Basics of R for Data Analysis

Ehssan Ghashim[1], Patrick Boily[1,2,3]

Abstract
R has become one of the world's leading languages for statistical and data analysis. In this report, we provide a short description of its core functionality.

Keywords
R, R Studio, data manipulation, data wrangling, simple graphics, common statistical procedures

[1]Centre for Quantitative Analysis and Decision Support, Carleton University, Ottawa
[2]Department of Mathematics and Statistics, University of Ottawa, Ottawa
[3]Idlewyld Analytics and Consulting Services, Wakefield, Canada
Email: patrick.boily@carleton.ca

Contents

1. Introduction

R is a powerful language that is widely-used for data analysis and statistical computing. It was developed in the early 90s by Ross Ihaka and Robert Gentleman. Since then, continuous efforts have been made to improve R's user interface. The journey of R language from a rudimentary text editor to interactive R Studio and more recently Jupyter Notebooks has engaged many data science communities across the world.

This was made possible in part because of generous contributions by R users. The inclusion of sophisticated packages (such as `dplyr`, `tidyr`, `readr`, `data.table`, `SparkR`, `ggplot2`, etc.) has made R both more powerful and more useful, allowing for smart data manipulation, visualization, and computation.

Why Use R?

Here are some benefits that potential users might note:

- the style of coding is intuitive;
- R is open source and free;
- more than 7800 packages, customized for various computation tasks, are available (as of October 2017);
- the R community is overwhelmingly welcoming and useful to new users and experienced users alike (you can browse and ask questions at StackOverflow, and consult worked-out examples on R-bloggers, for instance);
- high performance computing experience is possible (with the appropriate packages), and
- is is one of the highly sought skills by analytics and data science companies.

Installing R / R Studio

You can download and install the vanilla version of R, but RStudio provides a much better coding experience, in our opinion. It is available to Windows user from Vista onward.

The following steps will allow you to install R Studio:

1. Visit **rstudio.com/products/rstudio/download/**
2. In the 'Installers for Supported Platforms' section, select the R Studio installer based on your operating system. The download should begin as soon as you click.
3. Follow the instructions until the download is complete.
4. Start R Studio by clicking on the desktop icon or use 'search windows' to access the program. Once opened, the R Studio GUI will display 4 windows (see Figure 1).
    - **Console:** this area shows the output of code that has been run (either from the command line in the console or from the script window).
    - **Script:** as the name suggests, this is the window one would typically use to write code. Lines can be run by first selecting them (right-clicking) and pressing 'Ctrl + Enter' simultaneously. Alternatively, you can click on the little 'Run' button located at the top right corner of the script window.
    - **Environment:** this space displays the set of external elements that have been added. This

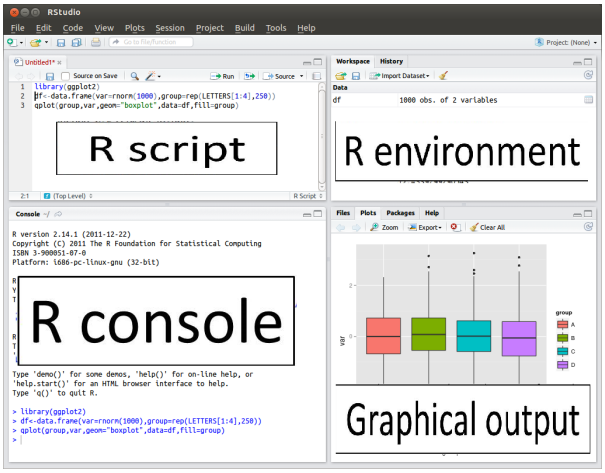

Figure 1. The R Studio GUI.

includes data set, variables, vectors, functions etc. This area allows the user to verify that data has been loaded properly.

- **Graphical Output:** this space display the graphs created during exploratory data analysis, or embedded help on package functions from R's official documentation.

2. First Steps

Installing R Packages

To install a package, simply type:

```
install.packages(``package name'')
```

You can type this code directly in the console, followed by a carriage return, or enter it in the R Script window and click Run in the menu at the top.

The base distribution already comes with some high-priority add-on packages, namely

KernSmooth	MASS	boot	class
foreign	lattice	mgcv	nlme
rpart	spatial	survival	base
grDevices	graphics	grid	methods
stats	stats4	tcltk	tools
cluster	nnet	datasets	splines

The packages listed here implement standard statistical functionality, for example linear models, classical tests, a huge collection of high-level plotting functions, and tools for survival analysis.

Computations in R

Let's begin with basics. To get familiar with the R coding environment, start with some basic calculations. R Console can be used as an interactive calculator too.

Type the first line of each group in your console, followed by a carriage return to confirm that R works as one would expect of a calculator:

```
> 2 + 3
   5

> (3*8)/(2*3)
   4

> log(12)
   1.07

> sqrt(121)
   11
```

Similarly, you can experiment with various combinations of calculations and get the results.

Should you want to modify or repeat a prior calculation, press the 'Up Arrow' when in R Console to cycle through previously executed commands. Pressing 'Enter' re-runs the selected computation.

On the other hand, you can avoid scrolling through a wall of computations by creating a variable.

In R, this is done using the variable assignment symbols <− or =. Once a variable exists in memory, the output does not get printed directly unless you call the variable or surround the variable assignment with a pair of parentheses.

```
> x <- 8 + 7
> x
   15

> (x <- 8 + 7)
   15
```

Variables can be named using any combination of alphanumeric symbols, but the name has to start with a letter (a-z, A-Z) and cannot contain spaces and punctuation marks (except for periods and dashes).

R Essentials

Everything you see or create in R is an **object**: vectors, matrices, data frames, even variables are objects. R allows 5 basic classes of objects:

- Character
- Numeric (real numbers)
- Integer (whole numbers)
- Complex
- Logical (True / False)

Each of these classes has **attributes**. Think of attributes as their 'identifier', a name or number which aptly identifies them. An object can have the following attributes:

- names, dimension names
- dimensions

- class
- length
- etc.

An object's various attributes can be accessed using the `attributes()` function. We will have more to say on this topic.

The most basic R object is the **vector**. An empty vector can be created using `vector()`. A vector contains various objects all of the same class.[1]

Vectors are often created using the concatenate operator `c()` (which makes it a singularly bad idea to use `c` as a variable name).

```
> a <- c(1.8, 4.5) # numeric
> b <- c(1 + 2i, 3 - 6*i) # complex
> d <- c(23, 44) # integer
> e <- vector("logical", length = 5) #
    logical
> f <- c("abc","def") # character
```

Comments can be introduced in R code *via* the # symbol: all characters following a pound (or sharp) symbol is ignored by R until the next line of code.

R Data Types

Vector A vector contains objects of the same class. You may have the need to mix objects of different classes in a list – this can be done by **coercion**. This has the effect of 'converting' objects of different types to the same class. For instance:

```
> qt <- c("Time", 24,"October", TRUE,
    3.33) # coercion to character
> ab <- c(TRUE, 24) # coercion to numeric
> cd <- c(2.5,"May") # coercion to
    character
```

To check the class of any object, use the `class('name')` function.

```
> class(qt)
  "character"
```

To convert the class of a vector, you can use the `as.` command.

```
> bar <- 0:5 # create a vector of 6
    inteers
> class(bar) # find bar's class
  "integer"
> as.numeric(bar) # convert to numeric
> class(bar)
  "numeric"
> as.character(bar) # convert to
    character
> class(bar)
  "character"
```

Similarly, you can change the class of any vector. But, you should pay attention here – you can convert a numeric vector a character one, but going the other way will introduce NAs.

List A list is a special type of vector which contain elements of different data types.

```
> my_list <- list(22,"ab", TRUE, 1 + 2*i)
> my_list
   [[1]]
      [1] 22
   [[2]]
      [1]``ab''
   [[3]]
      [1] TRUE
   [[4]]
      [1] 1+2*i
```

As you can see, the output of a list differs from that of a vector, since all the objects are of different types. The double bracket `[[1]]` shows the index of the first element and so on. The elements of lists can be extracted by using the appropriate index:

```
> my_list[[3]]
   [1] TRUE
```

The single single bracket `[ ]` also has a role: it returns the list element with its index number, instead of the result above.

```
> my_list[3]
   [[1]]
      [1] TRUE
```

Matrices A vector for which rows and columns are explicitly identified is a **matrix**, a 2-dimensional data structure. All the entries of a matrix have to be of the same class.

The following code produces a 3 by 2 matrix.

```
> my_matrix <- matrix(1:6, nrow=3,
    ncol=2)
> my_matrix
       [,1] [,2]
   [1,] 1  4
   [2,] 2  5
   [3,] 3  6

> dim(my_matrix)
   [1] 3 2
```

[1] That can cause unforeseen difficulties as it is not always easy to distinguish between real numbers and integer visually. Furthermore, the digits of a number can be represented as character strings in some cases.

```
> attributes(my_matrix)
   [1] 3 2
```

The dimensions of a matrix can be obtained using either the `dim()` or `attributes()` commands. To extract a particular element from a matrix, simply use the appropriate indices. What might you expect to see from the following commands?

```
> my_matrix[,2] #extracts second column
> my_matrix[,1] #extracts first column
> my_matrix[2,] #extracts second row
> my_matrix[1,] #extracts first row
```

As an aside, it is straightforward to create a matrix from a vector, by assigning the dimensions using `dim()`.

```
> age <- c(23, 44, 15, 12, 31, 16) #
    read in a vector of ages
> age
   [1] 23 44 15 12 31 16

> dim(age) <- c(2,3) # reshape the
    vector as a 3 x 2 matrix
> age
       [,1] [,2] [,3]
   [1,] 23 15 31
   [2,] 44 12 16

> class(age)
   [1] "matrix"
```

