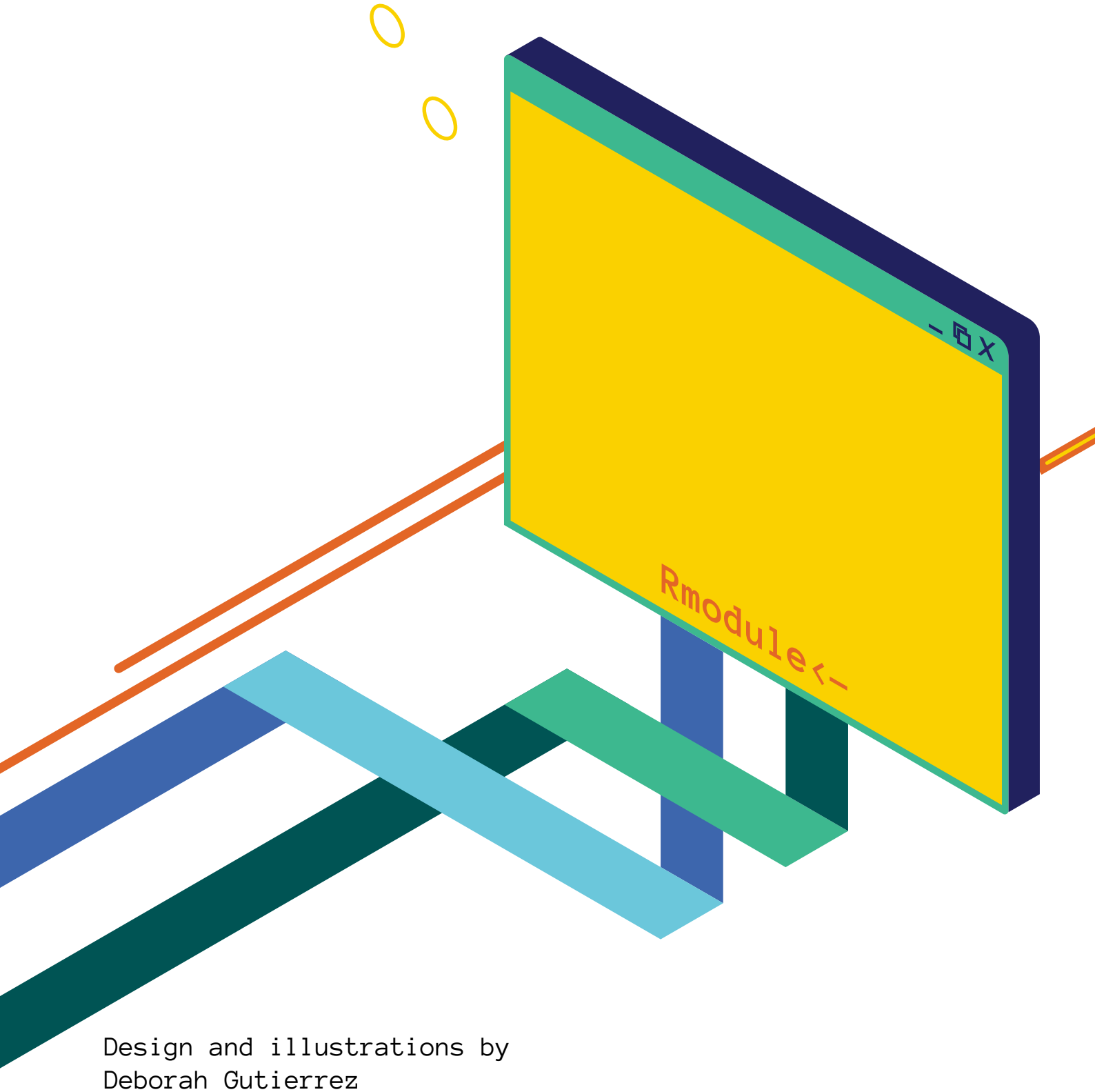
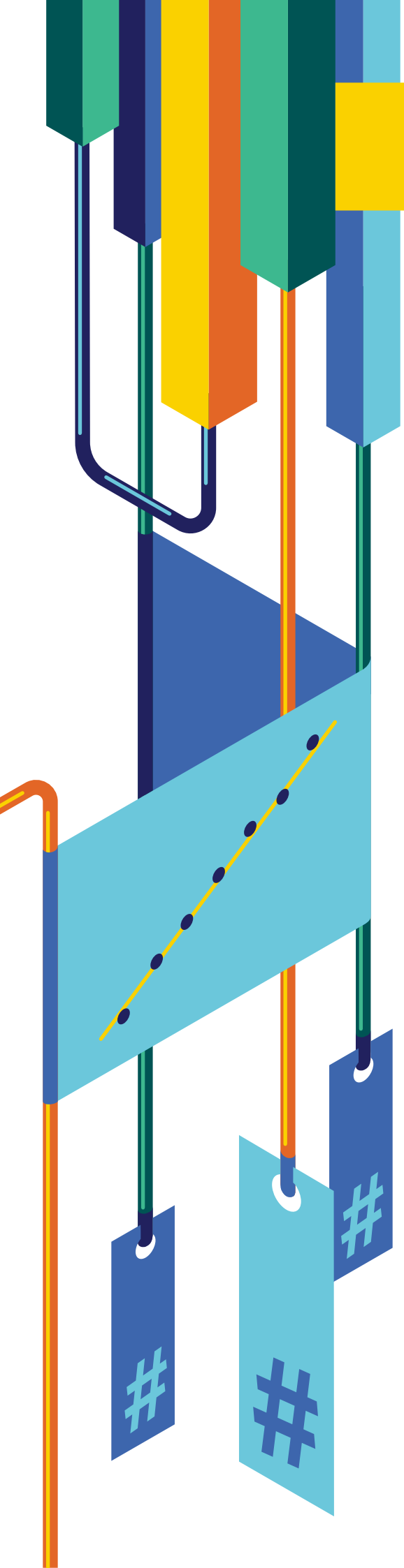
Dr. Lisa I. Pilkington



Design and illustrations by
Deborah Gutierrez



Preface

Welcome to R Module, a guide designed to get you using R for your analysis in no time at all. This module does not expect that you have had any previous experience with R or programming/coding – we will show and teach you all you need to know. Alternatively, if you have experience in R, this should fill any knowledge gaps that you might have and broaden your skill set.

R is one of the most popular and well-used statistical softwares available. It is used globally and has extraordinary scope in its functionality. It is for this reason that R is increasingly becoming the software to use in all aspects of statistical analysis. To know R will put you in a great position for any future direction you choose – this module, and R, is for everyone.

R Module has both written notes and accompanying videos to suit all types of learning and be complementary to all learning preferences. The videos have been created to show you R in action, while the notes, especially the example code, are comprehensive and detailed and available for you to quickly and conveniently refer to whenever you need. A range of asides and extra tips are also included to highlight certain points and offer extra information if you were interested in knowing more.

In addition to the analysis sections, R Module brings all the aspects you are taught throughout to guide you in what analysis techniques you should carry out (and in what order) for your statistical analysis process. The analysis depends on what data you have and what you want to find out. Some worked examples go through the entire process and you are also given some data sets to work through on your own, applying your new skills and knowledge.

Hopefully you enjoy working through R module and find it helpful!!

Outline:

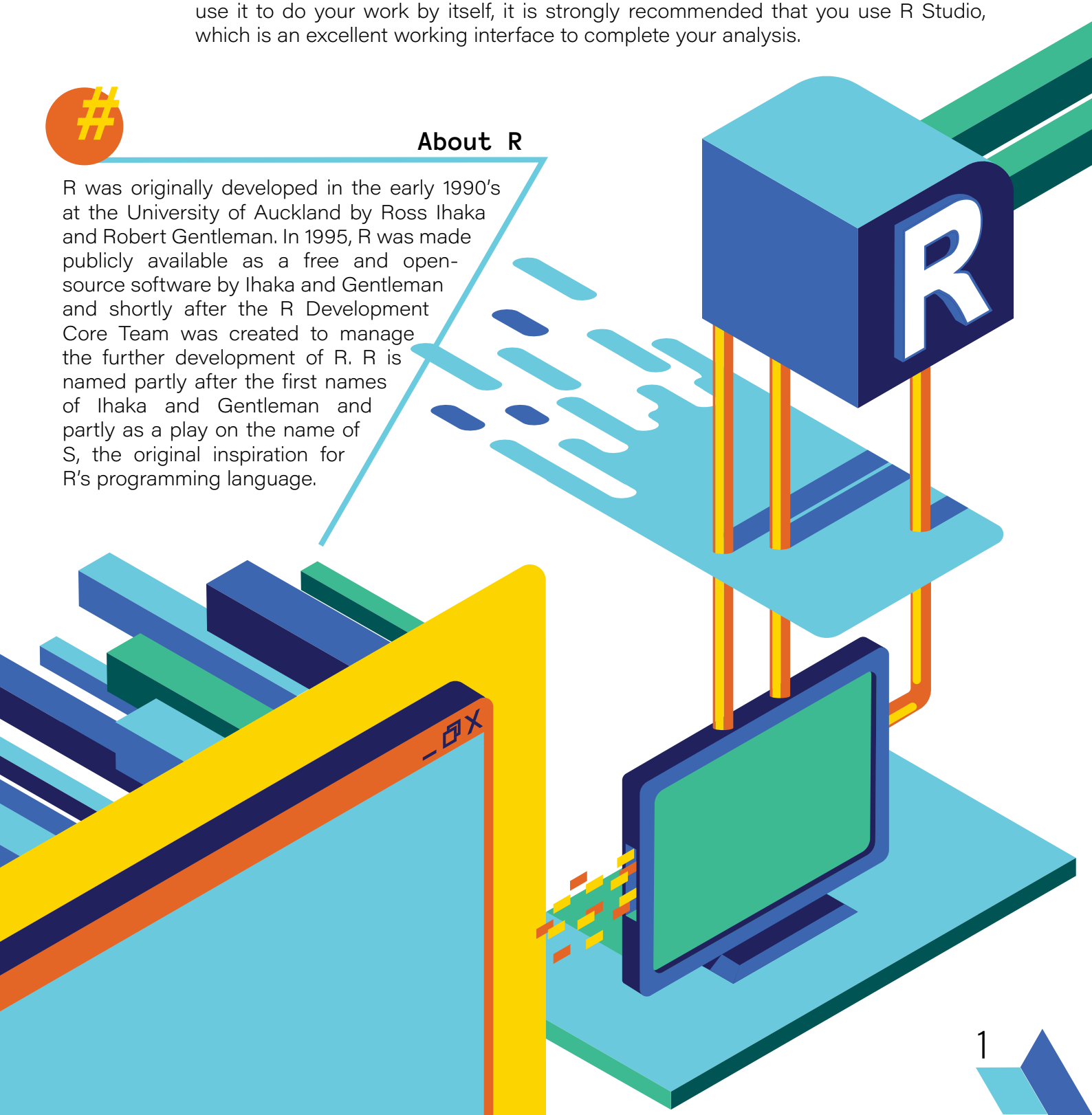
Section 1:

Getting Functional in R

The main statistical software that you will need to use is R and its commonly used, very user-friendly interface, RStudio. R is the computing "brains" and while you can use it to do your work by itself, it is strongly recommended that you use R Studio, which is an excellent working interface to complete your analysis.

About R

R was originally developed in the early 1990's at the University of Auckland by Ross Ihaka and Robert Gentleman. In 1995, R was made publicly available as a free and open-source software by Ihaka and Gentleman and shortly after the R Development Core Team was created to manage the further development of R. R is named partly after the first names of Ihaka and Gentleman and partly as a play on the name of S, the original inspiration for R's programming language.



Part I – Installing and setting up R and RStudio

First of all, you will need to install R and RStudio in your computer. You can follow the links below to complete this part:

- R: https://www.r-project.org
- RStudio: https://rstudio.com

If you need to install R and RStudio, please use this video as a guide:

›Click Here‹

Part II – Getting to know R and RStudio

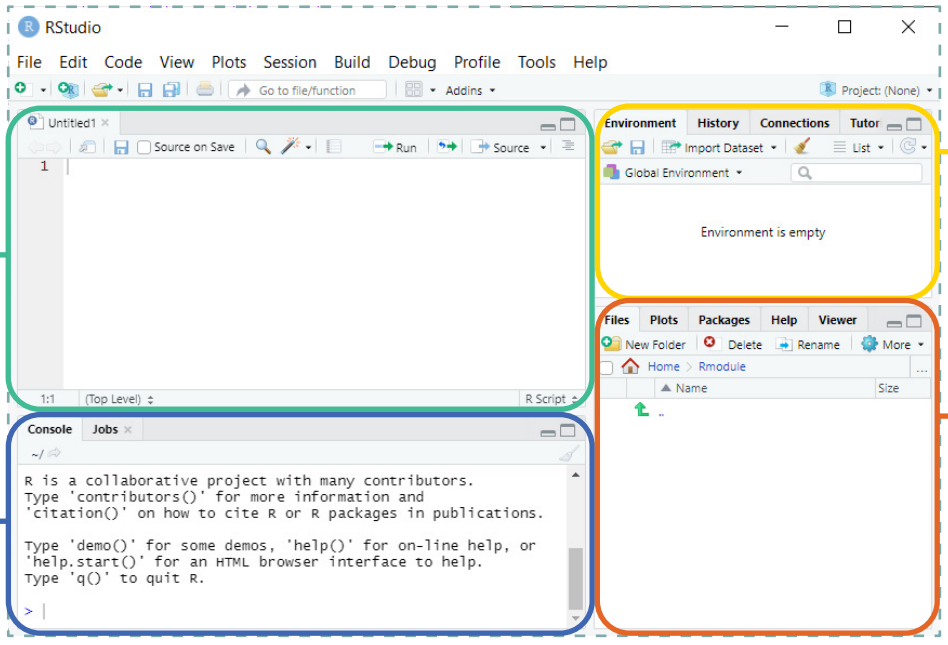
You should now have two R-related programs installed – R and RStudio. As R is the computing "brains", in theory you can run your code directly into this program, however it is much better to use the RStudio interface, which will be introduced in this section.

As mentioned in the video, there are four main sections/ components in the RStudio interface (these are also seen in the picture below):

Firstly you need to set up your RStudio interface – watch the video at the link below that shows you how to do this and also explains the various sections or components that you can view:

›Click Here‹

Code editor: This section is where you write and draft the code that you want to subsequently run in the console. When you write your code in here, it will not automatically run (unlike in the console, where you can directly write your code, but it will automatically run), and instead you can make sure that it is correct before you then run it. To run the code, have the blinking cursor on the line you want to run and hit the "Run" button along the top bar of this section.



Workspace and history: This section is where you can see any data that you input into R, stored and ready to use.

Plots and files: This section is where any graphs that you produce, will appear. This section is also where documentation appears when you search R for functions.

R console: This section is where the code runs, the output of your analysis appears here as well.

t i p s

In the code editor, you can write notes and organise your code by putting a # at the beginning of the line. Any line that has a hashtag (#) at the front will not be passed to the console and will instead be treated as text.

Part III <- How to start a project and save your work

When you start an analysis, you will want to create a new project that you can have all the related data, code and files in the one place. Projects can also be saved so you can continue your work at a later session without any loss of information.

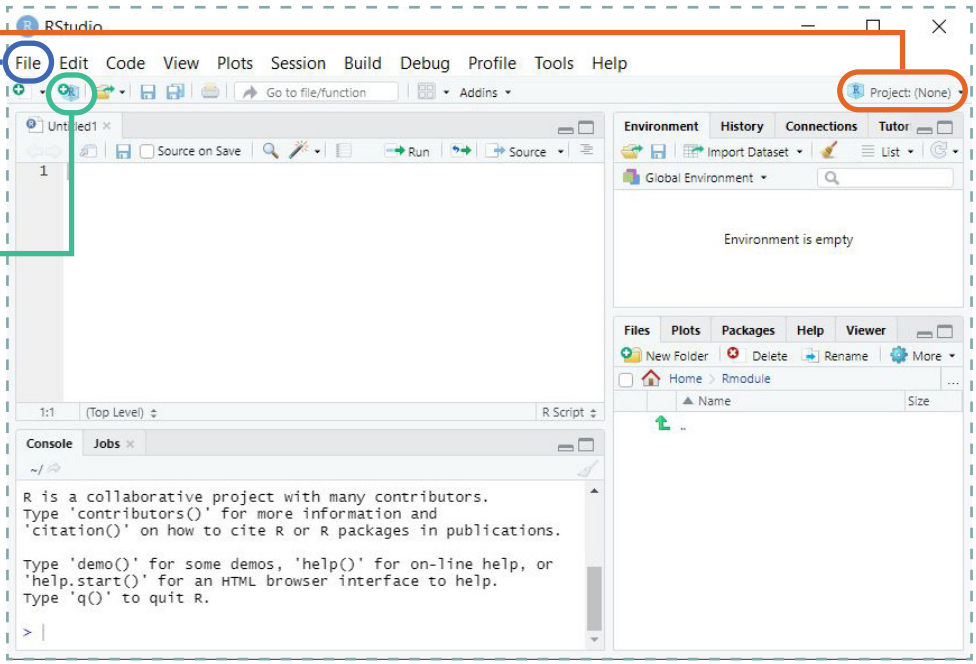
Watch the video at the link below that shows you how to create an R project and also explains how to save projects and code, open projects and exit from a project:

>Click Here<

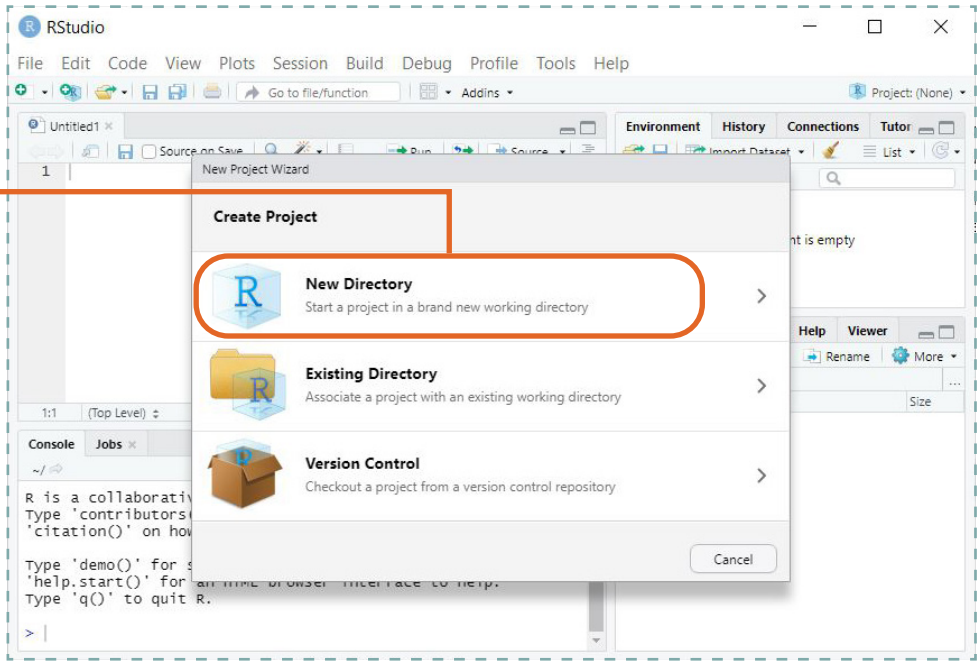
As mentioned in the video, the first thing that you need to do is to create a Project. This can be done by selecting the **New Project** command (available on the Projects menu and on the global toolbar). At that point a pop-up will appear and you choose where to create the project (normally a **New Directory**), select it as being a **New Project** and then set up the directory in a folder.

You can find the **New Project** command in both of these locations.

Or directly click this shortcut.

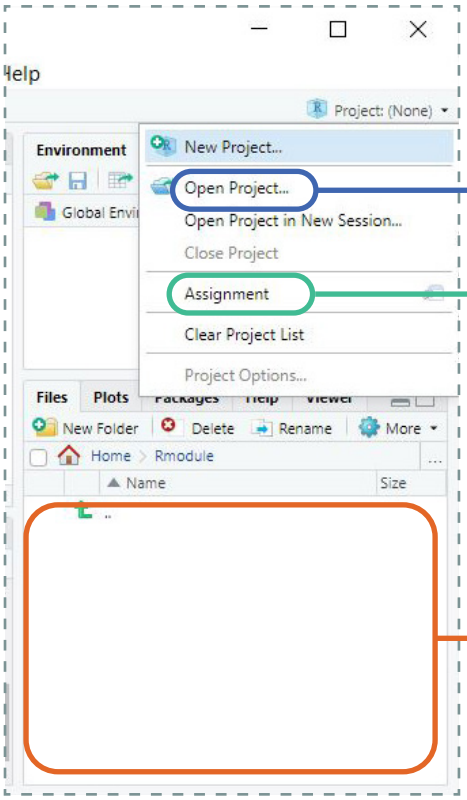


Click this to opt for a new directory to choose the location where you want to create your new project.



When you create a project in RStudio, the associated files are placed in the selected directory. The new project is also loaded in RStudio – you can check this as the name you have given it will appear in the Projects tool bar (found to the far right of the main toolbar).

The last project that you had open in RStudio will be the project that is opened when you start a new session in RStudio. To open a preexisting project that is not the one currently loaded, there are a few ways to do this:

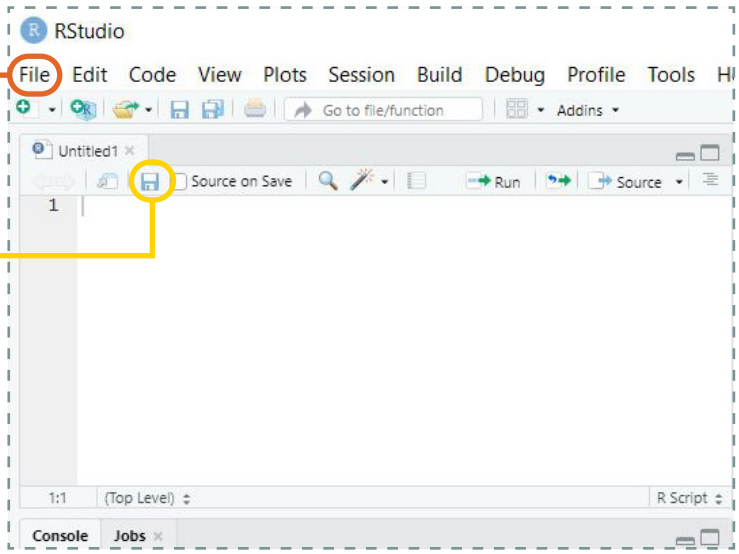


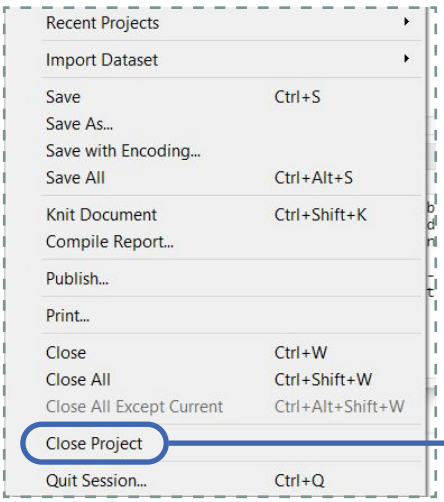
1) Select **Open Project** (available on the Projects menu and on the global toolbar) and then browse for, and then select, your desired existing project; or

2) If the project is one you have recently worked on, it will be in the list of most recently opened projects, available on the Projects menu and on the global toolbar ("Assignment" is the name of a recent project on this device); or

3) Double click on the project in file explorer.

To save the code that you are writing in the code editor, select the save button either selecting **Save** in the global tool bar (accessed through selecting File) or in the toolbar above the code editor. The file, which you name in the window that appears, will be stored in the same directory as your project.





The easiest way to save the rest of the elements in your project is to select to exit from RStudio (or to go to another project) and a pop-up will appear asking if you want to save your workspace image – select **Save** and all that you have done in that the project will be saved.

To exit from the project, you can select **Close Project** in the projects menu (or exit from R).

Part IV <- Features in R

In R, three of the important features that allow you to conduct analysis are Functions, Packages and Variables.

Watch the video that gives you information about some of these features, shows you how to search for functions and how to install packages:

>Click Here<

Functions

Functions are structures in R that take a set of data, perform an action on it and give you an output. This output could appear in the R console, or if the function is programmed to produce a graph or plot, this will appear in the plots and files window in the bottom right quadrant of RStudio.

Functions have names and the way they are used in R is to write its name, then in brackets state the name of the data that you want R to perform the function on.

A simple example of a function is the function called "`mean`" (see Section 2A Part II a for more information). If the following is run in R, you are instructing R to perform the function called "`mean`" (which is to calculate the mean) on the data set "`ExampleData`".

```
mean(ExampleData)
```

Sometimes functions require more details in addition to the name of the data that you want to perform it on. An example of such a function, that you will see in a later section is the `t.test` function which requires the data set, a value for mu and the confidence level (see Section 2C Part I for more details).

Many functions have already been made by R programmers and are either already available in R or can be loaded in by installing the package to which they belong (see information about Packages below). You can also write your own function. An example of how to create a function is in Section 2A Part III, where a function is built to calculate 95% confidence intervals.

