Hands-On Data Science with R
From Data to R

Graham.Williams@togaware.com

13th June 2015

Visit http://HandsOnDataScience.com/ for more Chapters.

Data is available in an enormous variety of formats and stored on our own data stores, from databases, and openly from the Internet through sites like http://data.gov in the US, http://data.gov.uk/ in the UK, and http://data.gov.au/ in Australia. A major task we face as Data Scientists is in transferring that data into R. Being open source and with a focus on the freedom of sharing between multiple tools R provides extensive capabilities for reading data.

In this chapter we explore some of the options for reading data into R. We start with a simple and widely used format by reading a dataset from a comma separated values (CSV) file. We will continue with various file formats, move on to accessing data from databases, and discuss the growing need to read data from the Internet.

The required packages for this chapter include:

```
library(xlsx)     # Read Excel spreadsheets.
library(RCurl)
library(foreign)
library(openxlsx)
library(readxl)   # The definitive Excel reader.
library(readr)    # Efficient reading of data.
```

As we work through this chapter, new R commands will be introduced. Be sure to review the command's documentation and understand what the command does. You can ask for help using the ? command as in:

```
?read.csv
```

We can obtain documentation on a particular package using the *help=* option of `library()`:

```
library(help=rattle)
```

This chapter is intended to be hands on. To learn effectively, you are encouraged to have R running (e.g., RStudio) and to run all the commands as they appear here. Check that you get the same output, and you understand the output. Try some variations. Explore.

1 To Do

Include Hadley's new haven package as useful for reading SAS, SPSS, and Stata files. http://blog.rstudio.org/2015/03/04/haven-0-1-0/.

Include Hadleys' new read_csv() from readr as well as read_excel() from readxl to read Excel spreadsheets.

Might also include read.nexus().

Include SQLite and perhaps make mention of sqldf to manipulate data using SQL although maybe that belongs in the BasicR chapter.

The foreign package may still be useful to access various format datasets.

2 Common Data Storage Formats

Data is available in a variety of forms and from a variety of sources.

.csv	Comma Separated Values	read.csv()

.

We generally store our data in R as a *data frame*. This is then the starting point for our data wrangling where we tidy our data ready for our analyses.

3 Reading from CSV

One of the simplest ways to load data into R is through the use of `read.csv()` which will read the contents of a CSV file and load them into an R data frame.

To illustrate we will read from a file in the `data` sub-directory of the current working directory. (Download the data file as http://onepager.togaware.com/heart.csv.) Check the current working directory using `getwd()`.

```
getwd()
```

```
## [1] "/home/gjw/projects/onepager"
```

The file itself will be listed as one of the CSV files in this directory, using `dir()`.

```
dir(path="data", pattern="*.csv")
```

```
##  [1] "austdm_firecomp.csv" "cmt_150514.csv"      "dvdtrans.csv"
##  [4] "heart.csv"           "ozdata.csv"          "stroke.csv"
##  [7] "WDI_Country.csv"     "WDI_CS_Notes.csv"    "WDI_csv.zip"
## [10] "WDI_Data.csv"        "WDI_Description.csv" "WDI_Footnotes.csv"
## [13] "WDI_Series.csv"      "WDI_ST_Notes.csv"    "weatherAUS.csv"
```

We now load the data from the CSV file using `read.csv()`.

```
heart <- read.csv(file=file.path("data", "heart.csv"))
```

For `read.csv()` we do not need to include the string `file=` part of the argument and in the commands below the `x=` and `object=` are also optional and can be dropped, relying on the position within the argument list to identify the formal argument. Once loaded review the data with `dim()`, `head()`, `tail()` and `str()`.

```
dim(x=heart)
```

```
## [1] 286  10
```

```
head(x=heart)
```

```
##   age    sex  chest_pain rest_bps chol fbs rest_ecg max_hr ex_ang  disease
## 1  31   male      asympt      120  270   f   normal    153    yes positive
## 2  33 female      asympt      100  246   f   normal    150    yes positive
## 3  34   male  typ_angina      140  156   f   normal    180     no positive
## 4  35   male atyp_angina      110  257   f   normal    140     no positive
## 5  36   male atyp_angina      120  267   f   normal    160     no positive
....
```

```
tail(x=heart)
```

```
##     age    sex  chest_pain rest_bps chol fbs              rest_ecg max_hr
## 281  45   male atyp_angina      140  224   t                normal    122
## 282  47   male      asympt      140  276   t                normal    125
## 283  48 female atyp_angina      120  251   t st_t_wave_abnormality    148
## 284  54 female atyp_angina      120  230   t                normal    140
## 285  55   male atyp_angina      120  256   t                normal    137
....
```

```
str(object=heart)
```

```
## 'data.frame': 286 obs. of  10 variables:
##  $ age       : int  31 33 34 35 36 37 38 38 38 41 ...
##  $ sex       : Factor w/ 2 levels "female","male": 2 1 2 2 2 2 2 2 2 2 ...
##  $ chest_pain: Factor w/ 4 levels "asympt","atyp_angina",..: 1 1 4 2 2 1 1...
##  $ rest_bps  : int  120 100 140 110 120 140 110 120 92 110 ...
##  $ chol      : int  270 246 156 257 267 207 196 282 117 289 ...
....
```