Matrices can also be created by joining two vectors (with matching dimensions) using `cbind()` or `rbind()`

```
> x <- c(1, 2, 3, 4, 5, 6)
> y <- c(20, 30, 40, 50, 60)
> cbind(x, y)
       x y
   [1,] 1 20
   [2,] 2 30
   [3,] 3 40
   [4,] 4 50
   [5,] 5 60
   [6,] 6 70

> rbind(x,y)
     [,1] [,2] [,3] [,4] [,5] [,6]
   x 1   2   3   4   5   6
   y 20 30 40 50 60 70

> class(cbind(x, y))
   [1] "matrix"

> class(rbind(x, y))
   [1] "matrix"
```

Data Frame: This is the most commonly used data type. It is used to store tabular data, but it is different from a matrix. In a matrix, every element must have the same class. A data frame can accommodate lists of vectors of different classes. In other words, each column of a data frame acts like a list. Every time data is read into R, it is first stored as a data frame.

```
> df <- data.frame(name =
    c("ash","jane","paul","mark"), score
    = c(67,56,87,91)) # create a data
    frame with columns "name" and "score"
> df
     name score
   1 ash 67
   2 jane 56
   3 paul 87
   4 mark 91

> dim(df)
   [1] 4 2

> str(df)
   'data.frame': 4 obs. of 2 variables:
   $ name : Factor w/ 4 levels "ash",
       "jane", "mark", ..: 1 2 4 3
   $ score: num 67 56 87 91

> nrow(df)
[1] 4

> ncol(df)
[1] 2
```

In the code above, `df` is the name of data frame, `dim()` returns the dimension of the data frame as 4 rows and 2 columns, `str()` returns the structure of the data frame (i.e. the list of variables stored in the data frame), and `nrow()` and `ncol()` return the number of rows and number of columns in the data frame, respectively.

Exercises

1. Calculate the following quantities:
   - The sum of 10.01, 25.003, and 37.5
   - The square root of 121
   - Calculate the 10-based logarithm of 90, and multiply the result with the cosine of π. Hint: see ?log and ?pi.
2. Type the following code, which assigns numbers to objects x and y.

```
x<-25
y<-55
```

   - Calculate the product of x and y
   - Store the result in a new object called z
   - Inspect your workspace by typing ls(), and by clicking the Environment tab in Rstudio, and find the three objects you created

- Make a vector of the objects x, y, and z. Use this command,

```
myvec<-c(x,y,z)
```

3. You have measured five cylinders. Their lengths are: 3.5, 5.6, 6.9, 7.8, 2.8, and the diameters are: 0.3, 0.9, 0.6, 0.1, 0.2. Read these data points into two vectors (give the vectors appropriate names). Calculate the volume of each cylinder($V = \text{length} \times \pi \times (\text{diameter}/2)^2$).
4. Input the following data, on space shuttle launch damage prior to the Challenger explosion. The set covers 6 launches out of 24 that were included in the pre-launch charts used to decide whether to proceed with the launch or not

Temp	Erosion	Blowby	Total
53	3	2	5
57	1	0	1
63	1	0	1
70	1	0	1
70	1	0	1
75	0	2	1

Enter these data into a data frame, with (for example) column names **temperature, erosion, blowby** and **total**.

3. Data Manipulation

Reading data into a statistical system for analysis, and exporting the results to some other system for report writing, can be frustrating tasks that take far more time than the statistical analysis itself, even though most readers will find the latter far more appealing.

This section describes the import and export facilities available either in R itself or via packages available from CRAN.

Reading Data

R comes with a few data reading functions:

- `read.table`, `read.csv` for tabular data
- `readLines` for lines of a text file
- `source`, `dget` to read R code files (inverse of `dump` and `dput`, respectively)
- `load` to read-in saved workspaces
- `unserialize` to read single R objects in binary form

There are, of course, numerous R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

`read.table()` The `read.table()` function is one of the most commonly used functions for reading data. The help file is worth reading (run `?read.table` in the console) in its entirety if only because the function gets so much use.

Here are its main arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, string indicating how the columns are separated
- `colClasses`, character vector indicating the class of each column in the dataset
- `nrows`, number of rows in the dataset (by default `read.table()` will read the entire file)
- `comment.char`, character string indicating the comment character (this defaults to "#"; if there are no commented lines in your file, it's worth setting this to be the empty string "")
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors? (this defaults to *TRUE* because back in the old days, strings represented levels of a categorical variable; now that text mining is an every day occurrence, that's not always the case)

For small to moderately sized datasets, you can usually call `read.table()` without specifying any other arguments

```
> data <- read.table("foo.txt")
```

In this case, R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table

Telling R all these things directly makes R run faster and more efficiently. The `read.csv()` function is identical to read.table except that some of the defaults are set differently (like the `sep` argument).

With much larger datasets, there are a few things that can be done to prevent R from choking on the data:

- read the help page for `read.table`, which contains many hints
- make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you should probably stop right here
- set `comment.char = ""` if there are no commented lines in your file

- use the `colClasses` argument; specifying this option can make `read.table` run MUCH faster, often twice as fast (in order to use this option, you have to know the class of each column in your data frame; if all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`)
- A quick way to figure out the classes of each column is to use the following code:

```
> initial <-
    read.table("datatable.txt",
    nrows = 100)
> classes <- sapply(initial, class)
> tabAll <-
    read.table("datatable.txt",
    colClasses = classes)
```

- set `nrows` – this doesn't make R run faster but it helps with memory usage (a mild overestimate is okay, you can use the Unix tool `wc` to calculate the number of lines in the file).

In general, when using R with larger datasets, it's also useful to know a few things about your system:

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- What operating system are you using? Some operating systems can limit the amount of memory a single process can access

For example, suppose you have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly speaking, how much memory is required to store this data frame? Well, on most modern computers **double precision floating point numbers** are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

$$\begin{aligned} 1,500,000 \times 120 \times 8 \text{ bytes} &= 1,440,000,000 \text{ bytes} \\ &= 1,440,000,000/220 \text{ bytes/MB} \\ &= 1,373.29 \text{ MB} \\ &= 1.34 \text{ GB}. \end{aligned}$$

Reading in a large dataset for which one does not have enough RAM is an easy way to freeze your computer (or at the very least your R session). This is usually an unpleasant experience that requires one to kill the R process, in the best case scenario, or reboot the computer, in the worst case. It's always a good idea to do a rough memory requirements calculation before reading in a large dataset.

`txt`, `csv`, and Other Formats

- **Fixed format text files**

```
ds =
    read.table("dir_location\\file.txt",
    header=TRUE) # Windows only
ds =
    read.table("dir_location/file.txt",
    header=TRUE) # all OS
    (including # Windows)
```

Forward slash is supported as a directory delimiter on all operating systems; a double backslash is supported under Windows. If the first row of the file includes the name of the variables, these entries will be used to create appropriate names (reserved characters such as '$' or '[' are changed to '.') for each of the columns in the dataset. If the first row doesn't include the names, the header option can be left off (or set to FALSE), and the variables will be called V1, V2,..., Vn. A limit on the number of lines to be read can be specified through the `nrows` option The `read.table()` function can support reading from a URL as a filename or browse files interactively using **`read.table(file.choose())`**.

Sometimes data arrives in irregularly-shaped data files (there may be a variable number of fields per line, or some data in the line may describe the remainder of the line). In such cases, a useful generic approach is to read each line into a single character variable, then use character variable functions to extract the contents.

```
ds = readLines("file.txt")
ds = scan("file.txt")
```

The `readLines()` function returns a character vector with length equal to the number of lines read. A limit on the number of lines to be read can be specified through the `nrows` option. The `scan()` function returns a vector, with entries separated by white space by default. These functions read by default from standard input, but can also read from a file or URL.

- **Comma-separated value (CSV) files**

```
ds =
    read.csv("dir_location/file.csv")
```

- **Read sheets from an Excel file**

```
library(gdata)
ds =
    read.xls("http://www.amherst.edu/
~nhorton/r2/datasets/help.xlsx",
    sheet=1)
```

The sheet can be provided as a number or a name.

- **Reading datasets in other formats**

```
library(foreign)
ds = read.dbf("filename.dbf") # DBase
ds =
    read.epiinfo("filename.epiinfo")
    # Epi Info
ds = read.mtp("filename.mtp") #
    Minitab portable worksheet
ds = read.octave("filename.octave")
    # Octave
ds = read.ssd("filename.ssd") # SAS
    version 6
ds = read.xport("filename.xport") #
    SAS XPORT file
ds = read.spss("filename.sav") # SPSS
ds = read.dta("filename.dta") # Stata
ds = read.systat("filename.sys") #
    Systat
```

The `foreign` package can read Stata, Epi Info, Minitab, Octave, SPSS, and Systat files (with the caveat that SAS files may be platform dependent). The `read.ssd()` function will only work if SAS is installed locally

Manual Data Entry The `data.entry()` function invokes a spreadsheet that can be used to edit or otherwise change a vector or data frame.

In this example, an empty numeric vector of length 10 is created to be populated. The `data.entry()` function differs from the `edit()` function, which leaves the objects given as unchanged arguments, returning a new object with the desired edits.

```
x = numeric(10)
data.entry(x)
```

or

```
x1 = c(1, 1, 1.4, 123)
x2 = c(2, 3, 2, 4.5)
```

Writing Data

There are analogous functions for writing data to files

- `write.table`, to write tabular data to text files (i.e. CSV) or connections
- `writeLines`, to write character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `serialize`, for converting an R object into a binary format for outputting to a connection (or file).

Using Textual for Data Storage There are numerous ways to store data, including structured text files like CSV or tab-delimited, or more complex binary formats. However, there is an intermediate format that is textual, but not as simple as something like CSV. The format is native to R and is somewhat readable because of its textual nature.

One can create a more descriptive representation of an R object by using the `dput()` or `dump()` functions. The `dump()` and `dput()` functions are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable. Unlike writing out a table or CSV file, `dump()` and `dput()` preserve the metadata (sacrificing some readability), so that another user doesn't have to specify it all over again. For example, we can preserve the class of each column of a table or the levels of a factor variable.

`dput()` and `dump()` One way to pass data around is by deparsing the R object with `dput()` and reading it back in (parsing it) using `dget()`.

