**Introduction to ‘R’ statistical software for   
data analysis**

## George Savva

## Quadram Institute Bioscience

## James Brown

## John Innes Centre

# Preface

This course will introduce R statistical software, some basics of how it works, guides to performing common operations, where to go for further support and some tips for good practice.

The aim is to become familiar with the R/RStudio environment and some common functions so that you can learn the specific functions that you need on your own or with further training. Specifically this course will bring you up to the level you need to attend the externally run statistics courses provided at by NBI.

Note: there is no discussion of the statistical validity of any of the procedures described here, simply how to execute them using R.

The data you need to complete the training exercises is in the accompanying file (Introstat.xlsx). All of the commands for the worked examples and the exercises is in the file TutorialScript.R. The commands for the worked examples are also typed out here.

# Learning objectives

Specific tasks:

Day 1:

* Making a project
* Loading packages
* Loading a dataset from an Excel file
* Exploring your data; calculating descriptive statistics
* Hypothesis tests

Day 2:

* Estimating and diagnosing a regression model
* *Making graphics using base graphics and the ggplot2 package – not in the current handout*

How to use R

* R and RStudio
* RStudio ‘projects’ and a good workflow
* The command line
* Using scripts for reproducibility
* Getting help

How R works

* Objects and functions
* Data frames and vectors
* Types of data: numerics, factors, and strings
* Representing and handling ‘missing’ values
* The importance of ‘tidy’ data

# Introduction

## What are R and RStudio?

R is an open source statistics package, initially developed during the 1990s, and that has now become the world’s most widely used and comprehensive statistical software. R calls itself a ‘programming language *and* environment for statistic computing’.

That is, ‘R’ refers both to the software itself and the programming language that you use to interact with it.

RStudio is a *integrated development environment* (IDE) for R that makes working R much easier. Most R users use RStudio and I recommend using RStudio instead of just using R for new users.

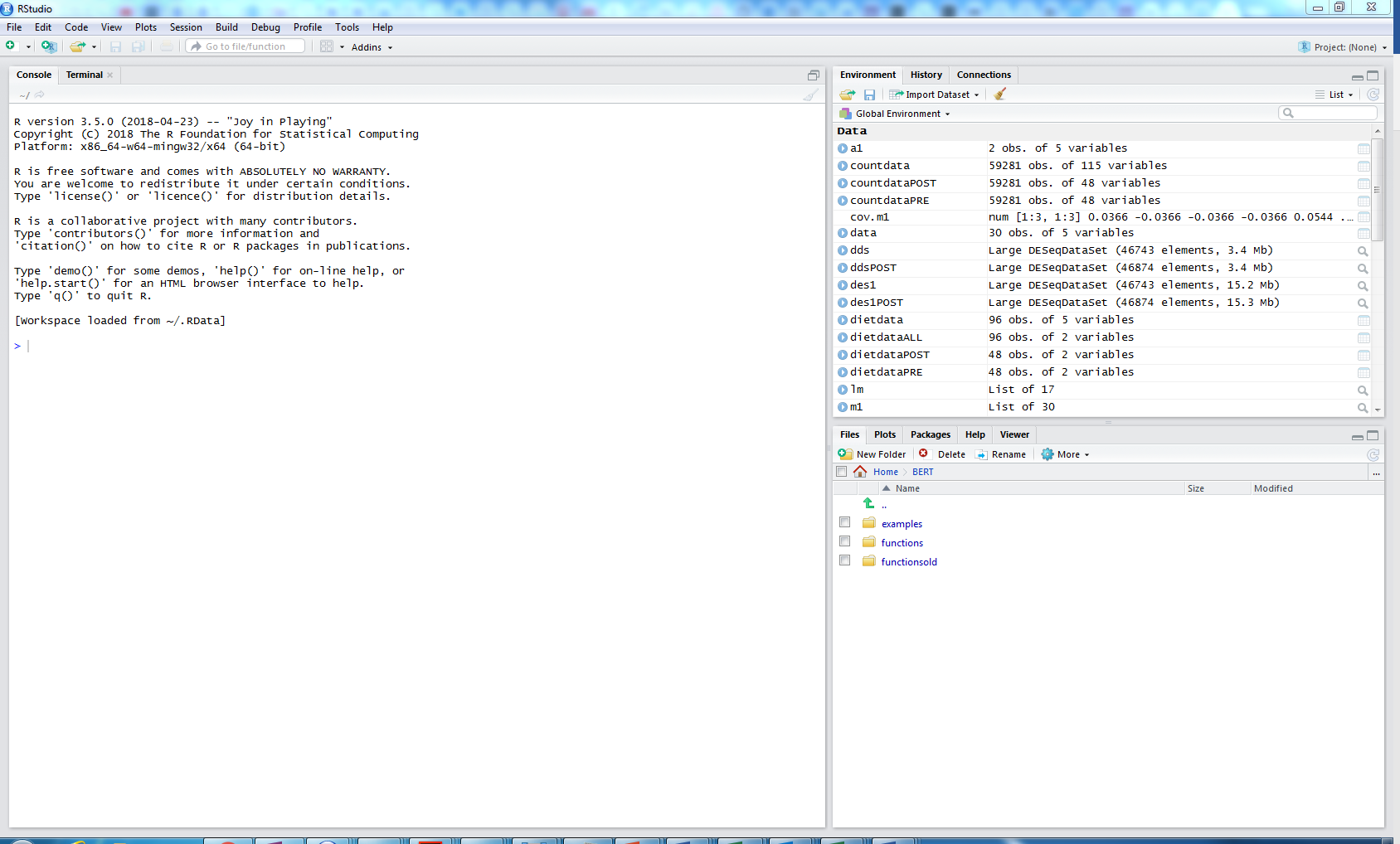
The great strength of R is in its contributed packages, these are community written add-ons that provide functions to perform almost any statistical, programming, or data-related task.

