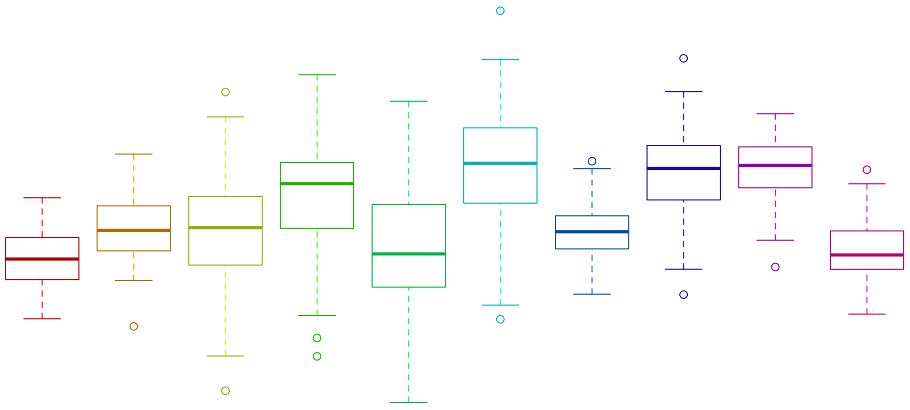


WORKBOOK: INTRODUCTION TO R

Durham University Research Methods Centre summer school

Materials produced by:
Dr Sally Street (Durham University)

Workbook edited and appended for the Department of Psychology by:
Prof Lynda Boothroyd (Durham University)

Contents

Acknowledgements

Throughout the worksheets I've drawn upon Andy Field's (2012) R textbook 'Discovering Statistics Using R' (Sage): https://uk.sagepub.com/en-gb/eur/discovering-statistics-using-r/book236067. This is also our course-recommended textbook for our undergraduate stats teaching in R in Anthropology. I'd recommend it highly as a supporting resource for R beginners.

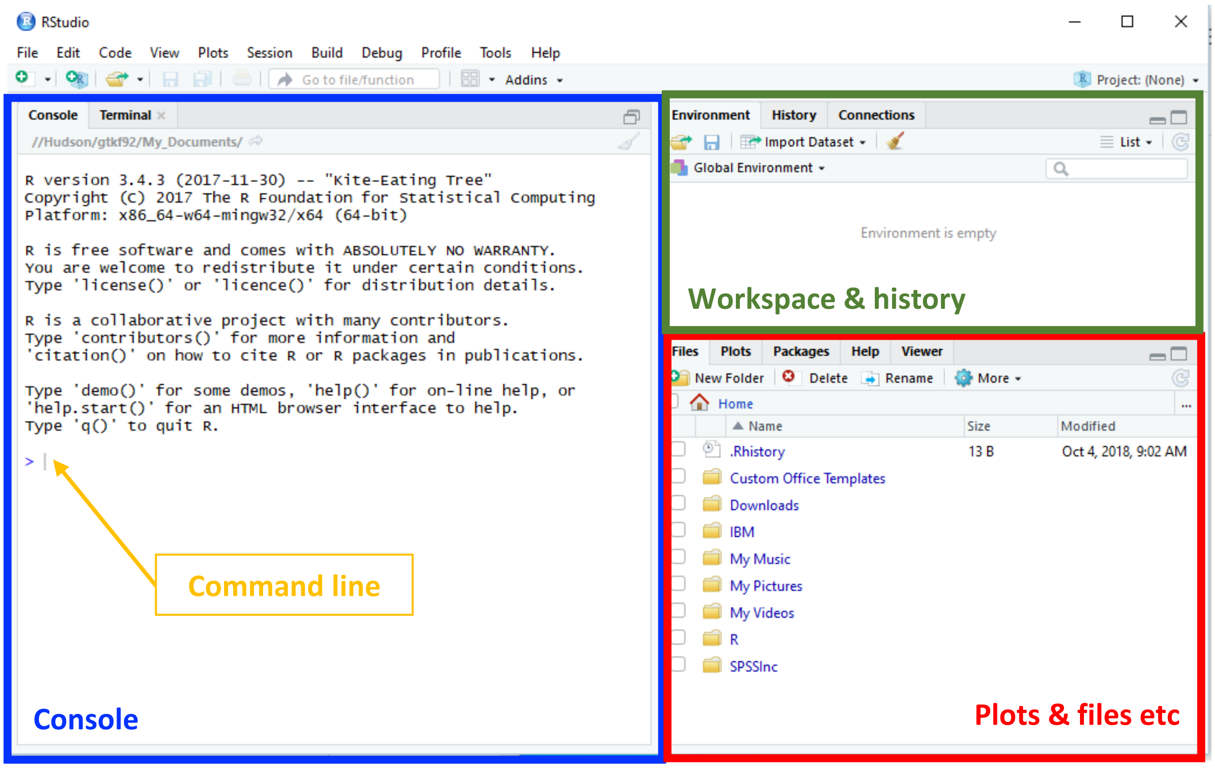
Session 1: Introduction to R & RStudio

Launching the software

When using a computer on the University network, you can launch the software using the AppHub – this lets you use the software without downloading it, like a streaming service. Go to the AppHub (https://apphub2.durham.ac.uk/), scroll down to RStudio (look out for the logo on the right) and click either version to launch the program. R is the software that does all the work behind the scenes, while RStudio provides a more convenient interface. You do not need to launch R separately – it will run automatically in the background when you launch RStudio[1].

Interacting with the software

You should see a window open like the one below, containing three smaller windows: the larger one on the left is the **console** containing the **command line**, the top right one contains information about the **workspace** and **history**, while the bottom right one is used primarily to view **plots** and **files**.



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Name	Size	Modified
.Rhistory	13 B	Oct 4, 2018, 9:02 AM
Custom Office Templates		
Downloads		
IBM		
My Music		
My Pictures		
My Videos		
R		
SPSSInc		

For now, the most important window to focus on is the **console**. In R you will be interacting with the program by entering commands rather than clicking on menu options, as you may be used to with statistical software such as SPSS. Commands are entered on the **command line** immediately after the chevron and blinking cursor.

[1] If you want to use the software on your own computers however, you will need to install both R and RStudio separately. Both are available for free from the following links, respectively: https://www.r-project.org/ https://www.rstudio.com/. If you need help installing the software there are many useful resources online, including some provided by the University (see here: https://www.dur.ac.uk/carod/ittrainingcourses/#r).

Interacting with R is a little like a one-sided conversation – we will be feeding it instructions, and it will return results. A large part of learning to use R is adapting to its own particular programming language and its syntax, as well as its quirks – similarly to learning a human language!

To get used to the interface, we'll start by entering a few basic instructions. Try, for example, typing some simple arithmetic using numbers, mathematical symbols (+, -, *, /) and round brackets on the command line. Press enter, and R should return the result on the line below, e.g.

```
(35*7)/40
```

And R will return the answer, 6.125.

So far, so good – but of course R can do many, much more useful things for you than basic maths. In order to make best use of R we'll need to get to grips with some of the basics of its language.

Functions

Functions are specific actions that we can instruct R to do, such as calculating a mean, plotting a graph, running a particular statistical test and so on. R comes ready installed with a huge library of functions (referred to as **base** functions), but you can expand its capabilities further by installing specialised **packages** of functions (see below) or by writing your own.

Functions in R need to be entered using commands structured in the following format:

function(arguments)

Where 'function' will be the name of the function, and the round brackets will contain items (called **arguments**) you want to perform the function on (such as a list of numbers, a set of data, etc.).

For example, we can use the `sum` function to add up a list of numbers. Enter the command below to try this out (**Note:** you can copy-paste commands directly from the worksheet into R):

```
sum(1,2,3,4,5)
```

The arguments inside the brackets need to be entered in various ways specific to each particular function. They should always, however, be separated by commas.

Objects

One of the first things that is important to understand about R is that it is an **object-oriented language**. This means that we primarily interact with the software by creating and storing **objects** in the software's memory. Objects can contain a huge variety of different things, from individual numbers or words to entire datasets, plots and statistical results.

To create an object, we need to enter commands in this format:

object<-items

Where 'object' stands in for your chosen name for the object, and 'items' refers to whatever you want to store inside it. The 'arrow' sign (<-) indicates assignment – so R is assigning any items to the right of the arrow to the named object on the left of the arrow. You can call objects whatever you like, but you should use alphanumeric characters only (i.e. no spaces or special characters, although full-stops and underscores are ok).

For example, we could create an object called ‘x’ that contains the number 100 like this:

```
x<-100
```

When you create an object successfully, it will appear as if nothing has happened. This is because R has done as you’ve instructed and stored the object in its memory, with nothing further to report yet. If you want to check the object really is there, just enter the name of the object and R will return whatever is inside it on the line below. Try this out now for ‘x’ to see for yourself.

The vast majority of commands in R use functions and objects together, using the following format:

object<-function(arguments)

For example, bringing all the above together we could create an object called ‘y’ containing the sum of a list of numbers as follows:

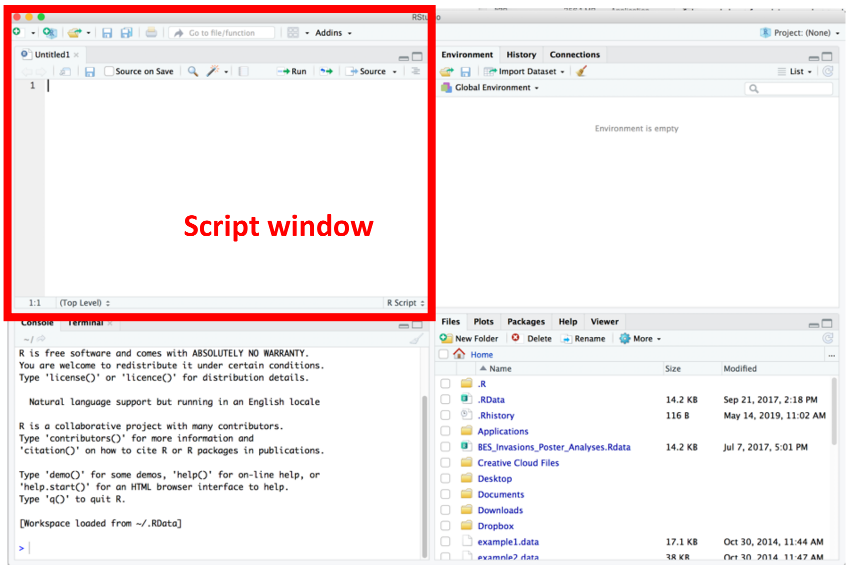
```
y<-sum(1,2,3,4,5)
```

Tip: to scroll through previous commands, click on the command line and press the up/down keys.

Scripts

It is not usually a good idea to type commands directly into the console window, because it is very easy to lose track of what you have been doing and make mistakes. Instead, you should create a **script** file to store all the commands you will use for a particular project. A script is simply a text file where you can type and save your commands so that you have a record of what you have done, which you can return to at a later date and pick up where you left off.

To create a new script file, click File>New File>R Script, and you will see a new window open in the top left-hand corner (see below). Then, click File>Save As to save a copy of your script file.



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

Name	Size	Modified
.R		
.RData	14.2 KB	Sep 21, 2017, 2:18 PM
.Rhistory	116 B	May 14, 2019, 11:02 AM
Applications		
BES_Invasions_Poster_Analyses.Rdata	14.2 KB	Jul 7, 2017, 5:01 PM
Creative Cloud Files		
Desktop		
Documents		
Downloads		
Dropbox		
example1.data	17.1 KB	Oct 30, 2014, 11:44 AM
example2.data	38 KB	Oct 30, 2014, 11:47 AM

To run commands from the script window, first type your command on a new line in the script window. Then, place your cursor on the same line as the command and press **control+enter**, and the command will shoot down from the script to the console window.

Try this now using one of the commands we have encountered so far, such as:

```
y<-sum(1,2,3,4,5)
```

Always make sure you save your script file regularly, or you'll lose your work if (or when!) R crashes.

Annotation

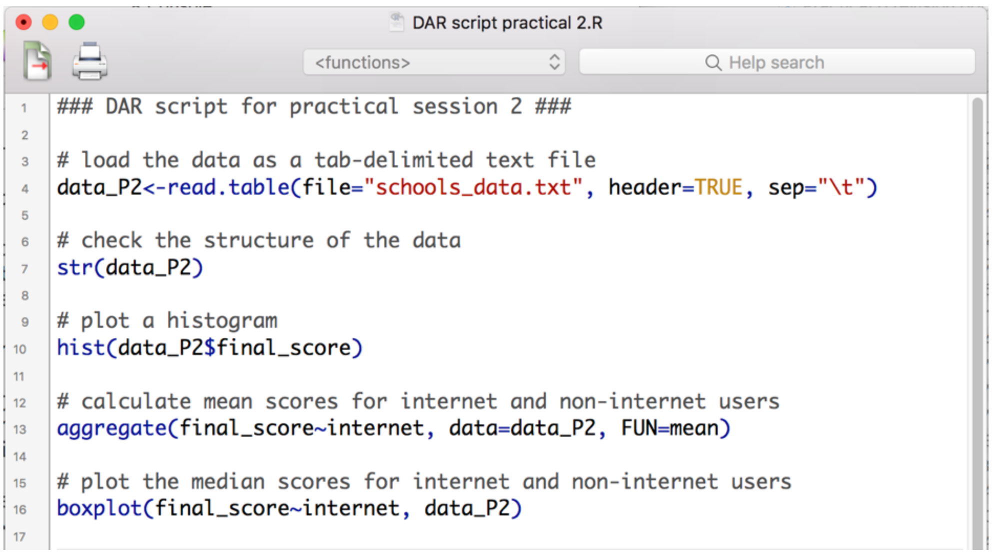
Even if you save all your commands in the script window and feel confident that you know what you're doing at the time, you will probably find it difficult to make sense of things if you return to a script after several weeks or months. Therefore, you should always **annotate** your code. This means adding notes to your script that explain what the commands are doing, in a way that is readable by a human rather than a computer. Be kind to your future self and write whatever will be most helpful!

