17.9: Reading Unusual Data Files

In this section I'm going to switch topics (again!) and turn to the question of how you can load data from a range of different sources. Throughout this book I've assumed that your data are stored as an `.Rdata` file or as a "properly" formatted CSV file. And if so, then the basic tools that I discussed in Section 4.5 should be quite sufficient. However, in real life that's not a terribly plausible assumption to make, so I'd better talk about some of the other possibilities that you might run into.

17.9.1 Loading data from text files

The first thing I should point out is that if your data are saved as a text file but aren't *quite* in the proper CSV format, then there's still a pretty good chance that the `read.csv()` function (or equivalently, `read.table()`) will be able to open it. You just need to specify a few more of the optional arguments to the function. If you type `?read.csv` you'll see that the `read.csv()` function actually has several arguments that you can specify. Obviously you need to specify the `file` that you want it to load, but the others all have sensible default values. Nevertheless, you will sometimes need to change them. The ones that I've often found myself needing to change are:

- `header`. A lot of the time when you're storing data as a CSV file, the first row actually contains the column names and not data. If that's not true, you need to set `header = FALSE`.
- `sep`. As the name "comma separated value" indicates, the values in a row of a CSV file are usually separated by commas. This isn't universal, however. In Europe the decimal point is typically written as `,` instead of `.` and as a consequence it would be somewhat awkward to use `,` as the separator. Therefore it is not unusual to use `;` over there. At other times, I've seen a TAB character used. To handle these cases, we'd need to set `sep = ";"` or `sep = "\t"`.
- `quote`. It's conventional in CSV files to include a quoting character for textual data. As you can see by looking at the `booksales.csv}` file, this is usually a double quote character, `"`. But sometimes there is no quoting character at all, or you might see a single quote mark `'` used instead. In those cases you'd need to specify `quote = ""` or `quote = "'"`.
- `skip`. It's actually very common to receive CSV files in which the first few rows have nothing to do with the actual data. Instead, they provide a human readable summary of where the data came from, or maybe they include some technical info that doesn't relate to the data. To tell R to ignore the first (say) three lines, you'd need to set `skip = 3`
- `na.strings`. Often you'll get given data with missing values. For one reason or another, some entries in the table are missing. The data file needs to include a "special" string to indicate that the entry is missing. By default R assumes that this string is `NA`, since that's what *it* would do, but there's no universal agreement on what to use in this situation. If the file uses `???` instead, then you'll need to set `na.strings = "???"`.

It's kind of nice to be able to have all these options that you can tinker with. For instance, have a look at the data file shown pictured in Figure 7.1. This file contains almost the same data as the last file (except it doesn't have a header), and it uses a bunch of wacky features that you don't normally see in CSV files. In fact, it just so happens that I'm going to have to change all five of those arguments listed above in order to load this file. Here's how I would do it:

```
data <- read.csv( file = "./rbook-master/data/booksales2.csv",  # specify the name o
                  header = FALSE,           # variable names in the file?
                  skip = 8,                 # ignore the first 8 lines
                  quote = "*",              # what indicates text data?
                  sep = "\t",               # what separates different entries?
                  na.strings = "NFI" )      # what is the code for missing data?
```

If I now have a look at the data I've loaded, I see that this is what I've got:

```
head( data )
```

```
##          V1 V2  V3   V4
## 1   January 31   0 high
## 2 February  28 100 high
## 3    March  31 200  low
## 4    April  30  50  out
## 5    May    31  NA  out
## 6    June   30   0 high
```

Because I told R to expect `*` to be used as the quoting character instead of `"` ; to look for tabs (which we write like this: `\t`) instead of commas, and to skip the first 8 lines of the file, it's basically loaded the right data. However, since `booksales2.csv` doesn't contain the column names, R has made them up. Showing the kind of imagination I expect from insentient software, R decided to call them `V1` , `V2` , `V3` and `V4` . Finally, because I told it that the file uses "NFI" to denote missing data, R correctly figures out that the sales data for May are actually missing.

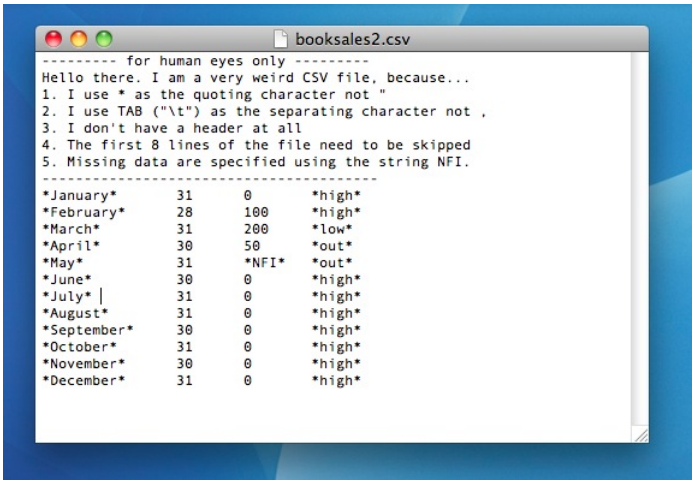

Figure 7.1: The `booksales2.csv` data file. It contains more or less the same data as the original `booksales.csv` data file, but has a lot of very quirky features.