## Getting R and RStudio

Download and install the latest version of R from <https://cran.r-project.org/>

Then download and install RStudio from <https://www.rstudio.com/>

Start RStudio. It will detect your installation of R, and you should see a screen like this:



On the left is the console window, where you type commands and see output. The windows on the right hold various useful tabs, here the top pane is showing the data I happen to currently have loaded (yours will be empty) and part of my filesystem at the bottom. These right-hand windows can also show graphs, help files, and your command history.

First task: check RStudio is working. Click in the console window and type:

1+2

Press return on your keyboard. You should see:

> 1+2

[1] 3

This is the basic way in which R works. We enter commands at the command prompt (or via a script), and we get the output in the console window.

To chain together multiple commands, and to keep our work safe we will write scripts, which are just sequences of commands that are executed in order.

Try a few other mathematical functions at the R console.

# Using ‘projects’

Before we go any further, we are going to start an RStudio ‘project’ to organise our work. Using projects helps us to keep all of the data and analysis for a particular piece of work in the same place.

Click on ‘New -> New project’ in the toolbar. Click ‘Start a new project in a brand new working directory’. Then click ‘new project’ on the next screen.

Now you can choose where to create the new directory for your R project, and what to call it. Make a project called ‘Rtraining’ or something like that, somewhere in your personal filestore.

Now when you return to the main RStudio window you are working within your project. Notice that the working directory has switched to the new directory that you created. Keep the R scripts and the data associated with this training course in that directory, so that you can access them easily.

See:

<https://r4ds.had.co.nz/workflow-projects.html#rstudio-projects>

<https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>

for more information on using projects

# Worked Example 1: Simple object assignments.

**Functions**

Everything in R is done by executing ‘functions’. When you typed 1+2 at the console above you were executing the + function, with 1 and 2 as its arguments, and the result was printed in the console window.

**Objects**

Instead of directly displaying the *value* of the function (‘value’ is what R calls the result of a function), you can give it a name and store it for later use:

x = 1+2

The = in this case represents ‘assignment’. The line above says:

‘evaluate 1+2, and store the result in an object called x’ or ‘assign the value of 1+2 to x’.

You might find it easier to understand:

x <- 1+2

This does exactly the same thing; some R users use <- instead of = for assignment (for historical reasons), so both forms will come up when you’re looking at help or other people’s code. I try to use = where I can, but old habits die hard so I am likely to use both interchangeably.

So now you have an *object* called ‘x’ that holds the number 3. You can ask R to display the value of ‘x’ by just entering x (just entering the name of an object prints that object):

> x

[1] 3

Or do something else with x

> x\*2

[1] 6

Note your new object and its value should now have appeared in the ‘environment window’ in RStudio.

To see the class of an object, use the class function.

> class(x)

Objects of different classes store different kinds of information.

## Exercise “More functions, and assigning values to variables”

1. make a new object called y which has the value of x+3. Then display y.
2. Now change the value of x (eg using x <- 6 ). Does the value of y change?
3. Objects can hold text strings instead of numbers. Try:

> myname <- “George” (or whatever your name is).