To understand what a premade function does, you can read documentation about how it works and the theory behind it. You can find the documentation by typing into the console "?" or "??" then the function name. For example, to read the documentation for the `mean` function, you would type either:

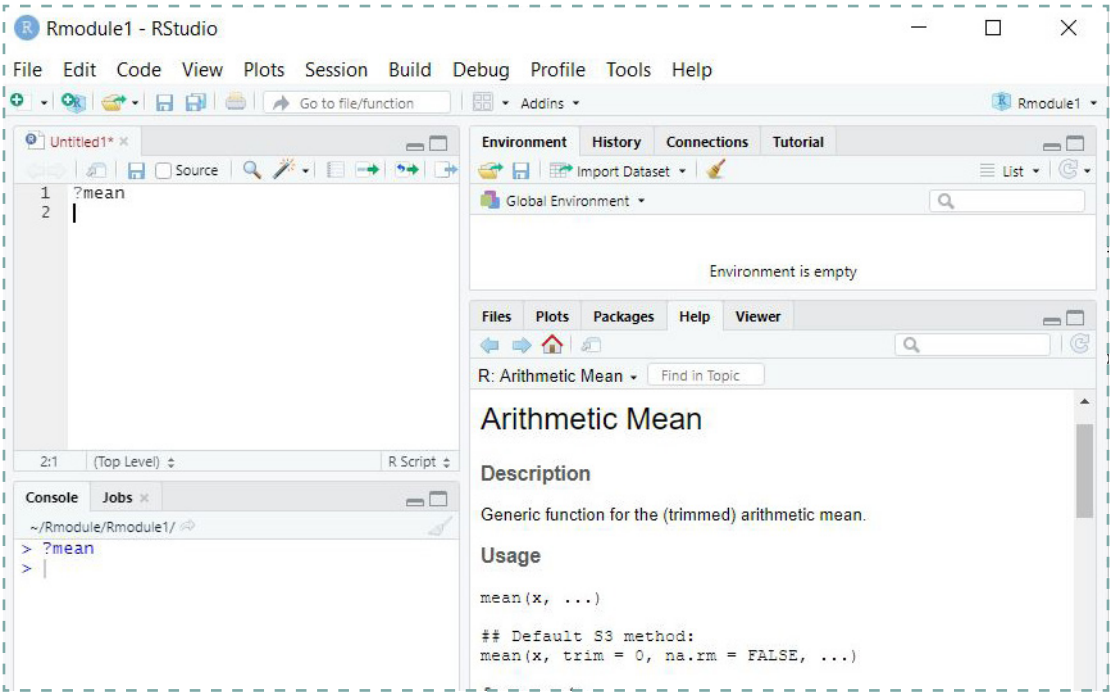
```
?mean
```

or

```
??mean
```

The R documentation for this function will then appear in the Plots and Files window in R Studio. The "?" option looks at the installed functions, while the "??" searches the entire database of available functions to find a matching term.

The below screenshot is from looking up the documentation for the `mean` function. When the command is run in the console (bottom left window), the documentation will appear in the bottom right window.



The documentation may be a bit wordy or difficult to understand at first but with practise it becomes a lot easier to understand.

Packages

All R functions and preloaded datasets are stored in packages. Some packages are preloaded into R, but you need to install others to use them. There are thousands of packages available for download (most at no cost). The current full list of R packages is available here (https://cran.r-project.org/web/packages/available_packages_by_name.html). This list is made available by the Comprehensive R Archive Network (CRAN).

Information and documentation about packages can be found by selecting the package of interest at the above link.

If the package you want to use is not one that is preloaded into R, you need to install it – this only needs to be done once on a device. When you want to use the functions available in a package, once it has been installed, you need to load the package – this needs to be done every time you open R and want to use the package.

CRAN is a worldwide network of servers that stores identical and current versions of the code and documentation required for R. CRAN is managed and maintained by the R Foundation and the R Development Core Team.

As mentioned in the text, CRAN is also the storage and source for all the packages you can access and use in R. CRAN was first released and announced in 1997, with 12 packages available. In 2021 there are currently 17,379 available packages to use!

Anyone can contribute and use packages to the CRAN Repository.

To install and use a package in R, the code is:

```
install.packages("PackageName")
require("PackageName")
```

- The term "`install.packages("PackageName")`" installs the "PackageName" package in R – this only needs to be done **once** on a device.
- The term "`require("PackageName")`" loads the "PackageName" package in R – this needs to be done **every time** you open R and use the package.

Please note that PackageName is not the name of a real package.

See the video and Section 2A Part I for an example of a real package – installing and using the outliers package.

Variables

In R, a variable is used to store information that functions and packages can manipulate - it is a data structure. Variables are named and can store a range of different types of information. The five main data/information types in R include; character, numeric, integer, logical or complex numbers.

The most basic type of variable in R is a single item of data, i.e. if it is numeric, a single value. The most common type of variable in R is a set of data/values i.e. a vector (see below). Variables can also be a group of vectors or a combination of a range of different structures available in R. The next section introduces the main data structures, variables, that you will encounter in R.

Part V ← Data structures and entering data into R

Data structures

Two of the main data structures that are used in R are vectors and Dataframes. A vector is a single list of values or characters while a dataframe is a larger data set. A dataframe is a combination of vectors bound together – this can be visualised by thinking of vectors as columns in a table, with the table being a dataframe.

Importing/inputting data into R

In R Module, there are two ways that you will be taught to input data – one is inputting directly into R, while the other is importing information from a file made in Excel.

These methods to import data are both explained in a video at the link below:

›Click Here‹ #

As mentioned in the video, firstly, we will look at directly inputting the data into R – this involves typing in the values, which are stored in R as vectors.

Say you have a list of values for a given measurement as follows:

The levels of chloride (in ppm) of 20 freshwater samples were as follows:

3, 4, 5, 8, 2, 6, 0, 1, 11, 6, 8, 7, 7, 7, 3, 9, 10, 4, 5, 1

This is a single vector of numbers and we can input it as such into R. The code for inputting the data as a vector is as shown, where this example adds the list of 20 values and calls the vector "`ExampleData`" – this name, however, can be anything that you like although it is advisable that you name it something logical.

To input data as a vector, the code is:

```
ExampleData <- c(3,4,5,8,2,6,0,1,11,6,8,7,7,7,3,9,10,4,5,1)
```

- The term "`ExampleData`" is the name of the vector – this can be changed to whatever you want it to be.
- The term "`<-c(...)`" is essential each time, where;
  - "`<-`" assigns the vector to that name.
  - "`c(...)`" constructs the vector.
- Each number (or character) is separated by a comma.

You can also store characters (i.e. letters or words) in a vector – this is useful for groups in data frames – see "`conditions`" in the following example.

tips

- It is recommended that you name your data with meaningful names related to the data that it portrays – this will come in handy when you are dealing with multiple sets of data. In this case, a good name would have been something like: chorideLevel
- Also, when your chosen name has two words, DO NOT include a space in the name, instead capitalise each word, e.g. instead of "example data", name your vector "`ExampleData`".
- A valid name can consist of letters, numbers and dot (.) or underline (_) characters. No other special characters (or spaces, as mentioned above), are allowed.
- It should also be noted that R is capitals-sensitive, e.g. R considers exampledata to be different to `ExampleData`, so please be careful of this – it is a common and easy mistake to make!

As mentioned above, vectors are a single list of values/characters, while dataframes are a series of vectors. The code for inputting a vector is above. One way you can create a dataframe in R, is add the vectors in separately and then "bind" them together.

An example of when you would need to do this is when you have a scenario as follows:

The percentage sugar content in a range of Manuka honeys from different suppliers, as determined by using a refractometer, is given in the table below.

sugar content (measurement)	supplier (i.e. conditition)
85.4	A
86.9	A
89.1	A
88.4	A
87.3	A
88.7	A
90.3	B
85.4	B
88.2	B
81.0	B
79.3	B
87.7	B
83.1	C
82.4	C
81.0	C
78.7	C
79.5	C
82.0	C

To add the information in this table, the code can be found below, where the two vectors "measurements" and "conditions" are added in separately, then bound together as columns in a dataframe called "ExampleDataframe.df".

The names of the vectors used in the example code below are purposefully non-specific to make it easier for you to apply the code to your own question/situation. In theory, I should have named the "measurement" vector as "sugarContent" and the "condition" vector as "supplier" to best represent the data.

Given you have already constructed two vectors called "measurements" and "conditions", like the following;

```
measurements <- c(85.4, 86.9, 89.1, 88.4, 87.3, 88.7, 90.3,
85.4, 88.2, 81.0, 79.3, 87.7, 83.1, 82.4, 81.0, 78.7,
79.5, 82.0)
conditions <- c("A","A","A","A","A","A","B","B","B","B",
"B","B","C","C","C","C","C","C")
```

To combine two (or more) vectors into a single data frame, the code is:

```
ExampleDataframe.df<-data.frame(measurements, conditions)
```

- The term "ExampleDataframe.df" is the name of the dataframe you are creating.
- The term "<-data.frame(...)" is essential each time, where;
  - "<-" assigns the dataframe to the specified name.
  - "data.frame(...)" constructs the dataframwith the specified vectors (in this case the vectors are called measurements and conditions). The name of each vector is separated by a comma.

t i p s

- In addition to the naming conventions described in the previous tips, it is often recommended that you also indicate what kind of structure the data has, in its name. Vectors traditionally do not have this, but other structures, like dataframes do. Dataframes are often indicated by including ".df" at the end of its name, e.g. "ExampleDataframe.df".

The second way in which data can be input into R is through importing an excel file. The best type of excel file to input is a .csv file, so please only try to import files of that type. Firstly, you need to get the excel file looking like you want it to look (this can be any number of rows or columns), and then:

To input data from an Excel (.csv) file, the code is:

```
ExampleInput.df <- read.table(file.choose(), sep=",",
header = TRUE)
```

- The term "`ExampleInput.df`" is the name of the dataframe you are creating.
- The term "`<-read.table(file.choose(), sep=",", header = TRUE)`" instructs R that you want to import a file of your choosing and is essential each time.
  - "`<-`" assigns the dataframe to that name.
  - "`read.table(file.choose(), sep=",", header = TRUE)`" can be broken down further into its parts:
    - "`read.table(...)`" is saying to read in a "table" of data from a file into R.
    - "`file.choose()`" is saying that you want to choose the file to add.
    - "`sep=","`" is saying that the separator between values is a comma (as you are inputting a comma separated value (.csv) file.
    - "`header = TRUE`" is saying that in the file you are adding, you have headers, i.e. the first row of the file is not data, but column headings.

  This command opens up a file explorer window (you will either see this pop up in front, like in the video, or it will open up another RStudio tab that you be able to open on the computer taskbar) and you select the .csv to be imported.

Regardless of whichever method you use, you should always inspect what your data looks like in R, to make sure it has been input/imported correctly. You can do this by either typing in the name in the console, pressing enter and it will appear in the console. The other way that you can use to check dataframes is by clicking on the name of the vector/dataframe in the top right window and it will appear where the R script window is (the top left).

Part VI <- Exporting output from R

The results of your analysis in R can be easily exported to other documents. You can either export individual pieces of output that you produce, into a separate document, or you can ask R to produce an editable document with both the code you used and the output you generated.

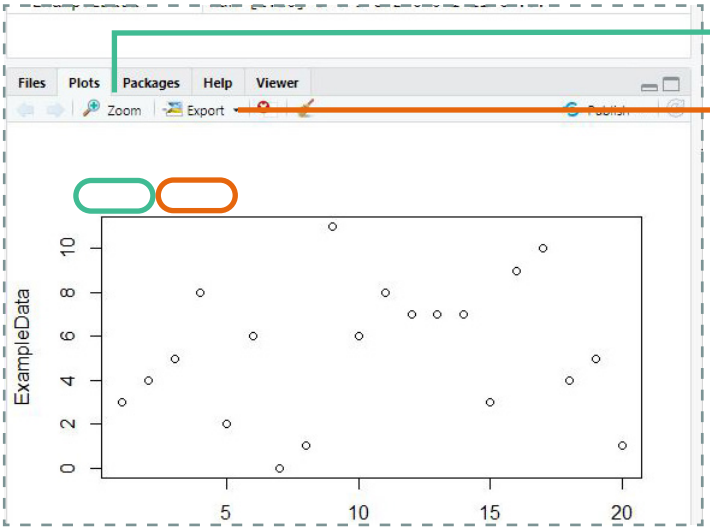
Watch the video at the link below that gives you information about the ways to export output that you produce in R:

>Click Here<

To export individual output

Console: The easiest way to export output that appears in the console is to copy and paste the numerical output to the document that you are working in.

Graphs and plots: Figures that appear in the Plots and Files window can be exported/ saved in a number of ways:

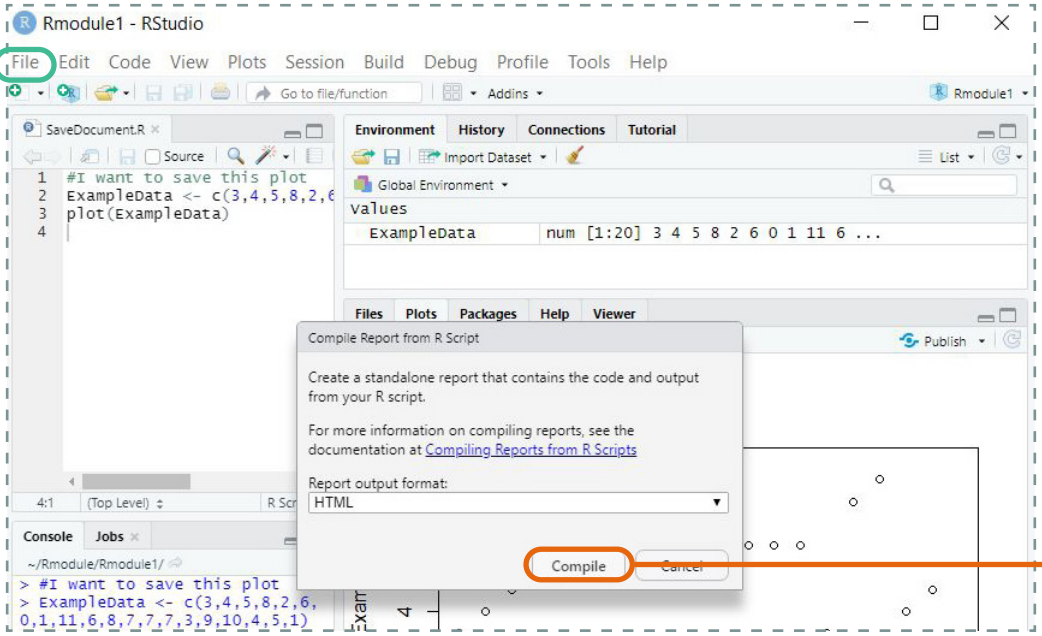


1) In the plot toolbar, click **Zoom**, which opens up the plot in a separate window . Right-click on the graph/plot and select to either copy the image (to then paste it in your document) or save the image; or

2) In the plot toolbar, there is an option to **Export**– select this and you can save the graph as an image or pdf.

To produce an editable document with code and output

In the File tab in the global toolbar, select "**Knit document**" which gives you the option of producing a HTML, PDF or MS Word document. A MS Word file is highly recommended as you can then easily edit the file. Select which file type you want to produce and then select **Compile**. The output file will appear in the folder that your project is stored in.



If you have input data from an Excel file in your code, once you ask R to compile, a window will appear, in which you will needto find and select the data file. Also, in case you have an `install.packages()` command in your code, remove before compiling the document.

t i p s

- This is the best way to produce your analysis for an assignment or task as this gives both the output and the code used to generate it, in one document.

Section 2A:

Basic Statistical Analysis

Once you have input your data into R, there is a huge range of different analyses that you can do. This section will focus on many of the basic analyses of single sets of data that you would do when you first start your analysis - assessing if there are any outliers in the data set (that would need to be removed) and calculating descriptive statistics and confidence intervals.

A video to accompany the notes given in this section, given below, can be found here:

>Click Here<

Part I – Outlier detection

The first thing that you should do for most data types when you start your data analysis process is determine if there are any outliers. In fact, it is preferable that this is actually done when you are taking your measurements (so if you need to discard an outlier, you are able to measure another data point to replace it) but this is not always possible, particularly if you have limited time or are given the data to analyse.

In R Module we will go through two different outlier tests – the Dixon's Q-test and the Grubbs test, both of which can be performed using R:

To conduct outlier tests in R, the code is:

```
install.packages("outliers")
require("outliers")
dixon.test(ExampleData)
grubbs.test(ExampleData)
```

- The line "`install.packages("outliers")`" installs the "outliers" package in R – this only needs to be done **once** on a device.
- The line "`require("outliers")`" loads the "outliers" package in R – this needs to be done **every time** you open R and use the package.
- The term "`dixon.test(...)`" instructs R to perform a Dixon's Q-test on the specificed vector (in this case the vector is called `ExampleData`).
- The term "`grubbs.test(...)`" instructs R to perform a Grubb's test on the specified vector (in this case the vector is called `ExampleData`).

The output from the tests will show up in the console, which you will then be able to interpret.

Part II – Calculating descriptive statistics

Once you have determined if there are any outliers in the data (and removed them/ corrected them if there are), you can move on to calculating descriptive statistics (mean, standard deviation, variance), using R:

To calculate descriptive statistics, the code is:

```
mean(ExampleData)
sd(ExampleData)
var(ExampleData)
```

- The term "`mean(...)`" instructs R to calculate the mean of the values of the specified vector (in this case the vector is called `ExampleData`).
- The term "`sd(...)`" instructs R to calculate the standard deviation of the values of the specified vector (in this case the vector is called `ExampleData`).
- The term "`var(...)`" instructs R to calculate the variance of the values of the specified vector (in this case the vector is called `ExampleData`).

As for the outlier tests, the output values for these calculations appear in the console.

Part III <- Calculating confidence intervals

To calculate a confidence interval in R, there is no available pre-made function, like there is for calculating the mean and standard deviation, so I have constructed a function called "confidence.interval". Once you load in the function, you will not need to do so again in your session/project.

See Section 1 Part IV for an introduction to functions. All of the functions that you can use in R have been made like the confidence interval function that you see here.

As you can see, a lot of code and actions can be included in a function. Fortunately, by creating functions we do not have to write all this code out each time, instead we only need to write out the command to use the function and specify the data (and other details, if required), to use it.

The code that you need to input into R to create the function is as follows:

```
confidence.interval <- function(vector) {
xbar <- mean(vector)
s <- sd(vector)
n <- length(vector)
se <- s/sqrt(n)
t <- qt(0.975,df=n-1)
margerror <- se*t
lower <- xbar-margerror
upper <- xbar+margerror
cat("The 95% confidence interval is (",lower,",",upper,")")
}
```

Once this function has been input into R, to calculate the 95% confidence interval, the code is:

```
confidence.interval(ExampleData)
```

- The term "`confidence.interval(...)`" instructs R to use the function we have constructed to calculate the 95% confidence interval of the values of the specified vector (in this case the vector is called `ExampleData`).

You do not need to understand the code in the function. If you copy and paste the function above, and then use it, the confidence interval will be calculated and appear in the console.

Section 2B:

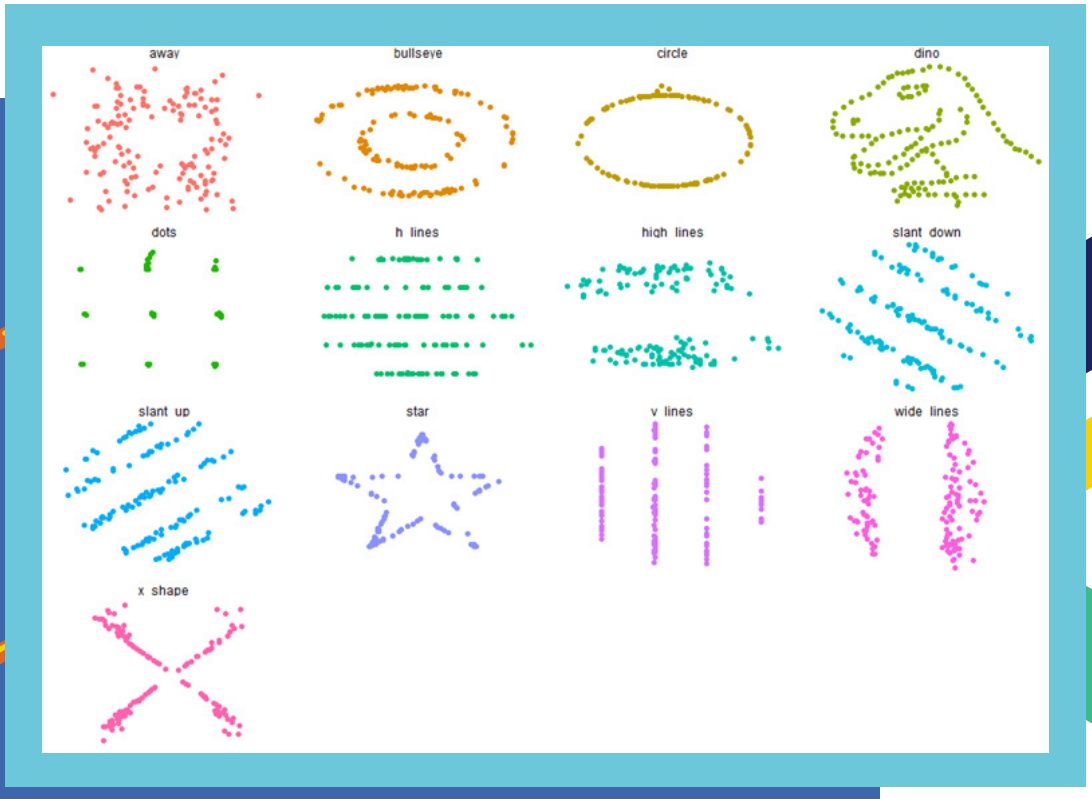
Basic Graphing/ Data Visualisation

One of the first steps that you should do before you conduct more-extensive analysis is to graph your data. This allows you to visualise your data and consequently learn a lot about your data that descriptive statistics alone will not necessarily tell you. The importance of this is particularly shown using the Datasaurus data available in a package in R.

A video to accompany the notes given in this section, given below, can be found here:

›Click Here‹

The Datasaurus data package contains 13 sets of x-y data, with each set having very similar descriptive statistics (i.e. almost identical mean of x, mean of y, standard deviation of x, standard deviation of y, and Pearson correlation between x and y). If you were to just look at the descriptive statistics, you would say that the data sets are essentially the same. However, when you graph the data set using scatter plots, it becomes very clear that each set looks very different!



The best graph to use to visualise your data depends on what kind of data you have:

- If you have a single set of values i.e. a vector of numbers, a one-dimensional dotplot (for small sample sizes, see Part I) or a box and whisker/bar plot (for medium to large sample sizes is recommended).
- If you have sets of values that you want to compare, you can plot these plots side-by-side (see Part II).
- If you have x-y data (i.e. like the data above), then a scatter plot (see Part III).

The graphs in this section are made using the basic functions in packages preloaded into R. There are other packages available that have far more flexibility and options (and many say produce graphs that are much nicer-looking!) for graphing. One of the most widely-used of these packages is ggplot2 which will be covered in Section 4, along with other useful graphing packages.

Part I <- One-dimensional plots

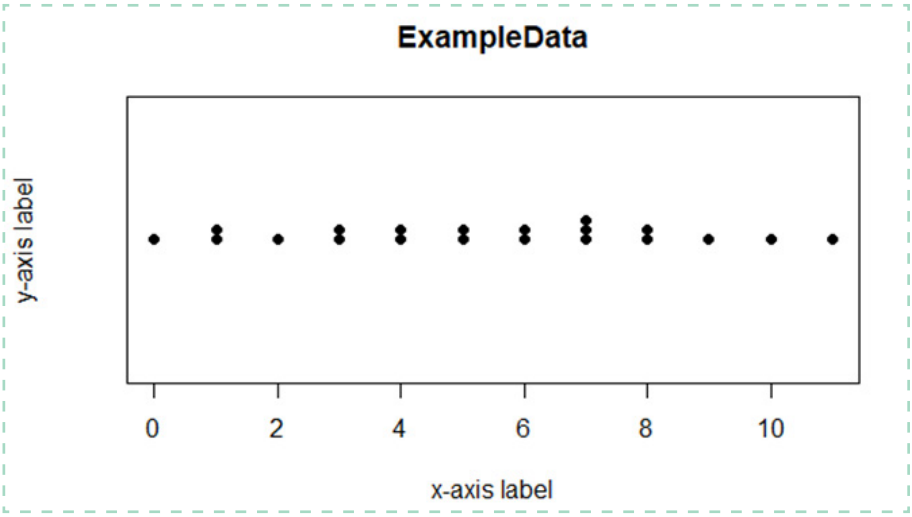
One-dimensional dot plots (sometimes known as strip charts) and box plots are excellent for visualising single data sets. Dot plots, in particular are good for small data sets.

To generate a one-dimensional dot plot, the code is:

```
stripchart(ExampleData, pch = 19, method = "stack", main = "ExampleData", xlab = "x-axis label", ylab = "y-axis label")
```

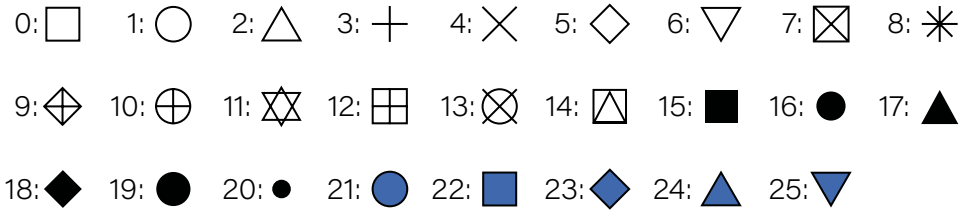
- The term "`stripchart(...)`" instructs R to create a one-dimensional dotplot of the specified vector (in this case the vector is called `ExampleData`).
  - "`pch = 19`" states the type of symbol that should be used in the plot. See tip over the page.
  - "`method = "stack"`" states that the symbols should be stacked on top of each other for repeated values.
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.

When this code is run, the graph will appear in the plot window, in the bottom right panel in RStudio. The plot generated by the above code can be found on the right:



t i p s

- R has a range of symbols that you can use in your graphics. These are stipulated for your graph using the `pch` part of the code. There are 25 possible symbols to choose from. To change the symbol in the graph, just state `pch` = the corresponding number (in the last example, the number was 19, corresponding to the filled-in circle).