You can annotate your commands in the script window by preceding text with a hash (#). For example, try running the command below. Use your cursor to highlight both lines together before running them, so that they will be sent down to the command line at the same time.

```
# sum the list of numbers and assign them to the object y
y<-sum(1,2,3,4,5)
```

You'll see that while R reads the command, it ignores the text you've entered preceded by the hash key. Because the text is not read by R, you can write whatever you like (including special characters).

You should **annotate all of your commands** during the workshop and for any project you do in R. The screenshot below shows you an example of a well-annotated script from one of our undergraduate classes here in Anthropology, so you can get an idea of what to aim for. You'll see that while the commands won't make much sense to you yet, you'll be able to understand something of what the student was doing with the data based on their annotations.



```
### DAR script for practical session 2 ###

# load the data as a tab-delimited text file
data_P2<-read.table(file="schools_data.txt", header=TRUE, sep="\t")

# check the structure of the data
str(data_P2)

# plot a histogram
hist(data_P2$final_score)

# calculate mean scores for internet and non-internet users
aggregate(final_score~internet, data=data_P2, FUN=mean)

# plot the median scores for internet and non-internet users
boxplot(final_score~internet, data_P2)
```

Vectors

Vectors are one of the most fundamental types of object in R. A **vector** is a sequence of elements which are all of the same type (such as numbers or characters). In R, the function `c` (for 'concatenate') is used to combine multiple elements into a vector. For example, try using the command below to create a vector 'z' containing round numbers from 1 to 10:

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

Alternatively, we could do the same using the colon operator to generate a sequence of integers from X to Y (inclusive):

```
z<-c(1:10)
```

When creating a vector containing character strings, all elements must be contained in double quotation marks, for example:

```
fruits<-c("mango", "papaya", "watermelon")
```

R's functions are typically **vectorized**, which means that if apply them to a vector, the function will be performed on all individual elements contained within the vector. For example, if you multiply z by 10 you should see the following output returned:

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

Indexing

Once you're able to create and store objects, you'll often find yourself needing to select specific items from them based on various criteria – this is called **indexing.**

Indexing requires a command in the following format:

object[criteria]

Where 'object' stands in for the name of your object (e.g. a vector), and 'criteria' the criteria by which you want to select the items. **Note**: you must use square rather than round brackets for indexing.

One way to index is by the numeric position of elements within the vector. For example, the first command below creates a new vector 'v', while the second selects the second and fifth elements:

```
v<-c("A", "B", "C", "D", "E")
```

```
v[c(2,5)]
```

Note the use of `c` within the square brackets on the second line – this is because the selection criteria themselves need to be combined into a vector, when there is more than one!

As before, we could also use the colon operator for the criteria if we wanted to select a list of elements from one number through to another, e.g.

```
v[c(1:4)]
```

Alternatively, you can select elements that meet certain criteria using **relational operators**, i.e.

Operator	**Description**
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to

Note: in R, 'is equal to' is indicated by double, rather than single, equals signs. The single equals sign can be used in place of the arrow (`<-`) to indicate assignment, but this is potentially confusing.

For example, using the vector z we created earlier we could find out which numbers in the list are 5 or more by entering:

```
z>=5
```

And we could extract these numbers from the list using:

```
z[z>=5]
```

The utility of indexing will become more obvious in the second session, where we will cover how to select specific observations from datasets based on various criteria.

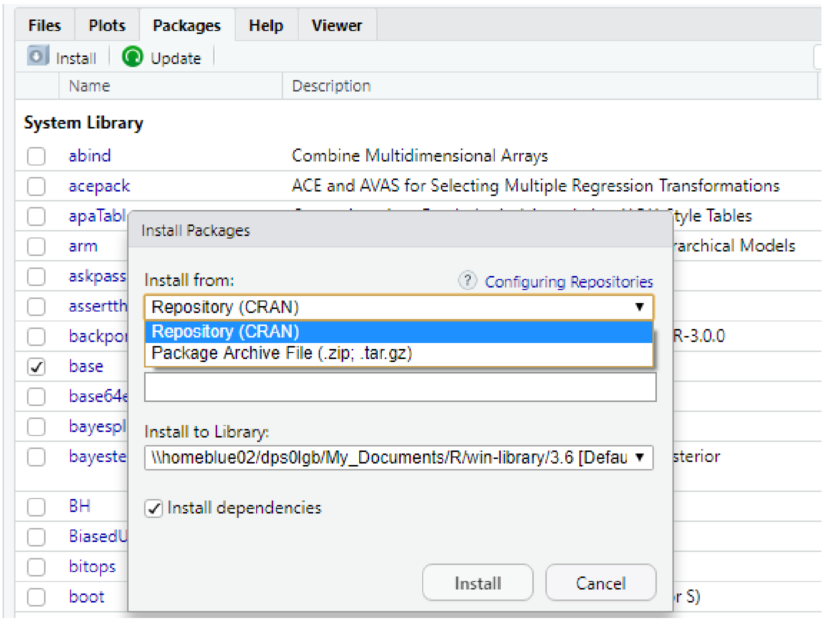
Packages

One of the benefits of R's open source nature is that many data analysts and computer programmers have written their own packages which expand R's functionality. Some of these packages are written in base R and some of them draw on other packages as well. What they all typically do, is allow you to run analyses or functions that would require extensive coding in base R if you were to do so from scratch. Many packages come ready installed with R. You can see a list of them in the packages tab on the bottom right.

You will eventually need to install and load additional packages yourself. For instance, later in session 4 we will use the *car* package. You can install it using a line of script like this:

```
install.packages("car")
```

You can also install packages by clicking on the install button in the Package window. This opens a dialogue box where you can type the name of the package. By default RStudio will download the package from the CRAN repository. However, in some cases you need to manually download the package yourself as a compressed file (you can usually find it via Google), and then load it from your local drive.



Install from: Repository (CRAN)

Install to Library: \\homeblue02/dps0lgb/My_Documents/R/win-library/3.6 [Defau

☑ Install dependencies

Note: occasionally some people are unable to install packages when using RStudio on the University network. If that's the case, don't worry – you can either skip or just read through sections using packages for now. It should work just fine when using the software on your own computers.

Once you've installed a package, you won't need to do this again unless you want to update it. Any time you want to use a package within a particular R session, however, you will need to load it using `library`, e.g.

```
library("car")
```

Once it is loaded you can use it for the rest of the session. If you go to the packages tab, you will see that car is now ticked to indicate that it is loaded.

Sometimes you may want to unload a package without restarting your R session (for instance if it clashes with another package later in your workflow.) This is done by using the detach function:

```
detach("package:car", unload=TRUE)
```

Check to see if *car* is now unticked in the packages window.

Error and warning messages

When things go wrong, R will (usually!) tell you by reporting an error or warning message in `red text`. Don't worry if this happens – it just means that you have probably not used the appropriate language in your commands. **Errors** mean that R cannot do what you have asked it to do, while **warnings** mean that R has been able to do what you have asked, but thinks this might cause problems.

For example, try typing in some random text into the command line and entering it. You'll see an error message returned like this:

```
Error: object 'sdfds' not found
```

R is telling you here that it cannot find any object called ‘sdfds’, because no such object exists.

To see an example of a warning, try the command below. The function `as.numeric` can be used to convert an object to a number or series of numbers.

```
as.numeric("apples")
```

R will return the following result and associated warning message:

```
[1] NA
Warning message:
NAs introduced by coercion
```

Here, we get a warning rather than an error because while syntactically correct, this command does not make much sense. We’ve asked R to transform the character string “apples” into a number but this is not a very clear instruction – how should R know which number we want to assign to “apples”? Should this be based on the number of letters in the word, the position of the first letter in the alphabet, etc.? So, instead it returns NA (‘not available’) and warns you because this probably isn’t what you wanted it to do.

Trouble-shooting

Dealing with errors, warnings and other unexpected behaviour in R can be frustrating, but is all part and parcel of learning to code. This section contains some useful tips for problem-solving when things go wrong.

Read the message

This might sound really obvious, but the first thing to do when you get an error or warning is to read it carefully. R will tell you precisely what the problem is, although it will do so in its own language, which takes some getting used to. Try to read the message in extremely literal terms and ask yourself ‘what have I told R to do’ and ‘what does R think I want to do’?

Double-check your commands

99% of the time, problems in R are caused by human input errors rather than anything going wrong with the software itself. Before doing anything else, simply read and re-read your command back to yourself to see if you can spot the problem[2]. Pay special attention to commas, quotes, brackets and capitalisation as these are very easy to get wrong.

Ask a friend

It is very easy to go ‘blind’ to our own coding errors, especially if we have been staring at a script for a while. Quite often a pair of fresh eyes solves the problem quickly, so it can be really helpful to ask a peer to check your code for you (and be prepared to do the same for them in return!).

[2] See also the ‘rubber duck debugging’ method: https://rubberduckdebugging.com/

Use the help function

In many cases, things go wrong because you are not totally sure what a function does or what arguments it needs to work. If you need to remind yourself of this, enter a question mark immediately followed by the name of a function, for example:

```
?mean
```

You will see R's help page for the function pop up in the bottom right window. Unfortunately, these help files are not super user-friendly for beginners as they make heavy use of R jargon, so do not worry if you can't make much sense of them. This does help, however, if you simply want to remind yourself what a particular function does and what arguments it requires. If you scroll down to the bottom of the help page you will also see some reproducible examples of the function in action.

Ask Google

The vast majority of the time, someone else will have experienced and resolved the same problem as you and posted it on a statistics forum such as Stack Overflow or Cross Validated. Just by typing in some key words or copy-pasting an error message into Google, you will likely find something helpful.

Note – do not worry if you cannot make sense of the forums, they can be very technical and take some getting used to. I also would not necessarily recommend you post on them yourself while getting to grips with R – some of the respondents can be a bit harsh with newbie queries…

Session 1: questions and exercises

1. A typical command in R involves creating an object using a function and arguments. Which of the options below correctly represents the format of a typical command in R?

A. *object->function(arguments)*
B. *object<-function(arguments)*
C. *object<-function[arguments]*
D. *object(function(arguments))*

2. Find the square root of 35, using the function `sqrt`

3. Which key should you use to preface annotations?

A. #
B. $
C. /
D. @

4. The following command will return an error. Why is this?

```
z<-(14, 25, 36)
```

5. How could you resolve the following error?

```
Error in sum(1532, 5435, 7385, ) : argument 4 is empty
```

6. What is the mean of 42, 79, 11 and 85? Find your answer by first assigning the numbers to a vector, then using the mean function on that vector.

7. What is the `c` function used for?

A. calling a function
B. combining different data types
C. cancelling a command
D. concatenating a sequence

8. What kind of brackets are used for indexing?

A. square
B. round
C. curly

9. Create a character vector (i.e. containing a sequence of items, each of which is a character string) and select a subset of the items based on their numeric position (e.g. the third and fourth elements).

10. Create a numeric vector (i.e. containing a sequence of numbers only) and select a subset of the items based on a relational operator such as equal to, greater than, less than, etc.

Session 2: Importing & handling data

In this section we'll tackle what can often be the most challenging step for new R-users: importing data and wrangling it into the right format for analysis. We'll illustrate exercises using a police dataset on crimes committed in County Durham in April 2019[3]. Go to https://data.police.uk/, select 'Download Police.uk data in batches', set the date range as April 2019 and select 'Durham Constabulary'. Download two of the available datasets ('crime' and 'outcome' data).

Setting up a new R session

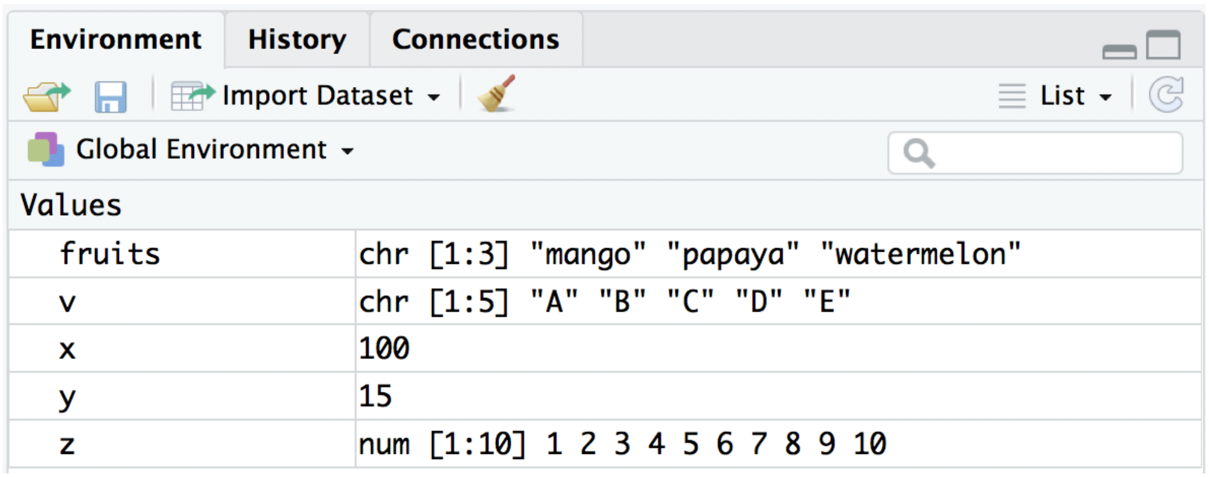
There are three things you will need to do routinely before starting any data analysis in a new R session: 1) create and save a new **script** file, 2) clear the **workspace** and 3) set a **working directory**. (This can also all be done by using R Projects, but we won't cover that here. See the Tidyverse sessions.)