> myname

What is the class of the ‘myname’ object?

1. **(difficult)** Look up the function to turn a text string into upper case (an internet search will help you). Use this function to make a new object which has the same text as ‘myname’ but in upper case.

# Classes and types

We saw two object of two different ‘classes’ in the previous exercise. These classes were ‘numeric’ and ‘character’. The class of an object defines what kind of data it can hold, and how other functions act on it.

There are four basic classes that you will commonly use and should be aware of. These are:

* ‘numeric’ – For keeping numerical data
* ‘logical’ – can only take the values (TRUE or FALSE)
* ‘character’ – for strings of text
* ‘factor’ – for labelled categorical variables (ordered or unordered)

In addition, every other kind of object you will use or generate has its own class, which defined the sort of information it contains and how other functions handle it. Later in this tutorial we will see objects of class:

* ‘data.frame’ – storing entire datasets
* ‘lm’ – stores the results of a linear model

**Character strings**

Character strings represent text rather than numbers. You might use strings as labels for categories in a dataset, to identify columns in a dataset, to make your outputs more readable, or you might find that part of your data has been entered as a string, for example patient identifiers or gene names in a database.

Strings are identified in R (and in most other programming languages) by enclosing them in quotes. Single quotes and double quotes can be used (and are treated almost identically), but double quotes are preferred. For example try:

> print("Hello")

[1] "Hello"

> print('Hello')

[1] "Hello"

> print(Hello)

Error in print(Hello) : object 'Hello' not found

A common mistake in R is to forget to enclose strings in quotes. In which case R tries to interpret your input as an object name, leading to an error message if that name doesn’t exist.

*Note: don’t try to copy-and-paste text with quotes from a Word file into R. Word changes standard quotes to forward or backward quotes to make them look nice, but this will not work in R.*

# Common mathematical functions

Most mathematical functions in R work on numeric objects as you would expect. Functions you might commonly need include:

The operators +, -, \*, /

Raising to a power:

> 3^2

[1] 9

> sqrt(16)

[1] 4

Taking log and antilog (be careful of the base, this is a common source of confusion)

> a = log(10)

> a

[1] 2.302585

> log(10, base=10)

[1] 1

> exp(a)

[1] 10

# Worked example 2 - Vectors

Objects can store more complex information than just a simple number.

Type the following:

> a <- c(3,4,5)

This assigns the *vector* (3,4,5) to a.

In other words this command says ‘create a vector from the numbers 3,4 and 5, and call this vector object ‘a’.

c() is a common function in R which simply ‘combines’ its arguments into a vector.

The *arguments* of a function are the values you pass to the function. In R you pass arguments in brackets after the function name, and they are separated by commas.

You can display a just as you displayed ‘x’ previously

> a

[1] 3 4 5

You can run more interesting functions on ‘a’. Make sure you understand what each of the functions below does *and* *why*:

> a + 1

[1] 4 5 6

> a \* 2

[1] 6 8 10

> a>3 # what class does the output from this function have?

[1] FALSE TRUE TRUE

> a + x

[1] 6 7 8

> mean(a)

[1] 4

> sum(a)

[1] 12

> summary(a)

Min. 1st Qu. Median Mean 3rd Qu. Max.

3.0 3.5 4.0 4.0 4.5 5.0

> plot(a) # where did your plot appear? (R ignores anything after a ‘#’)

You can ‘extract’ elements of a vector using the [ operator as follows.

> a[1]

[1] 3

> a[3]

[1] 5

What happens if you try a[4]. Why?

As well as specifying the particular elements you want, you can also select elements that meet a certain condition. Suppose we wanted the elements of a that were bigger than 3. Then we would write

> a[a>3]

This literally means “return the values of ‘a’ at the positions where ‘a’ is greater than 3”.

While this might seem like an odd thing to want to do now, it becomes important later, because this kind of ‘subsetting’ is one way to extract different parts of a dataset for analysis.

Individual elements of vectors can also be altered using the extract operator. For example:

> a[2] <- 10

> a

[1] 3 10 5

## Missing elements in vectors

Often your data will include missing values. R uses ‘NA’ to represent missing values. For example the following creates a vector with a missing value in the fourth position:

> myvector <- c(10,21,32,NA,54)

Try some other functions with myvector to see what impact the missing data point has.

> class(myvector)

> plot(myvector)

> myvector>20

> mean(myvector) # what happens here? Why? Can you fix it?