In real life you'll rarely see data this stupidly formatted.[125]

17.9.2 Loading data from SPSS (and other statistics packages)

The commands listed above are the main ones we'll need for data files in this book. But in real life we have many more possibilities. For example, you might want to read data files in from other statistics programs. Since SPSS is probably the most widely used statistics package in psychology, it's worth briefly showing how to open SPSS data files (file extension `.sav`). It's surprisingly easy. The extract below should illustrate how to do so:

```
library( foreign )                # load the package
X <- read.spss( "./rbook-master/data/datafile.sav" )   # create a list containing the
X <- as.data.frame( X )           # convert to data frame
```

If you wanted to import from an SPSS file to a data frame directly, instead of importing a list and then converting the list to a data frame, you can do that too:

```
X <- read.spss( file = "datafile.sav", to.data.frame = TRUE )
```

And that's pretty much it, at least as far as SPSS goes. As far as other statistical software goes, the `foreign` package provides a wealth of possibilities. To open SAS files, check out the `read.ssd()` and `read.xport()` functions. To open data from Minitab, the `read.mtp()` function is what you're looking for. For Stata, the `read.dta()` function is what you want. For Systat, the `read.systat()` function is what you're after.

17.9.3 Loading Excel files

A different problem is posed by Excel files. Despite years of yelling at people for sending data to me encoded in a proprietary data format, I get sent a lot of Excel files. In general R does a pretty good job of opening them, but it's bit finicky because Microsoft don't seem to be terribly fond of people using non-Microsoft products, and go to some lengths to make it tricky. If you get an Excel file, my suggestion would be to open it up in Excel (or better yet, OpenOffice, since that's free software) and then save the spreadsheet as a CSV file. Once you've got the data in that format, you can open it using `read.csv()`. However, if for some reason you're desperate to open the `.xls` or `.xlsx` file directly, then you can use the `read.xls()` function in the `gdata` package:

```
library( gdata )                  # load the package
X <- read.xls( "datafile.xlsx" )  # create a data frame
```

This usually works. And if it doesn't, you're probably justified in "suggesting" to the person that sent you the file that they should send you a nice clean CSV file instead.

17.9.4 Loading Matlab (& Octave) files

A lot of scientific labs use Matlab as their default platform for scientific computing; or Octave as a free alternative. Opening Matlab data files (file extension `.mat`) slightly more complicated, and if it wasn't for the fact that Matlab is so very widespread and is an extremely good platform, I wouldn't mention it. However, since Matlab is so widely used, I think it's worth discussing briefly how to get Matlab and R to play nicely together. The way to do this is to install the `R.matlab` package (don't forget to install the dependencies too). Once you've installed and loaded the package, you have access to the `readMat()` function. As any Matlab user will know, the `.mat` files that Matlab produces are workspace files, very much like the `.Rdata` files that R produces. So you can't import a `.mat` file as a data frame. However, you can import it as a list. So, when we do this:

```
library( R.matlab )                  # load the package
data <- readMat( "matlabfile.mat" )  # read the data file to a list
```

The `data` object that gets created will be a list, containing one variable for every variable stored in the Matlab file. It's fairly straightforward, though there are some subtleties that I'm ignoring. In particular, note that if you don't have the `Rcompression` package, you can't open Matlab files above the version 6 format. So, if like me you've got a recent version of Matlab, and don't have the `Rcompression` package, you'll need to save your files using the `-v6` flag otherwise R can't open them.

Oh, and Octave users? The `foreign` package contains a `read.octave()` command. Just this once, the world makes life easier for you folks than it does for all those cashed-up swanky Matlab bastards.

17.9.5 Saving other kinds of data

Given that I talked extensively about how to load data from non-R files, it might be worth briefly mentioning that R is also pretty good at writing data into other file formats besides it's own native ones. I won't discuss them in this book, but the `write.csv()` function can write CSV files, and the `write.foreign()` function (in the `foreign` package) can write SPSS, Stata and SAS files. There are also a lot of low level commands that you can use to write very specific information to a file, so if you really, really needed to you could create your own `write.obscurefiletype()` function, but that's also a long way beyond the scope of this book. For now, all that I want you to recognise is that this capability is there if you need it.

17.9.6 done yet?

Of course not. If I've learned nothing else about R it's that you're *never bloody done*. This listing doesn't even come close to exhausting the possibilities. Databases are supported by the `RODBC`, `DBI`, and `RMySQL` packages among others. You can open webpages using the `RCurl` package. Reading and writing JSON objects is supported through the `rjson` package. And so on. In a sense, the right question is not so much "can R do this?" so much as "whereabouts in the wilds of CRAN *is* the damn package that does it?"