3.1 Always Review the Data

We might have noticed some other CSV files in the data folder listed earlier. We now load another of those files (download from http://onepager.togaware.com/stroke.csv).

```
stroke <- read.csv(file.path("data", "stroke.csv"))
```

Notice that we have dropped the `file=` part of the argument and rely on the fact that `read.csv()` expects the file to be the first argument.

```
str(read.csv)
```

```
## function (file, header=TRUE, sep=",", quote="\"", dec=".",
##     fill=TRUE, comment.char="", ...)
```

Once loaded, we should always review the data. This is a very good habit to get into. As we have seen previously, `dim()`, `head()`, `tail()` and `str()` come in handy, as does `summary()`.

```
dim(stroke)
```

```
## [1] 829   1
```

```
head(stroke)
```

```
##   SEX.DIED.DSTR.AGE.DGN.COMA.DIAB.MINF.HAN
## 1     1;7.01.1991;2.01.1991;76;INF;0;0;1;0
## 2             1;.;3.01.1991;58;INF;0;0;0;0
## 3     1;2.06.1991;8.01.1991;74;INF;0;0;1;1
## 4   0;13.01.1991;11.01.1991;77;ICH;0;1;0;1
## 5   0;23.01.1996;13.01.1991;76;INF;0;1;0;1
....
```

```
tail(stroke)
```

```
##     SEX.DIED.DSTR.AGE.DGN.COMA.DIAB.MINF.HAN
## 824            0;.;23.12.1993;62;INF;0;0;0;1
## 825            0;.;26.12.1993;55;INF;0;1;1;1
## 826   0;20.06.1994;29.12.1993;93;INF;0;0;0;0
## 827   0;27.01.1994;31.12.1993;81;INF;1;0;0;0
## 828            0;.;31.12.1993;68;INF;0;0;0;1
....
```

```
str(stroke)
```

```
## 'data.frame': 829 obs. of  1 variable:
##  $ SEX.DIED.DSTR.AGE.DGN.COMA.DIAB.MINF.HAN: Factor w/ 829 levels "0;10.03...
```

```
summary(stroke)
```

```
##                    SEX.DIED.DSTR.AGE.DGN.COMA.DIAB.MINF.HAN
##  0;10.03.1992;1.03.1992;88;ID;1;0;0;1  :  1
##  0;10.03.1992;2.03.1992;87;INF;0;0;0;0 :  1
##  0;10.03.1993;19.02.1991;75;INF;0;0;0;1:  1
##  0;10.03.1993;8.02.1993;74;INF;0;1;0;1 :  1
##  0;.;10.04.1992;65;ID;0;0;0;0          :  1
....
```

Reviewing this data carefully we can see that it is not what we might be expecting. The data appears to have been read in as a single column, with a rather long column name beginning with "`SEX.D`" and finishing with "`F.HAN`". The individual columns have not been extracted.

3.2 Choosing the Separator

If we look at the contents of `stroke.csv` (and based on our observations of the data we have loaded above) we see that there are no commas separating the columns in the file. Instead the columns are separated using a semicolon.