> is.na(myvector) # what does this do?

> sum(is.na(myvector)) # can you explain what this does?

## Exercise “Get help!”

I can never remember the name of the function to calculate the standard deviation of a numeric vector. Look this up online (eg a google search for ‘standard deviation in R’), and use it to find the standard deviation of a.

Make another numeric vector called ‘b’ (or anything you like; object names don’t have to be single letters). Draw a histogram of that vector (look it up!).

**Difficult!**

Make a vector which is a sample of 100 values from a standard normal distribution. (The function rnorm will help you with this.). Now plot a histogram only of the values that are greater than 0.

# Worked example 3. Loading data into R from Excel part 1: installing a package

R is *not* used for data entry or storage; you are likely to have data stored as an Excel file, a csv file, in a database, or in some other format generated by a device.

So before we can do any analysis we need to import data. We do this in code like everything else in R.

Base R (‘base’ R is R as it comes when you first download it) cannot read data from Excel files. But there are several add-on ‘packages’ that can read Excel files (there are packages to read data from most common formats).

*Packages* are collections of functions and datasets related to a specific task. They are most often created by R users for their own use and then shared with the community through the Comprehensive R Archive Network (CRAN), which makes it very easy to find and install them.

Here we will import a dataset from Excel using the ‘readxl’ package.

First, readxl needs to be installed. You can do this through the tools menu or by typing:

install.package("readxl") # notice the quotes here

The first time you try to install a package, R will ask you where you would like to download packages from. Select any UK CRAN mirror.

Now the package is installed. We could access its functions by using the package name and the function name, but its easier to first type:

> library(readxl) # no quotes this time

‘library’ makes all of the functions in a package available for use without having to reference the package each time.

Now we have the function read\_excel() available, which reads data from an excel file.

Before we dive in and use it, we need to make sure our data is in a sensible place, and we understand how to use the function. First, save the example data for this tutorial into a ‘data’ subdirectory of your project.

## Exercise 3: “Read the help!”

We are nearly ready to import some data. But before using a new function its always good to read its documentation.

R and R packages are not as self-explanatory as other software, and so you should expect to spend a fair amount of time, particularly as you are learning R, reading documentation, vignettes, blogs, etc on what R can do, which packages exist, and how to use them.

read\_excel() has a few different options so first we should look at the help file:

> help(read\_excel) # where does the helpfile appear?

Most R help files are structured in the same way. They have a ‘Description’ section (what does the function do), a ‘Usage’ section (what is the syntax), an ‘Arguments’ section (detail of what all the options mean), a ‘Value’ section (what do I get when I run this) and some Examples. The examples are usually very helpful.

Notice that read\_excel() can extract data from different sheets and ranges of an Excel workbook, can use or ignore column names, and allows you to specify the type of data (numeric, dates, text etc) if you want to.

Many R packages also have vignettes or websites including simpler guides to their use in specific cases. readxl has a website that you might find helpful:

<https://readxl.tidyverse.org/>

Referring to R and R packages

It is important to cite R and R packages you use correctly, particularly where the package is essential to make your analysis reproducible. To find out how to cite a package use the ‘citation’ function.

How should you cite the ‘readxl’ package?

# Worked example 4: Loading data into R from Excel part 2: Data frames

Now we’ll load the data. We want to use the ‘tree species’ data from the introstat.xlsx spreadsheet. Open the spreadsheet in Excel and find this sheet. The data we want is in the sheet called P1-TreeSpeciesData.

From the read\_excel() help file we can deduce the syntax to load this data into R:

TreeData <- read\_excel(path="introstat.xlsx", sheet="P1-TreeSpeciesData",.name\_repair="universal")

(This assumes that the file ‘introstat.xlsx’ is in the current working directory. The current working directory is shown just above the R console window)

This line calls the read\_excel() function, with the arguments ‘path’, ‘sheet’, and ‘.name\_repair’ set. The other arguments will be set to their default values, which you can see from the help file. We could have set the range of the data in the spreadsheet, but read\_excel() can figure it out automatically most of the time; by default it uses all the data in the sheet which in this case is what we want.

We set .name\_repair to make sure that the names of the variables are valid R names that we can use in analysis commands. Note in the Excel sheet some names have spaces or other punctuation marks. These are not valid in variable names, so .name\_repair should be set to ‘universal’ to ensure column names are unique and valid.