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
structure(list(a = 1, b = structure(1L,
    .Label = "a", class = "factor")),
    .Names\
= c("a","b"), row.names = c(NA, -1L),
    class = "data.frame")
```

Notice that the `dput()` output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names. The output of `dput()` can also be saved directly to a file.

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y
a b
1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using `source()`.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

We can `dump()` R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
```

Year	NSW	Vic.	Qld	SA	WA	Tas.	NT	ACT	Aust.
1917	1904	1409	683	440	306	193	5	3	4941
1927	2402	1727	873	565	392	211	4	8	6182
1937	2693	1853	993	589	457	233	6	11	6836
1947	2985	2055	1106	646	502	257	11	17	7579
1957	3625	2656	1413	873	688	326	21	38	9640
1967	4295	3274	1700	1110	879	375	62	103	11799
1977	5002	3837	2130	1286	1204	415	104	214	14192
1987	5617	4210	2675	1393	1496	449	158	265	16264
1997	6274	4605	3401	1480	1798	474	187	310	18532

Table 1. Australian population figures

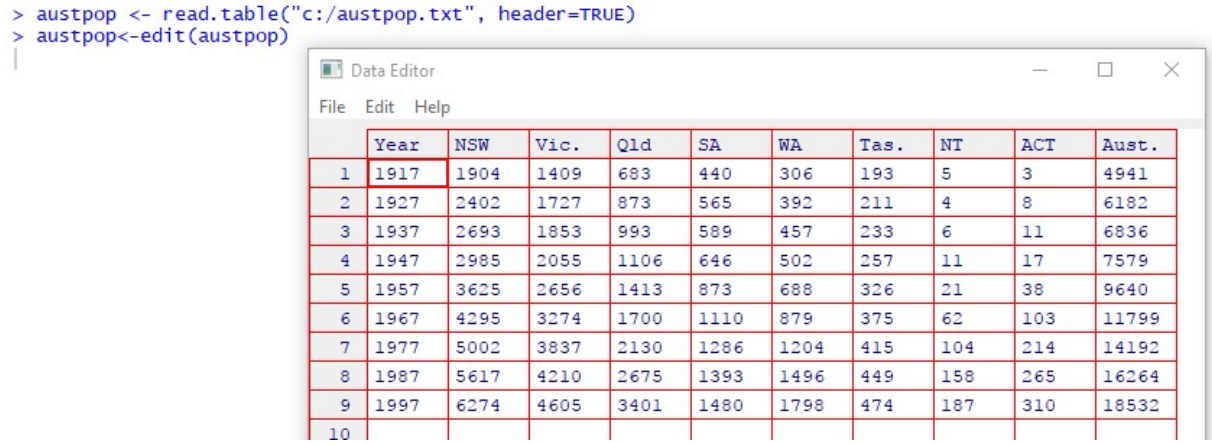

Figure 2. Editor window, showing the data frame `austpop`.

The inverse of `dump()` is `source()`.

```
> source("data.R")
> str(y)
'data.frame': 1 obs. of 2 variables:
$ a: int 1
$ b: Factor w/ 1 level "a": 1
> x
  [1] "foo"
```

Example

Consider the population figures for Australian states and territories (at various times since 1917) found in Table 1. The following code reads in the data from the file `austpop.txt` on the c: drive.[2]

```
> austpop <-
    read.table("c:/austpop.txt",
    header=TRUE)
```

The <− is a left diamond bracket (<) followed by a minus sign (−). It means "is assigned to". Use of header=TRUE causes R to use the first line to get header information for the columns. If the column headings are not included in the file, this argument can be omitted.

Now type in `austpop` at the command line prompt, displaying the object on the screen:

```
> austpop
  Year NSW Vic Qld SA WA Tas NT ACT Aust
1 1917 1904 1409 683 440 306 193 5 3 4941
2 1927 2402 1727 873 565 392 211 4 8 6182
. . .
```

To edit the data frame in a spreadsheet-like format, simply type, as can be seen in Figure 2.

```
> austpop<-edit(austpop)
```

Exercises

1. Read the following data into R (number of honeyeaters seen at the EucFACE site in a week). Give the resulting data frame a reasonable name. Type it into Excel or text file and save it as a CSV file or txt.

Day	nrbirds	Day	nrbirds
Sunday	3	Thursday	8
Monday	2	Friday	1
Tuesday	5	Saturday	2
Wednesday	0		

[2] Of course, that's not where your copy of the file might be saved.

Enter the following data as new observations of a different week starting on Sunday: 4, 3, 6, 1, 9, 2, 0.
2. Read the data from the space shuttle launch data into R, as in the above exercise.

4. Simple Data Visualizations

Whenever we analyze data, the first thing we should do is look at it. For each variable, what are the most common values? How much variability is present? Are there any unusual observations?

Producing graphics for data analysis is relatively simple. Producing graphics for publication is relatively more complex and typically requires a great deal of tweaking to achieve the desired appearance.

Our intent is to provide sufficient guidance so that most effects can be achieved, but further investigation of the documentation and experimentation will doubtless be necessary for specific needs.

R provides a number of functions for visualizing data. Table 2 summarizes a few important plot types. Advanced functionality is provided by Hadley Wickham's **ggplot2** (which is not covered in this bried outline).

The `plot()` Function

The most common plotting function in R is the `plot()` function. It is a generic function, meaning, it calls various methods according to the type of the object passed which is passed to it.

In the simplest case, we can pass in a vector and we get a scatter plot of magnitude vs index. More generally, we can pass in two vectors and a scatter plot of the points formed by matching coordinates is displayed.

For example, the command `plot(c(1,2),c(3,5))` would plot the points (1,3) and (2,5).

Here is a more concrete example where we plot the sine function in the range from $-\pi$ to π.

```
x <- seq(-pi,pi,0.1)
plot(x, sin(x))
```

The result is shown in Figure 3. We can add a title to our plot with the parameter main. Similarly, `xlab` and `ylab` can be used to label the x-axis and y-axis respectively (see Figure 4. The curve is made up of circular black points. This is the default setting for shape and colour. This can be changed by using the argument `type`. It accepts the following strings (with given effect)

- `p` – points
- `l` – lines
- `b` – both points and lines
- `c` – empty points joined by lines
- `o` – overplotted points and lines
- `s` – stair steps
- `h` – histogram-like vertical lines
- `n` – does not produce any points or lines

Similarly, we can specify the colour using the argument `col`. For instance, the following code produces the display found in Figure 5.

```
plot(x, sin(x),
 main="The Sine Function",
 ylab="sin(x)",
 type="l",
 col="blue")
```

Calling `plot()` multiple times will have the effect of plotting the current graph on the same window, replacing the previous one.

However, we may sometimes wish to overlay the plots in order to compare the results. This is made possible by the functions `lines()` and `points()`, which add lines and points respectively, to the existing plot.

```
plot(x, sin(x),
 main="Overlaying Graphs",
 ylab="",
 type="l",
 col="blue")
lines(x,cos(x), col="red")
legend("topleft",
     c("sin(x)","cos(x)"),
     fill=c("blue","red")
)
```

The `legend()` function allows for the appropriate display in Figure 6 display the legend.

The `barplot()` Function

Bar plots can be created in R using the `barplot()` function. We can supply a vector or matrix to this function, and it will display a bar chart with bar heights equal to the magnitude of the elements in the vector.

Let us suppose, we have a vector of maximum temperatures for seven days, as follows.

```
max.temp <- c(22, 27, 26, 24, 23, 26, 28)
```

Now we can make a bar plot out of this data using a simple R command (see Figure 7

```
barplot(max.temp)
```

This function can take on a lot of arguments, which you can read about by querying for help in R: `?barplot`. Frequently-used arguments include:

- `main` to specify the title
- `xlab` and `ylab` to provide labels for the axes

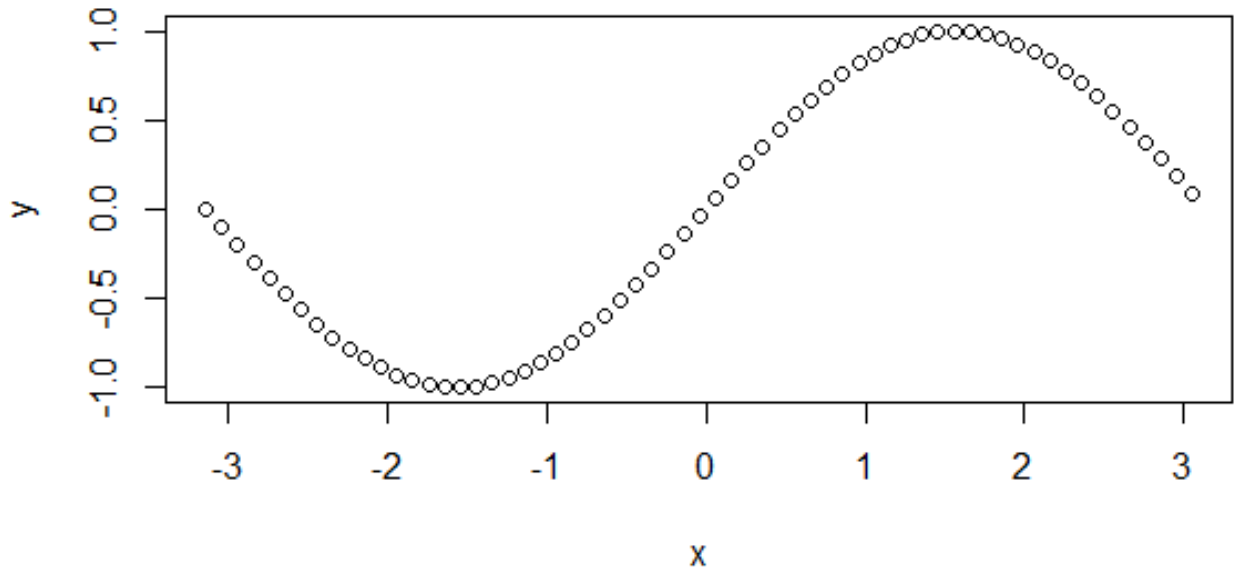

Figure 3. Sine curve.