Strictly speaking `stroke.csv` is not the usual kind of CSV (comma separated value) file. Nonetheless, this is readily catered for in R through the argument `sep=` of `read.csv()`. This argument allows us to specify the correct separator.

```
stroke <- read.csv(file.path("data", "stroke.csv"), sep=";")
dim(stroke)
```

```
## [1] 829   9
```

```
head(stroke)
```

```
##   SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 1   1  7.01.1991  2.01.1991  76 INF    0    0    1   0
## 2   1          .  3.01.1991  58 INF    0    0    0   0
## 3   1  2.06.1991  8.01.1991  74 INF    0    0    1   1
## 4   0 13.01.1991 11.01.1991  77 ICH    0    1    0   1
## 5   0 23.01.1996 13.01.1991  76 INF    0    1    0   1
....
```

```
str(stroke)
```

```
## 'data.frame': 829 obs. of  9 variables:
##  $ SEX : int  1 1 1 0 0 1 0 1 0 0 ...
##  $ DIED: Factor w/ 415 levels ".","10.02.1993",..: 374 1 179 61 214 61 46 ...
##  $ DSTR: Factor w/ 575 levels "10.01.1993","10.02.1991",..: 246 433 542 38...
##  $ AGE : int  76 58 74 77 76 48 81 53 73 69 ...
##  $ DGN : Factor w/ 4 levels "ICH","ID","INF",..: 3 3 3 1 3 1 3 3 2 3 ...
....
```

Note that `read.csv()` requires us to name the `sep=` argument in this instance. According to the manual page for `read.csv()` (and the output of `str()` below) the `sep=` argument is the third argument.

```
str(read.csv)
```

```
## function (file, header=TRUE, sep=",", quote="\"", dec=".",
##     fill=TRUE, comment.char="", ...)
```

We are using the argument in the second position in this call to `read.csv()`. The second argument position is reserved for `header=`.

The following are equivalent and it is useful to take a moment to understand what is happening here:

```
stroke <- read.csv(file.path("data", "stroke.csv"), sep=";")
stroke <- read.csv(file.path("data", "stroke.csv"), header=TRUE, sep=";")
stroke <- read.csv(file.path("data", "stroke.csv"), TRUE, ";")
```

The following results in an error, as we see:

```
stroke <- read.csv(file.path("data", "stroke.csv"), ";")
## Error in !header:  invalid argument type
```

3.3 Semicolon Separated Values

The use of semicolons to separate values within a row of a file is a special case of a CSV file. This is often used in countries where the comma is used as the decimal marker. R provides a special version of `read.csv()` called `read.csv2()` which defaults to using the semicolon as the field separator (`sep=";"`) and the comma as the decimal marker (`dec=","`).

```
stroke <- read.csv2(file.path("data", "stroke.csv"))
```

As always, check the data we have just read to ensure it is as we expected.

```
dim(stroke)
```

```
## [1] 829   9
```

```
head(stroke)
```

```
##   SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 1   1  7.01.1991  2.01.1991  76 INF    0    0    1   0
## 2   1          .  3.01.1991  58 INF    0    0    0   0
## 3   1  2.06.1991  8.01.1991  74 INF    0    0    1   1
## 4   0 13.01.1991 11.01.1991  77 ICH    0    1    0   1
## 5   0 23.01.1996 13.01.1991  76 INF    0    1    0   1
....
```

```
tail(stroke)
```

```
##     SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 824   0          . 23.12.1993  62 INF    0    0    0   1
## 825   0          . 26.12.1993  55 INF    0    1    1   1
## 826   0 20.06.1994 29.12.1993  93 INF    0    0    0   0
## 827   0 27.01.1994 31.12.1993  81 INF    1    0    0   0
## 828   0          . 31.12.1993  68 INF    0    0    0   1
....
```

```
str(stroke)
```

```
## 'data.frame': 829 obs. of  9 variables:
##  $ SEX : int  1 1 1 0 0 1 0 1 0 0 ...
##  $ DIED: Factor w/ 415 levels ".","10.02.1993",..: 374 1 179 61 214 61 46 ...
##  $ DSTR: Factor w/ 575 levels "10.01.1993","10.02.1991",..: 246 433 542 38...
##  $ AGE : int  76 58 74 77 76 48 81 53 73 69 ...
##  $ DGN : Factor w/ 4 levels "ICH","ID","INF",..: 3 3 3 1 3 1 3 3 2 3 ...
....
```

We will often be "pleasantly surprised" like this when using R. If we have a need to do something a little different, the chances will be that R supports it. Have a look at the documentation for `read.csv()` to glean a little of the flexibility of this particular function.

```
?read.csv
```

Notice `read.csv()` is actually just a call to the underlying `read.table()`, with some default options changed to suit CSV standard files.

```
read.csv
```

```
## function (file, header=TRUE, sep=",", quote="\"", dec=".",
##     fill=TRUE, comment.char="", ...)
## read.table(file=file, header=header, sep=sep, quote=quote,
##     dec=dec, fill=fill, comment.char=comment.char, ...)
## <bytecode: 0x2478ba8>
## <environment: namespace:utils>
....
```