Now we have a ‘data frame’ object called TreeData, which includes the data from the Excel sheet ready to process and analyse. It’s a good idea to check that the data has been extracted as you expected, and we can inspect it in various ways. Try:

class(TreeData)

What kind of an object is ‘TreeData’?

dim(TreeData)

This shows you the dimension of your data frame (how many rows and columns it has).

View(TreeData)

This opens a viewing window with your data in the top panel of RStudio

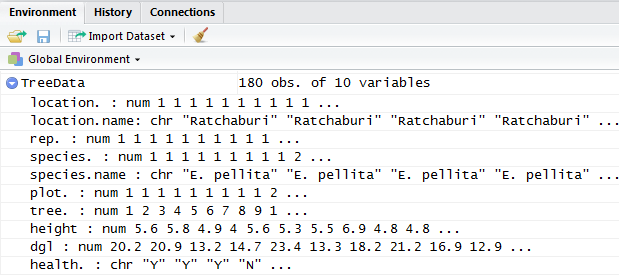
head(TreeData)

shows you the first few rows of the data file in the console window.

summary(TreeData)

shows the name and summary information for each variable, as the ‘summary’ function did in worked example 2. (You can use ‘summary’ on most kinds of R objects, it usually shows you something sensible).

Finally, clicking on the arrow next to the object name in the Environment window will show you details of each column that the data frame includes.



Note that the *class* for each variable is listed by each column name in the Environment window above.

This is the same output that you would get from the ‘str’ function.

# Worked Example 5: Accessing variables, and generating summary statistics

A data frame is essentially a set of vectors, all of the same length, that have been stuck together into a rectangle such that each row describes unit of one observation, and each column includes a particular attribute of that observation.

This ‘tidy’ format is how we must store *all* our datasets for analysis. If you have used SPSS, Stata or a relational database you will be familiar with this way of organising data. See the reference below on ‘tidy data’[[1]](#footnote-1) for how to organise your data into one or more tidy data frames, you may not immediately think this is possible for your dataset, but it is!

Once we have our data tidied and loaded into a data frame, it is much easier to specify how we want to work with each variable.

We can access the individual vectors using the $ accessor operator:

TreeData$height

So we can use any of the functions for vectors to summarise this variable. Try:

mean(TreeData$height)

boxplot(TreeData$height)

This can be a useful way to summarise individual variables, but it’s often more helpful to be able to summarise according to the levels of another factor.

Let’s get the heights by species. We can do this using the ‘aggregate’ function. But first we need to fix the names of the variables in the data frame. R *will* import data frame columns

> aggregate( data = TreeData, height ~ species.name, FUN=mean)

Group.1 x

1 A. aulococarpa 6.188889

2 A. polystachya 3.497222

3 C. cunninghamiana 8.138889

4 E. pellita 5.283333

5 M. viridiflora 11.280556

Look at the help file for aggregate() to understand how the arguments are specified.

For a categorical variable, a summary of frequency counts might be the most appropriate. We can get this with the table() function:

> table(TreeData$species.name)

A. aulococarpa A. polystachya C. cunninghamiana

36 36 36

E. pellita M. viridiflora

36 36

The ‘table’ function can also generate cross-tabs:

> table(TreeData$species.name, TreeData$health)

N Y

A. aulococarpa 7 29

A. polystachya 17 19

C. cunninghamiana 9 27

E. pellita 4 32

M. viridiflora 2 34

Tables of numbers are useful, but it might be more helpful to see the proportion of healthy trees by species. To get this we can pass the table we just made into the prop.table() function:

> table1 <- table(TreeData$species.name, TreeData$health)

> prop.table(table1)

N Y

A. aulococarpa 0.03888889 0.16111111

A. polystachya 0.09444444 0.10555556

C. cunninghamiana 0.05000000 0.15000000

E. pellita 0.02222222 0.17777778

M. viridiflora 0.01111111 0.18888889

The table above has calculated the ‘cell proportions’. If we want the row percentages we need to set the ‘margin’ option appropriately. And we could round this to 2 d.p. by passing the resulting proportion table into the round() function:

> proptab1 <- prop.table(table1, margin=1)

> round(proptab1, digits=2)

N Y

A. aulococarpa 0.19 0.81

A. polystachya 0.47 0.53

C. cunninghamiana 0.25 0.75

E. pellita 0.11 0.89

M. viridiflora 0.06 0.94

Remember ‘a[1]’ from earlier that returned the first element of the vector ‘a’? You can do a similar thing with two-dimensional structures like tables and data frames to return individual elements, rows or columns. For example:

> proptab1[1,1] # get the element on the first row and first column

[1] 0.1944444

> proptab1[,2] # this returns the second column only.