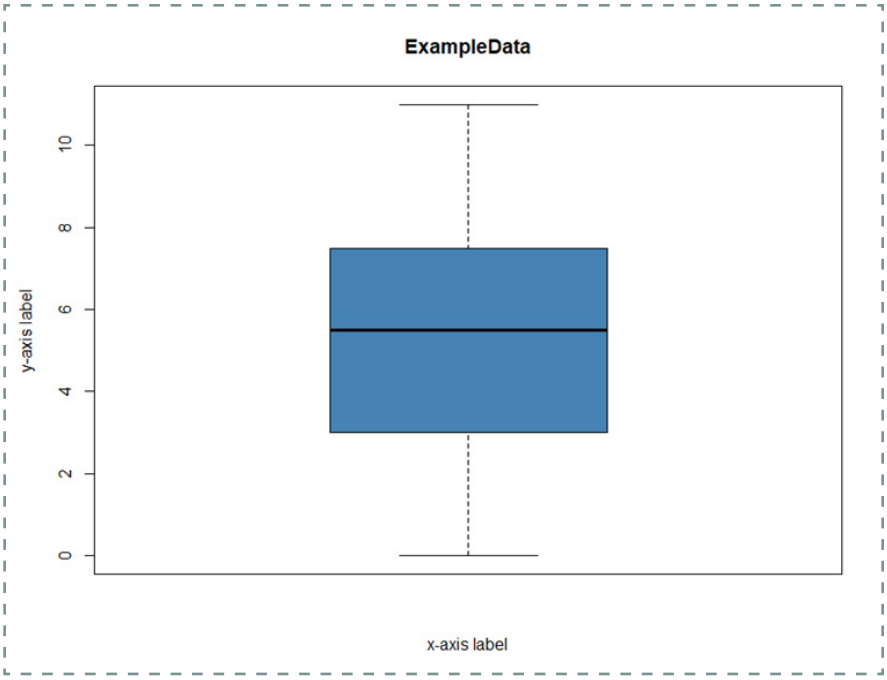
A boxplot can be produced using very similar code, omitting some of the components and changing the type of plot you want to produce in the command:

To generate a one-dimensional boxplot, the code is:

```
boxplot(ExampleData, main = "ExampleData", xlab = "x-axis label", ylab = "y-axis label", col = "steelblue")
```

- The term "`boxplot(...)`" instructs R to create a boxplot of the specified vector (in this case the vector is called `ExampleData`).
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.
  - "`col = "steelblue"`" says what the colour the boxplots should be. This can be any range of colours (see tip).

When this code is run, the graph will appear in the plot window, the bottom right panel in RStudio. The plot generated by the above code can be found on the right:



t i p s

- R has a range of colours that you can use in your graphics (like steelblue in the example about). To see all the available colours, type into R: `colors()` and all the names of the colours will be listed. To change the colour of the box in this graph, just replace the word in the speechmarks (i.e. `"steelblue"`) with the name of your desired colour, e.g. `"orangered"`.

  Popular colours with appropriate contrast include:

  - steelblue
  - darkcyan
  - orangered
  - mediumorchid1
  - deeppink2
  - goldenrod

Histograms and density plots (and box plots) are best used for larger data sets and are excellent at showing the distribution of the data. A similarly formatted command is needed for histograms:

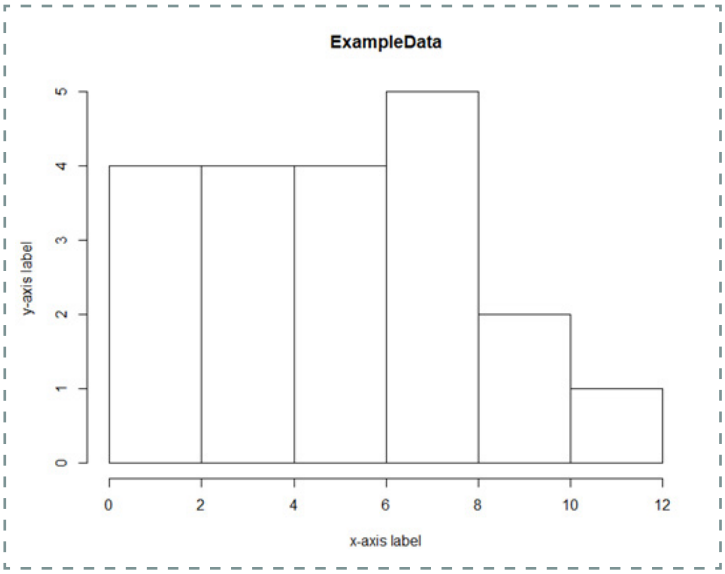
If you do not specify particular requirements about components (i.e. what colour you want the bars to be filled in, as shown in the boxplot above and how you could do for the histogram here by typing in "`col = "..."`") then the default colour will be used, which is typically either grey or no colour, for these graphs in the stats package.

To generate a histogram, the code is:

```
hist(ExampleData, main  = "ExampleData", xlab = "x-axis label",
ylab = "y-axis label")
```

- The term "`hist(...)`" instructs R to create a boxplot of the specified vector (in this case the vector is called `ExampleData`).
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.

When this code is run, the graph will appear in the plot window, the bottom right panel in RStudio. The plot generated by the above code can be found on the right:



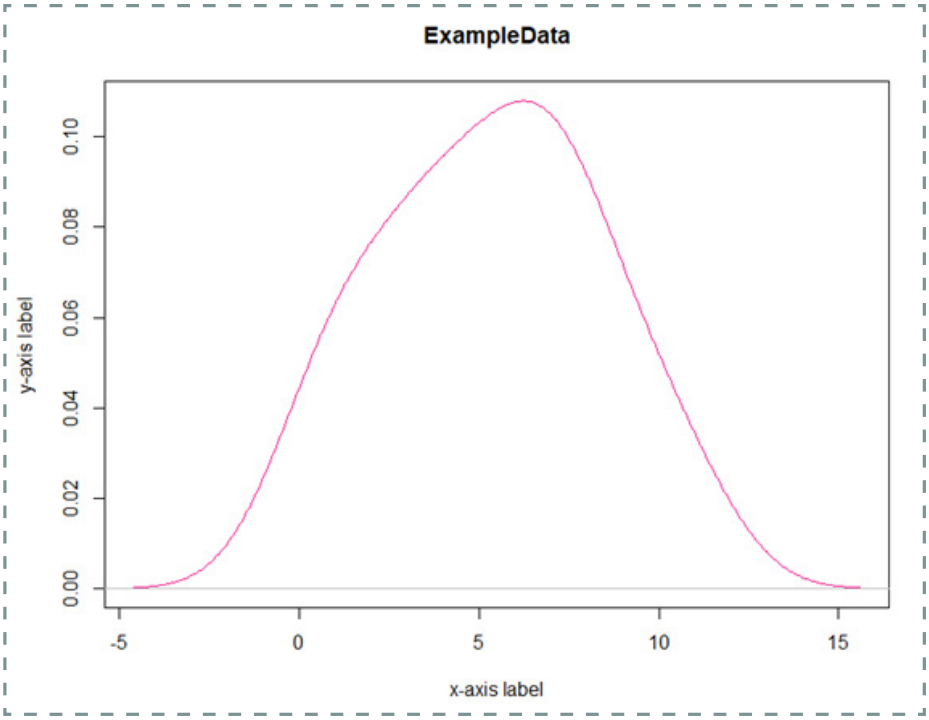
Density plots will provide similar data distribution information as histograms, but give a smoother distribution. The code to produce a density plot has two commands:

To generate a density plot, the code is:

```
dens <- density(ExampleData)
plot(dens,  col = "deeppink2", main  = "ExampleData", xlab
= "x-axis label", ylab = "y-axis label")
```

- The term "`dens<-density(...)`"instructs R to calculate the density values of the specified vector (in this case the vector is called `ExampleData`), giving the distribution.
  - R stores these densities as `dens`.
- The term "`plot(...)`" instructs R to create a boxplot of the supplied densities (in this case the densities are stored as `dens`).
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.
  - "`col = "deeppink2"`" says what the colour of the line density plot should be. This can be any range of colours - see tip.

The above code gives a density plot that looks as follows:



Part II <- Side-by-side plots

If you are comparing two or more single data sets (or a data that has different values for a given factor), to see what they look like compared to eachother/the effect of the factor, it is a good idea to produce side-by-side plots, i.e. boxplots or dot plots.

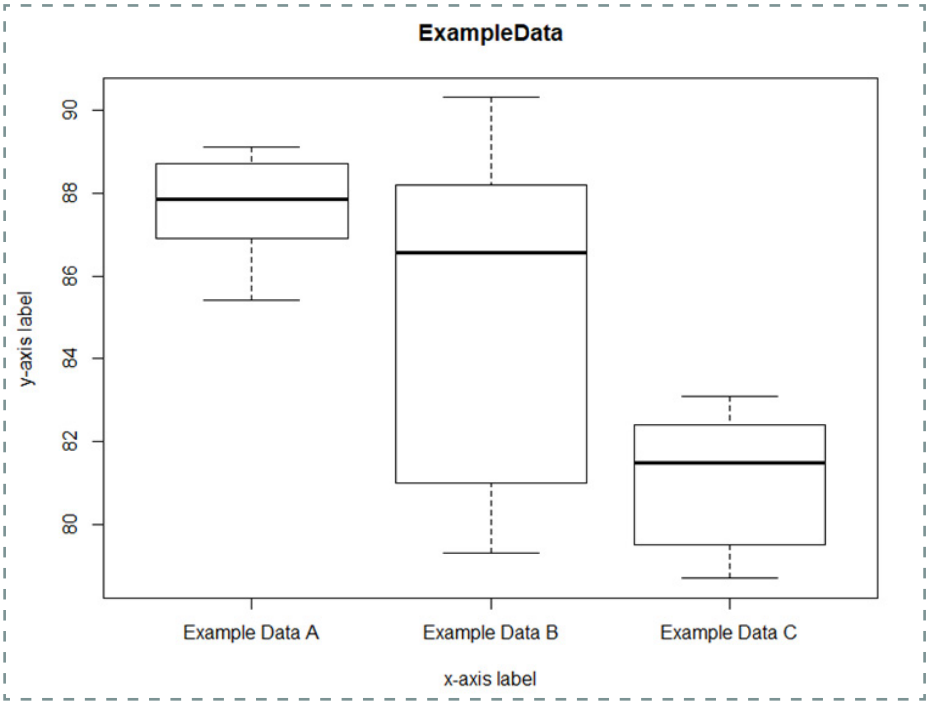
If you have two (or more) different data set vectors, side by side boxplots can be produced very simply by using the same code as earlier for one-dimensional plots, but specifying all the data vectors you want to plot, as is shown in the below code:

To generate a side-by-side boxplot from two or more data vectors, the code is:

```
boxplot(ExampleDataA, ExampleDataB, ExampleDataC, names=
c("Example Data A","Example Data B", "Example Data C"),
main  = "ExampleData", xlab = "x-axis label", ylab =
"y-axis label")
```

- The term "`boxplot(...)`" instructs R to create a boxplot of the specified vectors (in this case the vectors are called `ExampleDataA`, `ExampleDataB` and `ExampleDataC`).
  - "`names = c("Example Data A","Example Data B", "Example Data C")`" says what the names you want each of the boxes to be called, on the x-axis.
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply;av y-axis label.

The plot generated by the above code can be found below:



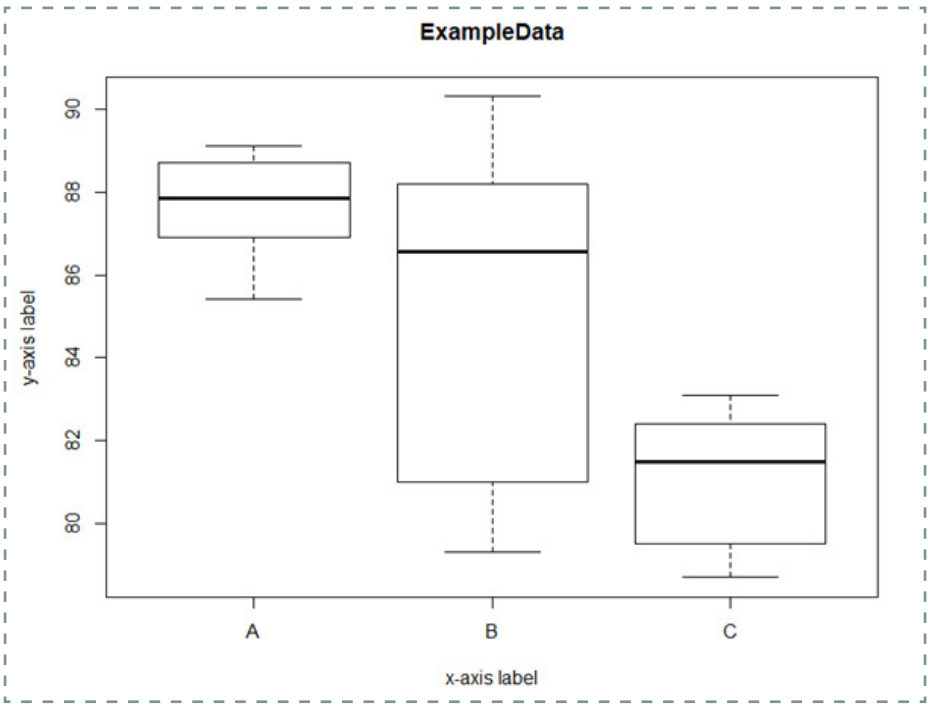
If you have a dataframe with numerical data as one column and a factor in another (i.e. the `ExampleDataframe.df` shown earlier in Section 1 Part V, where measurements is the numerical vector and conditions is the grouping factor/variable), side by side boxplots can be produced using a variation of the earlier code:

To generate a side-by-side boxplot from two or more data vectors, the code is:

```
boxplot(measurements~conditions, data = ExampleDataframe.df, main = "ExampleData", xlab = "x-axis label", ylab = "y-axis label")
```

- The term "`boxplot(...)`"instructs R to create a boxplot of the specified numerical vector (in this case the vector is called `measurements`) with a separate box for each level/group of the categorical vector (in this case the vector is called `conditions`), with both of these vectors in a dataframe (in this case the data frame is called `ExampleDataframe.df`).
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.

The plot generated by the above code can be found below:



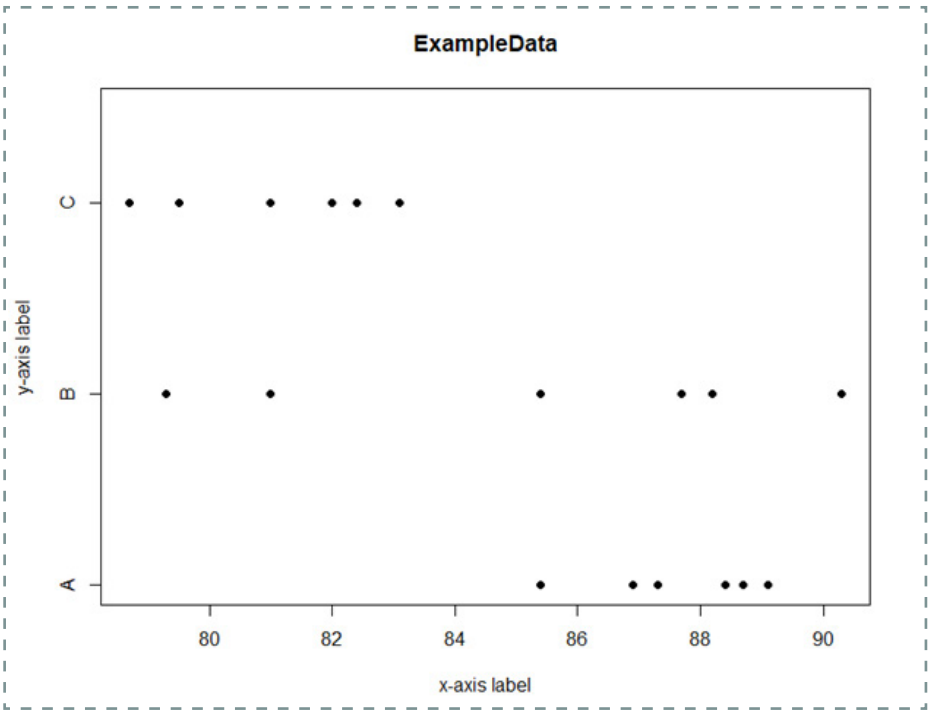
To do the same as above i.e. when you have a dataframe with numerical data as one column and a factor in another (e.g. the ExampleDataframe.df shown earlier in Section 1 Part V, where "measurements" is the numerical vector and "conditions" is the grouping factor/variable), side by side dotplots can be produced using a variation of the earlier code:

To generate a side-by-side dotplot from data in a dataframe, the code is:

```
stripchart(measurements~conditions, data=ExampleDataframe.df,
pch=19, method="stack", main="ExampleData", xlab="x-axis label",
ylab = "y-axis label")
```

- The term "stripchart(...)"instructs R to create a dotplot of the supplied numerical vector (in this case the vector is called measurements) with a separate group for each level/group of the categorical vector (in this case the vector is called conditions), with both of these vectors in a dataframe (in this case the data frame is called ExampleDataframe.df).
  - "pch = 19" specifies the type of symbol/dot that should be used in the plot.
  - "method = "stack"" specifies that the symbols should be stacked on top of each other for repeated values.
  - "main = "ExampleData"" says what the main title of the graph should be, in this case I have titled it simply ExampleData.
  - "xlab = "x-axis label"" says what the main title of the graph should be, in this case I have titled it simply x-axis label.
  - "ylab = "y-axis label"" says what the main title of the graph should be, in this case I have titled it simply y-axis label.

The plot generated by the above code using the ExampleDataframe.df data, can be found below:



Part III Scatter plots

If you have x-y data (i.e. like the data in the Datasaurus example), then a scatter plot is an excellent way to visualise the relationship between these two numerical data sets.

If you have x-y data, the two data sets you will either be given it in a file format that you can import into R as a dataframe or you can input the data vectors separately and combine them as two columns in a dataframe (see Section I Part V).

When you plot the results of regression models, this is done using a scatter plot. To the scatter plots, the line of the model that you fit can be added to this plot (see Section 2D Part III and Part IV).

To plot x-y data using a scatter plot, the code is:

```
plot(ExampleRDataframe.df$xvariable,
ExampleRDataframe.df$yvariable,   main   =   "ExampleData",
xlab = "x-axis label", ylab = "y-axis label")
```

- The term "`plot(...)`" asks R to plot the two quantitative variables (in this case "`xvariable`" and "`yvariable`") from a dataframe (in this case the dataframe is called "`ExampleRDataframe.df`"), with the first variable written being the variable on the x-axis and the second variable being the variable on the y-axis.
  - "`main = "ExampleData"`" says what the main title of the graph should be, in this case I have titled it simply; ExampleData.
  - "`xlab = "x-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; x-axis label.
  - "`ylab = "y-axis label"`" says what the main title of the graph should be, in this case I have titled it simply; y-axis label.

To see what this would look like with example data, see Section 2D Part III which uses the same code as shown above.

When you want to select columns in dataframes, you do so using a $ symbol.

An example is shown in the code – by writing `ExampleRDataframe.df$xvariable`, we are selecting a specific column called `xvariable` in the `ExampleDataframe.df` dataframe.

Section 2C:

Significance Tests

There are a range of statistical significance tests, where data is tested against a null hypothesis, these tests include:

- t-tests
  - one-sample t-test
  - two-sample t-test (not-significantly different variances)
  - two-sample t-test (significantly different variances)
  - paired-data t-test
- f-test
- ANOVA

R can conduct all of these tests for you, as will be shown in this section.

It is important to note that these tests assume that the data is normally distributed. This is not always the case, so it is important to check for this before you run these tests. The distribution of the data can be assessed by graphing techniques detailed in the previous section (Section 2B Part I and Part II).

A video to accompany the notes given in this section, given below, can be found here:

›Click Here‹

Part I T-tests

All t-tests in R use the same general code, with subtle variations depending on the type of t-test that you want to conduct.

The simplest situation is where you have one set of values (in R, as stated previously, they will be stored as a vector) and you are doing a one-sample t-test, assessing to see if the mean of the values is significantly different to a given value. Once the vector of values has been input into R, a one-sample t-test is conducted by the following:

To conduct a one-sample t-test, the code is:

```
t.test(ExampleData, mu = 0, conf.level = 0.95)
```

- The term "`t.test(...)`" instructs R to conduct a t-test.
  - "`ExampleData`" specifies the name of the vector undergoing the t-test.
  - "`mu = ...`" specifies the value is being are testing if the mean is equal to (in this case, this is `0`).
  - "`conf.level = 0.95`" specifies the confidence level. As part of the output for the t-test, R will give a confidence interval for the mean based on this confidence level. This will almost always be 95% (`conf.level = 0.95`), unless otherwise stated.

The output from this will appear in the console window, which will include a range of information, including the t-test statistic, p-value and a confidence interval. Examples of R output and how to interpret them are covered separately (i.e. in lectures).

The next examples of t-tests that you need to know are two-sample t-tests. These are t-tests that determine if two different sets of values have means that are significantly different to each other. There are two types of two-sample t-tests – when the variances of the two data sets are significantly different and when they are not significantly different. The test to see if variances are significantly different is an F-test (see more about how to do this in Part II of this section).

To carry out two-sample t-tests in R, the code is very similar for the two types, with only one difference:

To conduct a two-sample t-test where the variances of the data sets are not significantly different, the code is:

```
t.test(ExampleDataA, ExampleDataB, var.equal = TRUE,
conf.level = 0.95)
```

- The term "`t.test(...)`" instructs R to conduct a t-test.
  - "`ExampleDataA`" and "`ExampleDataB`" specify the names of the vectors for the t-test.
  - "`var.equal = TRUE`" is saying that the variances of the two data sets are not significantly different.
  - "`conf.level = 0.95`" specifies the confidence level. As part of the output for the t-test, R will give a confidence interval for the differences between the means based on this confidence level. This will almost always be 95% (`conf.level = 0.95`), unless otherwise stated.

To conduct a two-sample t-test where the variances of the data sets are significantly different, the code is:

```
t.test(ExampleDataA, ExampleDataB, var.equal = FALSE, conf.level = 0.95)
```

- The term "`t.test(...)`" instructs R to conduct a t-test.
  - "`ExampleDataA`" and "`ExampleDataB`" specify the names of the vectors for the t-test.
  - "`var.equal = FALSE`" is saying that the variances of the two data sets are significantly different.
  - "`conf.level = 0.95`" specifies the confidence level. As part of the output for the t-test, R will give a confidence interval for the differences between the means based on this confidence level. This will almost always be 95% (`conf.level = 0.95`), unless otherwise stated.

The last type of t-test is a paired sample t-test. As for the two-sample t-tests, there are two data vectors, but the code is slightly different:

To conduct a paired t-test, the code is:

```
t.test(ExampleDataA, ExampleDataB, paired = TRUE, conf.level = 0.95)
```

- The term "`t.test(...)`" instructs R to conduct a t-test.
  - "`ExampleDataA`" and "`ExampleDataB`" specify the names of the vectors for the t-test.
  - "`paired = TRUE`" is saying that the two data sets are paired and the t-test should be a paired t-test.
  - "`conf.level = 0.95`" specifies the confidence level. As part of the output for the t-test, R will give a confidence interval for the mean of the differences based on this confidence level. This will almost always be 95% (`conf.level = 0.95`), unless otherwise stated.

Part II <- F-test

As stated above, F-tests are used to assess if the variances of two different data sets are significantly different, or not. An F-test should be performed prior to a two-sample t-test – the results of the F-test inform you which two-sample t-test you should do.

To conduct an F-test, you can do the following:

To conduct an F-test, the code is:

```
var.test(ExampleDataA, ExampleDataB, conf.level = 0.95
```

- The term "`var.test(...)`" instructs R to conduct a F-test.
  - "`ExampleDataA`" and "`ExampleDataB`" specify the names of the vectors for the F-test.
  - "`conf.level = 0.95`" specifies the confidence level. As part of the output for the t-test, R will give a confidence interval for the differences between the means based on this confidence level. This will almost always be 95% (`conf.level = 0.95`), unless otherwise stated.

Part III ANOVA (Analysis of Variance)

The last type of statistical test that you should know is ANOVA (Analysis of Variance). The way that we do this in R is slightly different to the t-tests and F-test. For this test, we also need to use a data frame, as opposed to different vectors. This data frame will typically have a column for the numbers/values/measurements and another column that represents the grouping variable. An example of this is the data table referred to earlier when introducing data frames – the "measurements" column contains information about the numbers/values that you are assessing and the "conditions" column is the grouping variable (see Section I Part V). Once the data frame is in R, ANOVA can then be carried out, where first you need to fit a linear model and then you run the analysis on the model:

To conduct an ANOVA, the code is:

```
anovaexample <- lm(measurements~conditions, data = Example Dataframe.df)

anova(anovaexample)

anovaexample.aov<-aov(measurements~conditions,data=Example Dataframe.df)

TukeyHSD(anovaexample.aov)
```

- The term "anovaexample" is the name of a linear model.
  - "<-lm(...)" instructs R to create a linear model, using two columns of data – one column is the measurements/values (i.e. quantitative) and the other column is the grouping variable (i.e. a qualitative variable) - in this case "measurements" and "conditions" , respectively, in a dataframe (in this case the dataframe is called "ExampleDataframe.df").
- The term "anova(...)" instructs R to conduct an ANOVA analysis on the relationship that you defined as your linear model (in this case, the linear model is called "anovaexample").
- The term "aov(...)" instructs R to compute the analysis of variance of the model and save it as an aov data structure) in this case the aov file is called anovaexample.aov).
- The term "TukeyHSD(...)" instructs R to calculate a set of confidence intervals on the differences between the means of the levels of a factor with the specified family-wise probability of coverage. The intervals are based on the Studentised range statistic, Tukey's 'Honest Significant Difference' method. This is done using the aov data file which holds the results of the analysis of variance of the model (in this case, the aov data file is called "anovaexample.aov"). This function requires an aov data structure, which is why this is made and used for getting the Tukey HSD intervals, instead of the linear model made earlier.

This code will produce output in the console that you can then interpret.