1. Creating a script file

We've already seen how to create a **script** file – ordinarily it's a good idea to organise your work by having one script file per project or set of analyses, but you can do this however makes most sense to you. So, you can either keep working on your script file from session 1 or create a new one for session 2.

2. Clearing the workspace

The **workspace** refers to all of the objects created and stored within a given session. When using RStudio, you can see all of the objects stored in the current session by looking at the top left window under the 'Environment' tab. You should see something like this:



```
Values
  fruits      chr [1:3] "mango" "papaya" "watermelon"
  v           chr [1:5] "A" "B" "C" "D" "E"
  x           100
  y           15
  z           num [1:10] 1 2 3 4 5 6 7 8 9 10
```

Sometimes, stored objects can cause conflicts and unanticipated behaviour when moving between projects. It is very easy to forget what might be lurking in R's memory and end up doing analyses on the wrong dataset! For this reason, it is always a good idea to clear R's memory when starting a new script or set of analyses. You can do this by entering the command below:

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

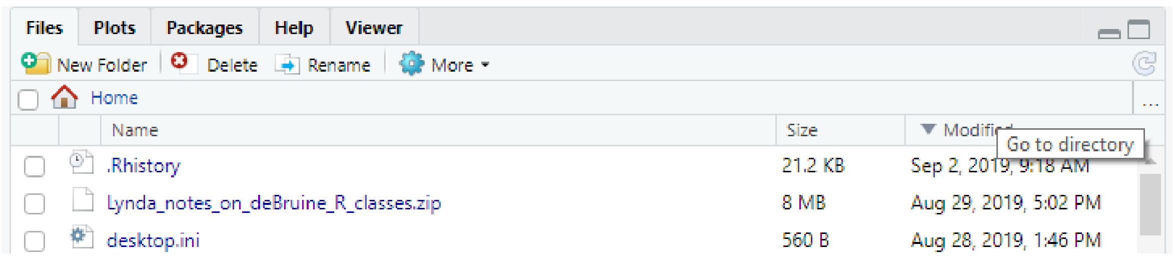
[3] https://data.police.uk/

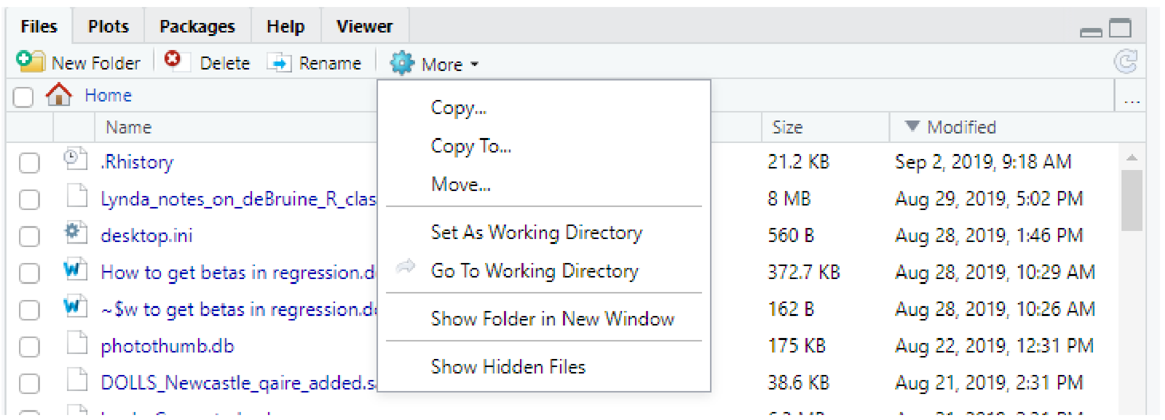
3. Setting a working directory

A **working directory** is a default folder that R will use to locate files within a given session. You'll need to set one whenever you are going to be reading in external files, such as data tables. We'll try this using the crime and outcome data you've just downloaded. Move both of these files to a conveniently located folder, such as your desktop, or create a new folder specifically for this workshop. Then, you can choose a location using the function `setwd`, with your chosen filepath inside the brackets (note that Windows and Macs may require / vs \ respectively), e.g.

```
setwd("C:/Users/Sally/Desktop")
```

If you don't know the filepath (which is not always obvious e.g. on a network), navigate to the right folder in the File window in RStudio (use the three dots in the corner if needed). You can then click on the settings blue-cog MORE button and select Set working directory from the drop down menu.





Alternatively, you can do this by clicking on a menu option– select the 'Session' tab, click 'Set working directory', then 'Choose directory'. You should see a command like the one above pop up on the command line to confirm the working directory has changed successfully.

Formatting data for R

R cannot easily import data directly from a spreadsheet program such as Excel. Instead, spreadsheet files should be converted to a plain text format such as .csv or .txt before you attempt to read them into R. Data should usually be formatted such that each row is a single observation and each column a single variable, reserving the top row for column headers if necessary.

You should format your data as clearly and simply as possible to avoid issues reading it into R. If you adhere to the suggestions below you should not encounter too many problems:

- Avoid special characters (use only letters, numbers, full-stops and underscores)
- Avoid spaces in the file or column names (use full-stops or underscores instead)
- Indicate missing data consistently with NA rather than leaving cells blank or using e.g. -999
- Including nothing else in the spreadsheet other than the data and column headers
- Input categorical variables as words, not numbers (e.g. ‘smoker’ vs. ‘non_smoker’, not 1 and 2)
- Within a column, consistently input numbers as numbers and words as words
- Avoid importing data containing Excel formulae, macros or other complexities
- Do not use colour-coding to indicate category membership as this will not be read into R

Below is an example of some hypothetical data appropriately formatted in Excel for R:

Participant_ID	**Age**	**Employment**	**Location**	**Smoker**
1	23	Employed	Urban	Yes
2	45	Employed	Urban	Yes
3	19	Unemployed	Urban	No
4	42	Unemployed	Rural	No
5	53	NA	NA	Yes
6	63	Employed	Rural	Yes

And here is an example of the same data formatted badly in Excel for R:

Participant ID#	**Age**	**Employment**	**Location**	**Smoker?**
1	23	Employed?	Urban	🟥
2	45	Employed?	Urban	🟥
3	Nineteen	Unemployed	Urban	🟦
4	42	Unemployed	Rural	🟦
5	53	-	Missing	🟥
6		Employed	Rural	🟥

If it is important to you to use non-R friendly formatting in your raw data, you should keep two versions – one original spreadsheet file (with as much fancy formatting as you like) and one ‘clean’ copy for R. In Excel, you can convert .xlsx to plain text files just by clicking ‘Save As’ and selecting an appropriate format, such as comma-separated or tab-delimited text files.

Importing data from spreadsheets

Assuming data are formatted appropriately for R, plain text files can be read into R fairly straightforwardly. The simplest way to do this is to use comma-separated (.csv) files, which can be imported into R using the function `read.csv`. Importing a dataset requires us to create a new object containing the data. Try for example reading in the crime data using the command below:

```
crime_data<-read.csv("2019-04-durham-street.csv")
```

If the command worked, nothing will appear to happen, but you should see the new object 'crime_data' appear in the Environment window. Helpfully, this will also report the numbers of rows (observations) and columns (variables).

Note: by default, `read.csv` will assume that the first row of your data contains the column headers. If for whatever reason your data does not have column headers, you should include an additional argument 'header=FALSE' inside the brackets.

It is absolutely essential at this stage that you check the data have been read in correctly, as mistakes with formatting and importing data are easy to make. Most fundamentally, you should check the version of the data that has been read into R against the original file, to make sure R has read in the correct number of rows and columns, in the correct format.

The functions `str` ('structure'), `head` and `tail` are very useful for examining the data to make sure nothing has gone wrong. `str` can be used on a wide variety of different objects and tells you some basic information about what an object contains. For example, try using it on the crime data:

```
str(crime_data)
```

And R should show you the following output:

```
'data.frame':	6849 obs. of  12 variables:
 $ Crime.ID             : Factor w/ 5339 levels "","00232212ea7
861 3003 2190 900 ...
 $ Month                : Factor w/ 1 level "2019-04": 1 1 1 1
 $ Reported.by          : Factor w/ 1 level "Durham Constabular
 $ Falls.within         : Factor w/ 1 level "Durham Constabular
 $ Longitude            : num  -2.9 -1.77 -1.73 -1.73 -1.74 ...
 $ Latitude             : num  53.2 54.9 54.9 54.9 54.9 ...
 $ Location             : Factor w/ 2403 levels "On or near A1(
 $ LSOA.code            : Factor w/ 432 levels "","E01003305",.
 $ LSOA.name            : Factor w/ 432 levels "","Cheshire Wes
 $ Crime.type           : Factor w/ 14 levels "Anti-social beha
 $ Last.outcome.category: Factor w/ 21 levels "","Action to be
 $ Context              : logi  NA NA NA NA NA NA ...
```

The top row tells you what kind of object R thinks it is (a data frame), and for data frames it will also report the number of rows and columns. Below, the column names are listed following the dollar signs. To the right of each variable name, `str` reports the type of variable R thinks each column contains, such as numeric ('num') integer ('int') or factor (together with the number of factor levels).

You should check carefully to make sure that variables are in the format you expect them to be in. Here, we can already see there might be some issues – R reports blank cells in the data ("") as a factor level for some categorical variables (e.g. Crime.ID), when these should in fact be read as missing data. Later on, we'll cover how to handle missing data but for present purposes this is fine.

`head` and `tail` can be used to display the top and bottom few rows of your data, respectively. While not particularly useful by themselves, they can be helpful for spotting quickly if something has gone terribly wrong when importing the data. For example, try using `head` on the crime data:

```
head(crime_data, n=10)
```

The n= argument indicates how many rows we wish to display, so you can adjust the value to whatever you wish. You can use `tail` in exactly the same way as `head`.

Next, we'll cover several useful ways in which you can manipulate your data in R prior to statistical analysis: subsetting, transforming/creating variables, merging and handling missing data.

Subsetting

The function `subset` lets you select subsets of your data based on a wide range of criteria. Subsetting in R involves creating a new object containing data that meet specified inclusion criteria. For example, let's say we wanted to create a subset of the crime data containing only certain types of crime. We can use the variable 'Crime.type' for this purpose. From the output we saw when we used `str` on the data, we can see that Crime.type is treated as a factor with 14 different levels (i.e. categories). We can ask R to report all the different levels of a factor using the `levels` function:

```
levels(crime_data$Crime.type)
```

R will display the names of all 14 crime types recorded in the dataset below. Remember that the dollar sign in R is used to select a named column from within a dataset.

Then, we could create a subset containing only e.g. bike thefts using `subset` as follows:

```
bike_thefts<-subset(crime_data, Crime.type=="Bicycle theft")
```

If this works correctly, you'll see the new data subset 'bike_thefts' appear in the Environment window. You can then use `str`, `head` and `tail` on this object if you wish to examine it further – this is often a good idea as it will confirm if subsetting has worked as you expected it to.

You can select observations based on a range of different criteria using **relational operators**, which we've encountered before:

Operator	Description
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to
`!x`	Not x

Additionally, we can use **logical operators** to select criteria, including to combine multiple criteria:

Operator	Description
`x | y`	x OR y
`x & y`	x AND y
`isTRUE(x)`	test if x is TRUE

For example, if you wanted to select only anti-social behaviour that occurred on or near Church Street, you would use:

```
church_St_ASB<-subset(crime_data, Location=="On or near Church Street" &
Crime.type=="Anti-social behaviour")
```

Note: you do not have to name your objects the same as my suggestions. You can name them whatever you like and whatever makes most sense to you. You should, however, avoid spaces and special characters (but underscores and full-stops are ok).

Alternatively, you can subset using indexing, but this can be a little trickier at first. When using indexing on a data frame, you can either index across the entire dataset using the following format:

data[criteria]

Or you can index based on criteria applied to rows and columns separately, using this format:

data[row_criteria, column_criteria]

As for vectors, indexing on dataframes can be done based on numeric positions. The command below, for example, would select rows 1 to 5 of the third column in the dataset:

```
crime_data[1:5, 3]
```

Typically, however, it is more useful and convenient to select cases based on whether they meet specific criteria, using relational and/or logical operators. Here's how we would extract a subset of the crime dataset containing only robberies, using indexing rather than the `subset` function:

```
robberies<-crime_data[crime_data$Crime.type=="Robbery",]
```

Because we want to retain only those cases whose *rows* contain the crime type "Robbery", here the selection criterion inside the square brackets should go before the comma. We leave the column criterion (after the comma) blank in this instance, because here we want to retain *all* columns in the data subset. Keep an eye on this comma – it is very easy to forget it or put it in the wrong place.

Transforming & creating new variables

Continuous variables

You can transform your variables in any number of useful ways in R. You can do so by performing some function on a column of data, and either over-writing the existing data with the transformed data, or creating a new column to contain the transformed data. I would always recommend doing the latter to avoid any confusion down the line.

For example, let's say that (for whatever reason) we wanted to round the latitude where each crime occurred to one decimal place. We could do so as follows, using the `round` function:

```
crime_data$Latitude_rounded<-round(crime_data$Latitude, digits=1)
```

You can change the number of decimal places by changing the number specified by the `digits` argument. To confirm that this has worked as expected, you can use `str` or `head`/`tail` to examine the data.

You can transform your variables in numerous other ways using functions, including natural log-transformation (`log`) and square-root transformation (`sqrt`), for example.