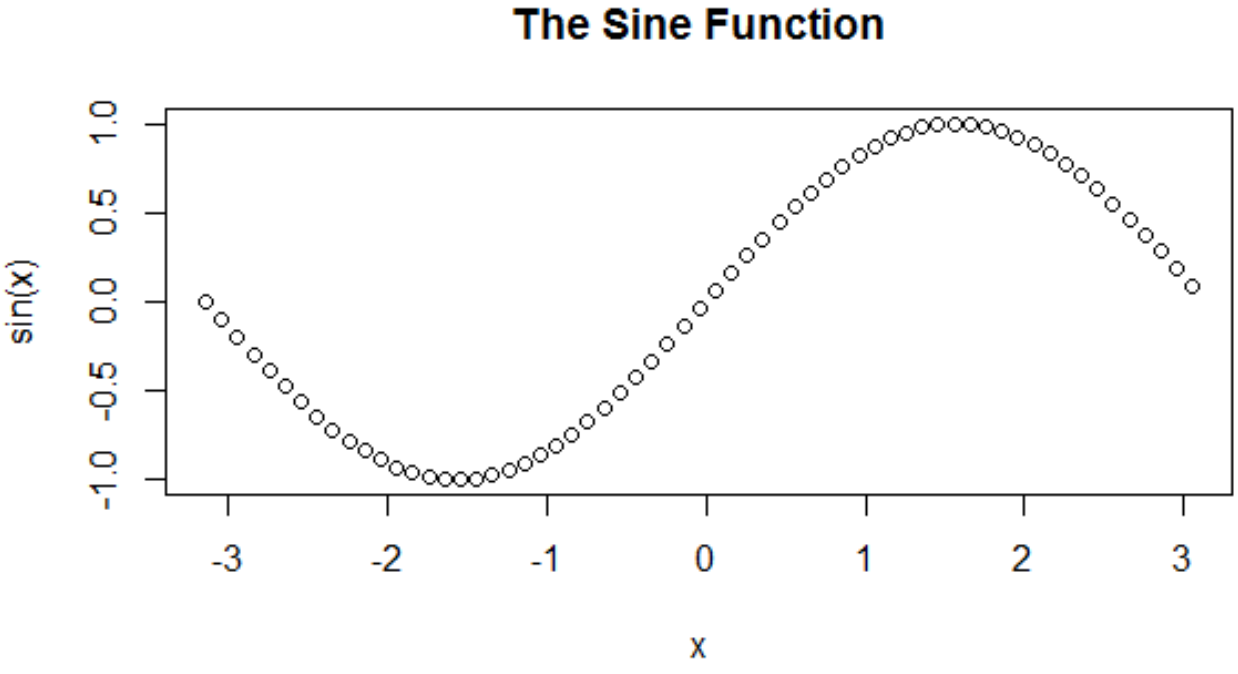

Figure 4. Fancy sine curve.

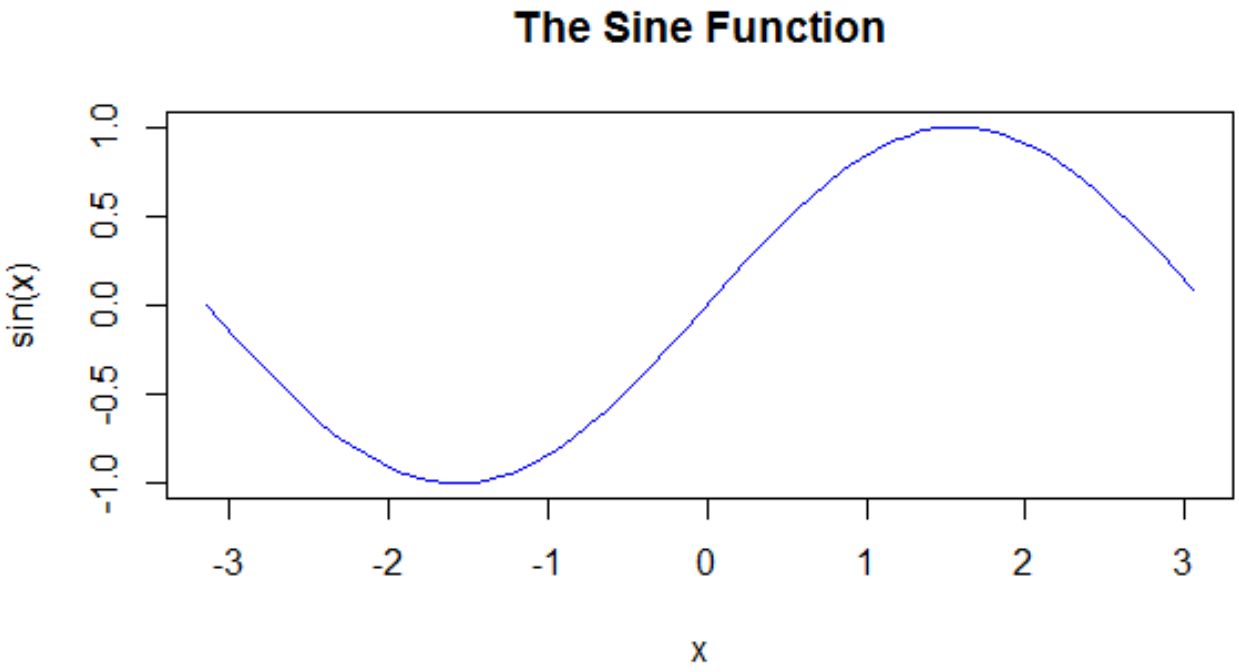

Figure 5. Fancier sine curve.

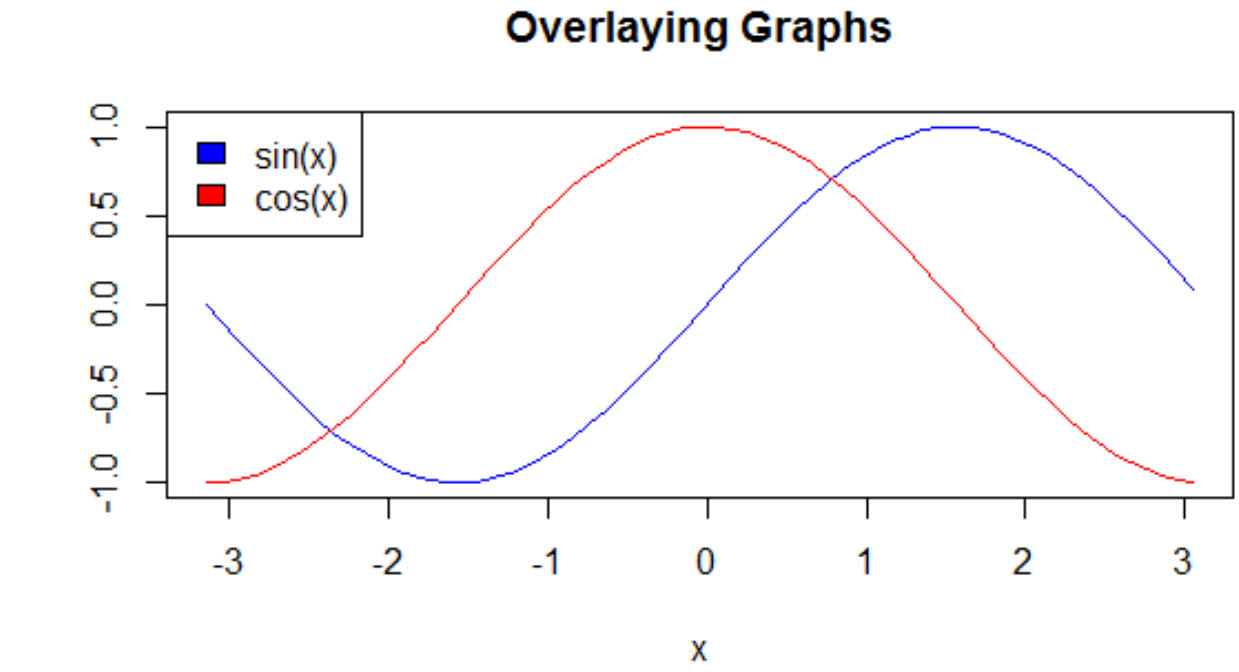

Figure 6. Overlaid fancier trigonometric curves.

Function	Graph type
`plot()`	Scatter plots and various others
`barplot()`	Bar plot (including stacked and grouped bar plots)
`hist()`	Histograms and (relative) frequency diagrams
`curve()`	Curves of mathematical expressions
`pie()`	Pie charts (for less scientific uses)
`boxplot()`	Box-and-whisker plots

Table 2. R plotting functions.

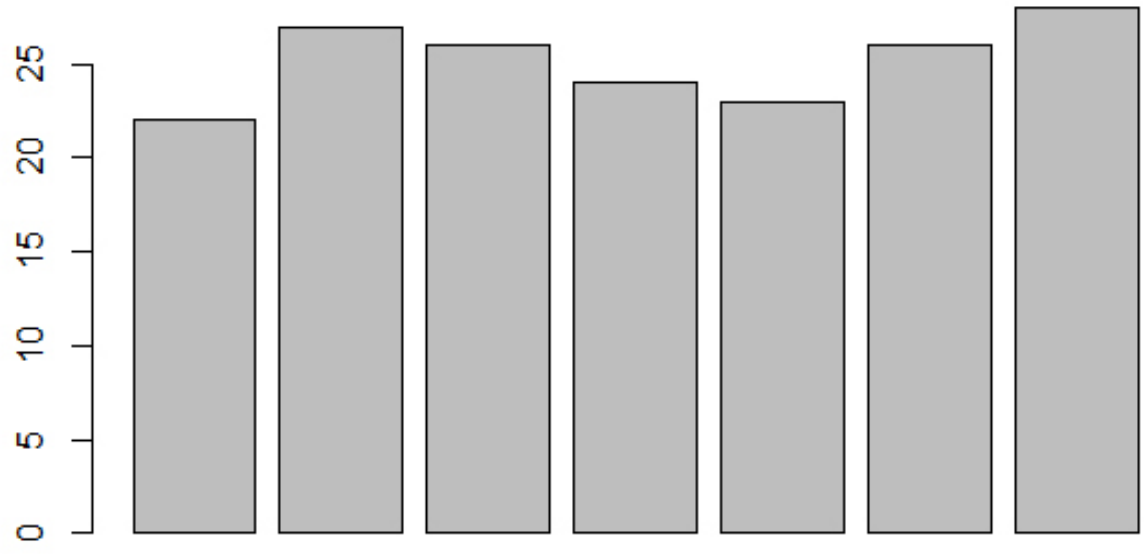

Figure 7. Barchart of daily temperature maximum.