Section 2D:

Regression Analysis

R is also very useful for displaying, modelling and analysing relationships between two quantitative variables.
An example of when you would need to do this is when you have a scenario as follows:

A video to accompany the notes given in this section, given below, can be found here:

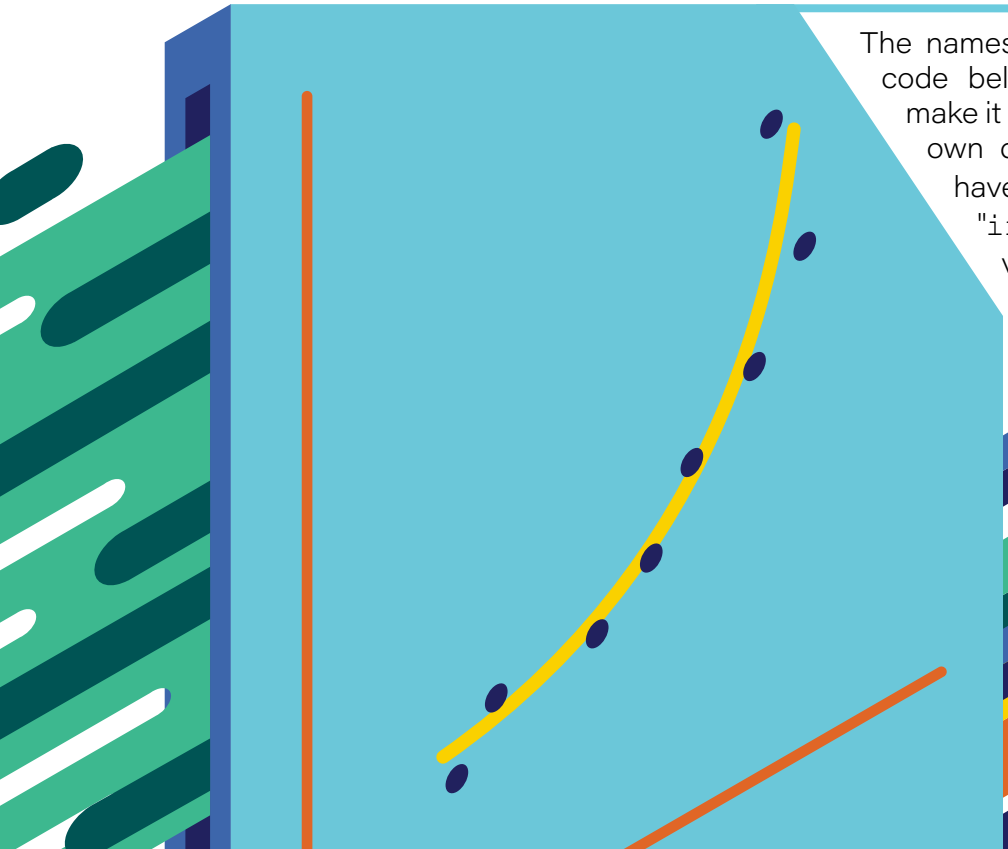
>Click Here<

Sulfaguanidine is a sulfonamide used to treat a range of bacterial infections, particularly in veterinary medicine. The following results were obtained during the analysis of sulphaguanidine by spectrofluorimetry:

Sulphaguanidine concentration (mg/L)	Intensity
0 (blank)	43
0.05	62
0.10	79
0.15	101
0.20	118
0.25	145
0.30	161

The x-variable is the explanatory/independent variable that you control (in this case it would be the sulphaguanidine concentration) and the y-variable would be the measured/dependent variable (in this case it would be the intensity).

The names of the vectors used in the example code below are purposefully non-specific to make it easier for you to apply the code to your own question/situation. In theory, I should have named the "`yvariable`" vector as "`intensity`" and the "`xvariable`" vector as "`sulphaguanidineConc`" to best represent the data.



Part I Unweighted linear regression

If the relationship between the variables is a linear relationship, a linear model can be fit and information can be gained from analysing the resultant equation that represents this relationship.

To fit a linear relationship between two quantitative variables in R, both of these variables are input as columns in a dataframe. A linear model is then fit to the data and you can view the specifics of the equation that represents this model. This can be done by the following:

To fit an unweighted linear model, the code is:

```
lmExample.lm <- lm(yvariable~xvariable, data = ExampleRDataframe.df)
summary(lmExample.lm)
```

- The term "`<- lm(...)`" instructs R to create a linear model (in this case "`lmExample.lm`"), using two columns of quantitative data (in this case "`yvariable`" and "`xvariable`" in a dataframe (in this case the dataframe is called "`ExampleRDataframe.df`")).
- The term "`summary(...)`" instructs R to provide information on the linear model between your the variables (in this case, the linear model is called "`lmExample.lm`").

Part II Weighted linear regression

While unweighted linear models are most common, sometimes unweighted linear is not appropriate for the data. This is commonly because the datapoints do not have equal uncertainty. Fortunately, fitting weighted linear models in R is very similar to fitting an unweighted linear model.

The first step that is required is that you need to create a vector of weights that you want to be applied. In this module there are two possible weightings shown that you could apply to your data.

To create a weighting vector for when the weighting is the reciprocal of the x-value, the code is:

```
weightsLm<-1/(ExampleRDataframe.df$xvariable)
```

- The term "`1/(ExampleRDataframe.df$xvariable)`" asks R to calculate the reciprocal of each of the x-values and store this as a new vector called `weightsLm`.

To create a weighting vector for when the weighting is the reciprocal of the standard deviation of the y-value, the protocol is to first fit an unweighted regression (as in Section 2D, Part I) and then calculate the ith squared residual as an estimate of the standard deviation^2, the code is:

```
weightsLm<-1/lm(abs(lmExample.lm$residuals)~lmExample.lm$fitted.values)$fitted.values^2
```

- The term "`1/lm(...)`" asks R to calculate the reciprocal of the ith squared residual from the unweighted linear model that was fit earlier (in this example called `lmExample.lm`), to each of the y-values and store this as a new vector called `weightsLm`.

Once you have calculated the vector of the weights, the next step is to fit the weighted linear model. This is very similar to fitting the unweighted linear model, with the only thing changing being that you add an extra `weights` term:

To fit a weighted linear model, the code is:

```
wlmexample.lm<-lm(yvariable~xvariable, weights = weightsLm,
data = ExampleRDataframe.df)

summary(wlmexample.lm)
```

- The term "`<-lm(...)`" instructs R to create a linear model (in this case called "`wlmexample.lm`"), using two columns of quantitative data (in this case "`yvariable`" and "`xvariable`" in a dataframe (in this case the dataframe is called "`ExampleRDataframe.df`")) and apply a weighting (`weightsLm` in this case).
- The term "`summary(...)`" instructs R to provide information on the equation that represents the weighted linear relationship between your two variables in the linear model (in this case, the linear model is called "`wlmexample.lm`").

Part III <- Linear Model Graphics

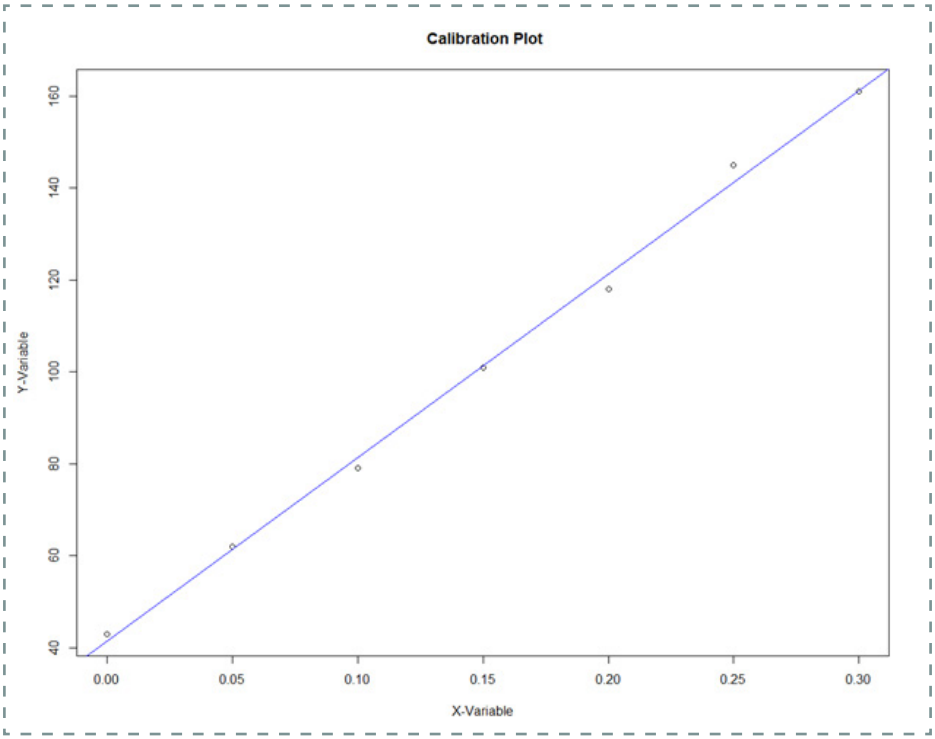
Once you have fit a (weighted or unweighted) linear model, you will need to graphically represent it – this can also be done using R:

To graph a linear model and calibration curve, the code is:

```
plot(ExampleRDataframe.df$xvariable, ExampleRDataframe.df$
yvariable, main = "Calibration Plot", xlab = "X-Variable",
ylab = "Y-Variable"

abline(lmExample.lm, col = "blue")
```

- The term "`plot(...)`" asks R to plot the two quantitative variables (in this case "`xvariable`" and "`yvariable`" in a dataframe (in this case the dataframe is called "`ExampleRDataframe.df`")), with the first variable written being the variable on the x-axis and the second variable being the variable on the y-axis.
  - "`main = "Calibration plot"`, "`xlab = "X-variable"`", "`ylab = "Y-variable"`" give the main title and the x-axis and y-axis labels.
- The term "`abline(...)`" plots the linear model (in this case `lmexample.lm`) that has been calculated previously – see Part I or Part II of this section. This command will plot a line on the plot that is active in the plot window.
  - "`col = "blue"`" says what the colour of the line should be. This can be any range of colours.

For the example data, the above code will give the following plot:



Once a linear model has been fit to our data, one of the key ways to assess the model to see if it is appropriate is to plot the residuals of the model. To do so, you would use this code:

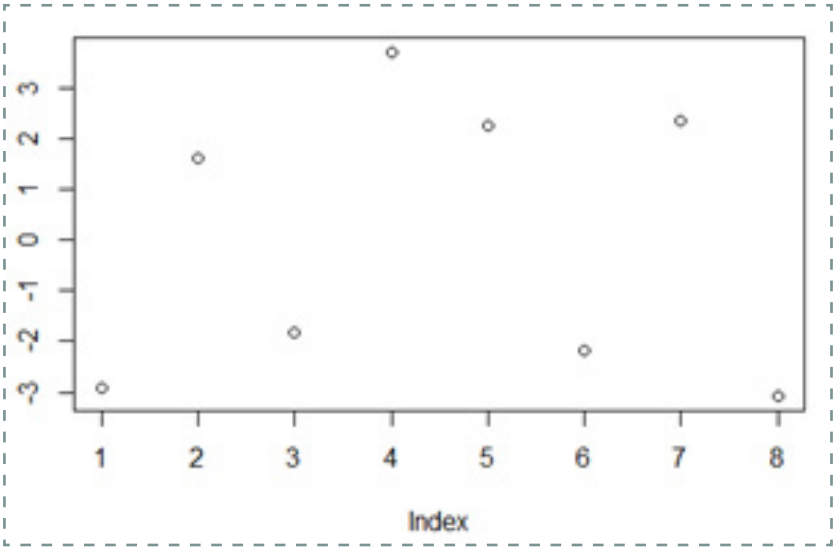
To plot the residual plot of a linear model, the code is:

```
plot(residuals(lmExample.lm))
```

- The term "`plot(...)`" asks R to plot the residuals of the fitted linear model (in this case called `lmxample.lm`) that has been calculated previously – see Part I of this section.

This is essentially a one-dimensional dot plot (but of the residuals) hence why you use a different function to the one described in Section 2B.

Residuals should be random and of constant variance and not have a notable trend or pattern. An example of residual plot that indicates the linear model is an appropriate fit would be:



Part IV <- Non-linear regression

Using R, you can fit a non-linear regression model, using the Gauss-Newton method. Before you fit the model, it is recommended that you plot the two variables to view their relationship. This can be done by using the code from Section 2B Part III.

To fit a non-linear regression model, you need to have the general equation that explains the relationship between the x and y variables which includes any unknown parameters as letters. An example of this kind of situation is described below:

The relationship between vapour pressure (Pres, in Torr) and the temperature (Temp, in degrees Celcius) can be described using the Antoine equation:

$$log_{10}Pres = a - \frac{b}{c+Temp}$$

Where a, b and c are parameters specific to a particular chemical. This above equation can be rearranged to give the following relationship between Pres and Temp:

$$Pres = 10^{a-\left(\frac{b}{c+Temp}\right)}$$

The vapour pressure (Pres) for ethanol was measured for a range of different temperatures (Temp) and the resulting data is given in the table below:

temperature	pressure
0	12
5	19
10	24
15	33
20	45
25	60
30	79
35	105
40	135
45	175
50	222
55	280
60	350
65	437
70	542
75	663
80	810
85	979
90	1180
95	1410

Estimates of the values for a, b and c are: a = 10, b = 1500, c = 200

The x-variable is the explanatory/independent variable that you control (in this case it would be the temperature) and the y-variable would be the measured/dependent variable (in this case it would be the pressure).

The names of the vectors used in the example code below are purposefully non-specific to make it easier for you to apply the code to your own question/situation. In theory, I should have named the "y" vector as "pressure" and the "x" vector as "temperature" to best represent the data.

To fit a non-linear model, the code is:

```
nlmexample.nls <- nls(y~10^(a-(b/(c+x)))),start=list(a=10,
b = 1500, c = 200), data = ExampleNLRDataframe.df)

summary(nlmexample.nls)
```

- The term "<-nls(...)" instructs R to create a non-linear model (in this case called "nlmexample.nls"), based on the given formula, using two vectors (in this case y and x) in accordance with the formula and the names of the vectors that have been input into R (these vectors are called x and y in the dataframe ExampleNLRDataframe.df).
  - "y~10^(a-(b/(c + x)))" is the formula describing the relationship between the two numerical vectors, with unknown parameters a, b and c. This will change for different relationships.
  - "start = list(...)" specifies the starting values for the unkown parameters to be optimised (in this case a = 10, b = 1500 and c = 200).
- The term "summary(...)" instructs R to provide information on the model that represents the relationship between the two variables in the model (in this case, the model is called "nlmexample.nls").

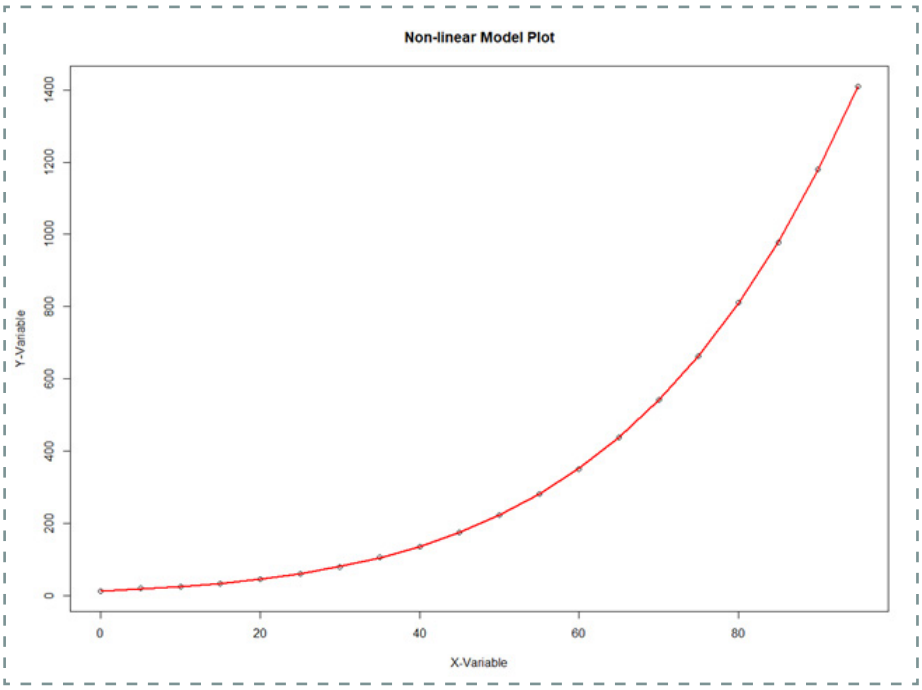
To plot the values with the non-linear model plotted to see how they match, you can use the following code (this code is very similar to that seen in Section 2D Part II:

To plot the data with the non-linear model, the code is:

```
plot(ExampleNLRDataframe.df$x, ExampleNLRDataframe.df$y,
main = "Non-linear Model Plot", xlab = "X-Variable", ylab =
"Y-Variable")

lines(ExampleNLRDataframe.df$x,predict(nlmexample.nls),
col="red", lwd = 2)
```

- The term "plot(...)" asks R to plot the two quantitative variables "x" and "y" which can be found in a dataframe (in this case called ExampleNLRDataframe.df).
  - "main = "Non-linear Model Plot"", "xlab = "X-Variable"", "ylab = "Y-Variable"" give the main title and the x-axis and y-axis labels, as seen previously.
- The term "lines(...)" plots the relationship described by the non-linear model "nlmexample.nls" on top of the datapoints, in red. This is different to abline used above for linear regression, as abline only plots straight lines – the lines function plots any relationship.

For the given example, the plot produced is:



Section 3A:

Using R for Unsupervised Learning

A video to accompany the notes given in this section, given below, can be found here:

>Click Here<

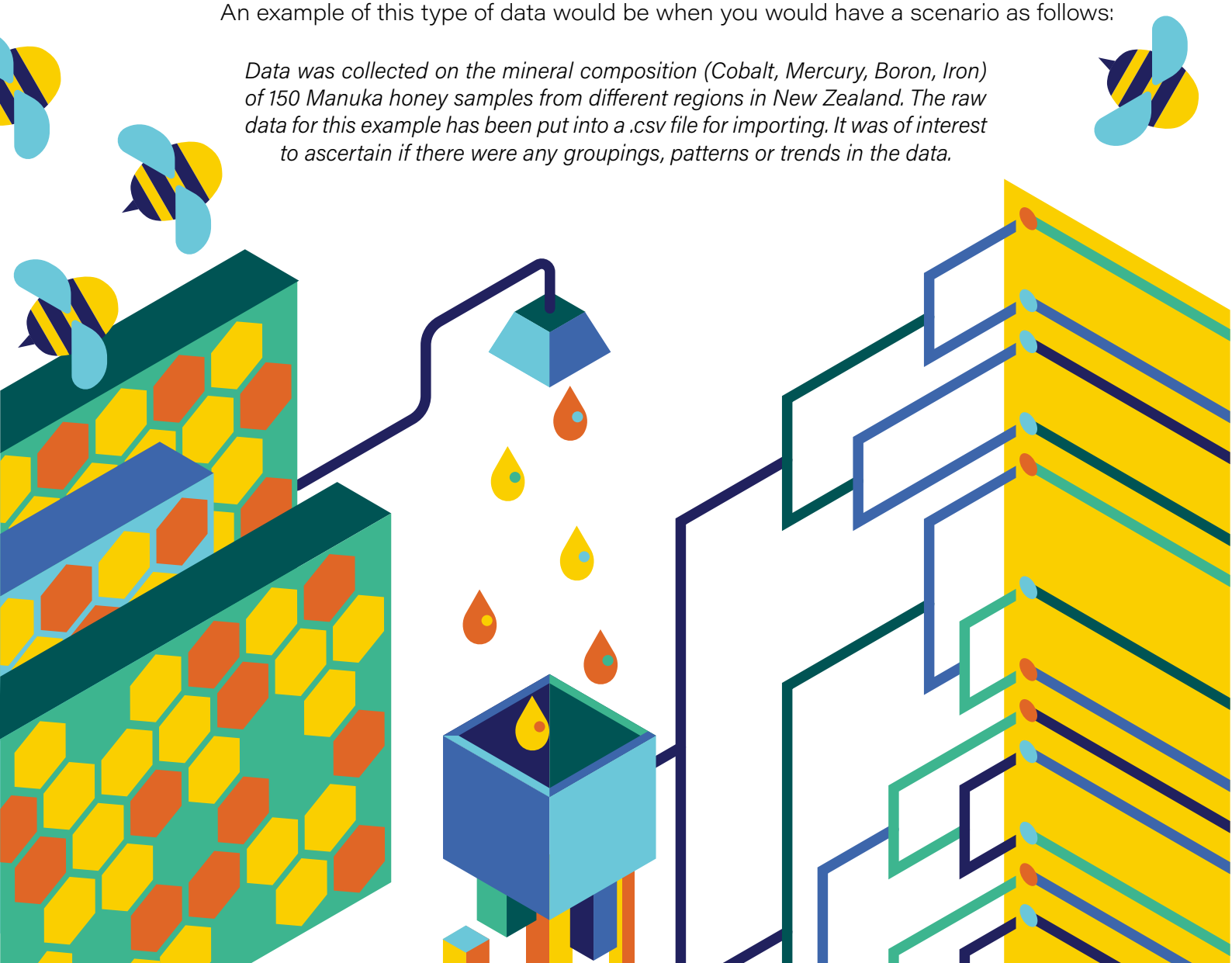
The previous analytical techniques have been concerned with analysing single variate data (i.e. analysing one variable changing – Section 2A and 2C) and relating two variables (i.e. regression – Section 2D).

In many situations, you need to analyse multivariate data (i.e. where several variables are measured for a single sample/observation) in such a way as to allow for all of these variables to be analysed simultaneously. This can be done using unsupervised (this section, Section 3A) and supervised (Section 3B) machine learning techniques.

Unsupervised machine learning (unsupervised pattern recognition) is multivariate exploratory data analysis and is concerned with identifying patterns in the data.

An example of this type of data would be when you would have a scenario as follows:

Data was collected on the mineral composition (Cobalt, Mercury, Boron, Iron) of 150 Manuka honey samples from different regions in New Zealand. The raw data for this example has been put into a .csv file for importing. It was of interest to ascertain if there were any groupings, patterns or trends in the data.



Part I <- Data manipulation and large data set management

In a chemistry-related, chemometric context:

- The objects might be samples, molecules, materials, findings, etc.
- Typical features or variables of those objects will be elemental patterns, spectra, concentrations, structural features, or physical properties.

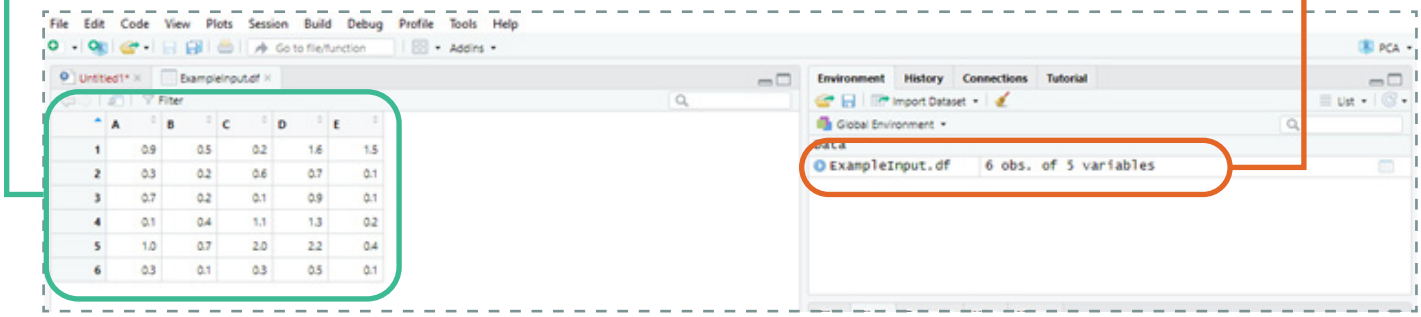
Before you can carry out any multivariate analysis, you first need to import the data into R.

In general, multivariate analytical data can be arranged as a data matrix of n objects (rows) and p features (columns).

The best, most recommended way to handle data sets of this type is to first create an Excel file. This is also how you are likely to be given data that you do not create/generate yourself. In your excel file, the data should be formatted as stated above, with columns being features and each row being a separate observation.

Once you have the data in this format (remember to save your file as a .csv file), you can import it just like previously described earlier (Section I Part V).

It is very important to check the data you have imported, particularly with larger data files where sometimes mistakes happen. The easiest way to check the imported data is to click on the name of the imported data frame in the global environment (the top right window of RStudio, orange box in screenshot). Selecting this will bring up the dataframe as a tab in the top left RStudio window (green box in screenshot).



You can use this to check to make sure that everything looks in order before moving on to the next steps. If something is not right, the easiest way to fix it is to go back to the .csv file, change it accordingly and then re-import it.

t i p s

- One of the most common errors that occurs when importing larger data sets is that sometimes extra cells are imported. These extra cells are empty and will have "NA" when you inspect the dataframe. The reason for this is that there is a space in the .csv file, so R thinks it is a cell that should be imported.

  The quickest way to fix this issue is to go to the .csv file and copy and paste only the cells you want, into a new file, save this file as a .csv and try to import it as you did earlier.