3.4 Strings as Strings

By default `read.csv` will treat columns consisting of strings as a factor with the levels corresponding to the different strings found in the data file. For data such as peoples names and addresses, we would want to retain these as strings rather than factors.

```
ds <- read.csv("stroke.csv", stringsAsFactors=FALSE)
```

4 White Space

We remove white space around values using `strip.white=TRUE`

5 Viewing the Data

```
View(stroke)
```

```
library(RGtk2Extras)
dfedit(stroke)
```

```
library(Deducer)
date.viewer()
```

6 Identifying Missing Values

A careful review of the stroke data we loaded above will identify that there are periods (i.e., ".") included in the data for `DIED`. Have a look at the tail of the dataset.

```
tail(stroke)
```

```
##     SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 824   0          . 23.12.1993  62 INF    0    0    0   1
## 825   0          . 26.12.1993  55 INF    0    1    1   1
## 826   0 20.06.1994 29.12.1993  93 INF    0    0    0   0
## 827   0 27.01.1994 31.12.1993  81 INF    1    0    0   0
## 828   0          . 31.12.1993  68 INF    0    0    0   1
....
```

Our guess would be that these are used to indicate missing values (a common practise).

We can tell `read.csv()` about this using the `na.strings=` argument.

```
stroke <- read.csv2(file.path("data", "stroke.csv"), na.strings=".")
```

We review the resulting dataset.

```
dim(stroke)
```

```
## [1] 829   9
```

```
head(stroke)
```

```
##   SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 1   1  7.01.1991  2.01.1991  76 INF    0    0    1   0
## 2   1       <NA>  3.01.1991  58 INF    0    0    0   0
## 3   1  2.06.1991  8.01.1991  74 INF    0    0    1   1
## 4   0 13.01.1991 11.01.1991  77 ICH    0    1    0   1
## 5   0 23.01.1996 13.01.1991  76 INF    0    1    0   1
....
```

```
tail(stroke)
```

```
##     SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
## 824   0       <NA> 23.12.1993  62 INF    0    0    0   1
## 825   0       <NA> 26.12.1993  55 INF    0    1    1   1
## 826   0 20.06.1994 29.12.1993  93 INF    0    0    0   0
## 827   0 27.01.1994 31.12.1993  81 INF    1    0    0   0
## 828   0       <NA> 31.12.1993  68 INF    0    0    0   1
....
```

```
str(stroke)
```

```
## 'data.frame': 829 obs. of  9 variables:
##  $ SEX : int  1 1 1 0 0 1 0 1 0 0 ...
##  $ DIED: Factor w/ 414 levels "10.02.1993","10.03.1992",..: 373 NA 178 60 ...
##  $ DSTR: Factor w/ 575 levels "10.01.1993","10.02.1991",..: 246 433 542 38...
##  $ AGE : int  76 58 74 77 76 48 81 53 73 69 ...
##  $ DGN : Factor w/ 4 levels "ICH","ID","INF",..: 3 3 3 1 3 1 3 3 2 3 ...
```

```
....
```

That looks better.

The value of the argument **na.strings=** can be a character vector, listing all the possibilities that we might come across to represent missing values in our file.

```
stroke <- read.csv2(file.path("data", "stroke.csv"), na.strings=c(".", "?", " "))
```

7 Specifying Data Types

```
sapply(stroke, class)
```

```
##       SEX      DIED      DSTR       AGE       DGN      COMA      DIAB
## "integer"  "factor"  "factor" "integer"  "factor" "integer" "integer"
##      MINF       HAN
## "integer" "integer"
```

```
classes <- c("factor", "character", "character", "integer", "factor",
             "factor", "factor", "factor", "factor")
stroke <- read.csv2(file.path("data", "stroke.csv"), na.strings=".",
                    colClasses=classes)
sapply(stroke, class)
```

```
##         SEX        DIED        DSTR         AGE         DGN        COMA
##    "factor" "character" "character"   "integer"    "factor"    "factor"
##        DIAB        MINF         HAN
##    "factor"    "factor"    "factor"
```

8 Reading Multiple Files

Here's some compact code to read all of the CSV files in a directory into one data frame. We assume they all have the same format:

```
fnames <- dir(pattern="\\.csv$")
dsl    <- lapply(fnames, read.csv)
ds     <- do.call("rbind", dsl)
```

We use to find all the csv files in a directory, load into a list and then collapse into a single data frame.