You can also use arithmetic operators to transform your variables, including:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

For example, imagine that we wanted to report latitude in tenths of degrees, rather than degrees. We could do so as follows:

```
crime_data$Latitude_tenths<-crime_data$Latitude*10
```

Categorical variables

When preparing categorical data for analysis, you will often wish to re-classify your variables by lumping categories together. One way to do this in R is by using if-else statements, with the function `ifelse` (although this can be a bit of a headache…)

We might, for example, wish to classify crime types as 'violent' or 'non-violent'. Although there are multiple ways to do this, we could decide that "Robbery" and "Violence and sexual offences" are the only two categories we wish to classify as violent crime. If so, we need to create an if-else statement that says crimes are to be classified as violent if they are robberies OR violent/sexual offences, and non-violent otherwise, as follows:

```
crime_data$Violent<-ifelse(crime_data$Crime.type=="Robbery" |
crime_data$Crime.type=="Violence and sexual offences", "Yes", "No")
```

So now we have the new variable 'Violent', which is "Yes" for crimes listed either as robberies or as violent and sexual offences, and "No" for all other types of crimes.

Note: when using ifelse statements, watch out for missing data as these will needed to be treated separately (i.e. R will need to know how to classify these in the new variable).

You can use ifelse statements to re-code categorical variables in numerous other ways, e.g. splitting categories into finer-grained groups or re-naming factor levels.

Changing variable types

We can also transform variables from one type to another (i.e. between numeric, integer, and factor), if appropriate. One common such transformation is splitting a continuous variable at some point, such as the mean or median, to create new binary categorical variable. Let's say for example that we wanted to treat the variable 'Latitude' as a categorical variable with two levels, "North" and "South". We could decide that all latitudes greater than or equal to the median are to be categorised as "North" and those less than the median "South". We could do this using the following if-else statement:

```
crime_data$Latitude_binary<-
ifelse(crime_data$Latitude>=median(crime_data$Latitude), "North", "South")
```

Merging

Next, we'll walk through how to merge two datasets together by a common identifier. For this exercise, you will need first to import the 'outcome' data. We will be merging this dataset with the main 'crimes' dataset using matching crime ID numbers. The 'outcome' file contains details of the outcomes of all crimes with status updates in the previous month[4], so it does not contain the exact same set of observations the original 'crime' dataset.

Now read in the 'outcomes' dataset using `read.csv`. Go back to the commands we covered earlier on in this session, copy-paste them and change the file name appropriately to do this.

Once you've done this, we can merge the crime and outcomes datasets together using the `merge` function, based on their common crime identification numbers (the 'Crime.ID') column. The command below assumes you've read in the outcomes data successfully and named it 'outcomes_data':

```
merged_data<-merge(crime_data, outcomes_data, by="Crime.ID")
```

Now use `str` on the merged dataset. From the output, you will see that the dataset contains a smaller number of rows (3248) than the original crime data (6849) – this is because not all of the crime IDs match between the datasets. By default, `merge` assumes that you want to drop all non-matching cases, but if you want to keep them, you can do so by adding extra arguments. all=TRUE will keep all non-matching cases, while all.x=TRUE will keep all cases in the first dataset even if they cannot be matched with the second, and vice versa for all.y=TRUE[5].

Handling missing data

Re-coding missing data

Missing data can be represented in a variety of different ways in spreadsheets (e.g. using NA, -999, blank cells, etc.). By default, R understands 'NA' to mean missing, so you will run into problems if you use anything other than NA for missing values. For example, see what happens when you run the command below, which checks if any missing values in the 'Crime.ID' column are missing using the `any` and `is.na` functions together:

```
any(is.na(crime_data$Crime.ID))
```

Note: the command above illustrates how functions can be nested within one another in R to perform multiple operations in a single line of code – this can be extremely useful.

R returns 'FALSE', indicating there are no missing values. However, there are in fact many entries with missing crime IDs! We can confirm this by combining the `length` and `which` functions to report how many observations in the crime ID column are equal to `""` (i.e. are blank).

[4] https://data.police.uk/about/#police-outcomes-file

[5] **NOTE:** I haven't provided examples of this here because some issues arise when merging these datasets that complicate things somewhat. Some crimes do not have ID numbers (anti-social behaviour, presumably because this is not actually a crime!) and some have duplicate ID numbers (for reasons unknown). We'll gloss over these issues for today due to time limitations, but this is a useful illustration of the importance of checking everything very carefully during any data manipulation in R…

```
length(which(crime_data$Crime.ID==""))
```

First we'll try directly manipulating the data using indexing, which will allow us to find and replace blank cells with 'NA' for the object containing the crime dataset. We can do this using the command below, which assigns NA to any cells meeting the criteria specified in the square brackets (equal to 'blank'):

```
crime_data[crime_data==""]<-NA
```

If you now examine the data using head or tail, you'll see that previously blank cells are now indicated by NA.

Important: this strategy involves directly manipulating the data frame (i.e. over-writing) so you should be **very** careful to check nothing unexpected happens along the way when using this approach. However, if things do go wrong, you can always re-import the data and start over. The original files are completely unaffected.

Alternatively, when we import the data we could add an argument to instruct R to convert blank cells to NAs, as follows:

```
crime_data<-read.csv("2019-04-durham-street.csv", na.strings="")
```

Removing missing data

Once we're sure that R is interpreting missing data correctly, we might want to remove it prior to analysis. We can do so either by removing rows missing observations for specific variables or by removing rows missing data for *any* variables.

For example, to select a subset containing only those observations with crime IDs, we could use:

```
crime_data_no_missing_IDs<-subset(crime_data, !is.na(Crime.ID))
```

Remember that the exclamation point in R stands for negation, so here we are creating a subset that contains only observations for which the crime identification number is NOT missing (NA).

'Crime.ID', however, is not the only column with missing data. If you wanted to drop rows missing data from multiple columns, you can do so using the AND operator, for example:

```
crime_data_no_missing_vars<-subset(crime_data, !is.na(Crime.ID) &
!is.na(LSOA.code))
```

Finally, if you want to exclude missing data altogether, you can use the `complete.cases` function to select only observations with data for all variables as follows:

```
crime_data_complete<-crime_data[complete.cases(crime_data),]
```

Note: in this instance, the complete dataset will contain no rows, because one column of crime data ('Context') is entirely missing!

Session 2: questions and exercises

1. In R, what should you use to represent missing values in data?

A. na
B. NaN
C. NA
D. nan

2. What symbol should you use to select a named column from a dataframe?

A. $
B. £
C. &
D. #

3. Which of the following means 'is equal to'?

A. !=
B. <-
C. =
D. ==

4. Which of the following represents OR?

A. ll
B. l
C. \
D. /

5. What is contained in the 100th row of the 7th column of the crime dataset? Use indexing to find the answer.

6. How many crimes in the dataset are EITHER burglaries or robberies? Use subsetting to find the answer.

7. How many crimes in the dataset involved theft (i.e. bike theft, burglary, other theft, robbery, shoplifting or theft from the person)? Use an if-else statement to find the answer.

8. How many observations in the crime dataset are missing data for the variable 'Last outcome category'?

9. The command below attempts to extract the crime identity numbers from the main crime dataset. Why does it return 'NULL'?

```
crime_data$Crime_ID
```

10. The command below is an attempt to select only instances of shoplifting from the main crime dataset. Why does it return an error?

```
shoplifting<-crime_data[crime_data$Crime.type=="Shoplifting"]
```

Session 3: Descriptive statistics & exploratory plots

In this session we will be using some recently compiled data on primate species and their conservation status (e.g. endangered, vulnerable, extinct, etc.)[6]. Download the file 'primate_extinction_data.csv' and move it to your working directory. Then import the data file into R using the command `read.csv`, naming the object 'primate_data'. Go back to the material on importing data in the previous session if you need to remind yourself how to do this.

This dataset contains information about 504 primate species including their common and scientific names, conservation indicators, body size, activity patterns and number of publications per species.

Sample sizes

One of the most basic ways in which we need to describe our data is to provide a sample size. To simply check how many rows are in a dataset, you can use `str` to examine the data:

```
str(primate_data)
```

The top row of the output will tell you the number of rows (obs.) and columns (variables):

```
'data.frame':	504 obs. of  9 variables:
```

You could also use `length` to report the number of observations in a single column of data, e.g.

```
length(primate_data$Mass_kg)
```

However, neither of the above commands exclude missing values, so they will not necessarily give you the sample size for your analyses. To count the number of non-missing observations for a single variable, you can ask for the sum of observations that are not missing as follows:

```
sum(!is.na(primate_data$Mass_kg))
```

Alternatively, if you want to check how many variables have complete rows across the whole dataset, you can use `complete.cases` together with `sum`, as follows:

```
sum(complete.cases(primate_data))
```

Here, `complete.cases` is checking through all the rows in the data and telling us which ones are complete (TRUE) and which are not (FALSE). Try running just `complete.cases` alone to see for yourself (although be warned that this will bring up a large output!):

```
complete.cases(primate_data)
```

When we feed the above command to the function `sum`, R will add up the number of TRUEs in the logical vector and thus tell us how many complete rows there are in the dataset.

[6] Estrada et al. (2017), *Science Advances*: https://advances.sciencemag.org/content/3/1/e1600946

If you want to figure out the number of rows with complete cases just for a subset of your data columns, you can use indexing (see Session 1 & 2) to select named columns from the dataset. For example, the command below will tell us how many species have data available for both body mass ('mass_kg') and activity pattern ('Activity', where N = nocturnal, C = cathemeral[7]).

```
sum(complete.cases(primate_data[,c("Mass_kg", "Activity")]))
```

Continuous data

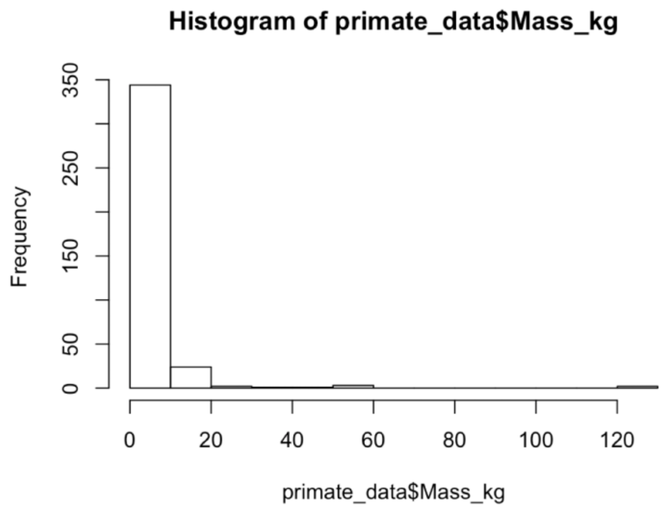
The primate extinction dataset contains two continuous variables we can use to explore descriptive statistics in R: species' average body mass ('Mass_kg') and a measure of the amount of research effort devoted to each species ('Publications'), calculated as the number of published articles on a given species recorded in the Web of Science academic literature database. While body mass is a continuous variable, number of publications is count data (takes only positive integers) and so different descriptive statistics are likely to be appropriate for each one.

Histograms

The most effective way of figuring out how best to describe data can be simply to plot out their distributions using histograms and visually examine them, which we can do using the command `hist`, e.g.

```
hist(primate_data$Mass_kg)
```

You should see a histogram pop up in the bottom right window, like the one below. Next, we will explore how to tweak graphical parameters so that R displays the plot in a more useful way.



The first problem with the above plot is that the data are broken up into pretty coarse 10kg 'bins'. All this really tells us is that the vast majority of primate species are 10kg or less, while a few unusual species are much larger. By default, R will choose break-points for you that it believes are sensible,

[7] Meaning irregular activity patterns (not strictly nocturnal or diurnal).

but you can alter them using the 'breaks' argument, which specifies how many bins you want to divide the data into. The bigger the number, the more breaks and the smaller the bins.

Try for example:

```
hist(primate_data$Mass_kg, breaks=20)
```

Behind the scenes, R actually uses an algorithm to break up the data so the number you give it may not correspond exactly to the number it shows, and at a certain point it will refuse to give you any narrower bins, but you can usually tweak this argument to get the data displayed in a more helpful way.

When you have a highly skewed distribution like this, you might want to plot just a limited range of the data, say species weighing up to 40kg. We can do this by adjusting the `xlim` plotting argument (i.e. 'x-axis limits') within `hist`. `xlim` needs two numbers combined (using `c`) in a vector, the first referring to the lower limit and the second to the upper limit, e.g.

```
hist(primate_data$Mass_kg, breaks=20, xlim=c(0,40))
```

Measures of central tendency

You can calculate means and medians using the functions `mean` and `median`. For example, you can calculate the mean body mass across the primate species using:

```
mean(primate_data$Mass_kg, na.rm=TRUE)
```

It is important to keep the 'na.rm' argument set to TRUE as this instructs R to remove missing values. If you do not include this, the command will just return 'NA'.

This tells us that the average primate species weighs around 5.19kg. However, as you might have anticipated, calculating a mean is probably not appropriate for this variable since the data are so skewed. The mean will be strongly pulled upwards by the small number of very big primate species (e.g. gorillas weighing >100kg). It would probably be more appropriate to calculate the median, which we can do in exactly the same way as the mean:

```
median(primate_data$Mass_kg, na.rm=TRUE)
```

As anticipated, the median is a much lower value (2.5kg), which probably better captures the weight of an 'average' primate species compared with the mean.

Like many distributions in biology, the primate body masses appear to have a **log-normal distribution**. This means that the log-transformed values will be approximately normally distributed. We can investigate this by plotting a histogram showing data on a log-10 scale, as below:

```
hist(log10(primate_data$Mass_kg), breaks=20)
```

This distribution actually looks more bimodal than normal, but had the data turned out normal we could potentially calculate a mean of the log-transformed data, if that would be helpful:

```
mean(log10(primate_data$Mass_kg), na.rm=TRUE)
```

Finally, for count data we may wish to calculate the mode instead of the mean or median. Annoyingly, there is no built-in function to calculate the mode in R (there is a function called ‘mode’, but it does something completely different). We can figure out the mode using a frequency table, with the function `table`, although unfortunately this is quite fiddly.