Part II ← Hierarchical cluster analysis (HCA)

To conduct hierarchical cluster analysis (HCA) on data, the first thing you need to do is mean-centre and scale the data, then you calculate a similarity matrix. One of the most common ways to calculate the similarity matrix is to calculate a Euclidean distance matrix which measures the Euclidean distance between points, which can be used as a measure of similarity. After this has been done, the next step is to carry out HCA using this Euclidean distance matrix and plot the resulting dendrogram. The following code is how to do this process:

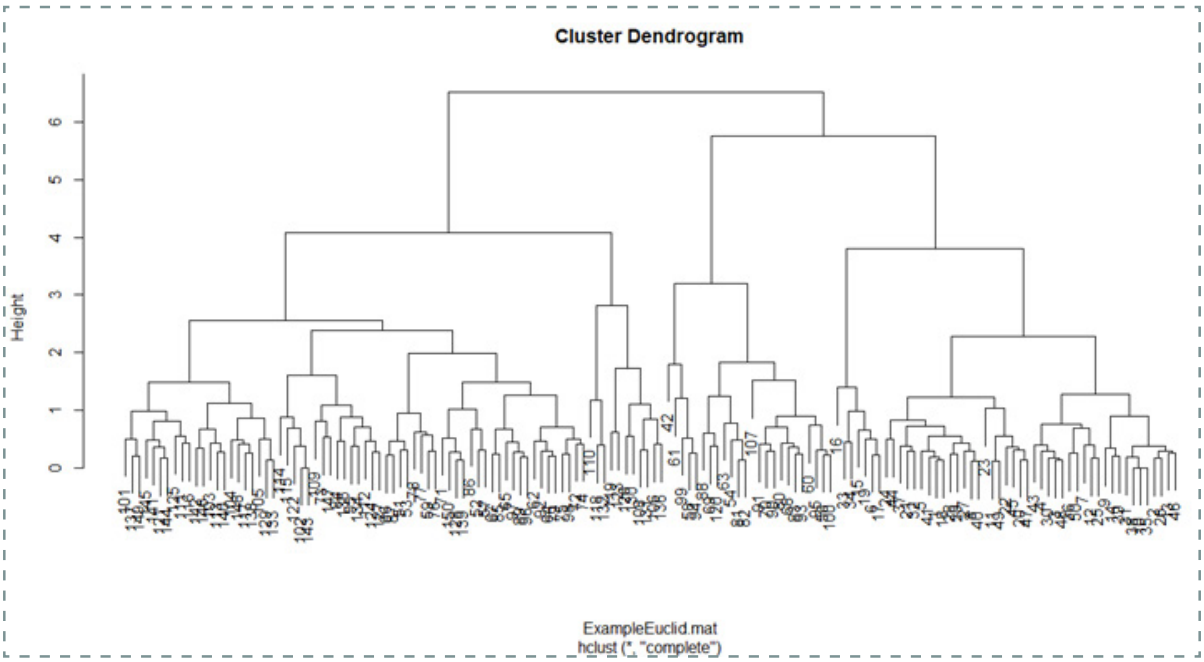
To carry out HCA on data and plot the results, the code is:

```
ExampleMeanCentre.df <- scale(ExampleMultiV.df, center = TRUE,
scale = TRUE)

ExampleEuclid.mat <- dist(ExampleMeanCentre.df)

ExampleEuclid.HCA <- hclust(ExampleEuclid.mat)

plot(ExampleEuclid.HCA)
```

- The term "`scale(...)`" asks R to mean-centre and scale the given dataframe (in this case `ExampleMultiV.df`) and call this new, transformed dataframe a new name (in this case `ExampleMeanCentre.df`).
- The term "`dist(...)`" asks R to calculate the Euclidean distance matrix for the dataframe, in this case "`ExampleMeanCentre.df`", and calls the resulting matrix "`ExampleEuclid.mat`".
- The term "`hclust(...)`" ask R to conduct HCA on the Euclidean matrix "`ExampleEuclid.mat`" and calls the resulting model "`ExampleEuclid.HCA`"
- The term "`plot(...)`" ask R to plot the resulting dendrogram from the HCA model "`ExampleEuclid.HCA`".

The plot of the dendrogram will appear in the bottom right window, available for interpretation and analysis.

The dendrogram for the honey samples is shown in the plot below:



Part III <- Principal component analysis (PCA)

To conduct a proncipal component analysis (PCA) in R is very straightforward and the results and output of the PCA can be generated using a range of commands. Conveniently, instead of needing to mean-centre and scale the data before analysis as we did for HCA, the PCA function `prcomp` does this for you. The following code states how to conduct PCA using R:

To carry out PCA on data and plot and provide the results, the code is:

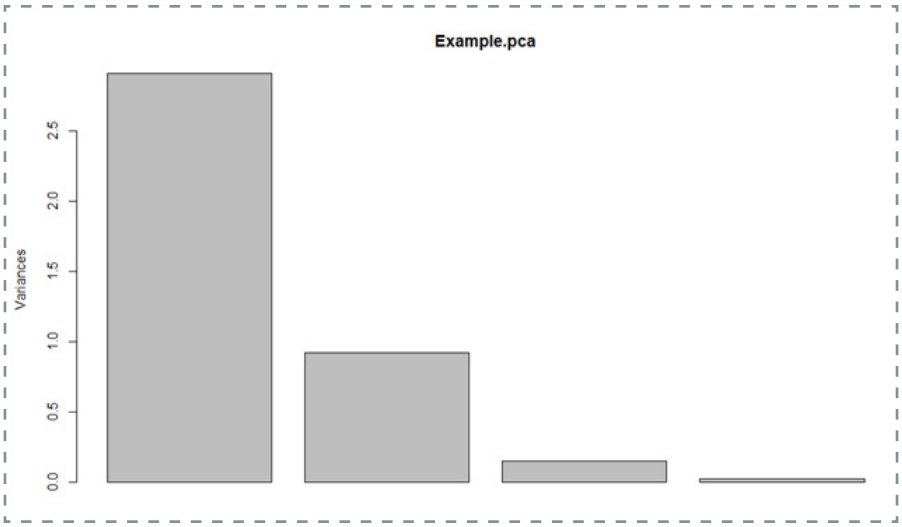
```
Example.PCA <- prcomp(ExampleMultiV.df, center = TRUE, scale = TRUE)
print(Example.PCA)
summary(Example.PCA)
screeplot(Example.PCA)
biplot(Example.PCA, scale = 0)
```

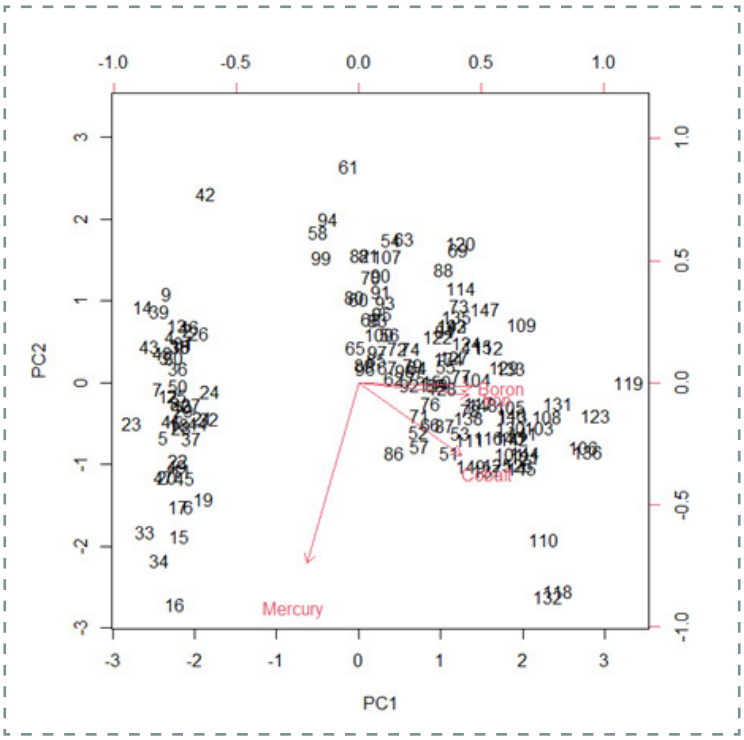
- The term "`prcomp(...)`" asks R to first mean-centre and scale the data in the dataframe "`ExampleMultiV.df`", then conduct PCA and call the resulting model "`Example.PCA`".
- The terms "`print(...)`" and "`summary(...)`" provides details about the PCA model, including the PC's (eigenvectors) and variance explained (eigenvalues) for the PCA model, in this case "`Example.PCA`".
- The term "`screeplot(...)`" asks R to plot the screeplot for the PCA model (in this case called "`Example.PCA`").
- The term "`biplot(...)`" ask R to plot the biplot for the PCA model (in this case called "`Example.PCA`").

t i p s

- One of the most common errors that might come up when you run your PCA is that there are non-varying/constant columns that are preventing analysis. The error will appear in the console when you try to run the code. If this does occur, inspect the data set that you have loaded in and check to see what the issue could be. This can often be due to a importing error where some of the columns are just blank "NA" cells, but otherwise it is likely you have a column that just has constant values and does not change for any of the observations. If this is the case, go back to the original .csv file, remove that column, and reimport the data.

The plots from conducting PCA analysis on the honey samples can be found below:





The graphs produced using the above code are sufficient, but their look can be improved by using an alternative graphing package – see Section 4 Part IV for more information on how to do this.

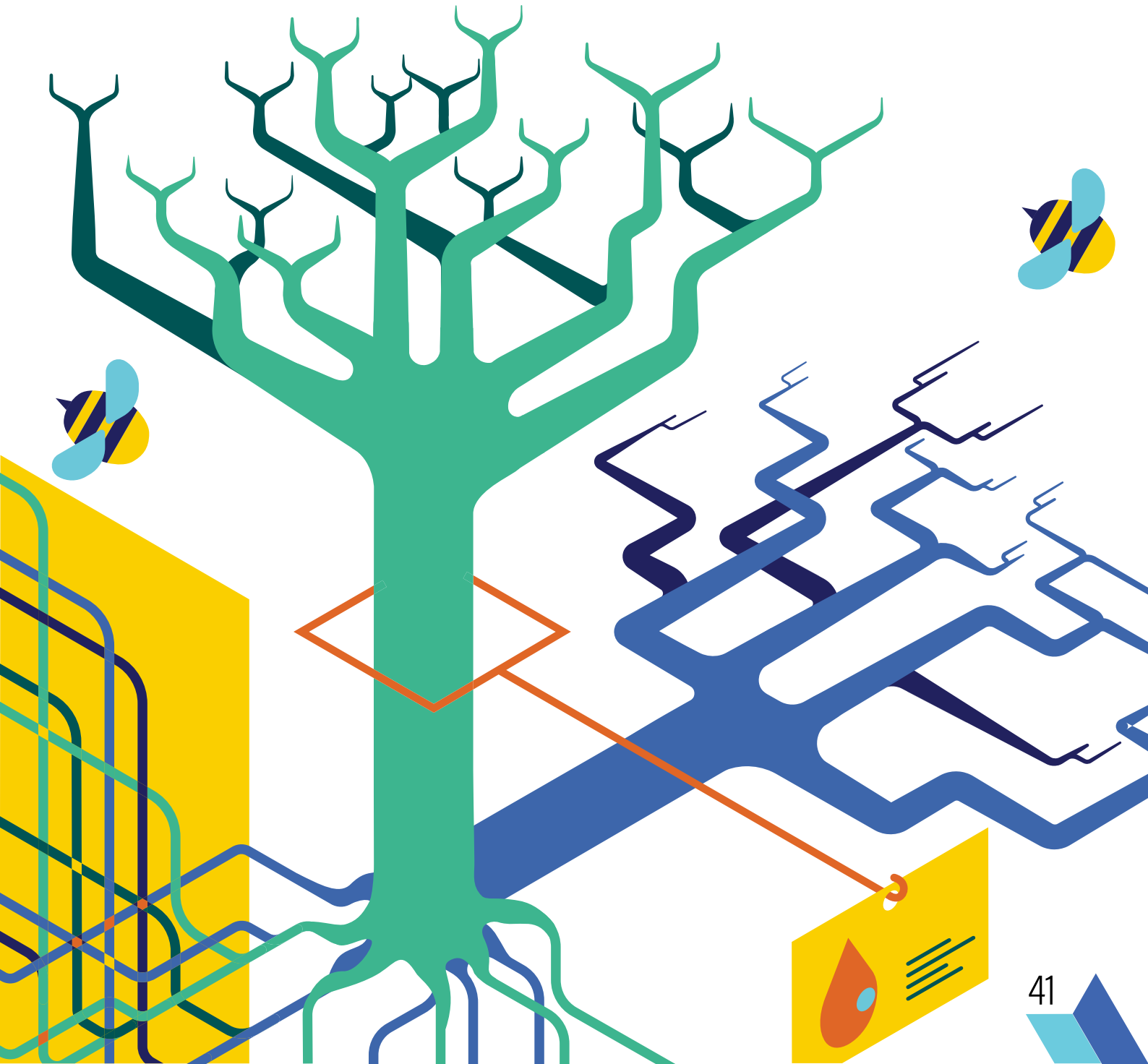
Section 3B:

Using R for Supervised Learning

Supervised machine learning (supervised pattern recognition) is concerned with identifying patterns when we know class membership as well as classifying unknowns into a class.

A video to accompany the notes given in this section, given below, can be found here:

>Click Here<



Part I ← Creating training and test sets

For data sets for supervised learning, you are often given two data sets – a training set and a test set. Both of these sets have a column which states the response (i.e. in most cases the class) that each of the samples has (belongs to for a classification situation). This response is often the first column of the data set.

When you conduct supervised learning techniques such as random forest, kNN and discriminant analysis, you need to have the response/class and explanatory variables, separate. Therefore, the first thing you need to do when you load in your data is to split the dataframe into the explanatory variables and the response variable (i.e. class). This can be done by the following:

To split a dataset into the explanatory variable and the response, the code is:

```
ExampleDataExplanatory.df <- ExampleMultiV.df[,-c(1)]
ExampleDataResponse <- as.factor(ExampleMultiV.df[,1])
```

- The term "`...[,-c(1)]`" asks R to make a new data frame (in this case `ExampleDataExplanatory.df`) that is the same as the original data frame (in this case "`ExampleMultiV.df`"), without the first column. This new dataframe contains only the explanatory variables.
- The term "`as.factor(...[,1])`" asks R to make a new data structure (in this case `ExampleDataResponse`) that is just the first column of the original data frame (in this case "`ExampleMultiV.df`"). This new data structure is actually a vector and is your response/class that should be defined as a factor.

You will need to do this for both the training data set and the test dataset.

t i p s

- Do not call your training and test data sets the same name when you import them into R!!! Be descriptive with your data set names – this will help you in the long run.

Part II ← Random forest models

When fitting a random forest model, you need to make sure that you have first installed the randomForest package. Once this has been installed, you then need to load it. Once you have done this, it is very straightforward to fit the random forest model to your training data, as well as generate the output and plot the most important variables. All of this can be done, as follows:

To fit a random forest model to a training set of data, the code is:

```
install.packages("randomForest")

require("randomForest")

ExampleModel.rf<-randomForest(x=TrainingDataExplanatory.df,
y = TrainingDataResponse)

print(ExampleModel.rf)

varImpPlot(ExampleModel.rf)
```

- The line "`install.packages("randomForest")`" installs the "`randomForest`" package in R – this only needs to be done once on a device.
- The line "`require("randomForest")`" loads the "`randomForest`" package in R – this needs to be done every time you open R and use the package.
- The term "`randomForest(...)`" instructs R to create a random forest model using your training data, calling the random forest model the name you assign it (in this case `ExampleModel.rf`).
  - "`x=...`" specifies the name of the dataframe of the training set explanatory variables (in this case called `TrainingDataExplanatory.df`).
  - "`y=...`" specifies the name of the dataframe/vector of the training set response/classes (in this case called `TrainingDataResponse`).
- The term "`print(...)`" asks R to print the output and information about the random forest model (in this case called `ExampleModel.rf`).
- The term "`varImpPlot(...)`" asks R to plot a graph to show the most important variables in the random forest model (in this case called `ExampleModel.rf`).

After the random forest model has been fit to your training data, you can then assess how good the model is by testing it with your test data explanatory variables and comparing the predicted class with the actual class. This can be done by the following:

To test a random forest model on a test set of data, the code is:

```
TestDataRfPredictions <- predict(ExampleModel.rf, newdata =
TestDataExplanatory.df, type = "class")

print(table(TestDataResponse, TestDataRfPredictions))
```

- The term "`predict(...)`" instructs R to predict the class of the test data based on the random forest model (in this case `ExampleModel.rf`), using the explanatory variables for the test data (in this case `TestDataExplanatory.df`). R calls the predictions for the test set the name you assign it (in this case `TestDataRfPredictions`).
- The term "`print(table(...))`" instructs R to print the confusion matrix for the test set of data (i.e. a table of the results comparing the actual class (in this case `TestDataResponse`) and the predicted class using the random forest model (in this case `TestDataRfPredictions`).

Part III ← kNN (k Nearest Neighbours) analysis

When using k Nearest Neighbours (kNN) analysis to predict the class of the training data, you need to make sure that you have first installed the class package. Once this has been installed, you then need to load it. After this, it is very straightforward to use kNN with your training data, to predict the class of a test set and then assess the results. All of this can be done, as follows:

To analyse your test data using kNN training data, the code is:

```
install.packages("class")

require("class")

TestDataKnnPredictions <- knn(train = TrainingDataExplanatory.df,
test = TestDataExplanatory.df, cl = TrainingDataResponse, k = 6)

print(table(TestDataResponse, TestDataKnnPredictions))
```

- The line "`install.packages("class")`" installs the "`class`" package in R – this only needs to be done once on a device.
- The line "`require("class")`" loads the "`class`" package in R – this needs to be done every time you open R and use the package
- The term "`knn(...)`" instructs R to conduct a kNN analysis using your training data. R conducts this analysis to predict the class of the test set of data (this is the results of the command, in this case `TestDataKnnPredictions`).
  - "`train = ...`" specifies the dataframe of the training set explanatory variables (in this case called `TrainingDataExplanatory.df`).
  - "`test = ...`" specifies the explanatory variable data frame for the test set (in this case `TestDataExplanatory.df`).
  - "`cl = ...`" specifies the dataframe/vector of the training set response/classes (in this case called `TrainingDataResponse`).
  - "`k = ...`" specifies the number of nearest neighbours used in the class assignment by the model (in this case, 6).
- The term "`print(table(...))`" instructs R to print the confusion matrix for the test set of data (i.e. a table of the results comparing the actual class (in this case `TestDataResponse`) and the predicted class using the kNN analysis (in this case `TestDataKnnPredictions`).

Part IV <- Discriminant analysis

Discriminant analysis (DA) is another method of supervised learning that can be used to predict the probability of a sample belonging to a class/category based on its explanatory variables.

Commonly, linear discriminant analysis (LDA) is used but this technique requires certain assumptions to be met. A more flexible (but more complex) variant is quadratic discriminant analysis (QDA).

Before you conduct DA, you need to mean-centre and scale both the training and test explanatory variables. This can be done in the same was as you saw for HCA, but make sure to mean-centre and scale the training and test explanatory variable dataframes together. You can combine the explanatory training and test variables, do the mean centering, and separate the training and test dataframes, as shown below:

To mean-centre and scale the training and test data, the code is:

```
AllDataExplanatory.df    <-    rbind(TrainingDataExplanatory.df,
TestDataExplanatory.df)

AllDataExplanatoryMS.df <- scale(AllDataExplanatory.df, center =
TRUE, scale = TRUE)

TrainingDataExplanatory.df<-AllDataExplanatoryMS.df[c(1:120),]

TestDataExplanatory.df <- AllDataExplanatoryMS.df[-c(1:120),]
```

- The term "`rbind(...,...)`" combines the two data frames with explanatory variables, row-wise (in this case `TrainingDataExplanatory.df` and `TestDataExplanatory.df`)
- The term "`scale(...)`" asks R to mean-centre and scale the given dataframe (in this case `AllDataExplanatory.df`) and call this new, transformed dataframe a new name (in this case `AllDataExplanatoryMS.df`).
- The lines "`...dataframe[c(1:120),]`" asks R to split the mean-centered and scaled data frame back into its training and test sets.
  - "`[c(1:120),]`" specifies the rows of the combined data frame that correspond to the training data set. In this case, it is rows 1 to 120 – the number 120 will change depending on the number of observations/samples are in the training set (for the example data, there were 120 observations/samples in the training set).
  - "`[-c(1:120),]`" specifies the rows of the combined data frame that correspond to the test data set. In this case, it is all rows except 1 to 120 – the number 120 will change depending on the number of observations/samples are in the training set (for the example data, there were 120 observations/samples in the training set).

When using a DA model to predict the class of the training data, you need to make sure that you have first installed the mass package. Once this has been installed, you then need to load it. Once you have done this, it is very straightforward to build an LDA/QDA model with your training data, to predict the class of a test set and then assess the results.

First looking at creating an LDA model:

To carry out LDA on data and plot the results, the code is:

```
install.packages("MASS")

require("MASS")

ExampleModel.lda <- lda(x = TrainingDataExplanatoryMS.df,
grouping = TrainingDataResponse)

print(ExampleModel.lda)

plot(ExampleModel.lda)
```

- The line "`install.packages("MASS")`" installs the "`MASS`" package in R – this only needs to be done once on a device.
- The line "`require("MASS")`" loads the "`MASS`" package in R – this needs to be done every time you open R and use the package
- The term "`lda(...)`" instructs R to create a linear discriminant model on your training data (in this case calling the model `ExampleModel.lda`).
  - "`x = ...`" specifies the dataframe of the training set explanatory variables (in this case called `TrainingDataExplanatoryMS.df`).
  - "`grouping = ...`" specifies the dataframe/vector of the training set response/classes (in this case called `TrainingDataResponse`).
- The term "`print(...)`" asks R to print the output and information about the LDA model (in this case called `ExampleModel.lda`).
- The term "`plot(...)`" asks R to plot a graph plot the results of the generated LDA model (in this case called `ExampleModel.lda`).

Like for the random forest model, once you create the LDA model, the next thing to do is to test its performance using the test data set. This can be done by doing the following:

To test an LDA model on a test set of data, the code is:

```
TestDataLdaPredictions <- predict(ExampleModel.lda, newdata =
TestDataExplanatoryMS.df)

print(table(TestDataResponse, TestDataLdaPredictions
[["class"]]))
```

- The term "`predict(...)`" instructs R to predict the class of the test data based on the LDA model (in this case `ExampleModel.lda`), using the explanatory variables for the test data (in this case `TestDataExplanatoryMS.df`). R calls the predictions for the test set the name you assign it (in this case `TestDataLdaPredictions`).
- The term "`print(table(...))`" instructs R to print the confusion matrix for the test set of data (i.e. a table of the results comparing the actual class (in this case `TestDataResponse`) and the predicted class using the LDA model (in this case `TestDataLdaPredictions`).

The same process as described above can be carried out to perform quadratic discriminant analysis (QDA), with the only difference being the type of model you fit – instead of "lda", writing "qda", i.e.:

To carry out QDA on data and plot the results, the code is:

```
require("MASS")

ExampleModel.qda <- qda(x = TrainingDataExplanatoryMS.df,
grouping = TrainingDataResponse)

print(ExampleModel.qda)

TestDataQdaPredictions <- predict(ExampleModel.qda, newdata =
TestDataExplanatoryMS.df)

print(table(TestDataResponse, TestDataQdaPredictions
[["class"]]))
```

- The line "`require("MASS")`" loads the "MASS" package in R – this needs to be done every time you open R and use the package.
- The term "`qda(...)`" instructs R to create a quadratic discriminant model on your training data and calls the QDA model the name you assign it (in this case `ExampleModel.qda`).
  - "`x = ...`" specifies the dataframe of the training set explanatory variables (in this case called `TrainingDataExplanatoryMS.df`).
  - "`grouping = ...`" specifies the vector of the training set response/classes (in this case called `TrainingDataResponse`).
- The term "`print(...)`" asks R to print the output and information about the QDA model (in this case called `ExampleModel.qda`).
- The term "`predict(...)`" instructs R to predict the class of the test data based on the QDA model (in this case `ExampleModel.qda`), using the explanatory variables for the test data (in this case `TestDataExplanatoryMS.df`). R calls the predictions for the test set the name you assign it (in this case `TestDataQdaPredictions`).
- The term "`print(table(...))`" instructs R to print the confusion matrix for the test set of data (i.e. a table of the results comparing the actual class (in this case `TestDataResponse`) and the predicted class using the QDA model (in this case `TestDataQdaPredictions`).

Section 4:

Advanced Graphing/ Visual Representation

An alternative to graphing data using the basic "stats" package is to use alternative packages including "ggplot2". This package is extremely popular due to the aesthetics of the graphs that it produces. Like other packages, ggplot2 must be installed prior to use and then loaded each time you open your session.

A video to accompany the notes given in this section, given below, can be found here:

›Click Here‹ #

Part I ← ggplot2 package – one dimensional plots revisited

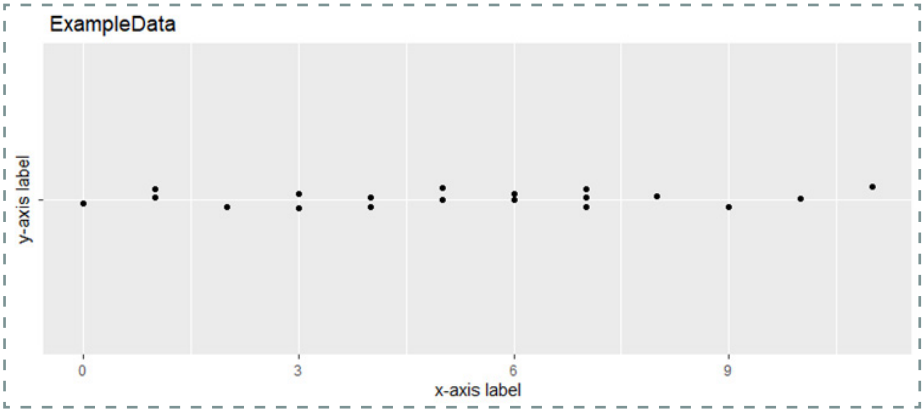
First, we will look at how to produce variants of the one dimensional plots introduced in Section 2B, Part I. ggplot2 works with dataframes, so even when you have one-dimensional data (i.e. a vector), you need to convert it to a dataframe (of only one variable) for ggplot2 to work with. Once you do this and load that ggplot2 package, the plot can be easily produced. This is exemplified with stripcharts/dotplots as follows on the next page.

To plot a stripchart with ggplot2, the code is:

```
ExampleData.df <- data.frame(ExampleData)

require("ggplot2")

ggplot(ExampleData.df, aes(x = ExampleData, y = "")) +
geom_jitter(shape = 19, width = 0, height = 0.05) +
labs(title = "ExampleData", x = "x-axis label", y = "y-axis
label")
```

- The term "`data.frame(...)`" instructs R to create a data frame (in this case called `ExampleData.df`) from the supplied vector (in this case the vector is called `ExampleData`).
- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`ggplot(...)`" instructs R and ggplot2 to create a one-dimensional dotplot of data from the supplied dataframe (in this case the dataframe called `ExampleData.df`).
  - "`aes(...)`" says what columns in the dataframe should be plotted.
    - "`x = ...`" says the x-axis corresponds to the values of the data in the dataframe – this is the name of the column in the dataframe (in this case, `ExampleData`).
    - "`y = ...`" says the y axis is blank which is noted by "".
  - "`geom _ jitter(...)`" describes the points that are plotted.
    - "`shape = 19`" states the type of symbol/dot that should be used in the plot – these are the same symbols as shown earlier in Section 2A.
    - The `height` and `width` values should not be changed from those stated in this example – these values says that points that are the same value should not overlap, but instead should be separate and gives parameters for this.
  - "`labs(...)`" states what the main and axis titles/labels should be.
    - "`title = "ExampleData"`" says what the main title of the graph should be (in this case I have titled it simply `ExampleData`).
    - "`x = "x-axis label"`" says what the x-axis label of the graph should be (in this case I have titled it simply `x-axis label`).
    - "`y = "y-axis label"`" says what the y-axis label of the graph should be (in this case I have titled it simply `y-axis label`).

