Introduction to R – Week 1

Contents

1 Required Software: R

You will need to make sure you have on your computer:

- R, RStudio, all required packages installed OR administrator privileges to install them.

All of the labs and other assessed tasks in this unit will make use of the statistical computing language R. You will need R installed on your computer, which can be achieved by downloading it from the following link:

https://cran.ms.unimelb.edu.au/

We will also use the RStudio IDE. Download the open source (free) edition of RStudio Desktop from the following link:

https://rstudio.com/products/rstudio/

Packages

There will be a number of packages you will need installed which will be introduced at the beginning of each lab (except this one, where they are introduced below in the section introducing some example code). For these purposes you will either need to make sure you have administrator privileges on your computer or that these packages have been previously installed on your computer by an administrator user.

2 Reading Data and Visualisation in R

2.1 Tips for Beginners on Working in R and RStudio

Working Directory

For those familiar with working in the command line, the R working directory is the same concept. This is a key concept for interacting with files outside of the R instance as it determines the location in the filesystem for relative paths. This is important when reading in data from a file, for example. The working directory can be checked and changed using `getwd()` and `setwd()` respectively.

However, we suggest that when you do some work that you start a new project in RStudio in a folder for that chunk of work (for example you might choose to make a separate project (i.e. directory) for each lab, assignment, etc.). This essentially will tell RStudio to set the working directory to the directory associated

to that bit of work when you open that project, and avoids a lot of "file not found" errors beginners often run into.

Organising Code

For the purposes of this unit, you will likely organise most of your code into two types of file: * Scripts, and * R Notebooks (Rmarkdown).

Scripts are simply files ending with the file extension `.R` which contain code. One of the most useful things to be aware of in RStudio is the keyboard shortcut to Run Selected Lines(s). This is usually[1] Ctrl + Enter and will essentially copy paste the selected code into the Console Tab and immediately run it in the current R instance. Rather than typing code directly into a command line, having it stored in a file is the first step towards reproducible research. Running your code section by section from a file also allows for a vast improvement to your software development cycle over typing code into a console. For these two reasons we strongly suggest you adopt this approach if you have not already.

R Notebooks are for making documents heavier on the text/ explanation with code interspersed, rather than mostly code with comments interspersed (for which scripts are more appropriate). R Notebooks are very useful for producing rapid reports for an employer, will be a useful tool for you to construct your assignments, and are how this document (and all lab documents) are written. Source code for all these documents is provided as examples so feel free to check that out. More useful keyboard shortcuts to know in RStudio are those that run chunks of code (a "chunk" is an embedded section of code in an R Notebook) such as Ctrl + Alt + N.

Note there are more types of documents, and in fact the "chunk" system in R Notebooks uses a package `knitr`, which supports a much wider variety of document types and programming languages. For writing a more formal document that needs more explicit typesetting and powerful functionality than what R Notebooks offer, you may end up learning about LaTeXfor example, which also supports (and was actually one of the original use-cases) for `knitr`.

Regardless of how you do it, keeping a record of your code as a way of reproducing your analysis is crucial. Reproducible of research and data analysis is paramount, for many reasons including:

- So that other people can take your results and build on them.
- So that when you find mistakes you can go back and fix them easily
- So that when you want to repeat a similar analyses you don't need to start from scratch
- Accountability and transparency when reporting results.

In this unit if you produce a result and submit it as part of an assessment you will be expected to be able to reproduce that result if asked to do so, and to provide the code to reproduce it.

Three Lines of Code to Start any Script

R operates by default placing everything in a global environment (namespace)[2]. For debugging purposes it can be useful to clear this environment before running code, to prevent any potential weirdness from objects sitting in the global environment interacting with your code. This can be achieved with the following command and it is good practice to have this at the beginning of any script you write:

```
rm(list = ls())
```

In the .Rmd file note the {r, eval = FALSE }.

You will **need to change** 'FALSE' to 'TRUE' if you want to execute the commands. You may also want to explore other options such as `echo` or `results`.