- `names.arg` to provide a name for each bar
- `col` to define colour, etc.

We can also transpose the plot to have horizontal bars by providing the argument `horiz = TRUE`. The barchart produced by the following code is shown in Figure 8.

```
# barchart with added parameters
barplot(max.temp,
  main = "Maximum Temperatures in a
     Week",
  xlab = "Degree Celsius",
  ylab = "Day",
  names.arg = c("Sun", "Mon", "Tue",
     "Wed", "Thu", "Fri", "Sat"),
  col = "darkred",
  horiz = TRUE)
```

Sometimes we may be interested in displaying the count or magnitude for each category. For instance, consider the following vector of age measurements for 10 college frosh.

```
age <- c(17,18,18,17,18,19,18,16,18,18)
```

Simply calling `barplot(age)` will not provide the required plot. It will plot 10 bars with appropriate heights (the students's age), but the display will not be available for each category. The values can be quickly found using the `table()` function, as shown below.

```
> table(age)
   age
   16 17 18 19
   1  2  6  1
```

Now plotting this data will produce the required barchart (see Figure 9). In the code below, the argument `density` is used to shade the bars.

```
barplot(table(age),
  main="Age Count of 10 Students",
  xlab="Age",
  ylab="Count",
  border="red",
  col="blue",
  density=10
)
```

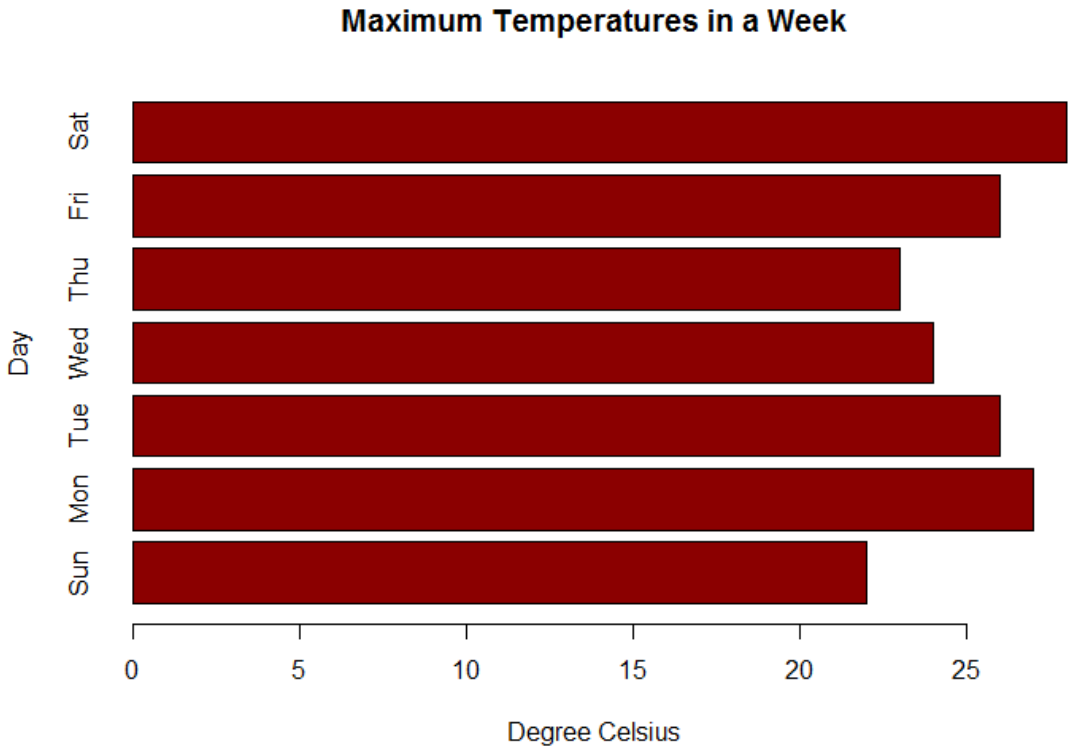

Figure 8. Fancy barchart of daily temperatures maximum.

Histograms

Histograms display the distribution of a continuous variable by dividing the range of scores into a specified number of bins on the *x*-axis and displaying the frequency of scores in each bin on the *y*-axis.

You can create histograms with the function `hist()`. The option `freq=FALSE` creates a plot based on probability densities rather than frequencies. The `breaks` option controls the number of bins. The default produces equally spaced breaks when defining the cells of the histogram.

For illustrative purposes, we will use several of the variables from the *Motor Trend Car Road Tests* (`mtcars`) dataset provided in the base R installation. The following listing provides the code for four variations on a histogram; the results are plotted in Figure 10.

```
par(mfrow=c(2,2))
hist(mtcars$mpg)
hist(mtcars$mpg,
    breaks=12,
    col="red",
    xlab="Miles Per Gallon",
    main="Colored histogram with 12
       bins")
hist(mtcars$mpg,
    freq=FALSE,
    breaks=12,
    col="red",
    xlab="Miles Per Gallon",
main="Histogram, rug plot, density
    curve")
rug(jitter(mtcars$mpg))
lines(density(mtcars$mpg), col="blue",
    lwd=2)
x <- mtcars$mpg
h<-hist(x,
      breaks=12,
      col="red",
      xlab="Miles Per Gallon",
      main="Histogram with normal curve
          and box")
xfit<-seq(min(x), max(x), length=40)
yfit<-dnorm(xfit, mean=mean(x), sd=sd(x))
yfit <- yfit*diff(h$mids[1:2])*length(x)
lines(xfit, yfit, col="blue", lwd=2)
box()
```

The first histogram demonstrates the default plot with no specified options: five bins are created, and the default axis labels and titles are printed.

In the second histogram, 12 bins have been specified, as well as a red fill for the bars, and more attractive and informative labels and title.

The third histogram uses the same colours, number of bins, labels, and titles as the previous plot but adds a

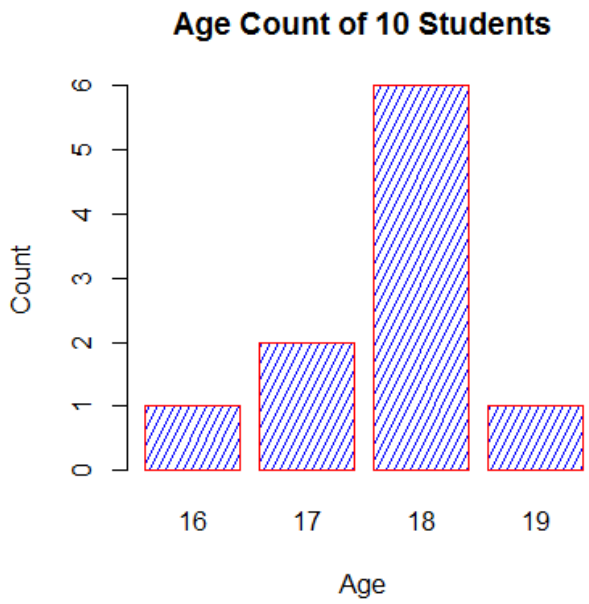

Figure 9. Fancier barchart of college frosh ages.

density curve and **rug-plot** overlay. The density curve is a kernel density estimate and is described in the next section. It provides a smoother description of the distribution of scores. The call to function `lines()` overlays this curve in blue and a width twice the default line thickness; a rug plot is a one-dimensional representation of the actual data values. If there are multiple repeated values, something like the following code will **jitter** the data, adding a small random value to each data point (a uniform random variate between ± `amount`) in order to avoid overlapping points.

```
rug(jitter(mtcars$mpag, amount=0.01))
```

The fourth histogram is similar to the second but with a superposed normal curve and a box around the figure. The code for superposing the normal curve comes from a suggestion posted to the R-help mailing list by Peter Dalgaard. The surrounding box is produced by the `box()` function.

The `curve` Function

Given an expression for a function $y(x)$, we can plot the values of y for various values of x in a given range. This can be accomplished using an R library function called `curve()`. We now plot the simple polynomial function $y = 3x^2 + x$ in the range $x = [1, 10]$, as follows:

```
curve( 3*x^2 + x, from=1, to=10, n=300,
    xlab="xvalue", ylab="yvalue",
          col="blue", lwd=2, main="Plot
             of (3x^2 + x)" )
```

This command produces the display found in Figure 11.

The important parameters of the `curve()` function are:

- the first parameter is the mathematical expression of the function to plot, written in the format for writing mathematical operations in or LaTeX;
- two numeric parameters (`from` and `to`) that represent the endpoints of the range of x;
- the integer `n` that represents the number of equally-spaced values of x between the `from` and `to` points;
- the other parameters (`xlab`, `ylab`, `col`, `lwd`, `main`) have their usual meaning.