The graph you will get will look like this:



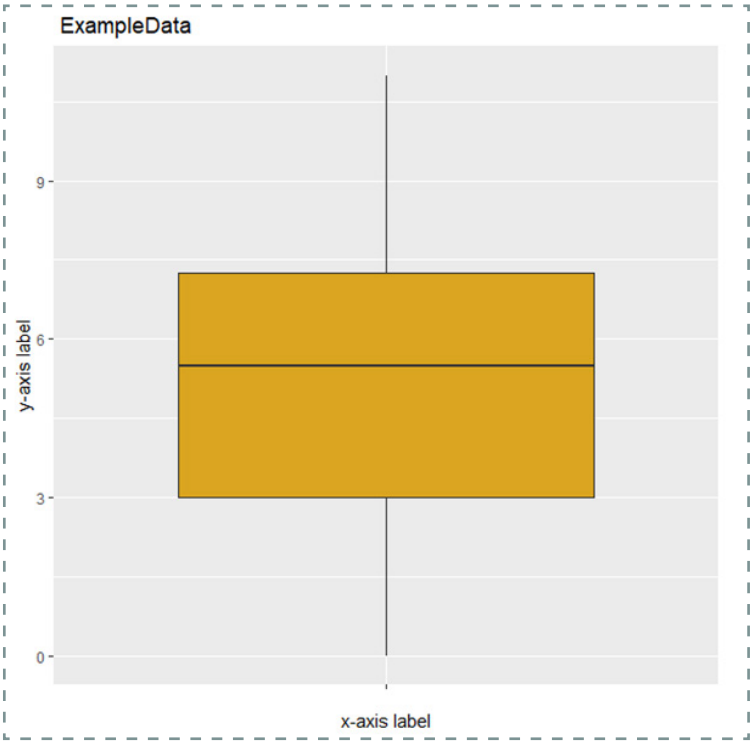
Dot plots/stripcharts are good for smaller data sets, but larger sample sizes are best represented using either boxplots or barplots (or density plots in some cases) – see the code and example output for producing these plots, below.

To plot a boxplot with ggplot2, the code is:

```
ExampleData.df <- data.frame(ExampleData)

require("ggplot2")

ggplot(ExampleData.df, aes(y = ExampleData)) + geom_boxplot
(fill = "goldenrod") + labs(title = "ExampleData", x =
"x-axis label", y = "y-axis label")
```

- The term "`data.frame(...)`" instructs R to create a data frame (in this case called `ExampleData.df`) from the supplied vector (in this case the vector is called `ExampleData`).
- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package
- The term "`ggplot(...)`" instructs R and ggplot2 to create a boxplot of data from the supplied data frame (in this case the dataframe is called `ExampleData.df`).
  - "`aes(...)`" specifies what column in the dataframe should be plotted.
    - "`y = ...`" specifies the y-axis corresponds to the values of the data in the dataframe – this is the name of the column in the dataframe (in this case, `ExampleData`)
  - "`geom _ boxplot(...)`" describes the box that is to be plotted.
    - "`fill = "goldenrod"`" specifies the colour that the box should be filled with – options for colours are the same as those described earlier in Section 2A.
  - "`labs(...)`" is as described on page 49.

The code above gives a boxplot that looks like this, using the example data:

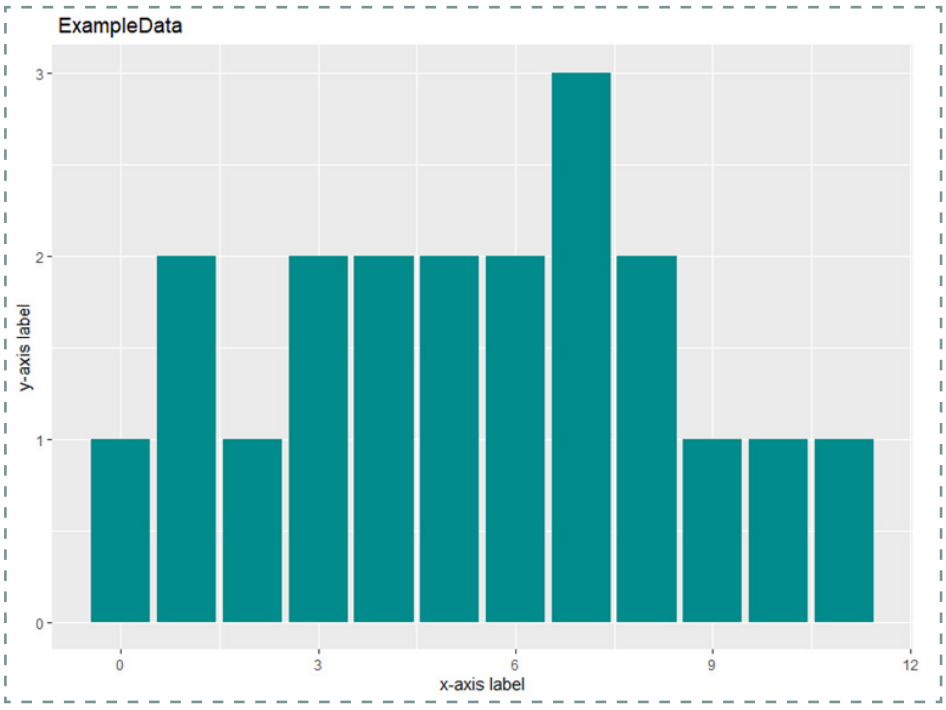


To plot a barplot with ggplot2, the code is:

```
ExampleData.df <- data.frame(ExampleData)

require("ggplot2")

ggplot(ExampleData.df, aes(x = ExampleData)) + geom_bar(fill
= "darkcyan") + labs(title = "ExampleData", x = "x-axis
label", y = "y-axis label")
```

- The term "`data.frame(...)`" instructs R to create a data frame (in this case called `ExampleData.df`) from the supplied vector (in this case the vector is called `ExampleData`).
- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package
- The term "`ggplot(...)`" instructs R and ggplot2 to create a barplot of data from the supplied data frame (in this case the dataframe is called `ExampleData.df`).
  - "`aes(...)`" specifies what columns in the dataframe should be plotted.
    - "`x = ...`" specifies the x-axis corresponds to the values of the data in the dataframe – this is the name of the column in the dataframe (in this case, `ExampleData`).
  - "`geom_bar(...)`" describes the look of the bars that should be plotted.
    - "`fill = "darkcyan"`" specifies the colour that the box should be filled with – options for these are the same as those described earlier in Section 2A.
  - "`labs(...)`" is as described on page 49.

The code above gives a barplot (using the example data) that looks like this:



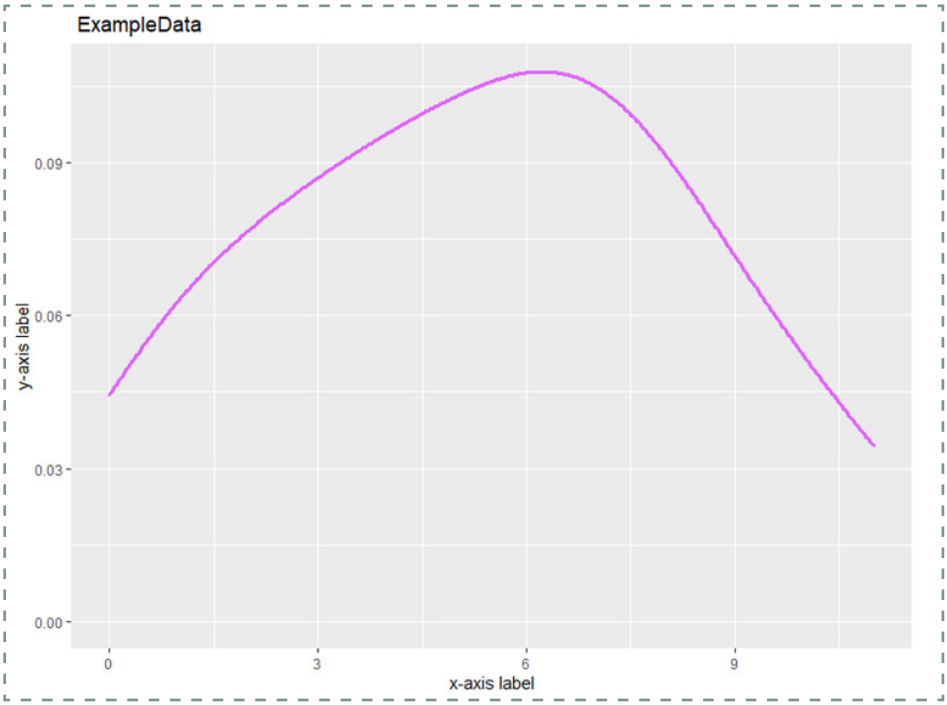
Density plots can also be produced using ggplot2 by implementing the following code:

To plot a density plot with ggplot2, the code is:

```
ExampleData.df <- data.frame(ExampleData)

require("ggplot2")

ggplot(ExampleData.df, aes(x = ExampleData)) +
geom_density(colour = "mediumorchid1", size = 1.2) +
labs(title = "ExampleData", x = "x-axis label", y = "y-axis
label")
```

- The term "`data.frame(...)`" instructs R to create a data frame (in this case called `ExampleData.df`) from the supplied vector (in this case the vector is called `ExampleData`).
- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`ggplot(...)`" instructs R and ggplot2 to create a density plot of data from the supplied data frame (in this case the dataframe is called `ExampleData.df`).
  - "`aes(...)`" specifies what columns in the dataframe should be plotted.
    - "`x = ...`" specifies the x-axis corresponds to the values of the data in the dataframe – this is the name of the column in the dataframe (in this case, `ExampleData`).
  - "`geom _ density(...)`" describes the look of the density plot and line that should be plotted.
    - "`colour = "mediumorchid1""` specifies the colour that the line should – options for these are the same as those described earlier in Section 2A.
    - "`size = 1.2`" specifies the width of the line on the graph.
  - "`labs(...)`" is as described on page 49.

This produces the following density plot, using the example data:



Part II <- ggplot2 package - side-by-side plots revisited

The ggplot2 package can also be used to produce side-by-side plots like those that were produced in Section 2A Part II. As mentioned above, the ggplot2 package best likes data in dataframes. Moreover, there is a specific way to format the dataframes that makes it best to plot – that is with all the values in one column and their grouping factor is in the other. This is the way that the example data frame for the data below was set up in Section 1 Part V.

sugar content (measurement)	supplier (i.e. conditition)
85.4	A
86.9	A
89.1	A
88.4	A
87.3	A
88.7	A
90.3	B
85.4	B
88.2	B
81.0	B
79.3	B
87.7	B
83.1	C
82.4	C
81.0	C
78.7	C
79.5	C
82.0	C

Side-by-side dot plots can be produced in a very similar way to single dot plots, but by also specifying the grouping factor for which you want separate dot plots to be produced by.

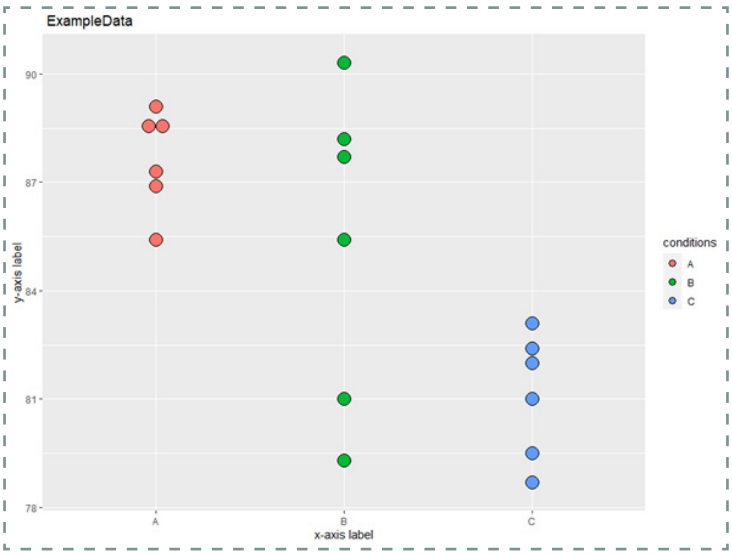
The names of the vectors used in the example code below are purposefully non-specific to make it easier for you to apply the code to your own question/situation. In theory, I should have named the "`measurement`" vector as "`sugarContent`" and the "`condition`" vector as "`supplier`" to best represent the data.

To plot a side-by-side dotplots with ggplot2, the code is:

```
require("ggplot2")

ggplot(ExampleData.df, aes(x = conditions, y = measurements,
fill = conditions)) + geom_dotplot(binaxis = 'y',
stackdir = 'center') + labs(title = "ExampleData", x =
"x-axis label", y = "y-axis label")
```

- The line "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`ggplot(...)`" instructs R and ggplot2 to create a side-by-side dotplot of data from the supplied data frame (in this case the dataframe called `ExampleData.df`).
  - "`aes(...)`" says what values in the dataframe should be on the x and y axis and how the dots should be coloured.
    - "`x = ...`" specifies the x-axis corresponds to the grouping variable - this is the name of the column in the dataframe (in this case, `conditions`).
    - "`y = ...`" specifies the y-axis corresponds to the measurement values - this is the name of the column in the dataframe (in this case, `measurements`).
    - "`fill = ...`" specifies the dots to be coloured according to the group (in this case, `conditions`). This term is optional and can be removed if you want all dots to be the same colour.
  - "`geom_dotplot(...)`" describes the way the points should be plotted.
    - "`binaxis = ...`" and "`stackdir = ...`" should not be changed from those stated in this example.
  - "`labs(...)`" is as described on page 49.

The graph produced using the above code and example data is given below. Here you can see that separate, side-by-side dot-plots have been plotted for each level of the grouping factor (in this case, "`conditions`". The dots have also been filled according to their group as an additional feature.



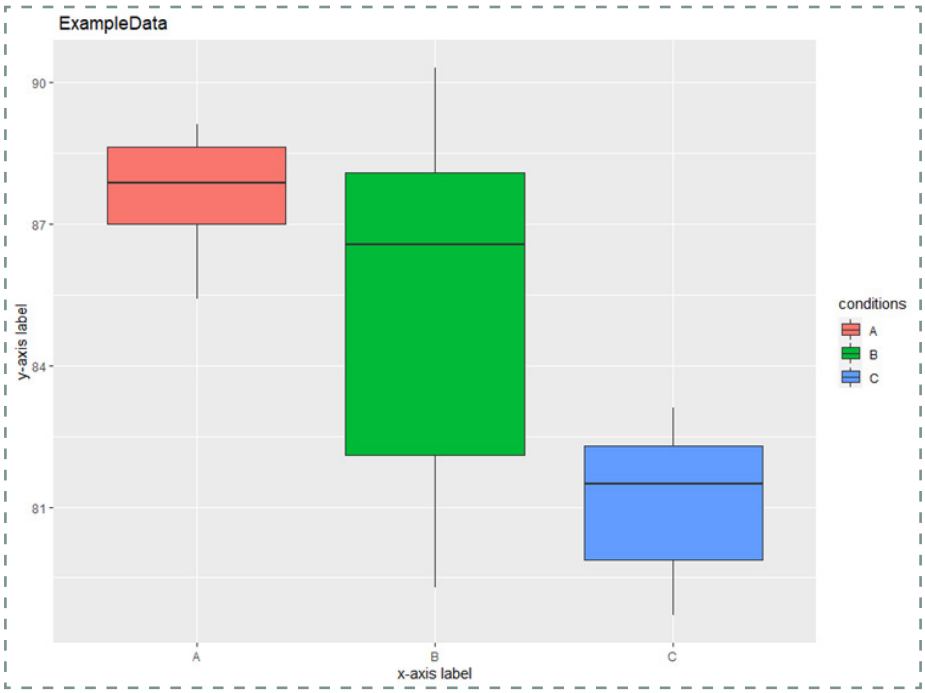
Side-by-side boxplots can also be produced in a very similar way, by using the following code:

To plot side-by-side boxplots with ggplot2, the code is:

```
require("ggplot2")

ggplot(ExampleData.df, aes(x = conditions, y = measurements,
fill = conditions)) + geom_boxplot() + labs(title =
"ExampleData", x = "x-axis label", y = "y-axis label")
```

- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`ggplot(...)`" instructs R and ggplot2 to create a side-by-side boxplot of the supplied data frame (in this case the dataframe is `ExampleData.df`).
  - "`aes(...)`" says what values in the dataframe should be on the x and y axis and how the boxes should be coloured.
    - "`x = ...`" specifies the x-axis corresponds to the grouping variable - this is the name of the column in the dataframe (in this case, `conditions`).
    - "`y = ...`" specifies the y-axis corresponds to the measurement values - this is the name of the column in the dataframe (in this case, `measurements`).
    - "`fill = ...`" specifies the boxes to be coloured according to the group (in this case, `conditions`). This term is optional and can be removed if you want all boxes to be the same colour.
  - "`geom _ boxplot()`" describes that a boxplot should be plotted.
  - "`labs(...)`" is as described on page 49.

The graph produced using the above code and example data is given below. Here you can see that separate, side-by-side boxplots have been plotted for each level of the grouping factor (in this case, "`conditions`"). The boxes have also been filled according to their group as an additional feature.



Part III ggplot2 package – Regression revisited

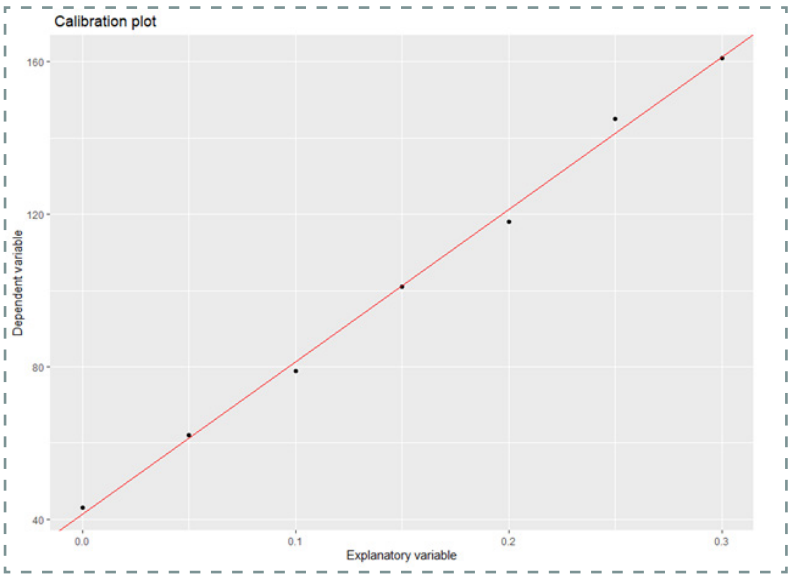
ggplot2 can also be used to plot linear models and calibration curves to show the relationship between two quantitative variables (one independent/explanatory and the other dependent). This can be done by using the following code:

To graph a linear model and calibration curve with ggplot2, the code is:

```
require("ggplot2")

ggplot(ExampleData.df, aes(x = xvariable, y = yvariable)) +
geom_point (shape = 19) + labs(title = "Calibration plot",
x = "Explanatory variable", y = "Dependent variable") +
geom_abline(intercept = 41.4, slope = 399.3, color="red")
```

- The term "`require("ggplot2")`" loads the "`ggplot2`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`ggplot(...)`" instructs R and ggplot2 to create a scatter plot of the data from the supplied dataframe (in this case the dataframe is called `ExampleData.df`).
  - "`aes(...)`" specifies what values in the dataframe should be plotted.
    - "`x = ...`" specifies the x-axis corresponds to the explanatory/ independent variable - this is the name of the column in the dataframe (in this case, `xvariable`).
    - "`y = ...`" says the y-axis corresponds to the dependent variable - this is the name of the column in the dataframe (in this case, `yvariable`).
  - "`geom _ point(...)`" describes the points that should be plotted.
    - "`shape = 19`" specifies the type of symbol/dot that should be used in the plot – these are the same symbols as shown earlier in Section 2A.
  - "`labs(...)`" specifies what the main and axis titles/labels should be
    - "`title = "Calibration plot"`" says what the main title of the graph should be (in this case I have titled it `Calibration plot`).
    - "`x = "Explanatory variable"`" says what the x-axis label of the graph should be (in this case I have titled it `Explanatory variable`).
    - "`y = "Dependent variable"`" says what the x-axis label of the graph should be (in this case I have titled it `Dependent variable`).
  - "`geom _ abline(...)`" describes the line of the linear model that should be plotted.
    - "`intercept = 41.4`" states what the y-intercept of the line should be. This value can be obtained from the linear model summary when the model is fit to the data - see Section 2D Part I and Part II.
    - "`slope = 399.3`" states what the slope of the line should be. This value can be obtained from the linear model summary when the model is fit to the data - see Section 2D Part I and Part II.
    - "`color="red"`" states what colour the line should be, in this case red. The options are the same as those described earlier in Section 2A.

This will produce a plot that looks like the following:



Part IV <- Factoextra package – PCA plots revisited

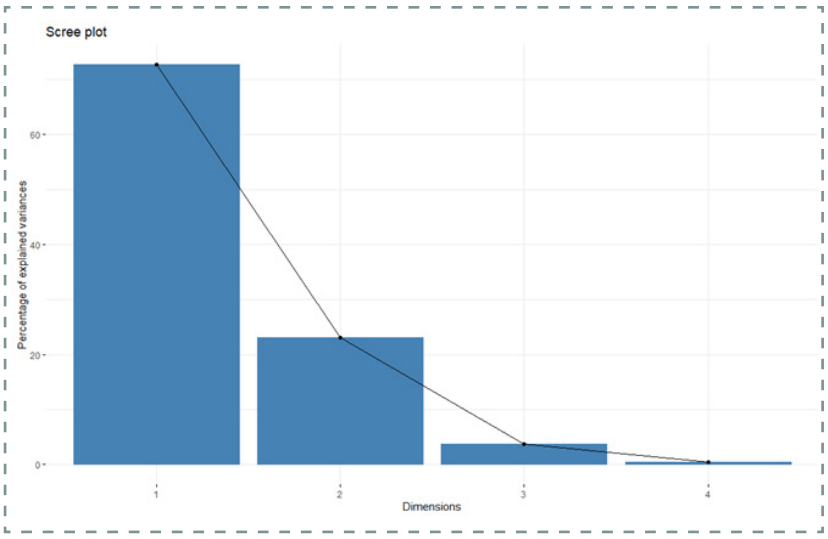
In Section 3A Part III, PCA was covered with the plots produced using the basic stats package. An alternative package to graph PCA output is the factoextra package (which actually uses ggplot2, behind the scenes). Once you have run the PCA using the code shown earlier, you can produce the various plots with factoextra, using the following code:

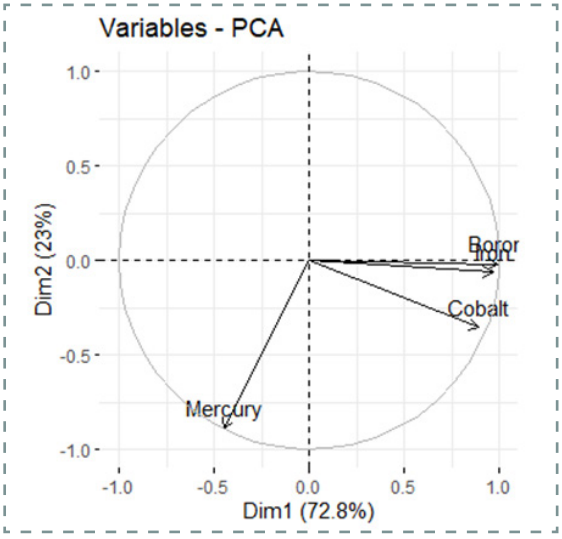
To plot the results of PCA using factoextra, the code is:

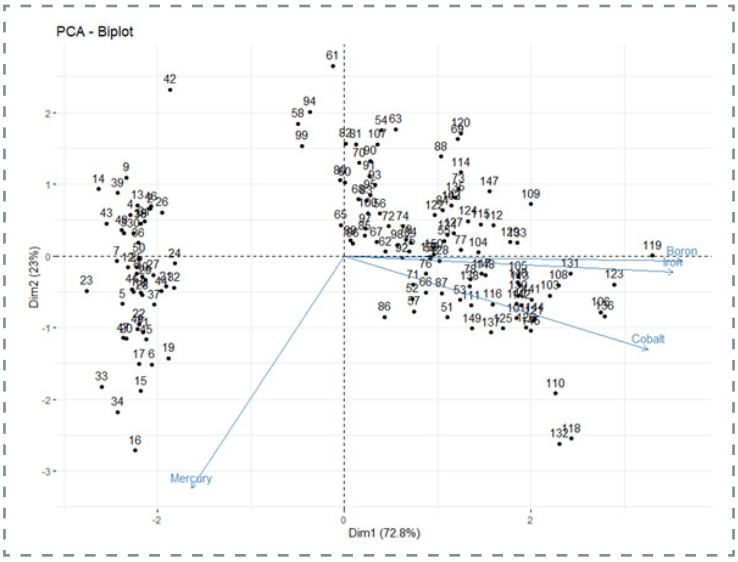
```
require("factoextra")

fviz_screeplot(Example.pca, ncp = 10)
fviz_pca_var(Example.pca)
fviz_pca_biplot(Example.pca)
fviz_pca_ind(Example.pca, geom = c("point", "text"),
pointsize = 3)
```

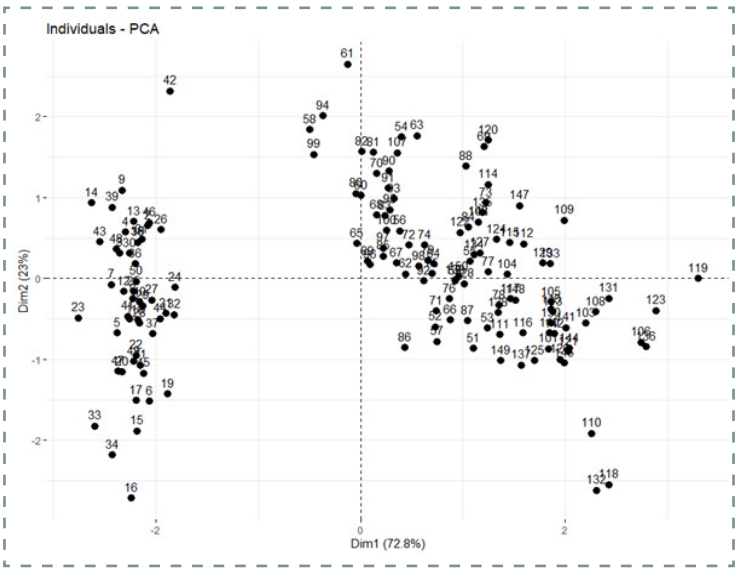
- The term "`require("factoextra")`" loads the "`factoextra`" package in R – this needs to be done **every time** you open R and use the package.
- The term "`fviz_screeplot(...)`" instructs R and factoextra to produce a screeplot for the PCA model "`Example.pca`".
  - "`ncp = 10`" indicates the maximum number of principal components that should be included in the screeplot, in this case 10.
- The term "`fviz_pca_var(...)`" instructs R and factoextra to produce a loadings plot for the PCA model, in this case called `Example.pca`.
- The term "`fviz_pca_biplot(...)`" instructs R and factoextra to produce a biplot for the PCA model, in this case called `Example.pca`.
- The term "`fviz_pca_ind(...)`" instructs R and factoextra to produce a scores plot for the PCA model, in this case called `Example.pca`.
  - "`geom = c(...)`" specifies that there should be labelled points for each sample in the scores plot.
  - The size of the point can be varied by changing the number for "`pointsize = 3`".