First let’s look at using `table` by itself. Use the command below to produce a frequency table for ‘Publications’:

```
table(primate_data$Publications)
```

In the table R returns, the odd numbered rows are the data and even numbered rows are the counts (so for example, two species each have 30 publications associated with them. Now, save this table as an object called ‘freqs’:

```
freqs<-table(primate_data$Publications)
```

Then, you can create a re-ordered table using the `order` command to sort the data from greatest to least (by setting the `decreasing` argument to TRUE):

```
freqs[order(freqs, decreasing=TRUE)]
```

Now the table is presented in descending order, so the top-left cell will show the most frequent value, which in this case is 1 – so most species have only a single scientific publication each. To make this somewhat less confusing, you could add [1] to the command above so that it displays only the first value (i.e. the one with the highest count):

```
freqs[order(freqs, decreasing=TRUE)][1]
```

Or, if you prefer to return the mode in a single (but potentially more confusing!) line of code, you can use:

```
table(primate_data$Publications)[order(table(primate_data$Publications),
decreasing=TRUE)][1]
```

Measures of variation

When reporting descriptive statistics for continuous variables, we typically need to report not only a measure of central tendency but also a measure of how much the data vary around that ‘average’ value. For normally distributed variables we can use the command `sd` to report the standard deviation, while for non-normally distributed variables it is probably more appropriate to use the `IQR` function to report the inter-quartile range. We can also report the range using `range`. These three commands should all be entered in the same format as `mean` and `median`, as follows:

function(data$variable, na.rm=TRUE)

Try now adapting the template above to calculate the standard deviation for primate body masses (although this probably is not appropriate given their distribution!) and the IQR for publications. Don’t forget to include the ‘na.rm’ argument and set it to TRUE, otherwise the commands will not return anything but ‘NA’ if there are missing values in the data.

Scatterplots

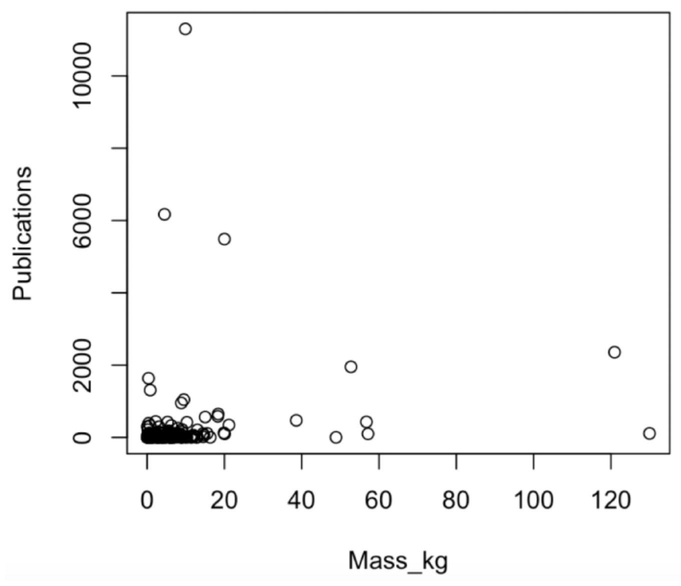
Often it can be very illuminating to visualise continuous data using scatterplots prior to any analysis. This can help us spot any unanticipated or concerning patterns in the data, potential data errors, outliers and so on. At this point, we will introduce the extremely useful `plot` command. `Plot` is a highly flexible function, which can produce many different types of plots for you depending on what objects and arguments you feed into it. If we give it two continuous variables, it will create a scatterplot by default. Let's first try making a basic scatterplot, for primate body masses against number of publications. We'll need to input a command using the format below:

plot(Y_variable~X_variable, data)

R uses the tilde (~) to distinguish the variable on the Y axis (to the left of the tilde) from the variable on the X axis (to the right of the tilde). Try the command below, placing publications on the Y axis and body masses on the X axis:

```
plot(Publications~Mass_kg, primate_data)
```

And you should see the following plot appear in the bottom-right corner:



Next, we'll look at some ways to display the data in a more useful way. First of all, it is hard to make sense of this plot since most of the data are clustered together in the bottom left corner. This is because both the variables have highly right-skewed distributions (i.e. many small values, few large values). It might therefore make more sense to plot out both variables on log-10 scales, as follows:

```
plot(log10(Publications)~log10(Mass_kg), primate_data)
```

Now we can see a pattern emerge in the data – it looks as if larger-bodied primate species generally are better studied than smaller-bodied species. This might be useful to know for subsequent analyses, as this could be an important source of bias in the data.

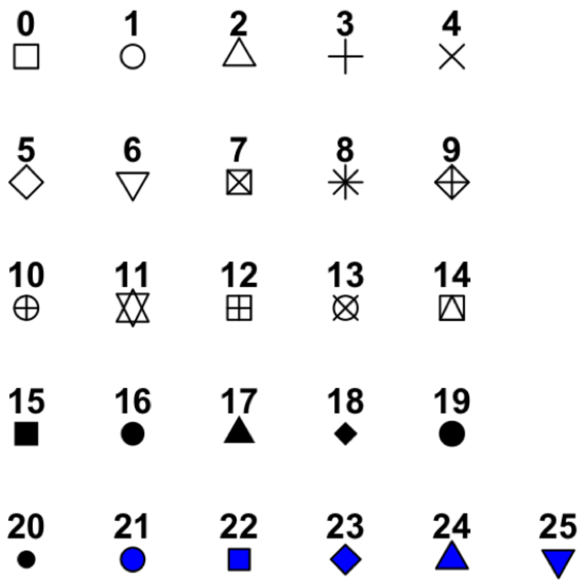
Before going further, we'll make a couple of graphical tweaks to improve the plot further. First, by default, R will display the names of the variables as specified in the dataset on the X and Y axis labels. We probably want to change these slightly, which we can do by specifying 'xlab' and 'ylab' arguments within `plot`. For example, try the command below:

```
plot(log10(Publications)~log10(Mass_kg), primate_data, ylab="Log10 Number
of publications", xlab="Log10 Body mass")
```

We might also want to rotate the Y axis numbers so that they are horizontally oriented. We can do this by adjusting the argument 'las' (which stands for 'label axis style'). Las requires a number from 0 to 3, in which 0 = both labels parallel to axes, 1 = Y axis perpendicular, X axis parallel, 2 = both perpendicular and 3 = Y axis parallel, X axis perpendicular. So in this case, we want to select 1, as follows:

```
plot(log10(Publications)~log10(Mass_kg), primate_data, ylab="Log10 Number
of publications", xlab="Log10 Body mass", las=1)
```

Finally, we might wish to change the symbol and colour of the plot points. To change the symbol, we need to add the argument 'pch' ('plotting character'), which takes a number from 0 to 25 corresponding to the following symbols:



To change the colour, we need to include the argument 'col'. We can specify colours in a number of different ways, including using words (e.g. "red", "blue", etc.[8]).

For example, the command below will change the plotting symbols to filled, purple circles:

```
plot(log10(Publications)~log10(Mass_kg), primate_data, ylab="Log10 Number
of publications", xlab="Log10 Body mass", las=1, pch=19, col="purple")
```

Sometimes when we have lots of plotting points overlapping, it can be useful to use transparent colours to get a sense of the density of the data. If we want to do this, we can set the colour using the `rgb` ('red green blue') function within the plotting command. `rgb` requires four arguments – a

[8] See e.g. http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf for a full list.

value for red, green, blue and 'alpha', which sets the transparency, all of which vary from 0 to 1. For example, the command below will specify purple plotting points with 50% transparency:

```
plot(log10(Publications)~log10(Mass_kg), primate_data, ylab="Log10 Number
of publications", xlab="Log10 Body mass", las=1, pch=19,
col=rgb(1,0,1,0.5))
```

For the plotting symbols 21-25, it is possible to set different colours for the outline and fill. 'Col' will set the colour of the outline, while 'bg' ('background') sets the colour of the filled area. The command below, for example, will plot grey circles with black outlines:

```
plot(log10(Publications)~log10(Mass_kg), primate_data, ylab="Log10 Number
of publications", xlab="Log10 Body mass", las=1, pch=21, col="black",
bg="grey")
```

Outliers

We can often spot potentially problematic outliers in the data just by visualising the data in scatterplots. However, if we want to use a more formal approach, we could look for values falling outside of a certain range of the data (e.g. more than 3 standard deviations from the mean, or falling outside of 95% of the values in the data). These values aren't necessarily going to be data entry errors or otherwise problematic, as with large distributions we would expect to get quite a few extreme values anyway. But you might wish to investigate such values further just in case.

The approach below is slightly long-winded – in practice you might wish to find a more 'ready-made' function to do this instead (try for example the package 'outliers') – but this way is probably more helpful for honing our R skills.

Let's say first we want to identify primates with body masses more than 3 standard deviations larger than the mean. First, we would need to calculate the standard deviation multiplied by three, as follows:

```
sd(primate_data$Mass_kg, na.rm=T)*3
```

Then we'd need to add this to the mean to find our cut-off value:

```
mean(primate_data$Mass_kg, na.rm=T)+sd(primate_data$Mass_kg, na.rm=T)*3
```

Now we know that we want to find out which species are larger than 38.6 kilograms. We could do this using the `subset` function, which we came across earlier:

```
subset(primate_data, Mass_kg>mean(primate_data$Mass_kg,
na.rm=T)+sd(primate_data$Mass_kg, na.rm=T)*3)
```

The above command is asking R to select only those rows for which the body mass is greater than the mean plus three standard deviations, and it should return two gorilla species, two orangutan species, the chimpanzee and a species of langur monkey. The inclusion of the latter species is potentially surprising as the list does not include some other large species (e.g. bonobos, gibbons), and so might warrant further investigation were you to use these data for further analysis.

If you wanted to identify values either more than *or* less than 3 SDs from the mean[9], you could modify the command as follows:

```
subset(primate_data, Mass_kg>mean(primate_data$Mass_kg,
na.rm=T)+sd(primate_data$Mass_kg, na.rm=T)*3 |
Mass_kg<mean(primate_data$Mass_kg, na.rm=T)-sd(primate_data$Mass_kg,
na.rm=T)*3)
```

This command is now getting a bit unwieldy and hard to follow! Here we've added an additional condition, specifying that we want to subset values either more than 3 SDs greater than the mean *or* (using '|') more than 3 SDs less than the mean.

Given the distribution of the primate body masses, however, it would make more sense to identify outliers using quantiles, as this does not require assuming that the data are normally distributed. For example, we might want to find out which species have body masses that fall outside of the range of 95% of the data. We can do this using the `quantile` function. If we want to identify values falling below or above the central 95% of the distribution, we want to identify the 2.5% and 97.5% quantiles, which we could do as follows (not forgetting to set na.rm to TRUE):

```
quantile(primate_data$Mass_kg, c(0.025, 0.975), na.rm=TRUE)
```

So now we know that 95% of the primate species in the sample weigh between 0.05 and 19.95 kg. Putting this together with the `subset` function, we can identify species who weigh either less than the 2.5% quantile OR more than the 97.5% quantile:

```
subset(primate_data, Mass_kg<quantile(primate_data$Mass_kg, 0.025,
na.rm=TRUE) | Mass_kg>quantile(primate_data$Mass_kg, 0.975, na.rm=TRUE))
```

From a quick glance at the data, these extreme values do not look particularly worrying – most of them are indeed species that are known to be very large (e.g. gorillas, baboons) or very small (mouse lemurs).

Categorical data

Frequencies & percentages

We can use R to calculate frequencies and percentages when describing categorical variables. The most important function to get to know here is `table`, which we already encountered earlier when calculating the mode. First, let's take a look back at the data to remind ourselves which are categorical variables:

```
str(primate_data)
```

We have a few to choose from, but let's start with 'Red_List_status' – this is the IUCN's classification of the species' conservation status. This is treated as a factor on 7 levels, which we can view using the `levels` function:

```
levels(primate_data$Red_List_status)
```

[9] Subtracting 3 standard deviations from the mean in this case gives us a negative number, which does not make much sense – you cannot have a primate weighing minus 5 kg for example! This nicely illustrates that you may get nonsensical results if you use means and standard deviations to describe skewed data.

CR = ‘critically endangered’, DD = ‘data deficient’, EN = ‘endangered’, LC = ‘least concern’, NE = ‘not evaluated’, NT = ‘near threatened and VU = ‘vulnerable’.

To create a frequency table, we just need to feed the variable to the function `table`, as follows:

```
table(primate_data$Red_List_status)
```

This variable happens not to have any missing values, but it is good practice to always include an additional argument telling table to also list how many cases are missing, if there are any, as follows:

```
table(primate_data$Red_List_status, useNA="ifany")
```

If we wanted to express the frequencies as proportions, we need to feed the whole command above to the function `prop.table`, as follows:

```
prop.table(table(primate_data$Red_List_status, useNA="ifany"))
```

Depressingly, this tells us that over a quarter of primate species are endangered. If we wanted to express these as percentages, we just need to multiply the whole thing by 100:

```
prop.table(table(primate_data$Red_List_status, useNA="ifany"))*100
```

Quite often we are interested in cross-tabulated frequencies for more than one variable. For example, perhaps we want to know the frequencies/proportions for conservation status by activity pattern (N = ‘nocturnal’ or C = ‘cathemeral’). We could do so by feeding both variables of interest to the `table` function, as follows:

```
table(primate_data$Red_List_status, primate_data$Activity, useNA="ifany")
```

Now that we have two variables, if we want to calculate proportions we need to specify whether we want the proportions across the rows or down the columns. We therefore need to include an extra argument ‘margin’ within `prop.table`, in which 1 indicates rows and 2 columns. For example, the command below will figure out the proportions across the rows, i.e. the proportion of nocturnal, cathemeral and NAs for each conservation status category:

```
prop.table(table(primate_data$Red_List_status, primate_data$Activity,
useNA="ifany"), margin=1)
```

Barplots

Barplots are one effective way to display frequencies, and thankfully basic barplots are relatively straightforward in R. In this section we’ll explore how to use the `barplot` function. You need to feed the `barplot` command a series of values to plot, which in this case are frequencies or proportions, but could alternatively represent, for example, a series of means. One way to do this is to feed `barplot` a frequency table for a single variable, for example:

```
barplot(table(primate_data$Red_List_status, useNA="ifany"))
```

There are many ways that we could improve this plot to make it both more visually appealing and useful. One of the first things we might wish to do is change the order of the bars. By default, R will plot them in alphabetical order, but it might be more helpful to arrange them by frequency. To make the code a bit less cumbersome, it might help to first save the frequency table as an object:

```
IUCN<-table(primate_data$Red_List_status, useNA="ifany")
```

We can re-order the categories from most to least frequent using the `order` function, which we encountered earlier:

```
IUCN[order(IUCN, decreasing=TRUE)]
```

Then, we can paste all of the above into the brackets after barplot plot the bars in order of their frequencies, from greatest to least:

```
barplot(IUCN[order(IUCN, decreasing=TRUE)])
```

If we wanted to order the bars from least to greatest, we would just set 'decreasing' to FALSE or leave it out (as this is the default setting anyway).