It can be useful to keep your namespace tidy and for that reason it can also be good practice to have the following command to un-attach any loaded packages so they don't interact with your namespace:

[1] Check the keyboard shortcuts in RStudio Under Tools > Keyboard Shortcuts Help. These will be very useful!
[2] This can occasionally problematic, so beware.

```
if (!is.null(sessionInfo()$otherPkgs)) {
    invisible(lapply(paste0("package:", names(sessionInfo()$otherPkgs)), detach,
        character.only = TRUE, unload = TRUE))
}
```

Lastly, there is an annoying setting in R that relates to importing data which converts strings automatically to a type called factor[3], which can result in some very confusing errors for beginners.

```
options(stringsAsFactors = FALSE)
```

Note that the some of the most recent releases of R have modified their defaults meaning if you are running one of those versions this command will do nothing.

Read the Documentation

It will be crucial that you learn to read R documentation. Often, particularly towards the second half of the unit, you will be asked to use an R function and in order to be able to do so you will need to be able to read and at least partly understand the documentation. For example, if you are told to use the function `mean()`, you would type:

```
`?`(mean)
```

into your R command line (in RStudio this appears in the Console tab), and this will bring up the documentation for that function (in RStudio this documentation will appear in the Help tab). Try reading the documentation for the function `mean()` right now as an exercise and see if you can figure out what it's optional arguments would be used for. If you can't yet, come back to this exercise at the end of the lab once you've had a bit more of a play and it might make more sense.

2.2 Example of Loading, Exploring, and Visualising Data

We will use the Iris data to illustrate some basic functions in R.

Packages

In this lab, we will use the following packages. If any of these packages are missing on your computer, install them (and their dependencies) using the `install.packages()` command or the `Install` option in the `Packages` tab of RStudio or by going to Tools in the menu bar. Try installing

```
# For making pretty graphs, part of tidyverse
library(ggplot2)

# For combining ggplot2 objects in intuitive (and magical) ways
library(patchwork)
```

and the package `reshape2` installed to run one of the commands below. Similarly, you will need the package `ISLR` for answering one of the questions.

Loading Data

The `iris` data is distributed with base R, but for the purposes of this tutorial we will demonstrate how to read it in from a CSV (comma separated values) file, which is available for download from LMS. Then so long as you have set up a project as discussed in section 3.1.1 "Working Directory" and put the file `irisdata.csv` in your project folder, the following line of code will read the data into R as a `data.frame` object named `df` (into the global environment by default).

```
df = read.csv("irisdata.csv")
```

Note that the setup of the previous sections is important here. Earlier we modified the default value of `stringsAsFactors` which gets passed to `read.csv()` (see the documentation by typing `?read.csv` in

[3]Read up on `?factor` if you like

your console). Similarly, setting the working directory through a project means we don't need to worry about specifying a more detailed file path[4], instead we just need `irisdata.csv` to be in the project folder. `read.csv()` is a convenient version of the more general `read.table` documented in the same file, and the same effect could be achieved with the slightly more complex command:

```
df = read.table("irisdata.csv", header = TRUE, sep = ",")
```

data.frames in R

In the previous section we read in the iris data from a CSV file into a global environment variable named `df` of type `data.frame`. Check out the documentation for `?data.frame` as this is the standard way to store data in R.

`head()` is a useful command for taking a look at a data.frame, eg:

```
head(df)
```

What does this command tell you?

What is the sepal length of the first observation?

What are the variable values of observation 150?

Try using `tail()` in the same way as `head()`, what does it do? By reading the documentation and/ or trying it, what does providing the optional argument `n` to these functions do?

Individual and ranges of entries of a data.frame can be accessed by their indices and columns can be identified by their headers. See some examples:

```
df[3, 2]
```

```
df[2:4, 3]
```

```
df[3, 2:4]
```

```
df[3, "Sepal.Width"]
```

```
df[c(2:4, 90:92), c("Sepal.Length", "Species")]
```

Find the entries of the 10, 15 and 20 observations corresponding to petal length.

Columns of a `data.frame` can also be accessed with the `$` operator or as a list, as follows and entries from these vectors can then also be returned:

```
df$Sepal.Length[2:4]
```

```
df[["Sepal.Length"]][2:4]
```

Note the single square brackets for accessing entries in vectors/ matrices and the double square braces for accessing entries in lists.

Exploring and Visualising

`str()` and `summary()` are useful functions for exploring an object:

```
str(df)
```

```
summary(df)
```

What is the median of the petal length?