Plots for the honey example introduced in Section 3A are given below:









Section 5A:

Bringing it all Together

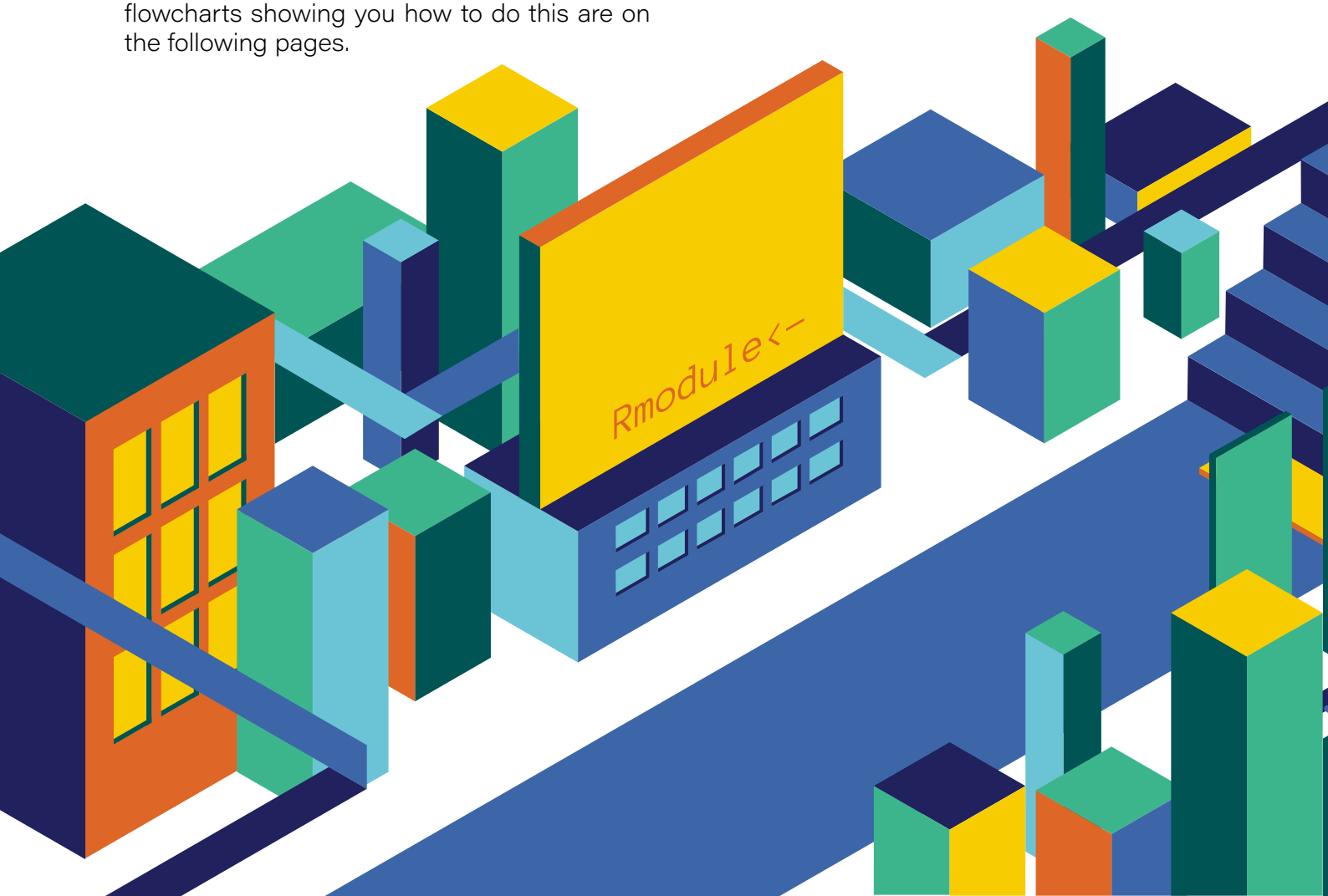
Part I <- Statistics workflows

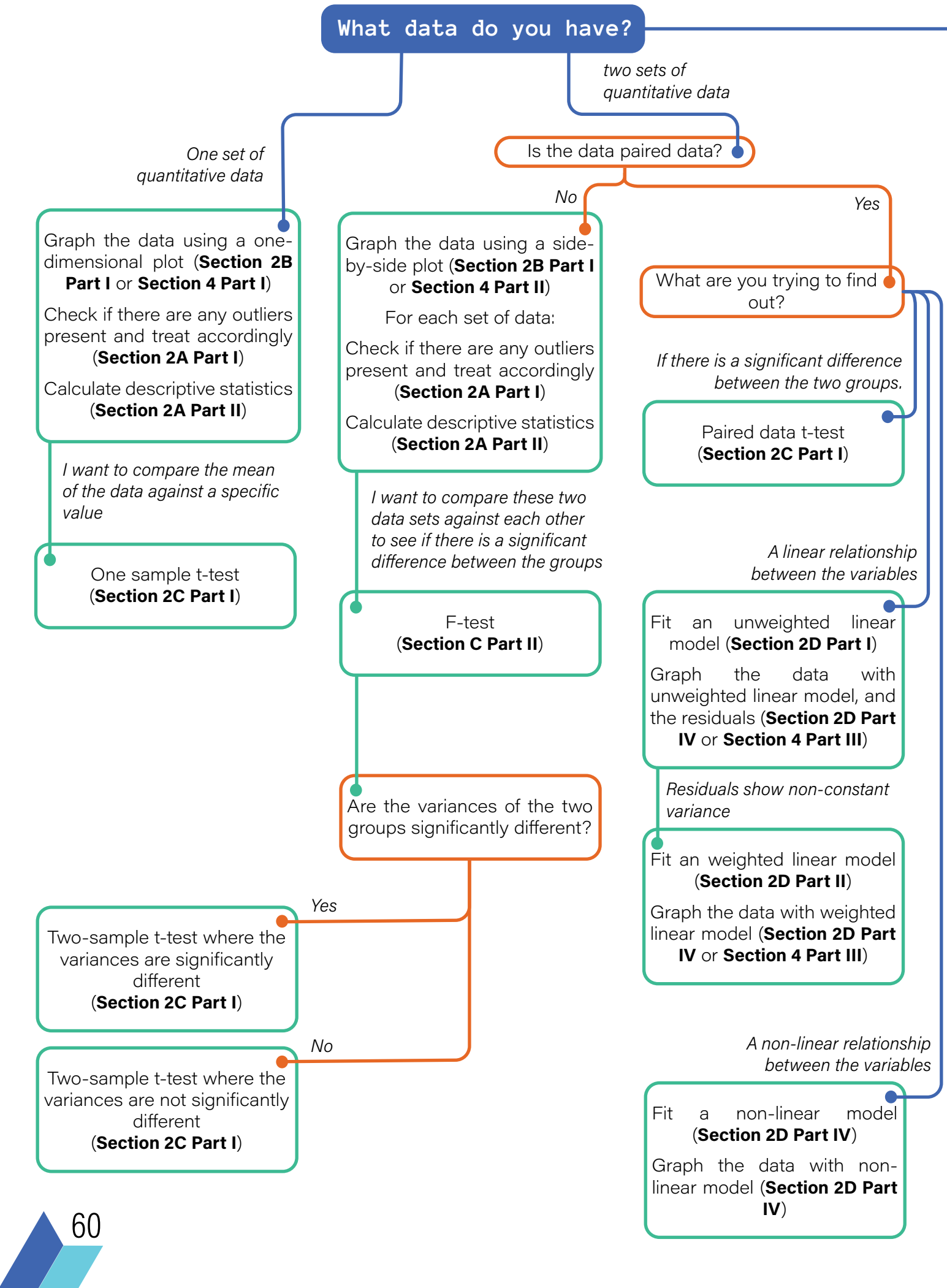
Now that we have gone through the different types of analyses that are possible with R, the next step is applying these methods to your data. Different data and situations do, however, require different type of analyses, so you need to decide on the correct workflow.

The first thing you need to do is to take note of what type of data you have and the second is to know what you are trying to deduce from your data. With this information, you can have a clear path as to what you need to do – flowcharts showing you how to do this are on the following pages.

There are two workflows, one is for if you have one or two sets of data and the other is if you have more than two sets of data/ measurements.

Each of the workflows communicates what sort of graph of the raw data you should produce if you should formally check for outliers, what statistics you should calculate and what sort of subsequent analysis you should be performing. It also refers back to the section of R module that you should consult for each task.




What data do you have?
One set of quantitative data
two sets of quantitative data
Graph the data using a one-dimensional plot (Section 2B Part I or Section 4 Part I)
Check if there are any outliers present and treat accordingly (Section 2A Part I)
Calculate descriptive statistics (Section 2A Part II)
I want to compare the mean of the data against a specific value
One sample t-test (Section 2C Part I)
Is the data paired data?
No
Yes
Graph the data using a side-by-side plot (Section 2B Part I or Section 4 Part II)
For each set of data:
Check if there are any outliers present and treat accordingly (Section 2A Part I)
Calculate descriptive statistics (Section 2A Part II)
I want to compare these two data sets against each other to see if there is a significant difference between the groups
F-test (Section C Part II)
Are the variances of the two groups significantly different?
Yes
Two-sample t-test where the variances are significantly different (Section 2C Part I)
No
Two-sample t-test where the variances are not significantly different (Section 2C Part I)
What are you trying to find out?
If there is a significant difference between the two groups.
Paired data t-test (Section 2C Part I)
A linear relationship between the variables
Fit an unweighted linear model (Section 2D Part I)
Graph the data with unweighted linear model, and the residuals (Section 2D Part IV or Section 4 Part III)
Residuals show non-constant variance
Fit an weighted linear model (Section 2D Part II)
Graph the data with weighted linear model (Section 2D Part IV or Section 4 Part III)
A non-linear relationship between the variables
Fit a non-linear model (Section 2D Part IV)
Graph the data with non-linear model (Section 2D Part IV)

> two sets of quantitative data

What are you trying to find out/do?

I want to compare the data/ measurements against each other to see if there is a significant difference between the data for the groups

Graph the data using a side-by-side plot (**Section 2B Part I** or **Section 4 Part II**)

For each set of data:

Check if there are any outliers present and treat accordingly (**Section 2A Part I**)

Calculate descriptive statistics (**Section 2A Part II**)

ANOVA (**Section 2C Part III**)

I want to simultaneously analyse a number of data/ measurements

Do you know groups/ classes for the data?

No

Unsupervised learning; do one or both of the following:

Conduct HCA on the data set (**Section 3A, Part II**)

Conduct PCA on the data set (**Section 3A, Part III** (and **Section 4 Part IV** for alternative graphing))

Yes

Supervised learning; if not already pre-done, split the data into training and test sets (**Section 3B Part I**), then do one, some or all of the following:

Fit a random forest model to the training set and test using the test set (**Section 3B, Part II**)

Use kNN with the training set and predict the class of the test set (**Section 3B, Part II**)

Fit a DA model to the training set and test using the test set (**Section 3B, Part IV**)

Part II ← Worked examples

The workflow charts in Section 5A Part I help to guide you as to what types of analysis you should do in a given situation. Below are four different workflow examples of how to follow this chart and complete the required analysis in R. The code and output formatting below has been generated using the "Knit Document" process mentioned in Section 1 Part VI.

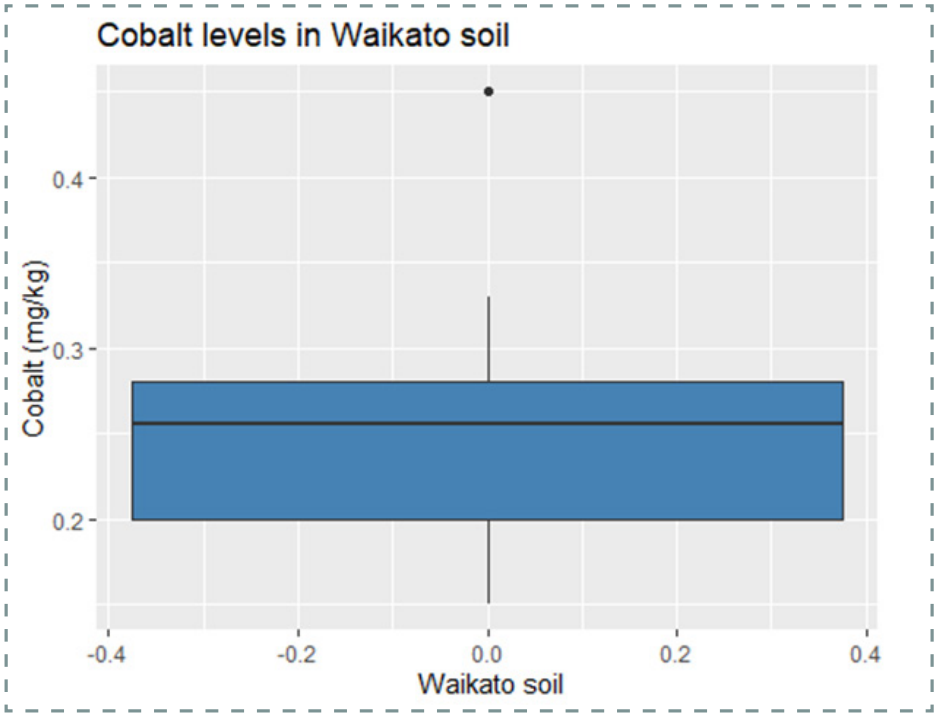
Example 1: Cobalt in soil

The cobalt levels in soils around the Waikato region were measured, giving the results shown in the table below:

Levels of cobalt in the soil (mg/kg)	0.15, 0.20, 0.31, 0.28, 0.26, 0.17, 0.45, 0.33, 0.25, 0.27, 0.22, 0.20, 0.17, 0.19, 0.30, 0.28, 0.24, 0.26

For the health of the sheep, sheep farming should only be where cobalt levels in soil are 0.3 mg/kg so the sheep do not get ill from cobalt deficiency. Determine if the soil in the Waikato region is appropriate for sheep farming.

```
##Loading in the data.

#First loading in the cobalt data

soilCobalt <- c(0.15, 0.20, 0.31, 0.28, 0.26, 0.17, 0.45, 0.33, 0.25, 0.27, 0.22, 0.20, 0.17, 0.19, 0.30, 0.28, 0.24, 0.26 )

##Graphing the data (this example using ggplot2)

#Creating a data frame of the vector as this is what ggplot2 prefers

soilCobalt.df <- data.frame(soilCobalt)

#One-dimension box plot

require("ggplot2")

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 3.6.3

ggplot(soilCobalt.df, aes(y = soilCobalt)) + geom_boxplot(-fill = "steelblue") + labs(title = "Cobalt levels in Waikato soil",x = "Waikato soil", y = "Cobalt (mg/kg)")
```




```
##Need to check for outliers and then calculate descriptive
statistics.

#Checking data for outliers

require("outliers")

## Loading required package: outliers

grubbs.test(soilCobalt)

##

##  Grubbs test for one outlier

##

## data:  soilCobalt

## G = 2.76845, U = 0.52264, p-value = 0.01349

## alternative hypothesis: highest value 0.45 is an outlier

#Results of the outlier test indicate that 0.45 is an outli-
er. Need to remove this from the data set:

soilCobalt <- c(0.15, 0.20, 0.31, 0.28, 0.26, 0.17,0.33,
0.25, 0.27, 0.22, 0.20, 0.17, 0.19, 0.30, 0.28, 0.24, 0.26 )

#Next calculating descriptive statistics for the data

mean(soilCobalt)

## [1] 0.24

sd(soilCobalt)

## [1] 0.05338539
```

```
var(soilCobalt)
## [1] 0.00285
##We want to test the data against the value of 0.3 mg/kg to see if there is a significant difference. Need to conduct a one-sample t-test
t.test(soilCobalt, mu = 0.3, conf.level = 0.95)
##
##  One Sample t-test
##
## data:  soilCobalt
## t = -4.634, df = 16, p-value = 0.0002758
## alternative hypothesis: true mean is not equal to 0.3
## 95 percent confidence interval:
##  0.2125518 0.2674482
## sample estimates:
## mean of x
##      0.24
```

Example 2: Uric acid in the blood

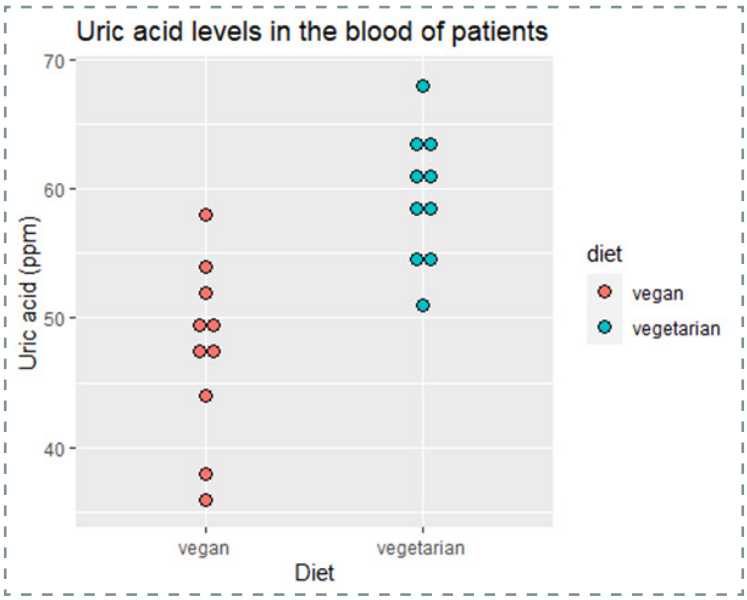
The levels of uric acid in human blood were measured in samples from vegetarian and vegan patients (10 people in each group), giving the results shown in the table below:

	Vegetarian	Vegan
Levels of uric acid in the blood (ppm)	54, 63, 58, 59, 61, 55, 61, 51, 68, 64	48, 52, 38, 36, 44, 54, 58, 49, 47, 50

Determine if there is a significant difference in levels of uric acid in vegetarian and vegan patients.

```
##Loading in the data.
#First loading in all Uric acid measurements as one vector and diet as the other, then creating a dataframe
UricAcid <- c(54, 63, 58, 59, 61, 55, 61, 51, 68, 64, 48, 52, 38, 36, 44, 54, 58, 49, 47, 50 )
diet<-c("vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegetarian","vegan","vegan","vegan","vegan","vegan","vegan","vegan","vegan","vegan","vegan")
UricAcidData.df <- data.frame(UricAcid, diet)
```

```
##Graphing the data (this example using ggplot2)

#Side by side dot plots of the two groups

require("ggplot2")

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 3.6.3

ggplot(UricAcidData.df, aes(x = diet, y = UricAcid, fill =
diet)) + geom_dotplot(binaxis = 'y', stackdir = 'center') +
labs(title = "Uric acid levels in the blood of patients",
x = "Diet", y = "Uric acid (ppm)")

## `stat_bindot()` using `bins = 30`. Pick better value with
`binwidth`.
```




```
##For each set of data, need to check for outliers and cal-
culate descriptive statistics.

#First, loading in vegetarian and vegan data as separate
vectors

vegetarianUricAcid <- c(54, 63, 58, 59, 61, 55, 61, 51, 68,
64)

veganUricAcid <- c(48, 52, 38, 36, 44, 54, 58, 49, 47, 50)

#Checking both for outliers

require("outliers")

## Loading required package: outliers

grubbs.test(vegetarianUricAcid)

##

##  Grubbs test for one outlier

##
```

```
## data:  vegetarianUricAcid
## G = 1.68516, U = 0.64941, p-value = 0.3566
## alternative hypothesis: highest value 68 is an outlier
grubbs.test(veganUricAcid)
##
##  Grubbs test for one outlier
##
## data:  veganUricAcid
## G = 1.70539, U = 0.64094, p-value = 0.3357
## alternative hypothesis: lowest value 36 is an outlier
#Both data sets have no evidence of outliers. Next calculat-
ing descriptive statistics for each
mean(vegetarianUricAcid)
## [1] 59.4
sd(vegetarianUricAcid)
## [1] 5.103376
var(vegetarianUricAcid)
## [1] 26.04444
mean(veganUricAcid)
## [1] 47.6
sd(veganUricAcid)
## [1] 6.801961
var(veganUricAcid)
## [1] 46.26667
##We want to compare the two data sets to see if there is a
significant difference.
#Need to conduct an F-test for variances first
var.test(vegetarianUricAcid,  veganUricAcid,  conf.level  =
0.95)
##
##  F test to compare two variances
##
## data:  vegetarianUricAcid and veganUricAcid
## F = 0.56292, num df = 9, denom df = 9, p-value = 0.4049
## alternative hypothesis: true ratio of variances is not
equal to 1
```

```
## 95 percent confidence interval:
##  0.1398214 2.2663137
## sample estimates:
## ratio of variances
##          0.5629203
#No evidence of a difference in variances for the two groups.
Need to calculate a two-sample t-test for when variances are
not significantly different:
t.test(vegetarianUricAcid, veganUricAcid, var.equal = TRUE,
conf.level = 0.95)
##
##  Two Sample t-test
##
## data:  vegetarianUricAcid and veganUricAcid
## t = 4.3881, df = 18, p-value = 0.0003547
## alternative hypothesis: true difference in means is not
equal to 0
## 95 percent confidence interval:
##   6.150468 17.449532
## sample estimates:
## mean of x mean of y
##      59.4      47.6
```

Example 3: Methyl violet

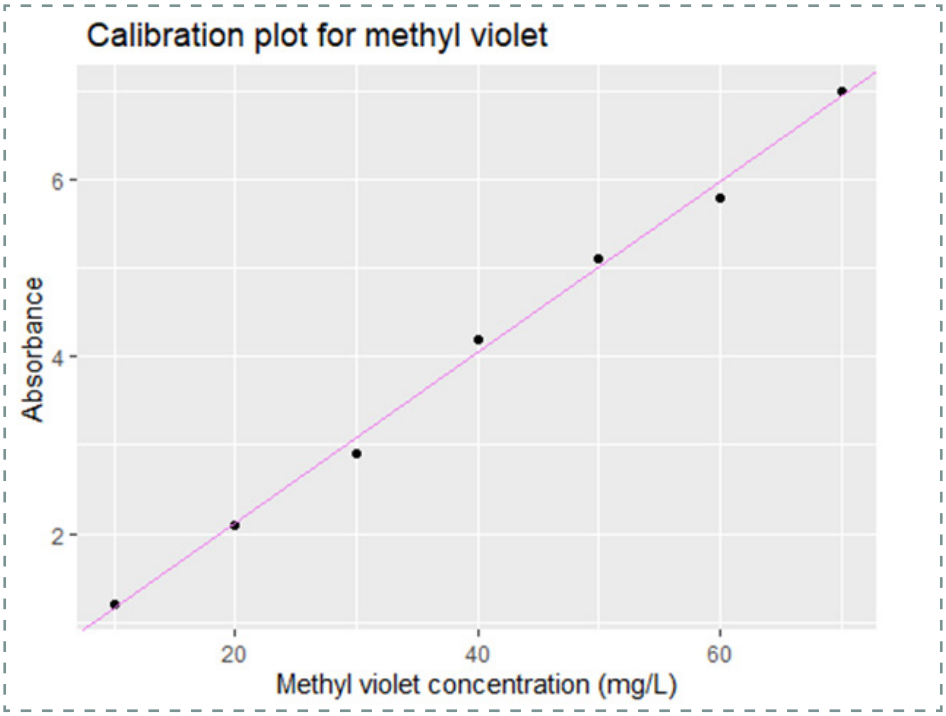
Methyl violet is an organic contaminant from industrial processes. The results from analysis to measure methyl violet using spectrophotometry are given in the table below where the absorbance at 583 nm of solutions containing various concentrations (in mgmL^{-1}) of methyl violet were measured.