We could add some nicer colours by using the `rainbow` function. If you feed `rainbow` a single number, it will create a vector of rainbow colours[10] from red to violet. So for example if we set colours as `rainbow(7)`, it will assign one colour of the rainbow to each of our 7 conservation categories:

```
barplot(IUCN[order(IUCN, decreasing=TRUE)], col=rainbow(7))
```

`barplot` shares many of the same arguments as `plot`, so we can use what we learned from creating scatterplots earlier to rotate the Y axis numbers and add X and Y axis labels:

```
barplot(IUCN[order(IUCN, decreasing=TRUE)], col=rainbow(7), las=1,
ylab="Frequency", xlab="IUCN Red List category")
```

We might also want to add a legend to decode the abbreviations on the X axis, which we can do using the function `legend`. `legend` will 'paste' a legend onto the existing plot, so if you make mistakes you will need to create the plot afresh before attempting to re-do the legend (otherwise it will just paste multiple legends on top of one another!). This function first requires a location, which can either be given as x and y coordinates or simply using words ("bottomright", "topright", "bottomleft" or "topleft") as the first argument, followed by the contents of the legend ('legend'), the plotting symbol (here 15 indicating filled squares) and the colours (again 7 colours from across the rainbow spectrum). We have to be really careful here that the colours and labels match up, as R won't help us out if we make a mistake – it will just blindly do what we tell it to!

```
legend("topright", legend=c("LC=Least Concern", "EN=Endangered",
"VU=Vulnerable", "CR=Critically endangered", "NE=Not evaluated", "NT=Not
threatened", "DD=Data deficient"), pch=15, col=rainbow(7))
```

Note that for the argument indicating the legend's location, we have not specified the name of the argument (i.e. 'x="topright"'). This is because R can either interpret arguments based on names or

[10] Rainbow colour schemes can be pretty, but they are not particularly helpful to the many people who have colourblindness! There are R packages available, however, that provide more colourblind-friendly colour palettes, such as viridis: https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

on the order in which they are received. `legend` ‘expects’ the first argument to contain the location, so unless we tell it otherwise, this is what it will assume. When learning R, however, it is often safer to specify arguments with names so that we have a better idea of what we are doing! You can use the help (e.g. `?legend`) command to remind yourself of the order of arguments for a given function.

In this case the legend comes out rather large compared to the rest of the plot. To make it smaller, we can introduce another very useful plotting argument – ‘cex’ (character expansion factor). ‘cex’ takes one number indicating the relative size of the plotting characters, where 1 is the default, 2 indicates twice the size, 0.5 half the size and so on. Try different values of cex until you get something that looks more reasonable:

```
legend("topright", legend=c("LC=Least Concern", "EN=Endangered",
"VU=Vulnerable", "CR=Critically endangered", "NE=Not evaluated", "NT=Not
threatened", "DD=Data deficient"), pch=15, col=rainbow(7), cex=0.5)
```

If we wanted to plot out the cross-tabulated frequencies for two variables, we can also use barplot, just by feeding it the appropriate table. For example, let’s say we wanted to plot out the proportions of species in each IUCN Red List category split by activity pattern (nocturnal or cathemeral). To keep things a bit simpler, we’ll exclude missing values. First, we’ll create the appropriate proportions table and save it as an object:

```
IUCN_activity_props<-prop.table(table(primate_data$Red_List_status,
primate_data$Activity), margin=2)
```

Note that we’ve now asked for the *column* rather than row percentages (because we want to show proportions of IUCN categories by activity pattern, not the other way around).

Then, we can create a stacked barplot by feeding the proportion table to `barplot`:

```
barplot(IUCN_activity_props)
```

And we could use the same graphical tweaks we covered earlier to alter the axes, plotting colours and so on. In this particular case it is most intuitive to use stacked bars, but if instead we wanted to plot all the bars side-by-side, we could just include the argument ‘beside’ and set it to TRUE:

```
barplot(IUCN_activity_props, beside=TRUE)
```

Descriptives for data subsets

In the last section for this session, we’ll cover how to calculate descriptive statistics for specific subsets of data. Some of this section will overlap with what we have already covered on subsetting and indexing. Here we’ll cover three ways to do this: subsetting, aggregating and indexing.

Perhaps the simplest way to calculate descriptives for data subsets is to create a subset, then use whatever functions we need to use on that subset to calculate the descriptives. For example, here is how you would calculate the mean body mass for nocturnal species using this method:

```
nocturnal_species<-subset(primate_data, Activity=="N")

mean(nocturnal_species$Mass_kg, na.rm=TRUE)
```

Although relatively straightforward, this method has the disadvantage of becoming pretty cumbersome if you want to calculate descriptives for lots of subsets. If that's the case, you might be better off using the function `aggregate` – this can be used to perform the same function across multiple data subsets all at once. For example, to calculate mean body masses for each IUCN Red List category, you could use `aggregate` as follows:

```
aggregate(Mass_kg~Red_List_status, primate_data, FUN=mean,
na.action=na.omit)
```

The first part inside the brackets specifies the formula you want to use to aggregate, where the variable on the right of the tilde is the grouping variable (think of it as standing for 'by' in this case). Then we include the name of the dataset, the function we wish to use ('FUN=') and finally what we want to do with missing data (in this case 'na.action=na.omit' i.e. exclude NAs[11]). This will produce a handy table (which we could also use to create a barplot if we wanted to). We can replace `mean` with any function we wish to, such as `median`, `sd`, `range` and so on.

Finally, we could calculate descriptives for subsets using indexing. This can be neater because it does not require you to create lots of objects (as does subsetting) and it doesn't provide with you with a whole lot of potentially irrelevant information (as might aggregating). It is however the least beginner-friendly option. First let's remind ourselves how to select subsets using indexing. We need the name of the dataset, followed by some square brackets containing the conditions used to select cases. For example, the command below will select just nocturnal species:

```
primate_data[primate_data$Activity=="N",]
```

Note the comma after "N" – it indicates we are selecting rows rather than columns, so is really important yet very easy to misplace! This will produce all of the rows that meet the criterion, but we are only interested in one column in particular – the body masses. So, we can add $Mass_kg on the end of the command to extract only those values:

```
primate_data[primate_data$Activity=="N",]$Mass_kg
```

Now, we just need to paste all of the above into the brackets following our chosen descriptive function, such as 'mean' (not forgetting the na.rm argument!). Putting this all together, we could use the following lines to calculate the mean body masses for nocturnal and cathemeral species separately:

```
mean(primate_data[primate_data$Activity=="N",]$Mass_kg, na.rm=TRUE)

mean(primate_data[primate_data$Activity=="C",]$Mass_kg, na.rm=TRUE)
```

Session 3: Questions and exercises

1. How many species have available data on activity pattern?

2. How many species have available data on both activity pattern and number of publications?

3. What is the median number of publications per species?

[11] Don't ask me why it isn't 'na.rm=TRUE' as it was for the other functions we encountered so far – I have no idea.

4. How much do the smallest and largest primate species weigh?

5. What is the inter-quartile range for number of publications?

6. What is wrong with the command below?

```
barplot(primate_data$Red_List_status)
```

7. What is wrong with the command below?

```
plot(Publications, Mass_kg, primate_data)
```

8. Which primates are larger than 99% of all other species?

9. What percentage of endangered primates are nocturnal (ignoring missing values)?

10. What is the mean body mass for endangered primate species?

Session 4: inferential tests & plotting results

Now that we have introduced descriptive statistics and exploratory plots, we are ready to move on to hypothesis testing using inferential tests and illustrating results with plots. In this section we'll walk through how to run a handful of basic inferential tests in R including chi-square tests, t-tests, ANOVA and correlations. This session isn't intended to give you an exhaustive overview of how to conduct these tests, nor the mathematical theory behind them, but it will help you get up and running efficiently when it comes to running your own statistical analyses in R.

For the remaining sessions, we'll be working with a publicly available dataset of babies' birth weights obtained from the US National Center for Health Statistics[12]: 'US_territories_birth_data_2018.csv'. This contains data for >25,000 births from US overseas territories (e.g. Puerto Rico, Guam and Samoa) for the year 2018. The main focus of our analyses will be to investigate how babies' birth weight may relate to parental characteristics such as age and smoking behaviour.

Save a copy of the file in your working directory and import it into R using `read.csv`, naming the object 'birth_data'. Then, examine the object using `str`. You should see that the dataset contains 25919 rows and 18 columns.

Chi-square tests

The first basic statistical test we will introduce is the **chi-square test-of-independence**. This test can be used to find out if there is a statistical association between two categorical variables. Here, we'll use it to find out whether smoking during pregnancy is associated with low birth weight. In the present dataset, the variable 'Smoker' indicates whether the babies' mothers smoked cigarettes during the pregnancy ('Y') or not ('N'), while 'Low_birth_weight' indicates whether babies meet the World Health Organisation's definition of low birth weight (2,499g or less = 'Y', otherwise 'N').

Before we run the test, create a frequency table using the command below:

```
table(birth_data$Low_birth_weight, birth_data$Smoker)
```

The first variable in the brackets corresponds to the rows of the table (i.e. low birth weight), and the second to the columns (i.e. smoker). To make things more straightforward when interpreting and plotting the results down the line, it is a good idea to treat the dependent variable as the rows and the independent variable as the columns. Then, to perform a chi-square test, we need to feed the above code for the frequency table into the function `chisq.test`, as follows:

```
chisq.test(table(birth_data$Low_birth_weight, birth_data$Smoker))
```

You should see the following output, which we'll walk through next:

```
	Pearson's Chi-squared test with Yates' continuity correction

data:  table(birth_data$Smoker, birth_data$Low_birth_weight)
X-squared = 28.011, df = 1, p-value = 1.206e-07
```

The first line of the output confirms which test was run. Yates's continuity correction is applied by default, which tweaks the calculation of the significance values in order to make the test more

[12] https://www.cdc.gov/nchs/data_access/vitalstatsonline.htm

conservative (i.e. less likely to produce a false positive). If you do not want to apply this correction, you just need to include an additional argument `correct=FALSE` inside the brackets. The second line just confirms what data went into the analysis, while the third line shows us the values we need to interpret the result. Here we have the test statistic, the degrees of freedom and the p-value. When p-values are very small, R displays them in scientific notation in which 'e' stands for 'multiplied by ten to the power of…'. So here we have a very small p-value, which definitely meets the conventional significance threshold of 0.05.

The output above strongly suggests that there is a statistical association between smoking during pregnancy and birth weight, but it does not tell us the directionality of this effect. The best way to figure this out is to simply look at the proportions, and visualise them using a barplot. First, calculate a proportions table using `prop.table` as below:

```
prop.table(table(birth_data$Low_birth_weight, birth_data$Smoker),
margin=2)
```

Remember that low birth weight corresponds to the rows, and smoking to the columns. Therefore, we need to calculate the proportions down the *columns* – i.e. the proportion of babies with low birth weight among non-smokers (column 1) versus smokers (column 2). If we were interested in the proportions across the rows, we would set the `margin` argument to 1 instead.

Then, we can feed the proportion table to the `barplot` function, including some graphical improvements that we covered earlier:

```
barplot(prop.table(table(birth_data$Low_birth_weight, birth_data$Smoker),
margin=2), las=1, col=c("blue", "red"), xlab="Smoking during pregnancy")
```

Looking at this plot together with the proportions, we can see that as we might have anticipated, low birth weight is more common among babies whose mothers smoked during pregnancy.

Because the chi-square test is a non-parametric test, it makes very few assumptions about the data. It does, however, assume that the expected frequencies are at least 5 in each cell of the table. R calculates the expected frequencies in the background when performing the chi-square test, so if we want to extract them, we can re-run the test adding '$expected' on to the end, as follows:

```
chisq.test(table(birth_data$Low_birth_weight, birth_data$Smoker))$expected
```

In this case, we do not have any issues, but if we had found any expected frequencies less than 5 we may have been better off using Fisher's exact test instead. This can be performed using the function `fisher.test` in the same format as the `chisq.test` function, for example:

```
fisher.test(table(birth_data$Low_birth_weight, birth_data$Smoker))
```

Independent-samples T-test

We can use the **independent-samples T-test** to find out whether there is a significant difference between two means. For example, we could use this test to investigate whether mothers who smoked during pregnancy give birth to, on average, lighter babies than those who did not smoke during pregnancy (now treating birth weight as continuous, rather than categorical as we did in the previous exercise).