What is the maximum sepal width?

[4] If and when you do need to specify file paths it is good practice to use `file.path()` as this is one of the requirements for your code to be run cross-platform.

For visualisation, some important `base` plotting functions are `plot()`, `hist()`, and `boxplot()`. `plot()` is a generic function, similar to `summary()` and will have very different behaviour for different inputs, for example you can provide the variables you want plotted in x and y axes:

```
plot(df$Sepal.Length, df$Sepal.Width)
```

```
plot(df$Sepal.Length, df$Sepal.Width, pch = as.numeric(factor(df$Species)))
```

What is the difference between the two plots?

More detail on methods for modifying `base` plots can be found on the l `?par` page.

Another interesting feature of `plot()` is that if you simply give it an entire `data.frame` the default behaviour is to make what is called a "pairs" plot, however it needs all the variables to be numeric. So we can just give it the first four variables and differentiate the species like this:

```
plot(df[, 1:4], pch = as.numeric(factor(df$Species)))
```

What do you see in this plot?

`hist()` and `boxplot` are also useful for making histograms and boxplots, see for example try

```
hist(df$Petal.Length)
```

```
hist(df$Petal.Length, 50)
```

```
boxplot(Petal.Length ~ Species, data = df)
```

Producing boxplots for sepal length, and display a histogram for it with the default number of bins and with 20 bins. Explain the difference in the shapes that you see.

What information can you get from a boxplot that is not available in a histogram?

What information can you get from a histogram that is not available in a boxplot?

Note the `~` in the boxplot command, this is R syntax for defining formula's — see the documentation `?formula`. These will be important later as they are the most common way to define models (such as linear regression models for example) in R.

Parallel Coordinate Plots

The pairs plot represent a reasonable way to visualise multivariate data with a small number of variables, but they only show 2 variables at a time. If we want to look at 3 or more variables, we use 'parallel coordinate plots'. Try `?MASS::parcoord()`[^6]

```
MASS::parcoord(df[, 1:4])
```

```
MASS::parcoord(df[, 1:4], col = c("green", "blue", "black")[as.numeric(factor(df$Species))])
```

Note the `::` operator to access the function `parcoord()` in the package `MASS` without loading it's entire namespace (which would be done with the command `library(MASS)`).

If you leave out the expression in square brackets, how does your plot change?

Optional: tidyverse and ggplot2

`tidyverse` is a group of packages that share a design philosophy that have recently become very popular in the data science area. There are advantages and disadvantages of using the `tidyverse`, but it can certainly be advantageous to at least be aware of it as you will likely come across examples of people using it and you will need to understand their code at some point or another. There is more information on `tidyverse` on the website. Much of it originated from a core group of work surrounding someone by the name of Hadley Wickham, who has written a number of books which are excellent tutorials into the topic.

For the purposes of this unit, we will be steering away from using `tidyverse` in our examples. This is largely for the purposes of minimising dependencies and required knowledge, which is useful to avoid issues with people installing different versions of packages and getting conflicting behaviour. While using `tidyverse` can be very effective it is also sometimes unstable, in the sense that packages are updated in a way that sometimes are not backwards compatible, etc.

One package that is a part of the `tidyverse` that is particularly useful is `ggplot2`, which we will provide a short demo of here. There can be a steep learning curve to get your head around the syntax, grammar, and design philosophy of `ggplot2`, but once you do it can be very useful (as with many of the `tidyverse` packages — `dplyr` being another notable and powerful example). Lets begin by reproducing some of the plots above using `ggplot2`:

```
p1 = ggplot(data = df, aes(x = Sepal.Length, y = Sepal.Width)) + geom_point()
print(p1)
```

Here `geom_point()` tells `ggplot2` that we want a scatterplot. Note the overloading of the `+` operator, this is an example of strange `tidyverse` syntax.

Previously we saw that some of the data are precisely overlapping, one way that `ggplot2` allows for us to more easily visualise this is by making our points transparent through the `alpha` argument:

```
p2 = ggplot(data = df, aes(x = Sepal.Length, y = Sepal.Width)) + geom_point(alpha = 0.4,
    size = 2)
print(p2)
```

Notice that now some of the points are darker than others — this is because multiple transparent points have been plotted over each other, this allows these overlapping points to be seen. Often when using `alpha` to make a plotting layer transparent it can be useful to also increase the `size` for visibility.