A. aulococarpa A. polystachya C. cunninghamiana E. pellita M. viridiflora

0.8055556 0.5277778 0.7500000 0.8888889 0.9444444

> proptab1[4,] # return the fourth row only

N Y

0.1111111 0.8888889

## Exercises

## 

What would be an easy way to convert these proportions to percentages?

Can you use prop.table() to generate column percentages instead of row percentages?

Further reading on the ‘extract’ operators: <https://github.com/lgreski/datasciencectacontent/blob/master/markdown/rprog-extractOperator.md>

# worked example 5b – the table1 package

The functions described above do the job of tabulating and summarising data, but are not particularly neat ways to generate summary statistics from datasets.

Surprisingly, ‘Base’ R does not really have a good set of functions to make descriptive tables, but there are much better functions in add-on packages for creating and displaying descriptive statistics.

The ‘table1’ package is a great example; it will describe a dataset it a way that is suitable for a ‘table 1’ in an academic paper.

Install the table1 package from CRAN as you did for readxl earlier.

> install.packages(“table1”)

Then either:

> table1::table1( data = TreeData, ~ health | species.name )

Or

> library(table1)

> table1( data = TreeData, ~ health | species.name )

This table is good to be directly pasted into a report. See the table1 package ‘vignette’ to learn more about how this works and how to change the output if you need to.

<https://cran.r-project.org/web/packages/table1/vignettes/table1-examples.html>

# Worked example 6: Making basic plots; formulas

We’ve already seen the box plot in the previous section. We can ask for this in a slightly different way, and stratify it by species

> boxplot(height ~ species.name, data=TreeData)

Notice here that we are no longer using the $ operator to access columns of the data frame. Instead we just need to specify the variable names, and then supply the argument ‘data=TreeData’ to the function. This argument tells boxplot which data frame to look in for the objects that we want to work with.

What happens if we forget to specify the dataset?

**Formulas**

Next notice how we’ve specified the relationship. Here, ‘height ~ species.name’ is an example of a *formula*. Formulas are objects that R uses to describe relationships between variables. This formula says ‘height depends on species.name’. boxplot() interprets this to mean that we want a box plot for height stratified by species.name, which seems reasonable!

The ‘table1’ function above also used formulas, but in a slightly different way.

We’ll make some improvements to the box plot we have created on day 2 when we discuss graphics.

# Worked Example 7: Simple hypothesis tests and estimation

Is height associated on tree health? We can generate a simple box plot as previously to examine this this time with some labels):

> boxplot( height ~ health., data=TreeData , xlab="Height", ylab="Health")

It looks like the two variables might be related, but we would like to perform a t-test to see if the difference is statistically significant, and to get an estimate (with confidence interval) for the difference.

The syntax is almost exactly the same as that used to generate the box plot. We use the same formula:

> t.test( height ~ health. , data=TreeData)

Welch Two Sample t-test

data: height by health.

t = -3.7752, df = 66.076, p-value = 0.0003448

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-2.8742163 -0.8857401

sample estimates:

mean in group N mean in group Y

5.405128 7.285106

Note that t.test has returned ‘Welch Two Sample t-test’. This is not the traditional t-test with equal variances assumes, by default R runs a t-test that does *not* assume equal variances across groups. This is probably better, but if you want the traditional test you’ll have to look at the help file to get it.

> t.test( height ~ health. , data=TreeData, var.equal=TRUE)

Two Sample t-test

data: height by health.

t = -3.5629, df = 178, p-value = 0.0004708

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-2.9212420 -0.8387143

sample estimates:

mean in group N mean in group Y

5.405128 7.285106

In general it’s useful to read the documentation for any new function that you are using, because the default R behaviour might not be what you expect.

*Note: Note from the t.test help file: the default options are shown first, so here var.equal=FALSE is the default which you had to override with var.equal=TRUE in the your function call.*

Exercise: How would you run a paired t-test? (No need to do it as we don’t have suitable data, but look at the help file.

# Worked example 8: Using a script

This is all starting to get a bit complicated. Rather than doing each of these steps one at a time it is much better to keep our code together in an R script.

That way, when we come back to it in six months or something changes in our data, or we want to tweak something in our analysis, we can remember what we did. An R script is just a text file with a series of R commands in, that load your data and conduct your analysis when they are run in the right order.