Box Plots

A box-and-whiskers plot describes the distribution of a continuous variable by plotting its five-number summary: the minimum, lower quartile Q_1 (25th percentile), median (50th percentile), upper quartile Q_3 (75th percentile), and the maximum. It display observations which may be identified as potential outliers (values outside the range of $[5/2Q_1 - 3/2Q_3, 5/2Q_3 - 3/2Q_1]$ for normally distributed variables). For example, the following statement produces the plot shown in Figure 12.

```
boxplot(mtcars$mpg, main="Box plot",
    ylab="Miles per Gallon")
```

Exercises Use the Australian data to to do the following

- Graph the New South Wales (NSW) population with all defaults using `plot()`. Redo the graph by adding a title, a line to connect the points, and some colour.
- Compare the population of New South Wales (NSW) and the Australian Capital Territory (ACT) by using the functions `plot()` and `lines()`, then add a legend to appropriately display your graph.
- Use a bar chart to graph the population of Queensland (QLD), add an appropriate title to your graph, and display the years from 1917 to 1997 on the appropriate bars.
- Create a light blue histogram for the population of South Australia (SA).

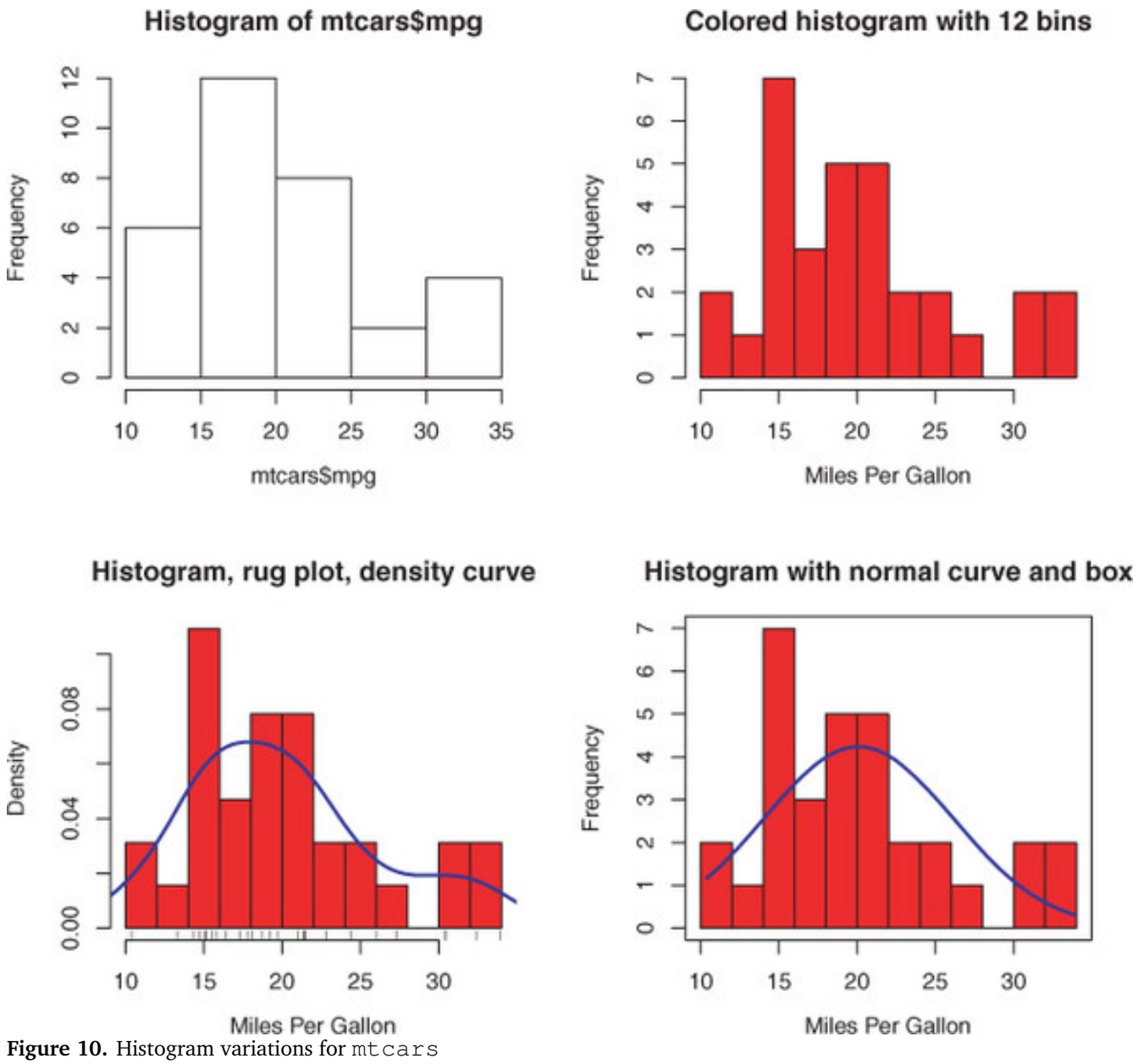

Figure 10. Histogram variations for `mtcars`

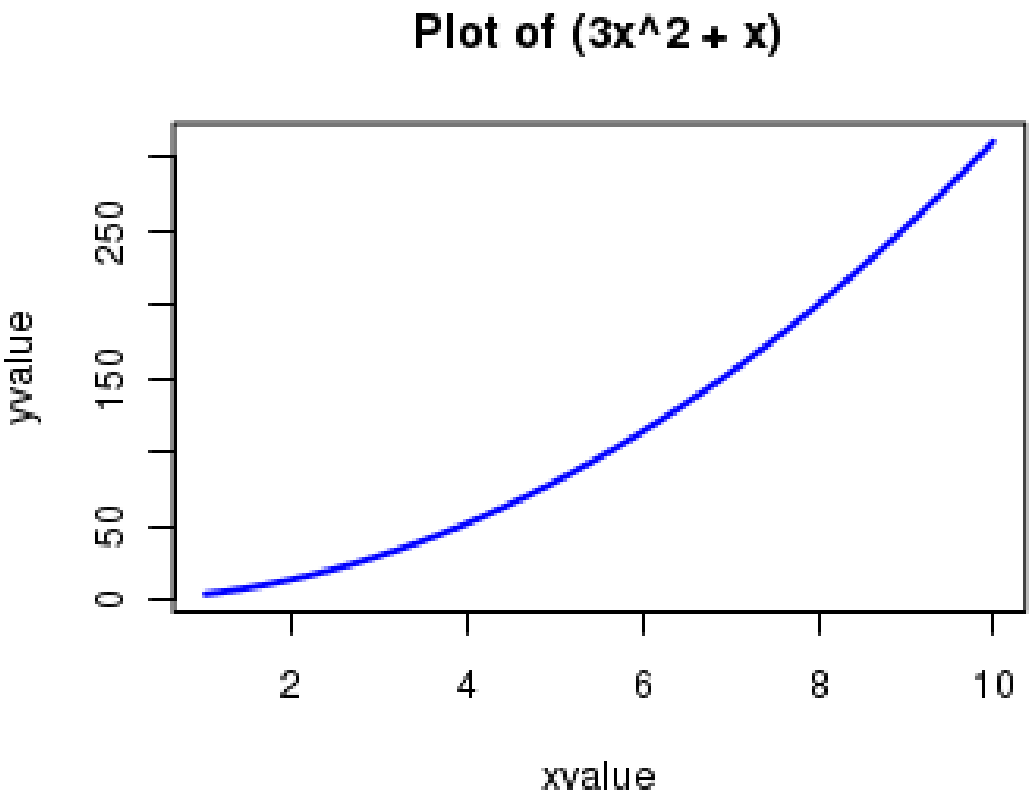

Figure 11. Polynomial curve: $y = 3x^2 + x$.

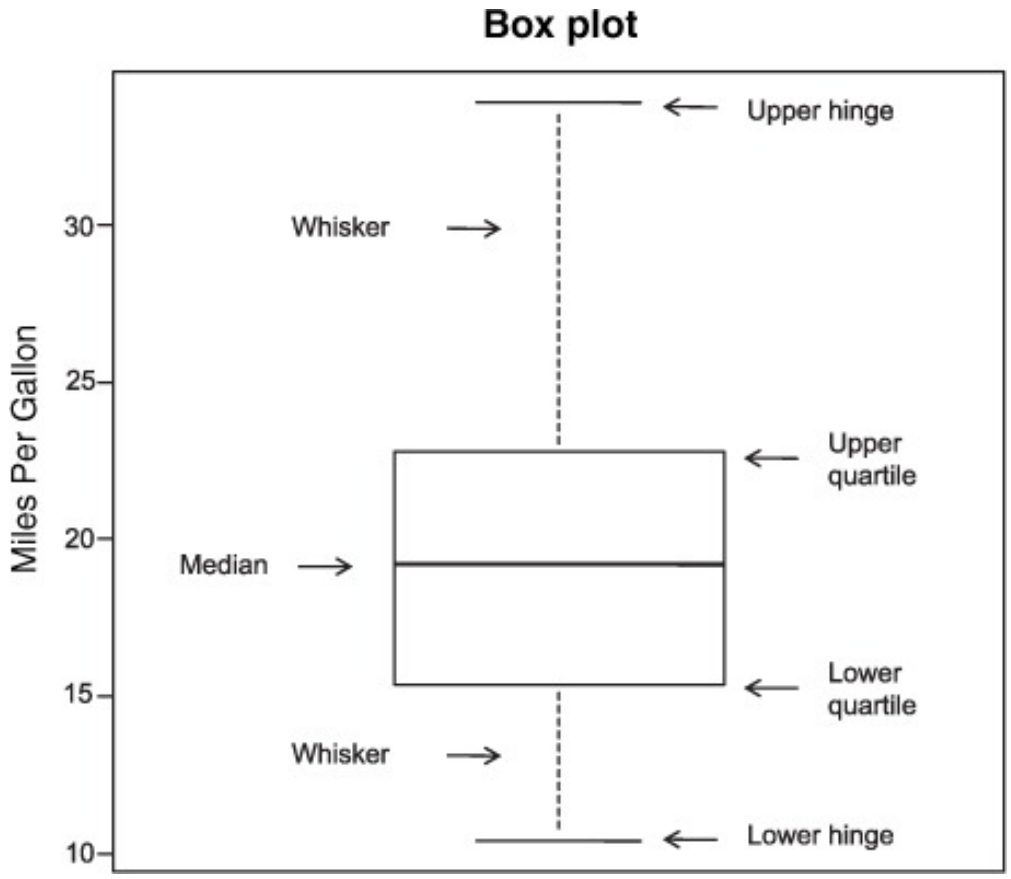

Figure 12. Box plot of `mtcars`'s `mpg`, with manual annotations.