9 Reading From the Clipboard

This is an extremely useful tip particularly when following examples from the Internet. You might see something that looks like a well structured table with column headings and aligned rows. You want to get that quickly into R. Select the table and copy it into the clipboard (which happens automatically on Linux or using Ctrl-C or the Copy menu on Windows). Then in R:

```
ds <- read.table("clipboard")
```

10 Reading Microsoft Excel Spreadsheets

To ready Microsoft Excel spreadsheets we use `readxl::read_excel()` from readxl (Wickham, 2015). One downside is that it can not select regions to read the data from unlike read.xlsx.

Several packages are available, including xlsx (Dragulescu, 2014) and openxlsx (?). The former depends on Java and the rJava (Urbanek, 2013) package, whilst openxlsx does not, whch may be an advantage given some of the common Java issues. An advantage of xlsx is that it can read specific row and column indicies with `rowIndex` and `colIndex`.

11 Writing to CSV

```
write(weather, file=file.path("data", "myweather.csv"), row.names=FALSE)
```

12 Saving RData

```
save(stroke, file=file.path("data", "stroke.RData"))
```

Exercise: Show size in memory and size on disk.

Exercise: Compare with dput() and dget() which convert r objects into an ascii text representation that is generally human readable. dget() recreates the R object.

13 Data from Spreadsheets

Exercise: Illustrate how to read libre office std format, MS/Excel format

14 Loading tab/txt Files

Exercise: Illustrate loading a tab delimited txt file.

15 Loading Fixed Width Files

Exercies: Illustrate reading a fixed width data file.

```
read.fwf()
```

16 Data from Internet Documents

A URL can be supplied to the `read.table()` family of functions, including `read.csv()`.

```
addr <- file.path("http://www.ats.ucla.edu/stat/r/examples/alda/data",
                  "tolerance1_pp.txt")
tolerance <- read.csv(addr)
```

As always, review the data.

```
dim(tolerance)
```

```
## [1] 80  6
```

```
head(tolerance)
```

```
##   id age tolerance male exposure time
## 1  9  11      2.23    0     1.54    0
## 2  9  12      1.79    0     1.54    1
## 3  9  13      1.90    0     1.54    2
## 4  9  14      2.12    0     1.54    3
## 5  9  15      2.66    0     1.54    4
....
```

```
tail(tolerance)
```

```
##      id age tolerance male exposure time
## 75 1552  15      1.55    0     1.04    4
## 76 1653  11      1.11    0     1.25    0
## 77 1653  12      1.11    0     1.25    1
## 78 1653  13      1.34    0     1.25    2
## 79 1653  14      1.55    0     1.25    3
....
```

```
str(tolerance)
```

```
## 'data.frame': 80 obs. of  6 variables:
##  $ id       : int  9 9 9 9 9 45 45 45 45 45 ...
##  $ age      : int  11 12 13 14 15 11 12 13 14 15 ...
##  $ tolerance: num  2.23 1.79 1.9 2.12 2.66 1.12 1.45 1.45 1.45 1.99 ...
##  $ male     : int  0 0 0 0 0 1 1 1 1 1 ...
##  $ exposure : num  1.54 1.54 1.54 1.54 1.54 1.16 1.16 1.16 1.16 1.16 ...
....
```

```
summary(tolerance)
```

```
##        id              age       tolerance          male
##  Min.   :   9.0   Min.   :11   Min.   :1.000   Min.   :0.0000
##  1st Qu.: 410.0   1st Qu.:12   1st Qu.:1.220   1st Qu.:0.0000
##  Median : 673.5   Median :13   Median :1.500   Median :0.0000
##  Mean   : 762.8   Mean   :13   Mean   :1.619   Mean   :0.4375
##  3rd Qu.:1009.8   3rd Qu.:14   3rd Qu.:1.990   3rd Qu.:1.0000
....
```

The data identifies a population of adolescents in a youth study. At particular ages their tolerance

to “deviant” behavior is recorded.

Having downloaded the data we may like to save it locally to file. Saving it as a binary R data file will use less disk space than the original CSV file, and retains the meta-data.

```
save(tolerance, file=file.path("data", "tolerance.RData"))
```

17 Data from Google Drive

We can load data into R from a spreadsheet stored on Google Drive by submitting a GET form that retrieves the data in a raw form. We can then parse the data into an R data frame. To access Internet based data we use RCurl (Temple Lang, 2015).