RStudio has good facilities for writing R scripts. To create a new one click on the  button in the top left corner or go to File -> New File -> R Script in the menus. Then you can enter your functions for loading libraries, getting data, data management descriptive analysis and plots etc.

Let’s write a short script to generate a new variable in our data frame. Suppose we want to log transform the height variable, and create a new variable called logHeight, then we want to plot logHeight against dgl. We can do this by adding the following lines to our script:

TreeData$logHeight <- log(TreeData$height)

plot( logHeight ~ dgl, data = TreeData)

Although TreeData$logHeight didn’t exist, it will be created by this command.

To run a whole script at once, press Ctrl+Shift+Enter.

To run a single command, place the cursor on the line you want to run and press Ctrl+Enter

To run one or more lines in a block, highlight the area you want to run and press Ctrl+Enter

**Best practice for working with R scripts:**

If you have your raw data saved, and you keep your scripts, then **you don’t need to save your results** **or any of your generated variables**. So long as the orginal data doesn’t change, running the script will completely reproduce all of your analysis and output. This is a better way of working than trying to save your environment with all of your results and tables in.

I have created a script including all the analyses from this tutorial, in TutorialScript.R. Load this and have a look around. Notice my comments to remind myself why I did things, this might be helpful when I next come to revise the analysis!

Day 2

# Worked example 9: Estimating a simple regression model

In the previous section we saw that dgl could be used to predict tree height. We can estimate a linear model for this relationship using the lm() command:

lm1 <- lm( height ~ dgl, data=TreeData )

Note this command has the same was of specifying a formula and reference to the data as the boxplot and t.test commands above.

This command creates a ‘linear model’ object and names it ‘lm1’. The linear model object contains the estimated linear regression model with ‘height’ as the outcome and ‘dgl’ as the predictor.

There are now lots of different functions you can use to extract elements of this model.

To see a description of all the information contained within the object, use:

> attributes(lm1)

$names

[1] "coefficients" "residuals" "effects" "rank" "fitted.values"

[6] "assign" "qr" "df.residual" "contrasts" "xlevels"

[11] "call" "terms" "model"

$class

[1] "lm"

But generally you won’t need to use these attributes directly. Different functions can be used to extract the information in readable and usable formats:

The summary command that we used earlier can also ‘summarise’ a linear model. It produces the standard linear regression output that you would expect from any statistics package.

> summary(lm1)

Call:

lm(formula = height ~ dgl, data = TreeData)

Residuals:

Min 1Q Median 3Q Max

-5.9033 -1.3254 -0.2229 1.4960 8.5181

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 2.03057 0.48493 4.187 4.43e-05 \*\*\*

dgl 0.34111 0.03182 10.721 < 2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.353 on 178 degrees of freedom

Multiple R-squared: 0.3924, Adjusted R-squared: 0.389

F-statistic: 114.9 on 1 and 178 DF, p-value: < 2.2e-16

To get the corresponding analysis of variance table, use the anova() function:

> anova(lm1) # Note by default R returns ‘type III’ ANOVA.

Analysis of Variance Table

Response: height

Df Sum Sq Mean Sq F value Pr(>F)

dgl 1 636.44 636.44 114.95 < 2.2e-16 \*\*\*

Residuals 178 985.55 5.54

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

To plot the data with the regression model added, use the ‘plot’ function then abline() to add the line:

> plot( height ~ dgl, data = TreeData)

> abline(lm1, col="red")

# Worked Example 10: Regression Predictions and diagnostics, categorical variables (factors)

After estimating a regression model we should inspect it to check its underlying assumptions are met. R will show you some useful diagnostic plots if you run:

> plot(lm1)

If we want to directly check the predicted values or residuals from the model we can use the predict and resid functions

> predict(lm1)

> resid(lm1)

These functions generate vectors with each element corresponding to a row in the original data frame. So element [1] of the vector created by predict() is the predicted (fitted) value of height for the first row in TreeData under our model.