5. Common Statistical Procedures

Once the data is properly organized and visual exploration has begun in earnest, the typical next step is to describe the distribution of each variable numerically, followed by an exploration of the relationships among selected variables. The objective is to answer questions such as:

- What kind of mileage are cars getting these days? Specifically, what's the distribution of miles per gallon (mean, standard deviation, median, range, and so on) in a survey of automobile makes and models?
- After a new drug trial, what is the outcome (no improvement, some improvement, marked improvement) for drug versus placebo groups? Does the gender of the participants have an impact on the outcome?
- What is the correlation between income and life expectancy? Is it significantly different from zero?
- Are you more likely to receive imprisonment for a crime in different regions of Canada? Are the differences between regions statistically significant?

Basic Statistics

When it comes to calculating descriptive statistics, R can basically do it all. Let's start with functions that are included in the base installation. We will then look for extensions that are available through the use of user-contributed packages.

For illustrative purposes, we will again use several of the variables from the *Motor Trend Car Road Tests* (`mtcars`) dataset provided in the base installation. We will focus on miles per gallon (`mpg`), horsepower (`hp`), and weight (`wt`):

```
> myvars <- c("mpg", "hp", "wt")
> head(mtcars[myvars])
                   mpg  hp   wt
Mazda RX4         21.0 110 2.62
Mazda RX4 Wag     21.0 110 2.88
Datsun 710        22.8  93 2.32
Hornet 4 Drive    21.4 110 3.21
Hornet Sportabout 18.7 175 3.44
Valiant           18.1 105 3.46
```

Let's first look at descriptive statistics for all 32 models. We will then examine descriptive statistics by transmission type (`am`) and number of cylinders (`cyl`). Transmission type is a dichotomous variable coded 0=automatic, 1=manual, while the number of cylinders can be 4, 5, or 6.

In the base installation, you can use the `summary()` function to obtain descriptive statistics. An example is presented in the following listing.

```
> myvars <- c("mpg", "hp", "wt")
> summary(mtcars[myvars])
      mpg              hp              wt
 Min.   :10.4   Min.   : 52.0   Min.   :1.51
 1st Qu.:15.4   1st Qu.: 96.5   1st Qu.:2.58
 Median :19.2   Median :123.0   Median :3.33
 Mean   :20.1   Mean   :146.7   Mean   :3.22
 3rd Qu.:22.8   3rd Qu.:180.0   3rd Qu.:3.61
 Max.   :33.9   Max.   :335.0   Max.   :5.42
```

The `summary()` function provides the minimum, maximum, quartiles, and mean for numerical variables, and the respective frequencies for factors and logical vectors. The functions `apply()` or `sapply()` can be used to provide any descriptive statistics. The format in use is:

```
> sapply(x, FUN, options)
```

where *x* is the data frame (or matrix) and `FUN` is an arbitrary function. If options are present, they're passed to `FUN`.

Typical functions that can be plugged here are

- `mean()`
- `sd()`
- `var()`
- `min()`
- `max()`
- `median()`
- `length()`
- `range()`
- `quantile()`
- `fivenum()`

The example in the next listing provides several descriptive statistics using `sapply()`, including **skew** and **kurtosis**.

```
> mystats <- function(x, na.omit=FALSE){
            if (na.omit)
               x <- x[!is.na(x)]
            m <- mean(x)
            n <- length(x)
            s <- sd(x)
            skew <- sum((x-m)^3/s^3)/n
            kurt <- sum((x-m)^4/s^4)/n -
                3
            return(c(n=n, mean=m,
                stdev=s, skew=skew,
                kurtosis=kurt))
          }

> myvars <- c("mpg", "hp", "wt")
> sapply(mtcars[myvars], mystats)
             mpg     hp     wt
n       32.000 32.000 32.0000
mean    20.091 146.688 3.2172
stdev    6.027 68.563 0.9785
skew     0.611  0.726 0.4231
kurtosis -0.373 -0.136 -0.0227
```

For cars in this sample, the mean `mpg` is 20.1, with a standard deviation of 6.0. The distribution is skewed to the

right (+0.61) and is somewhat flatter than a normal distribution (–0.37). This is most evident if you graph the data. Note that if you wanted to omit missing values, you could use

```
> sapply(mtcars[myvars], mystats,
    na.omit=TRUE).
```

The `Hmisc` and `pastecs` packages Several packages offer functions for descriptive statistics, including `Hmisc` and `pastecs`. Because these packages are not included in the base distribution, they need to be installed on first use. `Hmisc`'s `describe()` function returns the number of variables and observations, the number of missing and unique values, the mean, quantiles, and the five highest and lowest values. An example is provided in Table 3.

The pastecs package includes the function `stat.desc()` that provides a wide range of descriptive statistics. The format is

```
> stat.desc(x, basic=TRUE, desc=TRUE,
    norm=FALSE, p=0.95)
```

where *x* is a data frame or a time series. If `basic=TRUE` (the default), the number of values, null values, missing values, minimum, maximum, range, and sum are provided.

If `desc=TRUE` (also the default), the median, mean, standard error of the mean, 95% confidence interval for the mean, variance, standard deviation, and coefficient of variation are also provided.

Finally, if `norm=TRUE` (not the default), normal distribution statistics are returned, including skewness and kurtosis (with statistical significance) and the Shapiro–Wilk test of normality. A *p*—value option is used to calculate the confidence interval for the mean (.95 by default). The next listing gives an example.

```
> library(pastecs)
> myvars <- c("mpg", "hp", "wt")
> stat.desc(mtcars[myvars])
                mpg      hp     wt
nbr.val       32.00  32.000 32.000
nbr.null       0.00   0.000  0.000
nbr.na         0.00   0.000  0.000
min           10.40  52.000  1.513
max           33.90 335.000  5.424
range         23.50 283.000  3.911
sum          642.90 4694.000 102.952
median        19.20 123.000  3.325
mean          20.09 146.688  3.217
SE.mean        1.07  12.120  0.173
CI.mean.0.95   2.17  24.720  0.353
var           36.32 4700.867  0.957
std.dev        6.03  68.563  0.978
coef.var       0.30   0.467  0.304
```

Correlations

Correlation coefficients are used to describe relationships among quantitative variables. The sign ± indicates the direction of the relationship (positive or inverse), and the magnitude indicates the strength of the relationship (ranging from 0 for no linear relationship to 1 for a perfectly predictable linear relationship).

In this section, we look at a variety of correlation coefficients, as well as tests of significance. We will use the `state.x77` dataset available in the base R installation. It provides data on the population, income, illiteracy rate, life expectancy, murder rate, and high school graduation rate for the 50 US states in 1977. There are also temperature and land-area measures, but we drop them to save space. In addition to the base installation, we'll be using the `psych` and `ggm` packages.

R can produce a variety of correlation coefficients, including Pearson, Spearman, Kendall, partial, polychoric, and polyserial. The Pearson product-moment correlation assesses the degree of linear relationship between two quantitative variables. Spearman's rank-order correlation coefficient assesses the degree of relationship between two rank-ordered variables. Kendall's tau is also a nonparametric measure of rank correlation.

The `cor()` function produces all three correlation coefficients, whereas the `cov()` function provides covariances. There are many options, but a simplified format for producing correlations is

```
> cor(x, use= , method= )
```

Where *x* is a matrix or a data frame, and `use` specifies the handling of missing data. The options are `all.obs` (assumes no missing data), `everything` (any correlation involving a case with missing values will be set to missing), `complete.obs` (listwise deletion), and `pairwise.complete.obs` (pairwise deletion). The `method` specifies the type of correlation. The options are `pearson`, `spearman`, and `kendall`. The default options are `use ="everything"` and `method= "pearson"`. An example is provided in Table 4

The first call produces the variances and covariances. The second provides Pearson product-moment correlation coefficients, and the third produces Spearman rank-order correlation coefficients. You can see, for example, that a strong positive correlation exists between income and high school graduation rate and that a strong negative correlation exists between illiteracy rates and life expectancy.

A **partial correlation** is a correlation between two quantitative variables, controlling for one or more other quantitative variables. You can use the `pcor()` function in the `ggm` package to provide partial correlation coefficients. Again, this package is not installed by default, so be sure to install it on first use. The format is

```
> library(Hmisc)
> myvars <- c("mpg", "hp", "wt")
> describe(mtcars[myvars])
3 Variables  32 Observations
---------------------------------------------------------------------------
mpg
n missing unique Mean .05 .10  .25  .50  .75  .90  .95
32    0   25  20.09 12.00 14.34 15.43 19.20 22.80 30.09 31.30

lowest : 10.4 13.3 14.3 14.7 15.0, highest: 26.0 27.3 30.4 32.4 33.9
---------------------------------------------------------------------------
hp
n missing unique Mean .05   .10  .2    .50  .75  .90   .95
32     0    22  146.7 63.65 66.00 96.50 123.00 180.00 243.50 253.55

lowest : 52 62 65 66 91, highest: 215 230 245 264 335
---------------------------------------------------------------------------
wt
n missing unique Mean .05  .10   .25   .50   .75   .90  .95
32    0    29  3.217 1.736 1.956 2.581 3.325 3.610 4.048 5.293

lowest : 1.513 1.615 1.835 1.935 2.140, highest: 3.845 4.070 5.250 5.345 5.424
```

Table 3. Output from the `decribe()` function, `mtcars`.

```
> pcor(u, S)
```

where u is a vector of numbers, with the first two numbers being the indices of the variables to be correlated, and the remaining numbers being the indices of the conditioning variables (that is, the variables being partialed out), and S is the covariance matrix among the variables. An example will help clarify this:

```
> library(ggm)
> colnames(states)
[1] "Population" "Income" "Illiteracy"
    "Life Exp" "Murder" "HS Grad"
> pcor(c(1,5,2,3,6), cov(states))
[1] 0.346
```

In this case, 0.346 is the correlation between population (variable 1) and murder rate (variable 5), controlling for the influence of income, illiteracy rate, and high school graduation rate (variables 2, 3, and 6 respectively). The use of partial correlations is common in the social sciences.