The first step is to identify the unique key that is connected to the file on your Google Drive. This can be obtained from Google Drive.

```
key <- "0Aonsf4v9iDjGdHRaWWRFbXdQN1ZvbGx0LWVCeVd0T1E"
```

Next we get the actual raw data, saving it into a variable.

```
tt <- getForm("https://spreadsheets.google.com/spreadsheet/pub",
              hl="en_US", key=key, output="csv")
tt
```

```
## [1] "<HTML>\n<HEAD>\n<TITLE>Moved Temporarily</TITLE>\n</HEAD>\n<BODY BGCO...
## attr(,"Content-Type")
##                 charset
## "text/html"     "UTF-8"
```

Now we can read the data.

```
ds <- read.csv(textConnection(tt))
dim(ds)
```

```
## [1] 8 1
```

```
head(ds)
```

```
##                                                                        ...
## 1                                                                      ...
## 2                                                                      ...
## 3                                                                      ...
## 4                                                                      ...
## 5                                                                      ...
....
```

18 Data from Google Drive—read.csv() and https

Google changed over to serving all docs up using encryption (perhaps in 2012 or so) and thus automatically rewrites http: as https:. Unfortunately `read.csv()` can not handle encrypted connections (i.e., https:). So the method below will now result in an error.

First we might set up the URL to access:

```
library(RCurl)
key   <- "0AmbQbL4Lrd61dER5Qnl3bHo4MkVNRlZ1OVdicnZnTHc"
query <- curlEscape("select *")
addr  <- paste0("http://spreadsheets.google.com/tq?",
                "key=", key,
                "&tq=", query,
                "&tqx=out:csv")
addr
```

```
## [1] "http://spreadsheets.google.com/tq?key=0AmbQbL4Lrd61dER5Qnl3bHo4MkVNRl...
```

Here we constructed a URL with the required information. To access the document the URL needs to include the spreadsheet key. The query we pass along uses SQL to select all columns and rows from the spreadsheet, and is constructed for sending through the URL using `curlEscape()`. Finally we generate the appropriate string that is the URL to send out to the Internet.

Pasting the address into a browser will work to download the file `data.csv`. In R though we will see an error:

```
ds <- read.csv(addr)
```

```
## Error in file(file, "rt"):  cannot open the connection
```

19 Command Summary

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

Complete this list.

`readr::read_csv()` *Function from readr.* Load data from a CSV file. This is fast enough, and so faster than `utils::read.csv()` but not as fast as the options in the data.table (**?**) package.

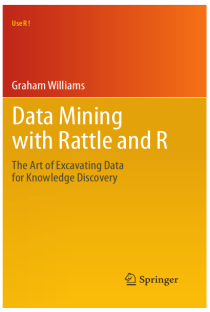
20 Exercises

Exercise .1 Read data from CSV file from the Internet

Exercise .2 Read data from a SQLite Database

21 Further Reading

The Rattle Book, published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from Amazon. Other documentation on a broader selection of R topics of relevance to the data scientist is freely available from `http://datamining.togaware.com`, including the Datamining Desktop Survival Guide.



This chapter is one of many chapters available from `http://HandsOnDataScience.com`. In particular follow the links on the website with a * which indicates the generally more developed chapters.

Other resources include:

- Any further reading?

22 References

Dragulescu AA (2014). *xlsx: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files*. R package version 0.5.7, URL http://CRAN.R-project.org/package=xlsx.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Temple Lang D (2015). *RCurl: General network (HTTP/FTP/...) client interface for R*. R package version 1.95-4.6, URL http://CRAN.R-project.org/package=RCurl.

Urbanek S (2013). *rJava: Low-level R to Java interface*. R package version 0.9-6, URL http://CRAN.R-project.org/package=rJava.

Wickham H (2015). *readxl: Read Excel Files*. R package version 0.1.0, URL http://CRAN.R-project.org/package=readxl.

Williams GJ (2009). "Rattle: A Data Mining GUI for R." *The R Journal*, **1**(2), 45–55. URL http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Williams.pdf.

Williams GJ (2011). *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*. Use R! Springer, New York.

This document, sourced from ReadO.Rnw revision 799, was processed by KnitR version 1.9 of 2015-01-20 and took 3.4 seconds to process. It was generated by gjw on nyx running Ubuntu 14.04.2 LTS with Intel(R) Xeon(R) CPU W3520 @ 2.67GHz having 8 cores and 12.3GB of RAM. It completed the processing 2015-06-13 11:51:19.