**A model with a categorical predictor**

Now suppose we wanted to model the relationship between height and species. Species is a categorical variable, rather than a continuous numeric as dgl was, but the way in which we estimate the model in R is the same. That is, we do this by running:

> lm2 <- lm( height ~ species.name, data=TreeData )

> summary(lm2)

Call:

lm(formula = height ~ species.name, data = TreeData)

Residuals:

Min 1Q Median 3Q Max

-5.0806 -0.9076 0.1139 0.9028 3.5194

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 6.1889 0.2346 26.375 < 2e-16 \*\*\*

species.nameA. polystachya -2.6917 0.3318 -8.111 8.46e-14 \*\*\*

species.nameC. cunninghamiana 1.9500 0.3318 5.876 2.07e-08 \*\*\*

species.nameE. pellita -0.9056 0.3318 -2.729 0.007 \*\*

species.nameM. viridiflora 5.0917 0.3318 15.344 < 2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.408 on 175 degrees of freedom

Multiple R-squared: 0.7861, Adjusted R-squared: 0.7813

F-statistic: 160.8 on 4 and 175 DF, p-value: < 2.2e-16

Notice that we did not need to create any dummy variables, R did this automatically, and chose the level with the lowest value (alphabetically) to be the reference value. Because the species.name variable was a ‘character’ variable, R understood that we wanted it to be analysed in categories (just as it did when we estimated the box plots).

If we had used the numeric species number variable, then R would not have automatically realised this, and we would get a non-sensical regression.

> lm3 <- lm( height ~ species., data=TreeData )

> summary(lm3)

Call:

lm(formula = height ~ species., data = TreeData)

Residuals:

Min 1Q Median 3Q Max

-6.1778 -1.4722 0.5222 1.5167 5.1333

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 2.6944 0.3978 6.773 1.77e-10 \*\*\*

species. 1.3944 0.1199 11.625 < 2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.276 on 178 degrees of freedom

Multiple R-squared: 0.4316, Adjusted R-squared: 0.4284

F-statistic: 135.1 on 1 and 178 DF, p-value: < 2.2e-16

To make R treat a numeric variable as categorical it needs to be turned into a ‘factor’ variable. A factor is a numeric variable where the numbers just represent groups, they lose their meaning as numbers. Rather than overwriting the species numeric variable, we’ll create a new one for the factor:

First we’ll check the class of TreeData$species and ask for a summary:

> class(TreeData$species.)

[1] "numeric"

> summary(TreeData$species.)

Min. 1st Qu. Median Mean 3rd Qu. Max.

1 2 3 3 4 5

Now make a new factor variable and add it to the dataframe:

> TreeData$speciesF <- factor(TreeData$species.)

Now check the class and the summary again. R realises that a factor should be summarised by a tabulation not mean, median etc.

> class(TreeData$speciesF)

[1] "factor"

> summary(TreeData$speciesF)

1 2 3 4 5

36 36 36 36 36

Finally the regression model should look correct again, that is with one row per category (compared to a reference group).

> lm4 <- lm( height ~ speciesF, data=TreeData )

> summary(lm4)

## Exercise

1. Are the regression coefficients the same as those estimated for model 2? If not, why not?
2. The function ‘tab\_model’ in the sjplot package provides nicely laid out summaries of regression models. Give it a try.

*Day 2 – Graphics will be sent separately*

# Answers and comment on exercises

1a.: Type y = x+3 or y <- x+3

1b.: Subsequently changing x *doesn’t* change the value of y. When y was created it used only the *value* of x at that time, and changing x afterwards doesn’t change y.

2: No clues with this one, but the final answer should be 1.

5: You could multiply the table of proportions by 100, either before or after you round it to 2 d.p. Compare:

> 100\*round(proptab1, digits=2)

> round(100\*proptab1, digits=2)

If you look up the help for prop.table(), you’ll notice the second argument controls which margin the proportions are calculated over. ‘1’ is the rows, ‘2’ is the columns. If you left out margins altogether, you’d get the cell proportions.

6: This was difficult. R’s barplot function doesn’t have the nice formula interface that plot and boxplot have. You need to calculate the proportions first, then pass them to barplot. You already made the proprtions using prop.table(), so you could pass the second column of proptab1 into the barplot function:

barplot(proptab1[,2], horiz=TRUE)

You could also do this with another package like ggplot2, which does have a function to directly create the bar plot, we’ll see this in Day 2.

8: You would add a ‘paired=TRUE’ argument (making sure that the data was set up correctly first).

10: No, the regression coefficients aren’t the same. The model is the same, but the reference group has changed. When we entered the names of the species as a character vector, R selected “A. aulococarpa” as the reference (alphabetically the first). When we created the factor from species., R assigned the lowest value to E. pellita, as this has the smallest numeric value (1) assigned to it, and this became the reference group for the second regression. We could use the relevel() function to change the reference group of a factor if we wanted the outputs to match exactly.

1. See Wickham (2014) for more on ‘tidy data’ <https://vita.had.co.nz/papers/tidy-data.html>. [↑](#footnote-ref-1)