[1] "http:[...]/dwo/201106/text/IDCJDW2801.201106.csv"
```

[2]Note that the URL here is correct at the time of publication but could change over time.

Note the use of the string "`[...]`" in the output—this is used so that the result is no wider than the printed page. We will use this notation often to indicate where data has been removed for typesetting purposes.

The most recent observations from the Bureau can now be read. Note that, for the benefit of stability, the actual dataset used below is from a specific date, June 2009, and the details of other data obtained at different times will differ. The first few lines of the downloaded file contain information about the location, and so we skip those lines by using the `skip=` argument of `read.csv()`. The `check.names=` argument is set to `FALSE` (the default is `TRUE`) so that the column names remain exactly as recorded in the file:

```
> dsw <- read.csv(bom, skip=6, check.names=FALSE)
```

By default, R will convert them into names that it can more easily handle, for example, replacing spaces with a period. We will fix the names ourselves shortly.

The dataset is not too large, as shown by `dim()` (consisting of up to one month of data), and we can use `names()` to list the names of the variables included:

```
> dim(dsw)

[1] 28 22

> head(names(dsw))

[1] ""
[2] "Date"
[3] "Minimum temperature (\xb0C)"
[4] "Maximum temperature (\xb0C)"
[5] "Rainfall (mm)"
[6] "Evaporation (mm)"
```

Note that, if you run this code yourself, the dimensions will most likely be different. The data you download today will be different from the data downloaded when this book was processed by R. In fact, the number of rows should be about the same as the day number of the current month.

B.1.2 Data Preprocessing

We do not want all of the variables. In particular, we will ignore the first column and the time of the maximum wind gust (variable number 10). The first command below will remove these two columns from the dataset. We then simplify the names of the variables to make it easier to refer to them. This can be done as follows using `names()`:

```
> ndsw <- dsw[-c(1, 10)]
> names(ndsw) <- c("Date", "MinTemp", "MaxTemp",
    "Rainfall", "Evaporation", "Sunshine",
    "WindGustDir", "WindGustSpeed", "Temp9am",
    "Humidity9am", "Cloud9am", "WindDir9am",
    "WindSpeed9am", "Pressure9am", "Temp3pm",
    "Humidity3pm", "Cloud3pm", "WindDir3pm",
    "WindSpeed3pm", "Pressure3pm")
```

We can now check that the new dataset has the right dimensions and variable names:

```
> dim(ndsw)

[1] 28 20

> names(ndsw)

 [1] "Date"          "MinTemp"       "MaxTemp"
 [4] "Rainfall"      "Evaporation"   "Sunshine"
 [7] "WindGustDir"   "WindGustSpeed" "Temp9am"
[10] "Humidity9am"   "Cloud9am"      "WindDir9am"
[13] "WindSpeed9am"  "Pressure9am"   "Temp3pm"
[16] "Humidity3pm"   "Cloud3pm"      "WindDir3pm"
[19] "WindSpeed3pm"  "Pressure3pm"
```

B.1.3 Data Cleaning

We must also clean up some of the variables. We start with the wind speed. To view the first few observations, we can use `head()`. We further limit our review to just three variables, which we explicitly list as the column index:

```
> vars <- c("WindGustSpeed","WindSpeed9am","WindSpeed3pm")
> head(ndsw[vars])

  WindGustSpeed WindSpeed9am WindSpeed3pm
1            24         Calm           15
2            31            6           13
3            22            9           17
4            11            6         Calm
5            20            6            7
6            39            2           28
```

Immediately, we notice that not all the wind speeds are numeric. The variable `WindSpeed9am` has a value of *Calm* for the first observation, and so R is representing this data as a categoric, and not as a numeric as we might be expecting. We can confirm this using `class()` to tell us what class of data type the variable is.

First, we confirm that `ndsw` is a data frame (which is R's representation of a dataset):

```
> class(ndsw)