Simple Linear Regression

In many ways, regression analysis is at the heart of statistics. It is a broad term for a set of methodologies used to predict a response variable (also called a dependent, criterion, or outcome variable) from one or more predictor variables (also called independent or explanatory variables).

In general, regression analysis can be used to identify the explanatory variables that are related to a response variable, to describe the form of the relationships involved, and to provide an equation for predicting the response variable from the explanatory variables.

For example, an exercise physiologist might use regression analysis to develop an equation for predicting the expected number of calories a person will burn while exercising on a treadmill.

The response variable is the number of calories burned (calculated from the amount of oxygen consumed), and the predictor variables might include duration of exercise (minutes), percentage of time spent at their target heart rate, average speed (mph), age (years), gender, and body mass index (BMI).

From a practical point of view, regression analysis would help answer questions such as:

- How many calories can a 30-year-old man with a BMI of 28.7 expect to burn if he walks for 45 minutes at an average speed of 4 miles per hour and stays within his target heart rate 80% of the time?
- What's the minimum number of variables you need to collect in order to accurately predict the number of calories a person will burn when walking?

R has powerful and comprehensive features for fitting regression models – the abundance of options can be confusing as well.[3] The basic function for fitting a linear model is `lm()`. The format is

[3] For example, in 2005, Vito Ricci created a list of more than 205 functions in R that are used to generate regression models (**mng.bz/NJhu**).

```
> states<- state.x77[,1:6]
> cov(states)
           Population Income Illiteracy Life Exp Murder HS Grad
Population   19931684 571230    292.868 -407.842 5663.52 -3551.51
Income         571230 377573   -163.702  280.663 -521.89  3076.77
Illiteracy        293   -164      0.372   -0.482   1.58    -3.24
Life Exp         -408    281     -0.482    1.802  -3.87     6.31
Murder           5664   -522      1.582   -3.869  13.63   -14.55
HS Grad         -3552   3077     -3.235    6.313 -14.55    65.24

> cor(states)
           Population Income Illiteracy Life Exp Murder HS Grad
Population     1.0000 0.208      0.108  -0.068  0.344  -0.0985
Income         0.2082 1.000     -0.437   0.340 -0.230   0.6199
Illiteracy     0.1076 -0.437     1.000  -0.588  0.703  -0.6572
Life Exp      -0.0681 0.340    -0.588    1.000 -0.781   0.5822
Murder         0.3436 -0.230     0.703  -0.781  1.000  -0.4880
HS Grad       -0.0985 0.620    -0.657    0.582 -0.488   1.0000
> cor(states, method="spearman")
           Population Income Illiteracy Life Exp Murder HS Grad
Population      1.000 0.125      0.313  -0.104  0.346  -0.383
Income          0.125 1.000     -0.315   0.324 -0.217   0.510
Illiteracy      0.313 -0.315     1.000  -0.555  0.672  -0.655
Life Exp       -0.104 0.324    -0.555    1.000 -0.780   0.524
Murder          0.346 -0.217     0.672  -0.780  1.000  -0.437
HS Grad        -0.383 0.510    -0.655    0.524 -0.437   1.000
```

Table 4. Various correlation coefficients for the `state.x77` dataset.

```
> myfit <- lm(formula, data)
```

where `formula` describes the model to be fit and `data` is the data frame containing the data to be used in fitting the model. The resulting object (`myfit`, in this case) is a list that contains extensive information about the fitted model. The formula is typically written as

$$Y \sim X_1 + X_2 + \cdots + X_k$$

where the $\sim$ separates the response variable on the left from the predictor variables on the right, and the predictor variables are separated by + signs.

In addition to `lm()`, Table 5 lists several functions that are useful when generating regression models. Each of these functions is applied to the object returned by `lm()` in order to generate additional information based on the fitted model.

Example The dataset `women` in the base installation provides the heights and weights for a set of 15 women aged 30 to 39. Let's say that you want to predict the weight from the height.[4] The analysis can be conducted as in Table 6. From the output, you see that the prediction equation is

$$\widehat{\text{Weight}} = -87.52 + 3.45 \times \text{ Height}.$$

Because a height of 0 is impossible, there is no sense in trying to give a physical interpretation to the intercept – it merely becomes an adjustment constant.

From the `P(>|t|)` column, we see that the regression coefficient (3.45) is significantly different from zero ($p < 0.001$) which indicates that there's an expected increase of 3.45 pounds of weight for every 1 inch increase in height. The multiple R-squared coefficient (0.991) indicates that the model accounts for 99.1% of the variance in weights.

Bootstrapping

Bootstrapping is a powerful and elegant approach to estimating the sampling distribution of specific statistics. It can be implemented in many situations where asymptotic results are difficult to find or otherwise unsatisfactory. Bootstrapping proceeds using three steps:

1. resample the dataset (with replacement) many times over (typically on the order of 10,000);
2. calculate the desired statistic from each resampled dataset;
3. use the distribution of the resampled statistics to estimate the standard error of the statistic (normal approximation method) or construct a confidence interval using quantiles of that distribution (percentile method).

[4] Having an equation for predicting weight from height could help identifying overweight or underweight individuals, or at least provide a red flag.

Function	Action
`summary()`	Displays detailed results for the fitted model
`coefficients()`	Lists the model parameters (intercept and slopes) for the fitted model
`confint()`	Provides confidence intervals for the model parameters (95% by default)
`residuals()`	Lists the residual values in a fitted model
`anova()`	Generates an ANOVA table for a fitted model, or an ANOVA table comparing two or more fitted models
`plot()`	Generates diagnostic plots for evaluating the fit of a model
`predict()`	Uses a fitted model to predict response values for a new dataset

Table 5. Other useful R linear fitting functions.

There are several ways to bootstrap in R.

As an example, let's say we want to estimate the standard error and 95% confidence interval for the **coefficient of variation** (CV), defined as σ/μ, for a random variable X. We generate normal data with a mean and variance of 1.

```
> x = rnorm(1000, mean=1)
```

The user must provide code to calculate the statistic of interest as a function.

```
> covfun = function(x) { # multiply CV
    by 100
return(100*sd(x)/mean(x))
}
```

The `replicate()` function is the base R tool for repeating function calls. Here, we nest within it a call to `covfun()` and a call to sample the data with replacement using the `sample()` function.

```
> options(digits=4)
> res2 = replicate(2000,
    covfun(sample(x, replace=TRUE)))
> quantile(res2, c(.025, .975))
   2.5% 97.5%
   98.85 116.07
```

The percentile interval is easy to calculate from the observed bootstrapped statistics. If the distribution of the bootstrap samples is approximately normally distributed, a t interval could be created by calculating the standard deviation of the bootstrap samples and finding the appropriate multiplier for the confidence interval. Plotting the bootstrap sample estimates is helpful to determine the form of the bootstrap distribution.

Exercises

For this question, use the `pupae` data attached to this report. This dataset comes from an experiment where larvae were left to feed on Eucalyptus leaves, in a glasshouse that was controlled at two different levels of temperature and CO2 concentration. After the larvae pupated (that is, turned into pupae), the body weight was measured, as well as the cumulative 'frass' (larvae excrement) over the entire time it took to pupate. Data courtesy of Tara Murray, and simplified for the purpose of this exercise.

- T_treatment - Temperature treatments ('ambient' and 'elevated')
- Co2_treatment - CO2 treatment (280 or 400 ppm).
- Gender - The gender of the pupae : 0 (male), 1 (female)
- PupalWeight - Weight of the pupae (g)
- Frass - Frass produced (g)

Perform a simple linear regression of *Frass* on *PupalWeight*. Produce and inspect the following:

- Summary of the model.
- Plots of the data.

References

[1] Kabacoff, R.I. [2011], *R in Action, Second Edition: Data analysis and graphics with R*, Live.

[2] Horton, N.J., Kleinman, K. [2016], *Using R and RStudio for Data Management, Statistical Analysis, and Graphics Second Edition*, CRC Press.

[3] Peng, R.D. [2015], *R Programming for Data Science*, Learnpub.

[4] Duursma, R., Powell, J., Stone, G. [2017], *A Learning Guide to R*, Scribd.

[5] Maindonald, J.H. [2008], *Using R for Data Analysis and Graphics Introduction, Code and Commentary*, Centre for Bioinformation Science.

[6] **Plot Parameters in R**

[7] **R Programming**

[8] **Complete Tutorial to Learn Data Science from Scratch**

```
> fit <- lm(weight ~ height, data=women)
> summary(fit)

Call:
lm(formula=weight ~ height, data=women)

Residuals:
   Min    1Q Median  3Q   Max
-1.733 -1.133 -0.383 0.742 3.117

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.5167 5.9369 -14.7 1.7e-09 ***
height     3.4500   0.0911  37.9 1.1e-14 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.53 on 13 degrees of freedom
Multiple R-squared: 0.991, Adjusted R-squared: 0.99
F-statistic: 1.43e+03 on 1 and 13 DF, p-value: 1.09e-14

> women$weight

 [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164

> fitted(fit)

    1     2     3     4     5     6     7     8     9
112.58 116.03 119.48 122.93 126.38 129.83 133.28 136.73 140.18
   10    11    12    13    14    15
143.63 147.08 150.53 153.98 157.43 160.88

> residuals(fit)

   1    2    3    4    5    6    7    8    9   10   11
 2.42 0.97 0.52 0.07 -0.38 -0.83 -1.28 -1.73 -1.18 -1.63 -1.08
  12   13   14   15
-0.53 0.02 1.57 3.12

> plot(women$height,women$weight,
     xlab="Height (in inches)",
     ylab="Weight (in pounds)")
> abline(fit)
```

Table 6. Regression analysis of the `women` dataset.