Methyl violet concentration	Absorbance
10 mg/L standard	1.0
20 mg/L standard	2.1
30 mg/L standard	2.9
40 mg/L standard	4.2
50 mg/L standard	5.1
60 mg/L standard	5.8
70 mg/L standard	7.2

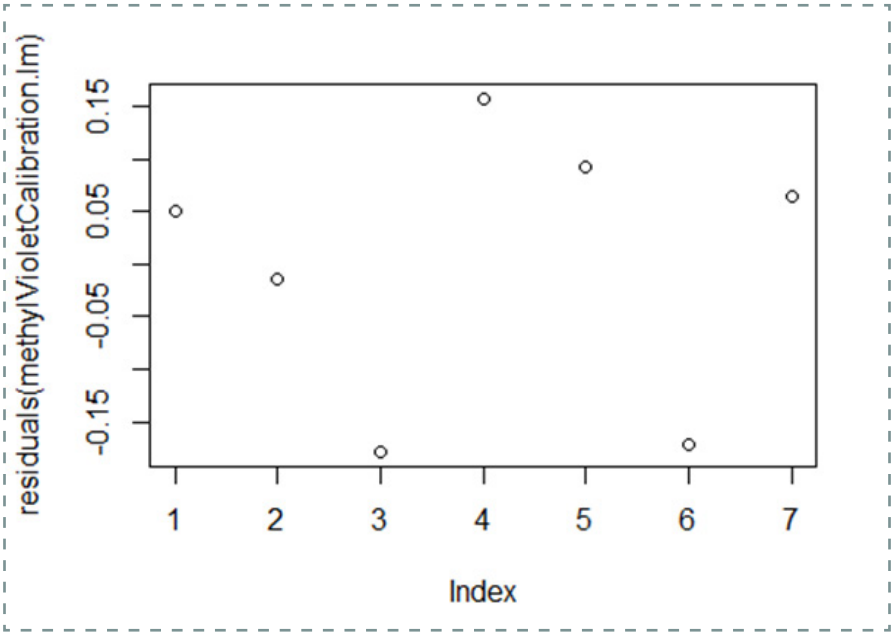
Determine if there is a linear relationship between these variables and if so develop and assess the linear regression model.

```
##Loading in the data.

#First loading in methyl violet concentration as one vector and
abosrbance as the other, then creating a dataframe

methylVioletConcentration <- c(10, 20, 30, 40, 50, 60, 70)

absorbance <- c(1.2, 2.1, 2.9, 4.2, 5.1, 5.8, 7.0)

methylViolet.df     <-     data.frame(methylVioletConcentration,
absorbance)

##Fitting an unweighted linear model

methylVioletCalibration.lm <- lm(absorbance~methylVioletConcen-
tration, data = methylViolet.df)

summary(methylVioletCalibration.lm)

##

## Call:

## lm(formula = absorbance ~ methylVioletConcentration, data =
methylViolet.df)

##

## Residuals:

##        1        2        3        4        5        6        7

##  0.05000 -0.01429 -0.17857  0.15714  0.09286 -0.17143  0.06429

##

## Coefficients:

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

## (Intercept)                0.185714   0.120374   1.543    0.184

## methylVioletConcentration 0.096429   0.002692 35.825 3.19e-07
***

## ---

## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.1424 on 5 degrees of freedom

## Multiple R-squared:  0.9961, Adjusted R-squared:  0.9953

## F-statistic:  1283 on 1 and 5 DF,  p-value: 3.19e-07

##Graphing the results

#Graphing the raw data with the unweighted linear model

require("ggplot2")

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 3.6.3
```

```
ggplot(methylViolet.df, aes(x = methylVioletConcentration, y = absorbance)) + geom_point (shape = 19) + labs(title = "Calibration plot for methyl violet", x = "Methyl violet concentration (mg/L)", y = "Absorbance") + geom_abline(intercept = 0.1857, slope = 0.0964, color = "violet")
```




```
#Graphing the residuals
plot(residuals(methylVioletCalibration.lm))
```




```
##Residuals show no signs of non-constant variance or a trend, so happy with the unweighted model
```

Example 4: Minerals in milk

Data was collected on the mineral composition (Calcium (Ca), Manganese (Mn), and Zinc (Zn)) of 39 Buffalo milk samples. The raw data for this is shown in the table below, but in this example the data has been put into a .csv file for importing.

Ca	Mn	Zn
161.39	69.8	49.53
230.54	119.99	49.77
252.1	120.82	50.15
324.61	113.08	48.16
239.94	112.88	50.74
190.82	132.44	51.1
311.66	140.95	51.37
243.82	151.69	51.66
367.42	195.6	49.07
290.91	205.41	48.2
339.67	190.09	49.76
272.1	202.04	48.29
394.26	250.24	48.53
383.87	240.85	51.65
165.3	175.87	55.07
128.5	165.84	55.07
125.65	200.24	55.63
109.21	210.93	55.42
165.8	218.47	56.29
160.14	222.54	56.38
116.11	210.08	56.53
164.29	251.94	56.98
154.9	266.15	54.32
108.48	273.37	54.28
101.92	267.23	55.45
193.49	309.72	55.61
189.5	280.11	55.82
162.72	289.9	55.77
180.2	314.24	54.91
356.16	375.54	43.41
392.41	377.96	42.36
381.3	379.79	40.04
372.67	383.4	43.79
357.97	356.52	40.8
371.55	375.79	43.28
382.13	365.17	43.61
362.01	375.18	41.74
354.71	356.82	44.47
375.74	352.51	43.39

Are there any underlying trends or groupings in the Buffalo milk and are there any relationships between the levels of the minerals?

```
##First loading in the buffalo milk .csv file
buffaloMilk.df <- read.table(file.choose(), sep=",", header = TRUE)
##Conducting HCA on the data
#Mean-centering and scaling the data, then converting it into a matrix
buffaloMilkMeanCentre.df <- scale(buffaloMilk.df, center = TRUE, scale = TRUE)
buffaloMilkEuclid.mat <- dist(buffaloMilkMeanCentre.df)
#Conducting HCA and plotting the results
buffaloMilk.HCA <- hclust(buffaloMilkEuclid.mat)
plot(buffaloMilk.HCA)
##Conducting PCA on the data
#Conducting PCA and printing the results
buffaloMilk.pca <- prcomp(buffaloMilk.df, center = TRUE, scale = TRUE)
print(buffaloMilk.pca)
## Standard deviations (1, .., p=3):
## [1] 1.4926552 0.7996825 0.3639896
##
## Rotation (n x k) = (3 x 3):
##            PC1        PC2         PC3
## Ca  0.6132685 -0.3957004 -0.68361022
## Mn  0.4783788  0.8747547 -0.07718752
## Zn -0.6285344  0.2796880 -0.72575420
summary(buffaloMilk.pca)
## Importance of components:
##                             PC1    PC2     PC3
## Standard deviation     1.4927 0.7997 0.36399
## Proportion of Variance 0.7427 0.2132 0.04416
## Cumulative Proportion  0.7427 0.9558 1.00000
#Plotting the results of the PCA (using the factoextra package)
```

```
require("factoextra")
```

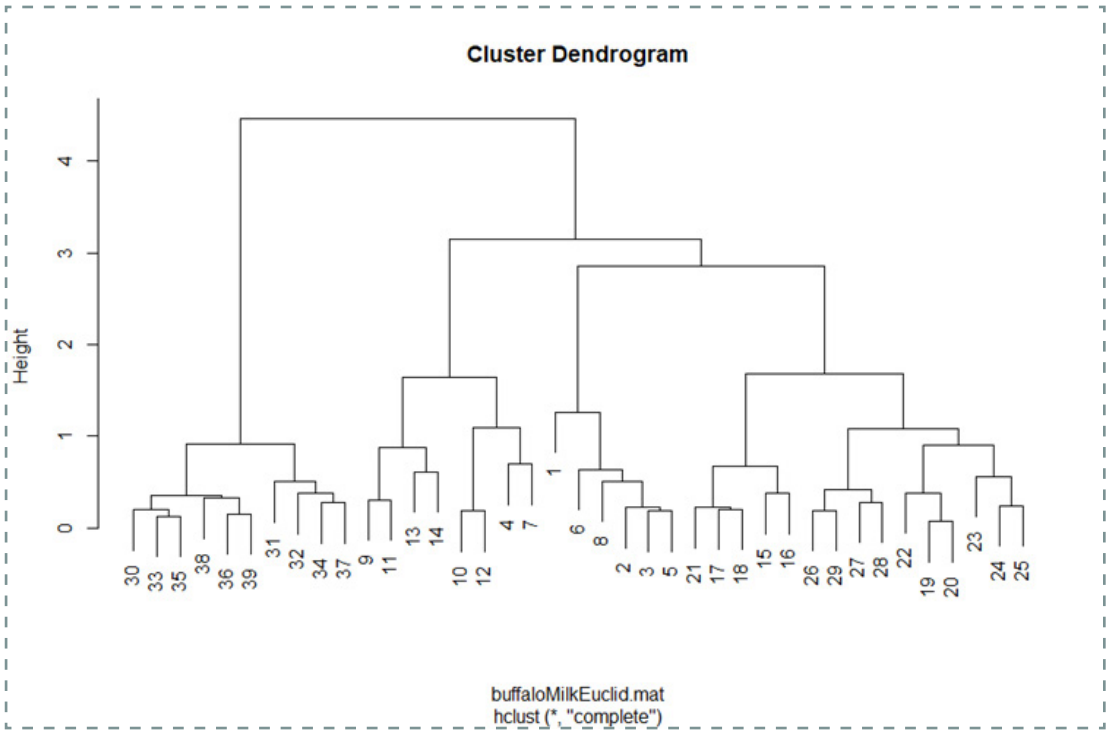
```
## Loading required package: factoextra
```

```
## Warning: package 'factoextra' was built under R version 3.6.3
```

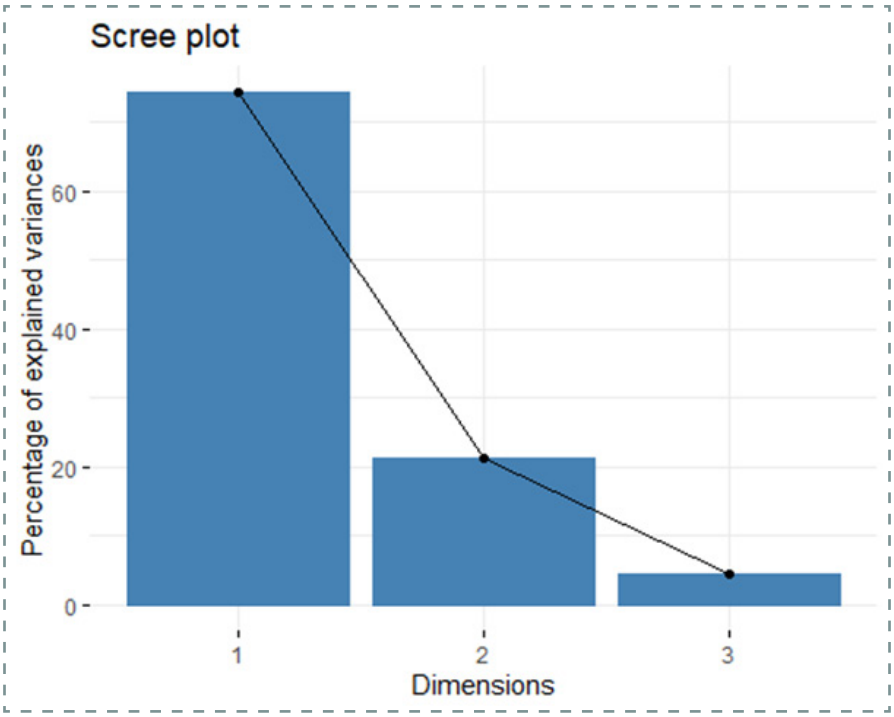
```
## Loading required package: ggplot2
```

```
## Warning: package 'ggplot2' was built under R version 3.6.3
```

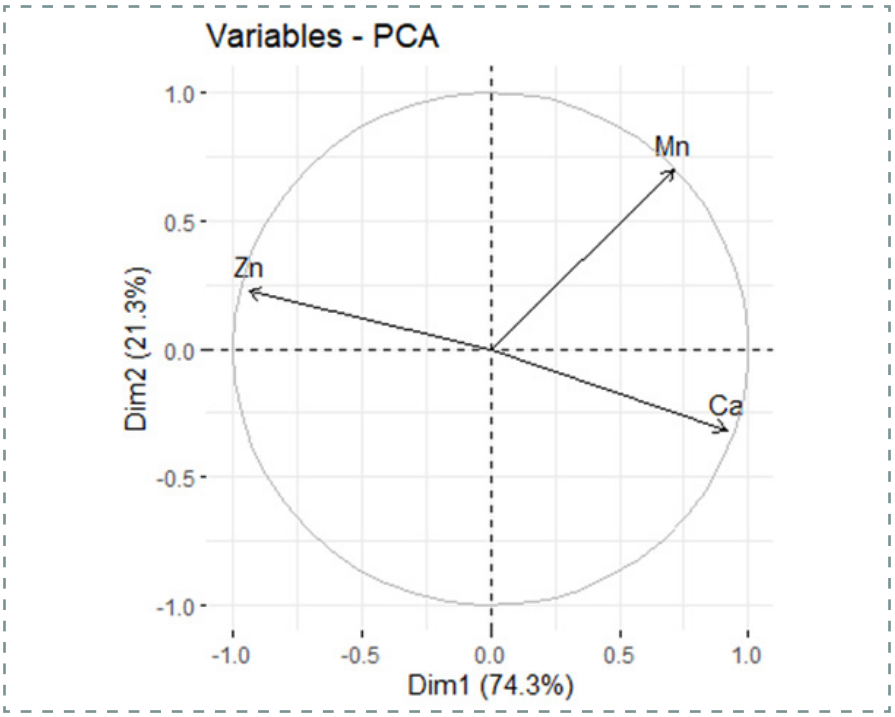
```
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
```



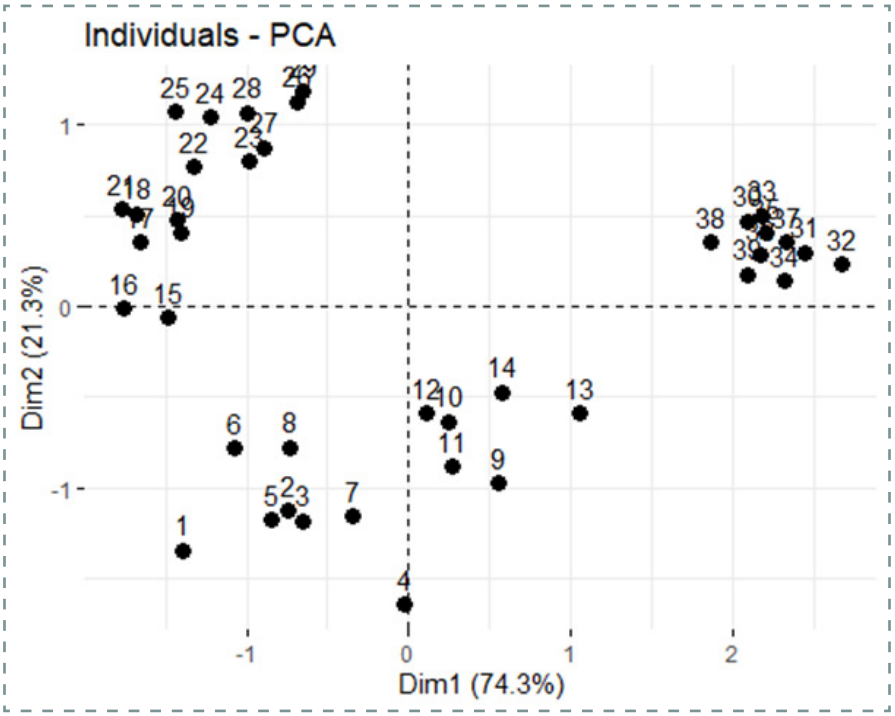

```
fviz_screeplot(buffaloMilk.pca, ncp = 10)
```



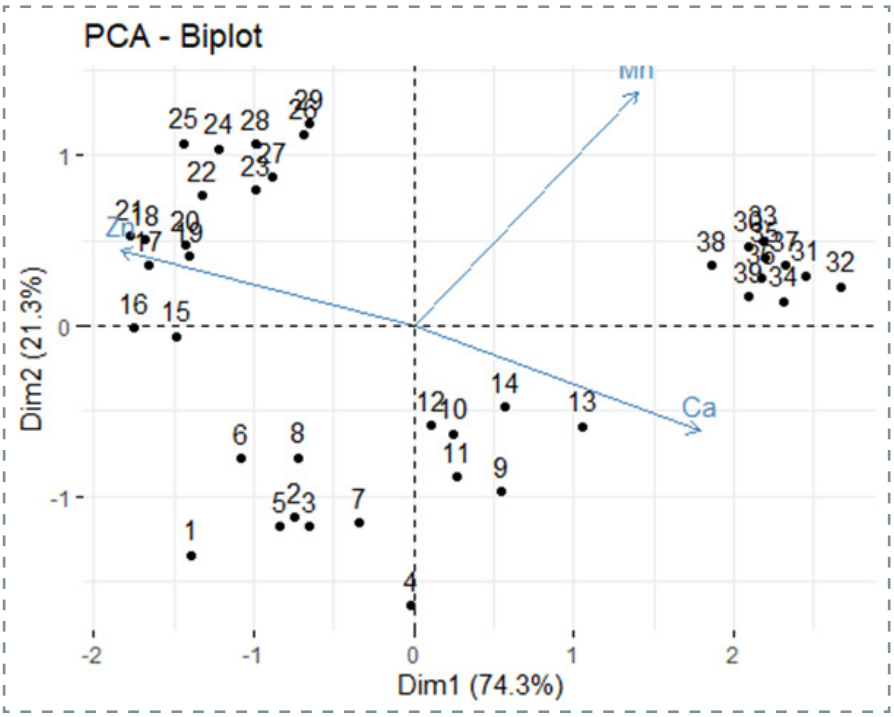
```
fviz_pca_var(buffaloMilk.pca)
```




```
fviz_pca_ind(buffaloMilk.pca, geom = c("point", "text"),
pointsize = 3)
```



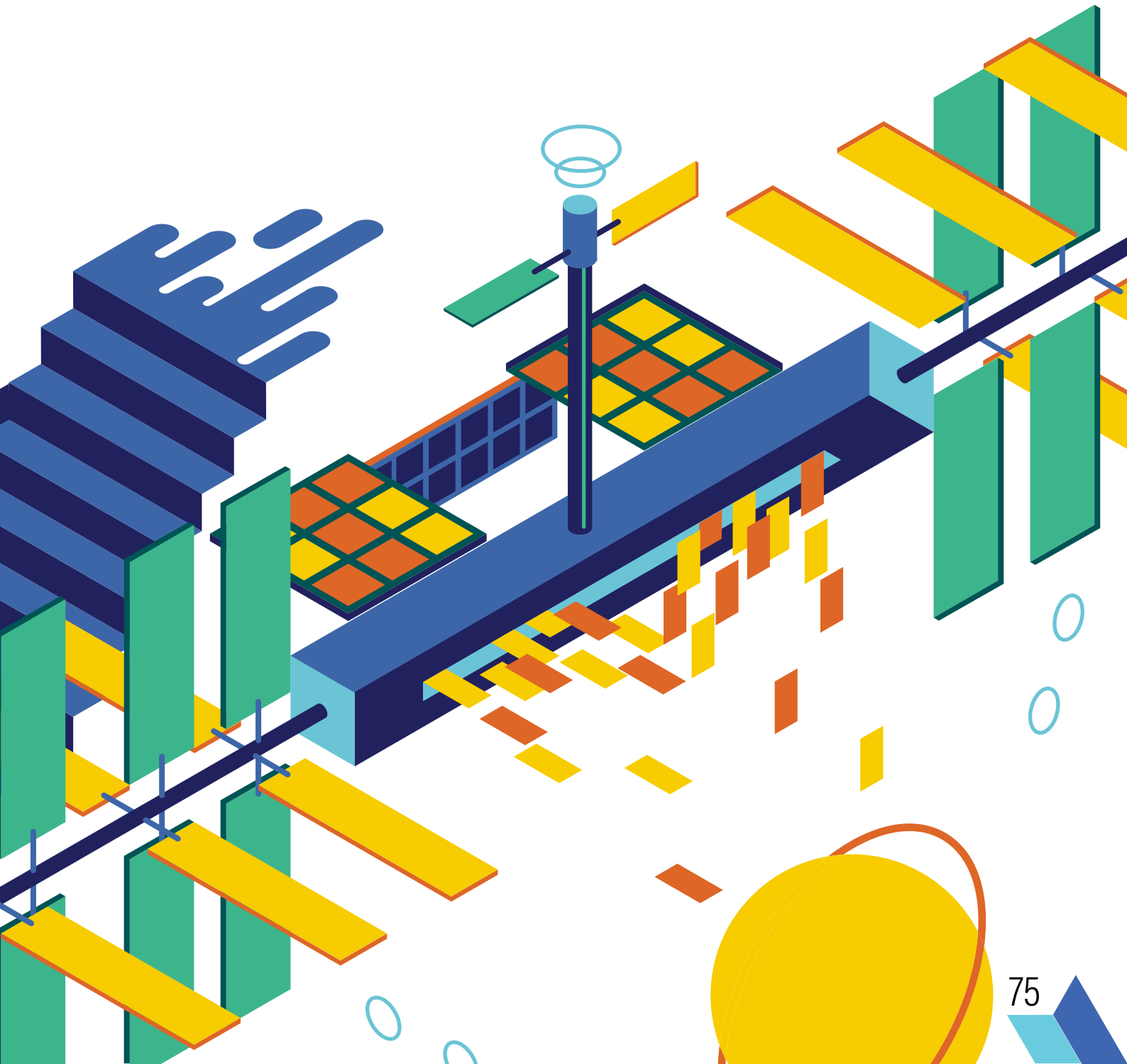
```
fviz_pca_biplot(buffaloMilk.pca)
```



Section 5B:

Moving Forward

Going through all the previous sections, you have now learned all of the fundamental processes in R to analyse and assess your data - well done! You have also been shown how to apply what you have learnt to real data, following appropiate workflows. You are now more than ready to use R in your own work and conduct a wide range of analyses - this section gives some suggestions of where to, from here, offering tips and ways to continue on your R journey. Thank you and good luck!



Part I – Debugging your code/interpreting errors

You will soon discover that it is inevitable that you will get some errors when you are using R – these normally come in dreaded red writing in the console. In this module I have tried to describe to you common causes of errors and given you tips to hopefully not make them in the first place.

If you do get an error, try your best to read the error message (which can sometimes be very challenging – they are not always very descriptive or diagnostic). Re-read your code to make sure that there is not something amiss like an errant (or absent) capital letter or space somewhere. If you cannot find the source of the error, try pasting the error into google and seeing what links come up – chances are, someone has asked on an online forum (see below) what the error means and there are often some very helpful solutions or suggestions on what is wrong and how it can be fixed.

With time and practice you will start being able to identify and figure out errors. Most importantly, do not be disheartened if you get them – it is all part of learning and they happen (regularly!) to the best of us.

Part II – Online community – R help forums

Being an open-source software that users all around the world use and even contribute to, it should be no surprise that R has a big online community, ranging from blogs to help forums, all of which have a range of tips and tricks.

- If you want to learn more about R, what it can do and useful snippets of analysis and techniques, R-bloggers (https://www.r-bloggers.com/) is a great place to regularly visit.
- There are other R communities and how-to's out there, including https://community.rstudio.com/ and http://www.sthda.com/english/
- Help forums like stackoverflow (https://stackoverflow.com/questions/tagged/r) and statckexchange (https://stats.stackexchange.com/questions/tagged/r) are very useful. If I have any trouble or want to do something in R, I will often just google my question and links to the same or similar question posted in one of these sites.

Part III – Available data repositories/recommended data sets

To practice the skills and techniques that you have been taught, there are a range of data sets that you can use.

R itself has a large range of built-in data sets. To see what data sets are available in R and then load one of the in-built data sets, you can do the following:

To load (and view) an in-built R data set, the code is:

```
data()
data("iris")
head("iris")
iris
```

- The term "`data()`" instructs R to list all the in-built data sets it has available for use
- The term "`data("...")`" loads the specified dataset in R (in this case, `iris`)– this needs to be done every time you open R and use the dataset.
- The term "`head("...")`" instructs R to show the first 6 lines of the dataset (in this case the `iris` dataset).
- The term "`iris`" instructs R to show the entire the dataset.

Of the available in-built R data sets, I have included a list of some appropriate data sets and what analysis you can use them to practice:

- PlantGrowth
  - This data set gives the plant yield (measured by dried weight of plants) obtained under a control and two different treatment conditions.
  - This is a great data set to work on to practice your ANOVA analysis (Section 2C Part III)
- InsectSprays
  - This data set gives the number of insects present after using different types of pesticide.
  - This is a great data set to work on to practice your ANOVA analysis (Section 2C Part III)
- Formaldehyde
  - This data set gives the results to prepare a standard addition curve for the analysis of formaldehyde by the addition of chromatropic acid and concentrated sulphuric acid. The measured response is the optical density for the resulting purple color by a spectrophotometer.
  - This is a great data set to work with to practice your regression analysis (Section 2D Part I).
- mtcars
  - This data set is one that was extracted from the 1974 Motor Trend US magazine. The data set has 11 variables for 32 different cars, including fuel consumption and aspects of automobile design and performance.
  - This is a great data set to work with to practice your unsupervised learning analysis (Section 3A).
- rock
  - This data set includes measurements on 48 rock samples from a petroleum reservoir. Measurements for each sample includes; permeability, total area of pores, total perimeter of pores, and shape.
  - This is a great data set to work with to practice your unsupervised learning analysis (Section 3A).

- USArrests
  - This data set is states the arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973 and also gives the percent of the population living in urban areas.
  - This is a great data set to work with to practice your unsupervised learning analysis (Section 3A).
- trees
  - This data set is made up of measurements of the diameter (labelled Girth in the data set), height and volume of timber in 31 felled black cherry trees.
  - This is a great data set to work with to practice your unsupervised learning analysis (Section 3A).
- iris
  - This data set gives the measurements of the variables; sepal length, sepal width, petal length and petal width, respectively, for 50 flower samples from each of three species (setosa, versicolor, and virginica) of iris.
  - This is a great data set to work with to practice your supervised learning analysis (Section 3B) once you split it into training and test datasets.

There are other data sets that are accessible in R, but they are associated with packages in R and therefore the parent package needs to be downloaded to access them. For a fairly comprehensive list of these available data sets and the package that they belong to, see this list: https://vincentarelbundock.github.io/Rdatasets/datasets.html

- This site gives a brief summary of the number and types of variables/ observations in the data set, as well as the package that it can be found in.
- This site also, very conveniently, allows you to download the .csv file of the data set so you do not need to install the parent package and instead can import the downloaded .csv file.

Otherwise, there are a large number of data repositories out there that you can manipulate and import into R to then analyse. A summary of the focus and features of other repositories can be found here: https://sr.ithaka.org/blog/data-repository-platforms-a-primer/ and others can be found through searching yourself.

You now have the tools and skills to analyse and understand an extraordinary number of data sets – go and enjoy!

Index of functions

Index of packages

1st edition, November 2022.