[1] "data.frame"
```

With the next example, we introduce `apply()` to apply `class()` to each of the variables of interest. We confirm that the variables are character strings:

```
> apply(ndsw[vars], MARGIN=2, FUN=class)

WindGustSpeed  WindSpeed9am  WindSpeed3pm
  "character"   "character"   "character"
```

The `MARGIN=` argument chooses between applying the supplied function to the rows of the dataset or to the columns (i.e., variables) of the dataset. The `2` selects columns, whilst `1` selects rows. The function that is applied is supplied with the `FUN=` argument.

To transform these variables, we introduce a number of common R constructs. We first ensure that we are treating the variable as a character string by converting it (although somewhat redundantly in this case) with `as.character()`:

```
> ndsw$WindSpeed9am  <- as.character(ndsw$WindSpeed9am)
> ndsw$WindSpeed3pm  <- as.character(ndsw$WindSpeed3pm)
> ndsw$WindGustSpeed <- as.character(ndsw$WindGustSpeed)
> head(ndsw[vars])

  WindGustSpeed WindSpeed9am WindSpeed3pm
1            24         Calm           15
2            31            6           13
3            22            9           17
4            11            6         Calm
5            20            6            7
6            39            2           28
```

B.1.4 Missing Values

We next identify that empty values (i.e., an empty string) represent missing data, and so we replace them with R's notion of missing values (`NA`). The `within()` function can be used to allow us to directly reference variables within the dataset without having to prefix them with the name of the dataset (i.e., avoiding having to use `ndsw$WindSpeed9am`):

```
> ndsw <- within(ndsw,
           {
             WindSpeed9am[WindSpeed9am   == ""] <- NA
             WindSpeed3pm[WindSpeed3pm   == ""] <- NA
             WindGustSpeed[WindGustSpeed == ""] <- NA
           })
```

Then, *Calm*, meaning no wind, is replaced with 0, which suits our numeric data type better:

```
> ndsw <- within(ndsw,
           {
             WindSpeed9am[WindSpeed9am   == "Calm"] <- "0"
             WindSpeed3pm[WindSpeed3pm   == "Calm"] <- "0"
             WindGustSpeed[WindGustSpeed == "Calm"] <- "0"
           })
```

Finally, we convert the character strings to the numbers they actually represent using `as.numeric()`, and check the data type class to confirm they are now numeric:

```
> ndsw <- within(ndsw,
          {
            WindSpeed9am  <- as.numeric(WindSpeed9am)
            WindSpeed3pm  <- as.numeric(WindSpeed3pm)
            WindGustSpeed <- as.numeric(WindGustSpeed)
          })
> apply(ndsw[vars], 2, class)

WindGustSpeed  WindSpeed9am  WindSpeed3pm
    "numeric"     "numeric"     "numeric"
```

The wind direction variables also need some transformation. We see below that the wind direction variables are categoric variables (they are technically *factors* in R's nomenclature). Also note that one of the possible values is the string consisting of just a space, and that the levels are ordered alphabetically:

```
> vars <- c("WindSpeed9am","WindSpeed3pm","WindGustSpeed")
> head(ndsw[vars])

  WindSpeed9am WindSpeed3pm WindGustSpeed
1            0           15            24
2            6           13            31
3            9           17            22
4            6            0            11
5            6            7            20
6            2           28            39

> apply(ndsw[vars], 2, class)

 WindSpeed9am  WindSpeed3pm WindGustSpeed
    "numeric"     "numeric"     "numeric"

> levels(ndsw$WindDir9am)

 [1] " "   "E"   "ENE" "ESE" "N"   "NNW" "NW"  "S"   "SE"
[10] "SSE" "SW"  "WNW"
```

To deal with missing values, which are represented in the data as an empty string (corresponding to a wind speed of zero), we map such data to `NA`:

```
> ndsw <- within(ndsw,
  {
    WindDir9am[WindDir9am == " "] <- NA
    WindDir9am[is.na(WindSpeed9am) |
               (WindSpeed9am == 0)] <- NA

    WindDir3pm[WindDir3pm == " "] <- NA
    WindDir3pm[is.na(WindSpeed3pm) |
               (WindSpeed3pm == 0)] <- NA

    WindGustDir[WindGustDir == " "] <- NA
    WindGustDir[is.na(WindGustSpeed) |
                (WindGustSpeed == 0)] <- NA
  })
```

B.1.5 Data Transforms

Another common operation on a dataset is to create a new variable from other variables. An example is to capture whether it rained today. This can be simply determined, by definition, through checking whether there was more than 1 mm of rain today. We use `ifelse()` to do this in one step:

```
> ndsw$RainToday <- ifelse(ndsw$Rainfall > 1, "Yes", "No")
> vars <- c("Rainfall", "RainToday")
> head(ndsw[vars])

  Rainfall RainToday
1      0.6        No
2      0.0        No
3      1.6       Yes
4      8.6       Yes
5      2.2       Yes
6      1.4       Yes
```

We want to also capture and associate with today's observation whether it rains tomorrow. This is to become our target variable. Once again, if it rains less than 1 mm tomorrow, then we report that as no rain. To capture this variable, we need to consider the observation of rainfall recorded on the following day. Thus, when we are considering today's observation (e.g., observation number 1), we want to consider tomorrow's observation (observation 2) of `Rainfall`. That is, there is a lag of one day in determining today's value of the variable `RainTomorrow`.

A simple approach can be used to calculate `RainTomorrow`. We simply note the value of `RainToday` for the next day's observation. Thus, we build up the list of observations for `RainToday` starting with the second observation (ignoring the first). An additional observation then needs to be added for the final day (for which we have no observation for the following day):

```
> ndsw$RainTomorrow <- c(ndsw$RainToday[2:nrow(ndsw)], NA)
> vars <- c("Rainfall", "RainToday", "RainTomorrow")
> head(ndsw[vars])

  Rainfall RainToday RainTomorrow
1      0.6        No           No
2      0.0        No          Yes
3      1.6       Yes          Yes
4      8.6       Yes          Yes
5      2.2       Yes          Yes
6      1.4       Yes           No
```

Finally, we would also like to record the amount of rain observed "tomorrow." This is achieved as follows using the same lag approach:

```
> ndsw$RISK_MM <- c(ndsw$Rainfall[2:nrow(ndsw)], NA)
> vars <- c("Rainfall", "RainToday",
            "RainTomorrow", "RISK_MM")
> head(ndsw[vars])

  Rainfall RainToday RainTomorrow RISK_MM
1      0.6        No           No     0.0
2      0.0        No          Yes     1.6
3      1.6       Yes          Yes     8.6
4      8.6       Yes          Yes     2.2
5      2.2       Yes          Yes     1.4
6      1.4       Yes           No     0.8
```

The source dataset has now been processed to include a variable that we might like to treat as a target variable—to indicate whether it rained the following day.

B.1.6 Using the Data

Using this historic data, we can now build a model (as we did in Chapter 2) that might help us to decide whether we need to take an umbrella with us tomorrow if we live in Canberra. (You may like to try this on local data for your own region.)

Above, we retrieved up to one month of observations. We can repeat the process, using the same code, to obtain 12 months of observations. This has been done to generate the *weather* dataset provided by Rattle. The *weather* dataset covers only Canberra for the 12 month period from 1 November 2007 to 31 October 2008 inclusive.

Rattle also provides the *weatherAUS* dataset, which captures the weather observations for more than a year from over 45 weather observation stations throughout Australia. The format of the *weatherAUS* dataset is exactly the same as for the *weather* dataset. In fact, the *weather* dataset is a subset of the *weatherAUS* dataset, and we could reconstruct it with the following R code using `subset()`:

```
> cbr <- subset(weatherAUS,
                Location == "Canberra" &
                Date >= "2007-11-01" &
                Date <= "2008-10-31")
```

The `subset()` function takes as its first argument a dataset, and as its second argument a logical expression that specifies the rows of the data that we wish to retain in the result.

We can check that this results in the same dataset as the *weather* dataset by simply testing if they are equal using `==`:

```
> cbr == weather
```

This will print a lot of `TRUE`s to the screen, as it compares each value from the *cbr* dataset with the corresponding value from the *weather* dataset. We could have a look through what is printed to make sure they are all `TRUE`, but that's not very efficient.

Instead, we can find the number of actual values that are compared. First, we get the two dimensions, using `dim()`, and indeed the two datasets have the same dimensions:

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

Then we calculate the number of data items within the dataset. To do this, we could multiply the first and second values returned by dim(). Instead, we will introduce two new handy functions to return the number of rows and the number of columns in the dataset:

```
> dim(cbr)[1] * dim(cbr)[2]
[1] 8784
> nrow(cbr) * ncol(cbr)
[1] 8784
```

Now we count the number of TRUEs when comparing the two datasets value by value. Noting that TRUE corresponds to the numeric value 1 in R and FALSE corresponds to 0, we can simply sum the data. We need to remove NAs to get a sum, otherwise sum() will return NA. We also need to count the number of NAs removed, which we do. Note that the totals all add up to 8784! The two datasets are the same.

```
> sum(cbr == weather, na.rm=TRUE)
[1] 8737
> sum(is.na(cbr))
[1] 47
> sum(is.na(weather))
[1] 47
> sum(cbr == weather, na.rm=TRUE) + sum(is.na(cbr))
[1] 8784
```

The sample *weather* dataset can also be downloaded directly from the Rattle Web site:

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

B.2 Audit

Another dataset we will use for illustrating data mining is the *audit* dataset, which is also provided by **rattle**. The data is artificial but reflects a real-world dataset used for reviewing the outcomes of historical financial audits. Picture, for example, your country's revenue authority (e.g., the Internal Revenue Service in the USA, Inland Revenue in the UK or the Australian Taxation Office). Revenue authorities collect taxes and reconcile the taxes we pay each year against the information we supply to them.

Many thousands of audits of taxpayers' tax returns might be performed each year. The outcome of an audit may be productive, in which case an adjustment to the information supplied was required, usually resulting in a change to the amount of tax that the taxpayer is liable to pay (an increase or a decrease). An unproductive audit is one for which no adjustment was required after reviewing the taxpayer's affairs.

The *audit* dataset attempts to simulate this scenario. It is supplied as both an R data file and a CSV file. The dataset consists of 2000 fictional taxpayers who have been audited for tax compliance. For each case, an outcome of the audit is recorded (i.e., whether the financial claims had to be adjusted or not). The actual dollar amount of any adjustment that resulted is also recorded (noting that adjustments can go in either direction). The *audit* dataset contains 13 variables, with the first variable being a unique client identifier. It is, in fact, derived from the so-called *adult* dataset.

B.2.1 The Adult Survey Dataset

Like the *weather* dataset, the *audit* dataset is actually derived from another freely available dataset. Unlike the *weather* dataset, the *audit* dataset is purely fictional. We will discuss here how the data was derived from the so-called adult *survey* dataset available from the University of California at Irvine's machine learning[3] repository. We use this dataset as the starting point and will perform various transformations of the data with the aim of building a dataset that looks more like an audit dataset. With the purpose of further illustrating the data manipulation capabilities of R, we review the derivation process.

[3]http://archive.ics.uci.edu/ml/.

First, we `paste()` together the constituent parts of the URL from which the dataset is obtained. Note the use of the `sep=` (separator) argument to include a "/" between the constituent parts. We then use `download.file()` to retrieve the actual file from the Internet, and to save it under the name `survey.csv`:

```
> uci <- paste("ftp://ftp.ics.uci.edu/pub",
               "machine-learning-databases",
               "adult/adult.data", sep="/")
> download.file(uci, "survey.csv")
```

The file is now stored locally and can be loaded into R. Because the file is a CSV file, we use `read.csv()` to load it into R:

```
> survey <- read.csv("survey.csv", header=FALSE,
                     strip.white=TRUE, na.strings="?",
            col.names=c("Age", "Workclass", "fnlwgt",
              "Education", "Education.Num",
              "Marital.Status", "Occupation",
              "Relationship", "Race", "Gender",
              "Capital.Gain", "Capital.Loss",
              "Hours.Per.Week", "Native.Country",
              "Salary.Group"))
```

The additional arguments of `read.csv()` are used to fine-tune how the data is read into R. The `header=` argument needs to be set to `FALSE` since the data file has no header row (a header row is the first row and lists the variable or column names—here, though, it is data, not a header). We set `strip.white=` to `TRUE` to strip spaces from the data to ensure we do not get extra white space in any columns. Missing values are notated with a question mark, so we tell the function this with the `na.strings=` argument. Finally, we supply a list of variable names using the `col.names=` (column names) argument.

B.2.2 From Survey to Audit

We begin to turn the *survey* data into the *audit* dataset, first by selecting a subset of the columns and then renaming some of the columns (reinforcing again that this is a fictitious dataset):

```
> audit <- survey[,c(1:2,4,6:8,10,12:14,11,15)]
> names(audit)[c(seq(2, 8, 2), 9:12)] <-
    c("Employment", "Marital", "Income", "Deductions",
      "Hours", "Accounts", "Adjustment", "Adjusted")
```

Here we see a couple of interesting language features of R. We have previously seen the use of `names()` to retrieve the variable names from the dataset. This function returns a list of names, and we can index the data items within the list as usual. The interesting feature here is that we are assigning into that resulting list another list. The end result is that the variable names are thereby actually changed within the dataset:

```
> names(audit)

 [1] "Age"        "Employment" "Education"  "Marital"
 [5] "Occupation" "Income"     "Gender"     "Deductions"
 [9] "Hours"      "Accounts"   "Adjustment" "Adjusted"
```

B.2.3 Generating Targets

We now look at what will become the output variables, `Adjustment` and `Adjusted`. These will be interpreted as the dollar amount of any adjustment made to the tax return and whether or not there was an adjustment made, respectively. Of course, they need to be synchronised.

The variable `Adjusted` is going to be a binary integer variable that takes on the value 0 when an audit was not productive and the value 1 when an audit was productive. Initially the variable is a categoric variable (i.e., of class *factor* in R's nomenclature) with two distinct values (i.e., two distinct *levels* in R's nomenclature). We use R's `table()` to report on the number of observations having each of the two distinct output values:

```
> class(audit$Adjusted)

[1] "factor"

> levels(audit$Adjusted)

[1] "<=50K" ">50K"

> table(audit$Adjusted)

<=50K  >50K
24720  7841
```

We now convert this into a binary integer variable for convenience. This is not strictly necessary, but often our mathematics in describing algorithms works nicely when we think of the target as being 0 or 1. The `as.integer()` function will transform a categoric variable into a numeric variable. R uses the integer `1` to represent the first categoric value and `2` to represent the second categoric value. So to turn this into our desired `0` and `1` values we simply subtract 1 from each integer:

```
> audit$Adjusted <- as.integer(audit$Adjusted)-1
> class(audit$Adjusted)

[1] "numeric"

> table(audit$Adjusted)

    0     1
24720  7841
```

It is instructive to understand the subtraction that is performed here. In particular, the `1` is subtracted from each data item. In R, we can subtract one list of numbers from another, but they generally need to be the same length. The subtraction occurs pairwise. If one list is shorter than the other, then it is recycled as many times as required to perform the operation. Thus, `1` is recycled as many times as the number of observations of `Adjusted`, with the end result we noted. The concept can be illustrated simply:

```
> 11:20 - 1:10

 [1] 10 10 10 10 10 10 10 10 10 10

> 11:20 - 1:5

 [1] 10 10 10 10 10 15 15 15 15 15

> 11:20 - 1

 [1] 10 11 12 13 14 15 16 17 18 19
```

Some mathematics is now required to ensure that most productive cases (those observations for which `Adjusted` is `1`) have an adjustment (i.e., the variable `Adjustment` is nonzero) and nonproductive cases necessarily have an adjustment of 0.

We first calculate the number of productive cases that have a zero adjustment (saving the result into the reference `prod`) and the number of nonproductive cases that have a nonzero adjustment (saving the result into the reference `nonp`):

```
> prod <- sum(audit$Adjusted == 1 & audit$Adjustment == 0)
> prod

[1] 6164

> nonp <- sum(audit$Adjusted == 0 & audit$Adjustment != 0)
> nonp

[1] 1035
```

This example again introduces a number of new concepts from the R language. We will break them down one at a time and then come back to the main story.

Recall that the notation `audit$Adjusted` refers to the observations of the variable `Adjusted` of the *audit* dataset. As with the subtraction of a single value, `1`, from such a list of observations, as we saw above, the comparison operator `==` (as well as `!=` to test not equal) operates over such data. It tests each observation to see if it is equal to, for example, `1`.

The following example illustrates this. Consider just the first few observations of the variables `Adjusted` and `Adjustment`. R notates logical variables with the observations `TRUE` or `FALSE`. The “`&`” operator is used for comparing lists of logical values pairwise:

```
> obs <- 1:9
> audit$Adjusted[obs]

[1] 0 0 0 0 0 0 0 1 1

> audit$Adjusted[obs]==1

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE

> audit$Adjustment[obs]

[1]  2174     0     0     0     0     0     0     0 14084

> audit$Adjustment[obs] == 0

[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

> audit$Adjusted[obs] == 1 & audit$Adjustment[obs] == 0

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
```

Notice that the final example here is used as the argument to `sum()` in our code above. This summation relies on the fact that `TRUE` is treated as the integer `1` and `FALSE` as the integer `0` when needed. Thus

```
> sum(audit$Adjusted[obs]==1 & audit$Adjustment[obs]==0)

[1] 1
```

so that there is only one observation where both of the conditions are `TRUE`. Over the whole dataset:

```
> sum(audit$Adjusted == 1 & audit$Adjustment == 0)

[1] 6164
```

For these 6164 observations, we note that they are regarded as being adjusted yet there is no adjustment amount recorded for them. So, for some majority of these observations, we want to ensure that they do have an adjustment recorded for them, as we would expect from real data.

The following formulation uses the integer division function `%/%` to divide `prod` by `nonp` and then multiply the result by `nonp`. This will usually result in a number that is a little less than the original `prod` and will be the number of observations that we will adjust to have a nonzero `Adjustment`:

```
> adj <- (prod %/% nonp) * nonp
```

The resulting value, saved as `adj` (5175), is thus an integer multiple (5) of the value of `nonp` (1035). The significance of this will be apparent shortly.

Now we make the actual change from `0` to a random integer. To do so, we take the values that are present in the data for adjustments where `Adjusted` is actually 0 (the `nonp` observations) and multiply them by a random number. The result is assigned to an adjusted observation that currently has a `0` `Adjustment`. The point around the integer multiple of `nonp`, noted above, is that the following will effectively replicate the current `nonp` observations an integer number of times to assign them to the subset of the `prod` observations being modified:

```
> set.seed(12345)
> audit[audit$Adjusted == 1 & audit$Adjustment == 0,
        'Adjustment'][sample(prod, adj)] <-
    as.integer(audit[audit$Adjusted == 0 &
                     audit$Adjustment != 0, 'Adjustment'] *
               rnorm(adj, 2))
```

There is quite a lot happening in these few lines of code. First off, because we are performing random sampling (using `sample()` and `rnorm()`), we use `set.seed()` to start the random number generation from a known point, and thus the process is repeatable (we will get the same random numbers each time). We could have used any number as the argument to `set.seed()` as long as we use the same number each time.

Next, we see quite a complex assignment. First, we index *audit* to include just those observations marked as adjusted yet having no adjustment. For these observations, we extract just the `Adjustment` variable (noting that all resulting observations will be `0`). The point of the expression, though, is to identify the locations in memory where this data is stored.

The variable is further indexed by a random sample (using `sample()`) of `adj` (5175) numbers between 1 and `prod` (6164). These are the observations for which we will be changing the `Adjustment` from 0 to something else.

The remainder of the assignment command calculates the replacement numbers. This time *audit* is indexed to obtain those nonproductive observations with a nonzero adjustment. These 5175 values are multiplied by a sequence of `adj` (5175) random numbers. The random numbers are normally distributed with a mean of 2 and are generated using `rnorm()`.

That is quite a complex operation on the data. With a little bit of familiarity, and breaking down the operation into its constituent parts, we can understand what it does.

We need to tidy up one last operation involving the `Adjustment` variable. Observations marked as having a nonproductive outcome should have a value of `0` for `Adjustment`. The following will achieve this:

```
> audit[audit$Adjusted == 0 & audit$Adjustment != 0,
        'Adjustment'] <- 0
```

B.2.4 Finalising the Data

The remainder of the operations we perform on the *audit* dataset are similar in nature, and we now finalise the data. The observations of `Deductions`, for nonadjusted cases, are reduced to be closer to 0, reflecting a likely scenario:

```
> audit[audit$Adjusted==0, 'Deductions'] <-
    audit[audit$Adjusted==0, 'Deductions']/1.5
```

To keep to a smaller dataset for illustrative purposes, we sample 2000 rows:

```
> set.seed(12345)
> cases <- sample(nrow(audit), 2000)
```

Finally, we add to the beginning of the variables contained in the dataset a new variable that serves the role of a unique identifier. The identifiers are randomly generated using `runif()`. This generates random numbers from a uniform distribution. We use it to generate 2000 random numbers of seven digits:

```
> set.seed(12345)
> idents <- as.integer(sort(runif(2000, 1000000, 9999999)))
> audit <- cbind(ID=idents, audit[cases,])
```

B.2.5 Using the Data

The final version of the *audit* dataset, as well as being available from **rattle**, can also be downloaded directly from the Rattle Web site:

```
> twaudit <- "http://rattle.togaware.com/audit.csv"
> myaudit <- read.csv(twaudit)
```

B.3 Command Summary

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

`apply()`	function	Apply a function over a list.
`as.character()`	function	Convert to character string.
`as.integer()`	function	Convert to integer.
`as.numeric()`	function	Convert to numeric.
audit	dataset	Sample dataset from **rattle**.
`class()`	function	Identify type of object.
`dim()`	function	Report the rows/columns of a dataset.
`download.file()`	function	Download file from URL.
`format()`	function	Format an object.
`head()`	function	Show top observations of a dataset.
`names()`	function	Show variables contained in a dataset.
`paste()`	function	Combine strings into one string.
`read.csv()`	function	Read a comma-separated data file.
`rnorm()`	function	Generate random numbers.
`sample()`	function	Generate a random sample of numbers.
`subset()`	function	Create a subset of a dataset.
`sum()`	function	Add the supplied numbers.
survey	dataset	A sample dataset from UCI repository.
`Sys.Date()`	function	Return the current date and time.
`table()`	function	Summarise distribution of a variable.
weather	dataset	Sample dataset from **rattle**.
weatherAUS	dataset	A larger dataset from **rattle**.
`within()`	function	Perform actions within a dataset.