Instead of using `geom_points` you may also like to explore `geom_jitter`, the arguments it takes and what it does.

`ggplot2` has "geom" i.e. geometry layers for histograms and boxplots:

```
p5 = ggplot(data = df, aes(x = Sepal.Length, fill = Species)) + geom_histogram(binwidth = 0.2)
print(p5)
```

Note the default for histograms with a grouping variable (such as `fill`) is to make a "stacked" histogram.

If instead you want overlapping histograms try:

```
p6 = ggplot(data = df, aes(x = Sepal.Length, fill = Species)) + geom_histogram(binwidth = 0.2,
    position = "identity", alpha = 0.3)
print(p6)
```

```
p7 = ggplot(data = df, aes(x = Species, y = Sepal.Length, colour = Species)) + geom_boxplot()
print(p7)
```

Compare the previous boxplots with those of p7. What is different?

For combining multiple plots, there is another useful package: `patchwork`, which can do some surprising things:

```
print(p6/(p7 + coord_flip()))
```

Explain what you see.

The other common way to make multiple plots in one with `ggplot2` is with `facets` but this will first require re-arranging our `data.frame` into "melted" form. For this we'll use a function from yet another package, `reshape2::melt()`. Note that it can be useful to have an `id` identifying rows in the original `data.frame`, and this is what we do here:

```
df$id = seq_len(nrow(df))
df.m = reshape2::melt(df, id.vars = c("id", "Species"))
```

Lets take a quick look at what this did:

```
head(df.m)
```

```
str(df.m)
```

```
nrow(df.m)
```

Why does `df.m` have 600 rows? Is this what you would have expected?

This melted `data.frame`, also called "long-form", can be used for faceting. For example histograms or boxplots for each variable:

```
p8 = ggplot(data = df.m, aes(x = value, fill = Species)) + geom_histogram(binwidth = 0.2,
    position = "identity", alpha = 0.3) + facet_grid(rows = vars(variable))
print(p8)
```

```
p9 = ggplot(data = df.m, aes(x = Species, y = value, colour = Species)) + geom_boxplot() +
    geom_jitter(width = 0.2, height = 0.1, alpha = 0.6, size = 1.4)
print(p9 + coord_flip() + facet_grid(rows = vars(variable)))
```

Finally, we construct a parallel coordinate plot quite easily which makes use of the species information.

```
p10 = ggplot(data = df.m, aes(x = variable, y = value, colour = Species, group = id)) +
    geom_line(alpha = 0.3)
print(p10)
```

Discuss what this plot shows. How does it differ from a similar display using the plot command?

Attach boxplots to our parallel coordinate plot:

```
print(p10/(p9 + facet_grid(cols = vars(variable)) + theme(axis.text.x = element_text(angle = 45,
    hjust = 1))))
```

3 Additional Resources for Reference

- R for Data Science (http://r4ds.had.co.nz/) provides an excellent introduction to data science using `tidyverse`.
- Advanced R (http://adv-r.hadley.nz/) and R Packages (http://r-pkgs.org/) are also available online and are excellent resources for more advanced R programming.
- Manuals: http://cran.ms.unimelb.edu.au/manuals.html
- Contributed Docs: http://cran.ms.unimelb.edu.au/other-docs.html

You may also find the following useful:

https://rstudio.com/resources/cheatsheets/

3.1 Resources on R Notebooks and Rmarkdown

- https://rmarkdown.rstudio.com/r_notebooks
- https://blog.rstudio.com/2016/10/05/r-notebooks/
- https://rviews.rstudio.com/2017/03/15/why-i-love-r-notebooks/
- https://bookdown.org/yihui/rmarkdown/notebook.html
- https://minimaxir.com/2017/06/r-notebooks/

- http://uc-r.github.io/r_notebook

4 Fitting Linear models.

For the data set power.txt fit a linear regression model to Cost against Units. Here Cost is the cost of production and Units is the units of power produced. Perform model diagnostics and comment on the goodness of your model. Produce as part of your model diagnostics a scatter plot of the data superimpose and the fitted line of regression.