Before we use the T-test, there are a couple of statistical assumptions we should verify – first that the data are normally distributed within both of the groups, and that there is roughly equal variance around the mean between the groups.

Checking for normality

Often we can get a pretty good sense of whether data are normally distributed simply by visually examining histograms. We can do this in R using `hist`, together with indexing to select specific subgroups of the data (alternatively we can use `subset`, but indexing is neater). To make sense of this, let's first remind ourselves how to plot a histogram for the dependent variable (birth weight) across the whole sample:

```
hist(birth_data$Birth_weight)
```

To use indexing to select specific cases, we will need to insert some square brackets after the name of the dataset, but before the dollar sign that selects a specific column. Inside the square brackets will be our conditions, which must be placed before a comma to indicate that these conditions apply to *rows* of the dataset rather than columns. So in this case, we need to paste in for example `[birth_data$Smoker=="Y",]` to select only rows containing smokers. Now, use these steps to plot out the histograms of birth weight for each group separately, and see if you think the data are approximately normal.

While visually examining the data is often very effective for getting a sense of how they are distributed, you might wish to confirm normality using a formal statistical test such as the **Shapiro-Wilk test**. Theoretically we can run this test quite straightforwardly using the function `shapiro.test`, which will compare an empirical distribution to a theoretical normal one with the same mean and standard deviation as the real data. We can perform the test on specific subgroups of the data using indexing, in the same way as we have just done for the histograms:

```
shapiro.test(birth_data[birth_data$Smoker=="Y",]$Birth_weight)
```

And you should see the output below:

```
	Shapiro-Wilk normality test

data:  birth_data[birth_data$Smoker == "Y", ]$Birth_weight
W = 0.97407, p-value = 0.0007729
```

All we really need to pay attention to here is the p-value. Since this is very low, this means the data are *significantly different from normal* i.e. violate one of the assumptions of the T-test.

When we try to perform the test for the non-smokers, however, we'll get a warning message:

```
shapiro.test(birth_data[birth_data$Smoker=="N",]$Birth_weight)
```

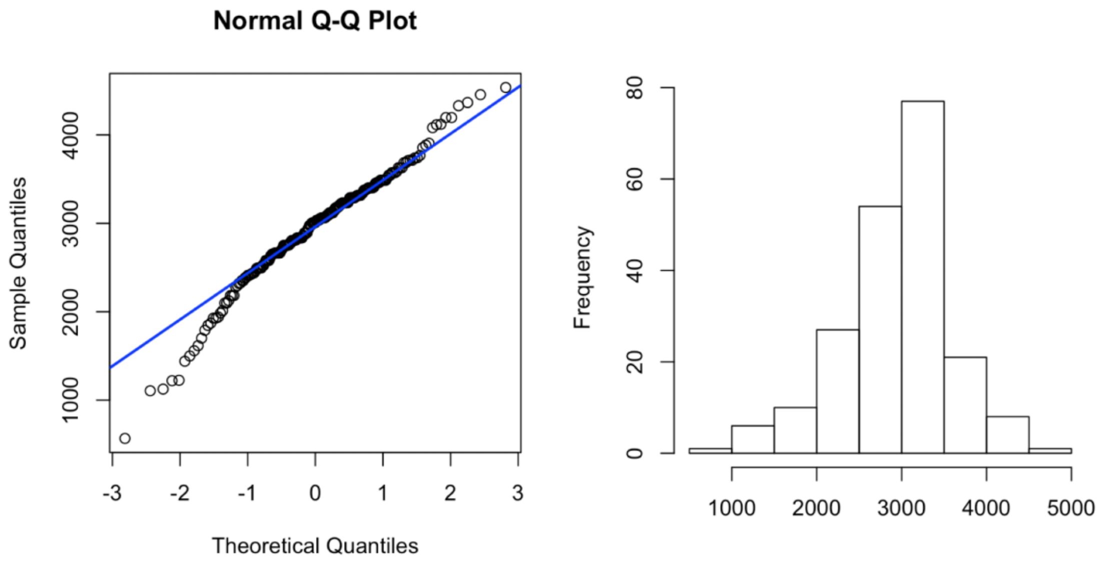
```
Error in shapiro.test(birth_data[birth_data$Smoker == "N", ]$Birth_weight)
  sample size must be between 3 and 5000
```

The function won't accept samples larger than 5000, while our dataset contains >25,000 non-smokers! This actually highlights one of the limitations of formal normality tests: they will inevitably report that distributions are significantly different from normal with sufficiently large sample sizes. In that case, you might be better off just visually examining the data after all! Quantile-quantile plots can help you do this more rigorously than just looking at histograms. QQplots sort the data in if order and compare the proportions of the data falling in each quantile with those expected should the data were normally distributed. We can create a qqplot using the functions `qqnorm` (to create the plot) and `qqline` (to add the line representing the theoretical normal distribution). The two commands below will create a qqplot for birthweight within the smoking group:

```
qqnorm(birth_data[birth_data$Smoker=="Y",]$Birth_weight)
```

```
qqline(birth_data[birth_data$Smoker=="Y",]$Birth_weight, col="blue", 
lwd=2)
```

The ‘lwd’ argument in the second line of code controls the line thickness (default = 1). Examining this plot (below left), we can tell that our sample has fewer observations at the lower end of the scale and more observations at the higher end of the scale than it should were it perfectly normally distributed. Looking back at the corresponding histogram (below right), this makes sense: the distribution does lean a little towards the right, with a bit of a tail pointing towards the left. If you were to do the same for the non-smoking group, you would find the same pattern.



A distribution that is not-quite, but possibly almost, normal, is a typical situation when performing statistical analysis in practice, and often one the textbooks do not well prepare you for! When deciding which tests to use based on the normality of your data, it is usually good practice not just to rely on one simple heuristic but rather consider multiple assessments together with some ‘common sense’ judgements based on your prior knowledge of the data – there won’t necessarily be a single right answer. Let’s assume, for now, however that these data are normal enough for the T-test!

Checking for homogeneity of variance

We can use **Levene’s test** to check for homogeneity (i.e. similarity) of variance between the groups. R does not have a built-in function for Levene’s test, but there is one available in the ‘car’ (‘Companion to Applied Regression’) package. To use it, we’ll need first to install (if you haven’t already done so) and load the package using the commands below (see Session 1 for additional help if needed):

```
install.packages("car")
library("car")
```

Now you're ready to check for homogeneity of variance using the function `leveneTest`:

```
leveneTest(Birth_weight~Smoker, data=birth_data)
```

The first argument contains the formula, in which as we've seen before, the variable to the left of the tilde indicates the dependent variable and the one to the right the independent variable. Then, the second argument contains the data. You should see the following output:

```
Levene's Test for Homogeneity of Variance (center = median)
         Df F value   Pr(>F)
group     1  8.6037 0.003358 **
      25578
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The p-value for this test is the value in the column 'Pr(>F)' (highlighted above). Since it is less than 0.05, following the conventional threshold we can conclude that the variance *significantly differs* between the groups and thus the assumption of homogeneity of variance is violated[13]. Note that R uses asterisks and full-stops to code different levels of significance in some outputs.

Running the test

Just to recap: we are planning to use an independent samples T-test to test the prediction that babies' birth weight (here treated as a continuous variable, measured in grams) is lower for those born to smoking than non-smoking mothers. We've found that the data do not meet all the assumptions of the T-test: they are not perfectly normally distributed, and do not have equal variance between the groups. For now we'll assume the data are normal enough and proceed with the test anyway, but we will need to include a correction for non-equal variances. We can do so using the function `t.test` as follows:

```
t.test(Birth_weight~Smoker, data=birth_data, var.equal=FALSE)
```

If we had found the variances did not significantly differ between the groups, we would have set 'var.equal' to TRUE instead. You should see the following output:

```
      Welch Two Sample t-test

data:  Birth_weight by Smoker
t = 4.2927, df = 206.41, p-value = 2.714e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 104.4796 281.9625
sample estimates:
mean in group N mean in group Y
       3132.982        2939.761
```

According to the results detailed in the third line of the output, we have a significant difference between the groups. Helpfully, the output also includes the group means so we can figure out the

[13] We should again, however, bear in mind that pretty much any difference in variance between groups will be significant for very large samples…

direction of effects: here, we can see that the mean birth-weight for babies born to non-smoking mothers (‘group N’) is greater (3,133g) than the mean birth-weight for smoking mothers (2,940g), confirming the prediction.

Non-parametric alternative

Alternatively, if we were not convinced that the data were sufficiently close to normally distributed to justify using a parametric test, we could use a non-parametric alternative to the independent samples t-test, such as the **Wilcoxon rank-sum test** (also known as the **Mann-Whitney test**). Like many other non-parametric tests, this one will perform a test on ranked data rather than the original values, to get around the assumption of normality. You can perform this test using a command in the same format as the t-test, as follows:

```
wilcox.test(Birth_weight~Smoker, data=birth_data)
```

Reassuringly, we find the same result – there is a significant difference in birth-weights between the smoking and non-smoking group. It looks as if in this case, the effect is strong enough that it will be consistently identified across different methods, which is often a good sign.

The results of T-tests can be effectively illustrated using points and error bars, but creating such a plot using R’s base functions is surprisingly difficult and time consuming. We will leave this for now until the next section on **ANOVA** to make things more efficient. Later on, Lisa will show you the numerous advantages of plotting using the ‘ggplot2’ package instead of the base functions.

If, however, we want to illustrate the results of a Wilcoxon rank-sum test, we can produce a plot showing means, interquartile ranges and extreme values fairly straightforwardly using the `boxplot` function. For example, try the command below:

```
boxplot(Birth_weight~Smoker, data=birth_data, las=1, ylab="Birth-weight
(grams)", xlab="Smoking during pregnancy", col=c("yellow", "cyan"))
```

Here the thick black bars represent the medians, the boxes the IQR and the whiskers the IQR*1.5, with the points representing extreme values that fall outside of IQR*1.5. If you want to tweak the value of this multiplier, you can do so by setting the argument ‘range=’ to a value of your choice.

ANOVA

We can use **one-way ANOVA** to find out whether means significantly differ between two or more groups. For example, we might want to look at whether (and how) birth weight differs between mothers who did not smoke during pregnancy, those who were light smokers and those who were heavy smokers. We can use the variable ‘Smoking_3’ for this purpose, which classifies mothers’ smoking behaviour as heavy, light or none, depending on whether they smoked 10 or more, 1-9 or 0 cigarettes per day during pregnancy.

Checking assumptions

Like the independent samples T-test, ANOVA assumes that the dependent variable is normally distributed within each group and that there is homogeneity of variance across the groups, so we would need to check if these assumptions are met before proceeding. However since we have covered this already for the T-tests, I'll save you some time by telling you that as before, the data are not perfectly normally distributed and we have unequal variance between the groups.

Running the test

For normally distributed data with equal variances we can run ANOVA using the function `aov`, while if we have unequal variances, we need to use `oneway.test` which includes a built-in correction. Since Levene's test was significant we need to use `oneway.test` in this case, but the formatting is similar for `aov` should you ever need to use this instead in future.

Performing ANOVA is a two-step process. The test itself tells you if there is *at least one* significant difference between the groups, but to find out *which* (if any) groups significantly differ, we need to use **post-hoc tests**. First, let's run the ANOVA using the code below:

```
oneway.test(Birth_weight~Smoking_3, birth_data)
```

You should see the following output:

```
	One-way analysis of means (not assuming equal variances)

data:  Birth_weight and Smoking_3
F = 10.343, num df = 2.00, denom df = 126.98, p-value =
6.893e-05
```

Based on this output, we can conclude that birth weight significantly varies between the categories of mothers' smoking behaviour. We now need to perform post-hoc tests to find out which differences in particular are significant. In principal, we could have just run multiple T-tests to find out which specific contrasts are significant, but this would risk inflating the risk of false positives. There are a variety of different post-hoc tests available, appropriate for different situations, and which use different approaches to reducing the probability of a type-I error. In particular, the Games-Howell post-hoc test is designed for ANOVA with unequal variances – although R does not have a built-in function for this test, we can borrow one from the package 'userfriendlyscience'.

First install and load the package:

```
install.packages("userfriendlyscience")
library("userfriendlyscience")
```

Once you've loaded the package, the function `posthocTGH` can be used to calculate the appropriate post-hoc tests. Unfortunately however it cannot handle missing data, so we would need to first create a subset of complete data for our chosen variables as follows:

```
complete<-subset(birth_data, complete.cases(Birth_weight, Smoking_3))
```

Then perform the test using the complete data subset, like this:

```
posthocTGH(complete$Birth_weight, complete$Smoking_3, method="games-
howell")
```

And you will see the following output:

```
          n means variances
Heavy    78  2858    392102
Light   124  2991    430177
None  25378  3133    300724

            diff ci.lo ci.hi   t  df    p
Light-Heavy  133 -85.3   351 1.4 169  .32
None-Heavy   275 105.5   445 3.9  77 <.01
None-Light   142   2.5   282 2.4 124  .05
```

The first table in the output provides us some useful summary statistics including the group means. Here we can see that as predicted, heavy smokers gave birth to the lightest babies and non-smokers to the heaviest babies. However, from the second table we can see that only two of the three contrasts are significant – those between non-smokers and heavy smokers, and non-smokers and light smokers. From these data it appears therefore that smoking 1-9 cigarettes per day has as much of an impact on baby birth weight as smoking 10 or more.

Plotting means & error bars

The results of T-tests or ANOVAs can be illustrated using bars or points to represent means, together with error bars to represent uncertainty, often standard error or confidence intervals (although increasingly these are seen as insufficient in Psychology). Plotting with error bars is *unreasonably* difficult using the base functions, so although I've included instructions on how to do this below and you might find it helpful from a pedagogical perspective, you are welcome to skip this part if you're pressed for time. It is mainly here to emphasise the benefits of using dedicated plotting packages such as ggplot2 (in the Tidyverse and Visualisation course) in comparison as it offers much better ways to represent data than simple means and errors.

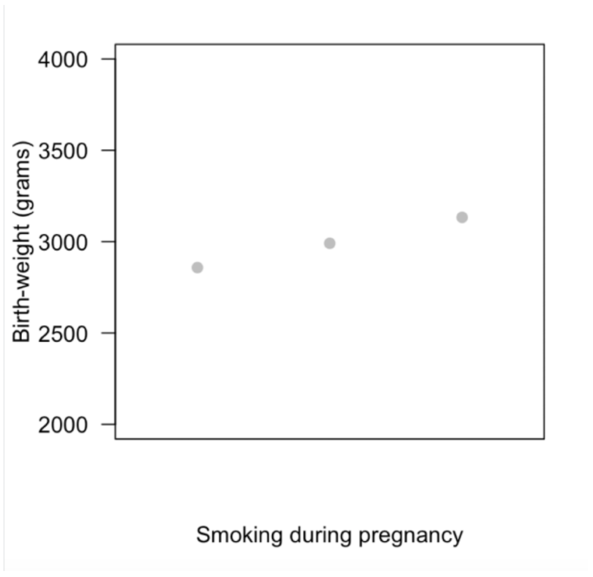
First, we need to identify the means that we want to plot and save them as a vector. We could do this using aggregate, but it might actually be simpler just to do it manually as follows (based on the output of the post-hoc tests), in the order "heavy", "light", then "none":

```
group_means<-c(2858, 2991, 3133)
```

Now, we can plot these means as points on a graph using the code below. We've encountered some of the graphical arguments before, but annotations are included for some less familiar ones. Make sure you highlight all lines at once when running the command:

```
plot(y=group_means, # plot the means on the Y-axis
     x=1:3, # position of means along the X-axis
     las=1,
     ylim=c(2000, 4000), # set the Y-axis limit
     xlim=c(0.5, 3.5), # set the X-axis limit
     pch=19,
     col="grey",
     ylab="Birth-weight (grams)",
     xlab="Smoking during pregnancy",
     xaxt="n") # suppress the X-axis numbers
```

You should end up with the following plot (below). We have suppressed the X-axis because by default, R will plot numbers while we want to use words to label the groups.



Next, we will add a custom X-axis using the function axis:

```
axis(side=1, # chooses the axis (1=X)
     at=c(1,2,3), # sets the position of the axis labels
     labels=c("Heavy", "Light", "None")) # sets the axis labels
```

Like the function legend, R will 'paste' the axis on top of the existing plot, so you will need to re-create both the plot and axis if you want to correct any mistakes.

Next, we need to add the error bars, which we'll do here using standard error. Surprisingly, R does not have a built-in function for standard error, but this provides a useful opportunity for us to introduce how to make your own functions in R. Roughly speaking, creating a function requires a command in the following format:

```
my_function <- function (x) {
        …
}
```

Where 'my_function' is the name you choose for your new function, 'function' is an existing function and 'x' is the placeholder for whatever your function will act upon. Inside the curly brackets, you would need to include what the function actually does to x – for example if your function takes some number x and multiplies it by 10, then you would put x*10 inside the curly brackets.

We can use the code below to create a new function, se, which will calculate the standard error of X, assuming X is a vector of numbers. The curly brackets contain the formula for standard error.

```
se<-function (x) {
  sd(x, na.rm=T)/sqrt(sum(!is.na(x)))
}
```

Once you have run the command above, se can be used like any other function. For example to calculate standard error for birth weight across the whole sample, you would use:

```
se(birth_data$Birth_weight)
```

To get the standard error within each group, we can feed `se` to `aggregate`, as follows:

```
aggregate(Birth_weight~Smoking_3, birth_data, FUN=se)
```

Then, we need to save the SEs as a vector (like we did for the means), making sure to do so *in the same order as the means* (i.e. Heavy, Light, None):

```
group_SEs<-c(70.900949, 58.899662, 3.442351)
```

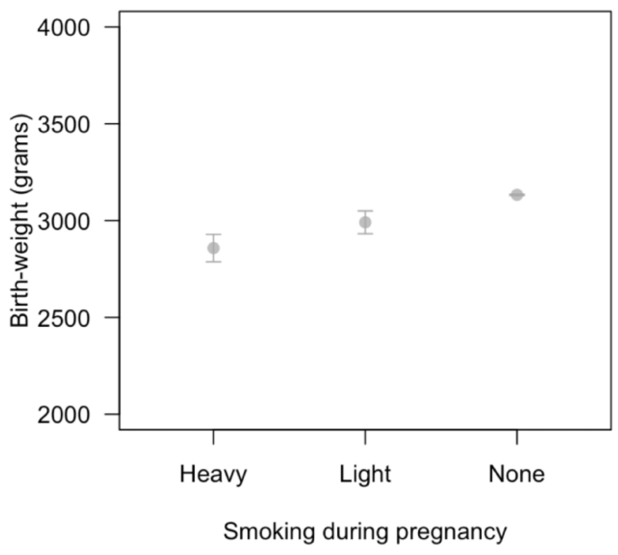
Now we need to calculate the top and bottom positions of the error bars, which is each mean plus and minus each SE, respectively:

```
error_top <- group_means+group_SEs
error_bottom <- group_means-group_SEs
```

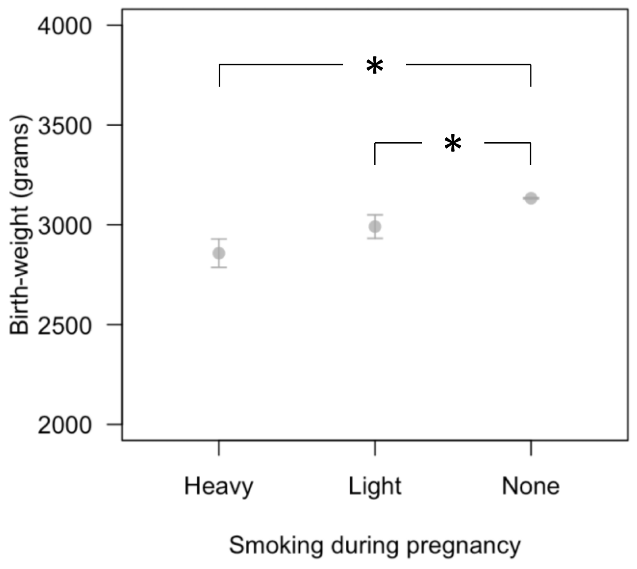
Then we can add the error bars using the arrows function. Individual arguments are explained below using annotations:

```
arrows(x0=1:3, # sets position on X-axis
       y0=error_bottom, # sets position of bottom of bar on Y-axis
       y1=error_top, # sets  position of top of bars on the Y-axis
       length=0.05, # sets width of whiskers
       angle=90, # sets angle of whiskers
       code=3, # specifies whiskers drawn at both ends
       col="dark grey") # selects the colour of the lines
```

Hopefully, you've ended up now with the pretty reasonable-looking plot below:



Lastly, you might want to add asterisks to the plot indicating which contrasts are significant. You could theoretically do this in R, but since it is such a pain, you might be better off doing it manually in a graphics program! For example, the one below was modified in PowerPoint:



Non-parametric alternative

If we had found that the data were not normally distributed, it might have been more appropriate to use a non-parametric alternative to the one-way ANOVA, the **Kruskal-Wallis test**. Like many other non-parametric tests, this test is performed on the ranked data rather than the raw values. We can do so using the `kruskal.test` function:

```
kruskal.test(Birth_weight~Smoking_3, birth_data)
```

Which will produce the results below:

```
	Kruskal-Wallis rank sum test

data:  Birth_weight by Smoking_3
Kruskal-Wallis chi-squared = 24.892, df = 2, p-value = 3.934e-06
```

As before, these results show that there is a significant difference between at least one of the groups. The Kruskal-Wallis has its own post-hoc tests ('Dunn's Test of Multiple Comparisons') but to avoid repetition, we won't go over these here[14].

As with the Wilcoxon rank-sum test, the appropriate way to illustrate the results of Kruskal-Wallis would be with medians and interquartile ranges rather than means and standard errors, e.g. using `boxplot`.

[14] There's no built-in function for Dunn's test, but it is implicated in several packages, see for example: https://cran.r-project.org/web/packages/dunn.test/dunn.test.pdf

Correlation

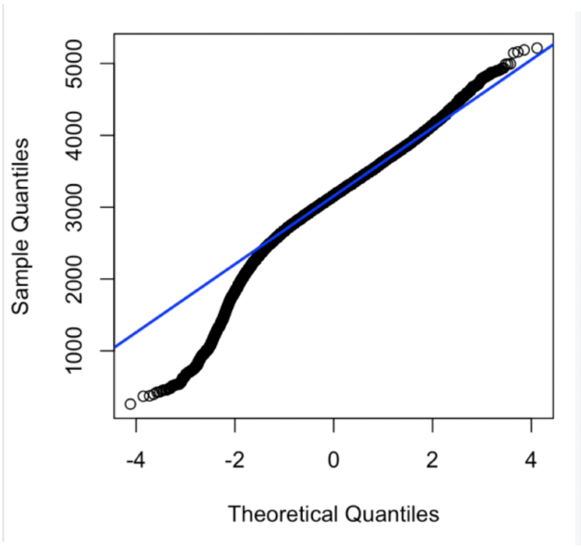
We can investigate relationships between two continuous variables using correlations. In particular, we can use the **Pearson's correlation** where data meet parametric assumptions, or otherwise a non-parametric alternative such as the **Spearman's correlation.** In this section, we will use correlations to explore the relationship between babies' birth weights and mothers' pre-pregnancy weights, anticipating these variables to be positively correlated.

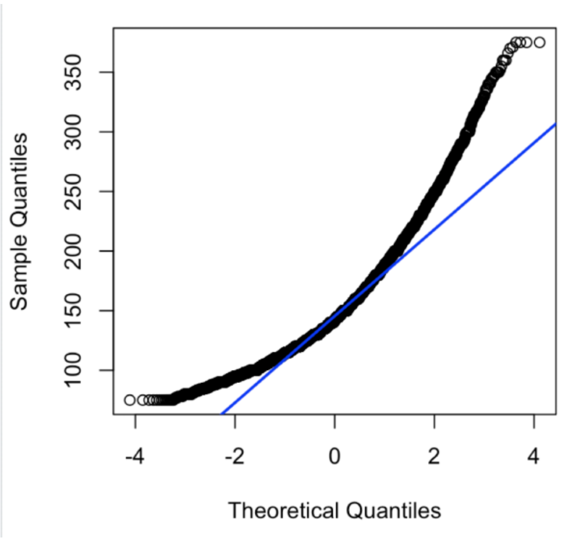
Checking assumptions

Let's start with the possible positive correlation between babies' and mothers' weights. Before we run any tests, we should check to see if data meet parametric assumptions. For Pearson's correlations, these are as follows:

1. Both variables are normally distributed
2. Linear relationship between variables
3. Homoscedasticity (equal variance over the range of the data)
4. Lack of strongly influential outliers

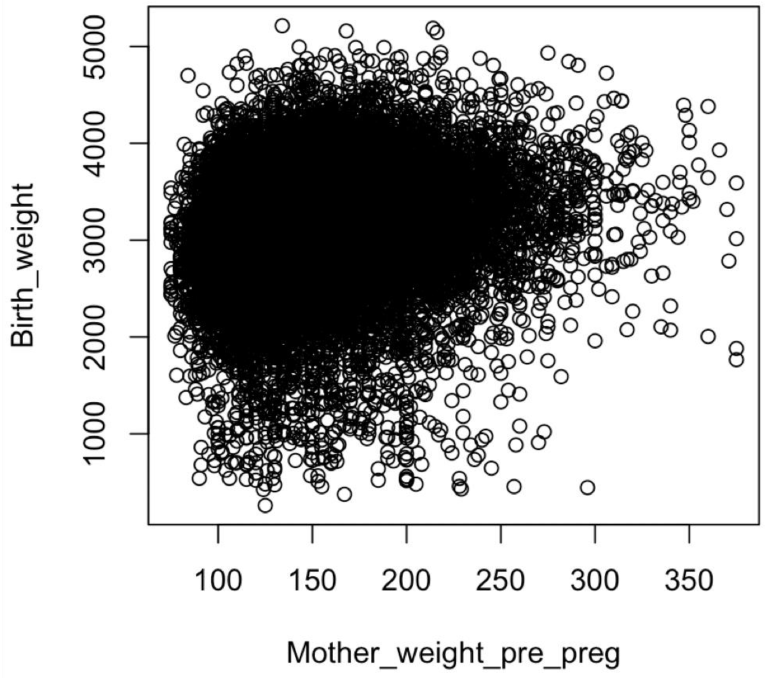
We can investigate the first assumption using histograms, qqplots and/or formal tests as we've seen before. Use `qqnorm` and `qqline` to produce qqplots for both birth weight and mothers' weight as below:





As you can see, babies' birth weights deviate from normality quite a bit, while mothers' birth weights are markedly different from normal – in the latter case, there are many more values at the extreme ends of the distribution than there ought to be were the data normally distributed.

There are formal tests for linearity, homoscedasticity and outliers, but I often find visualising the data is sufficient (I'll leave this up to you!). Examining a scatterplot, we can see some potential problems:



Though it’s hard to tell if we have any issues with linearity here, it looks as if we do have some heteroscedasticity – there is much more variation in mothers’ weight at the lower end than the higher end of the scale, so the data show a diagnostic ‘funnel’ shape. There are no obvious outliers, and it is probably quite unlikely that one or two extreme values will have a particularly undue influence over any patterns in the data given how large the sample is.

Overall then, the **Spearman’s correlation** would probably be more appropriate for the data, but we will walk through how to run and interpret both the Pearson’s and Spearman’s correlations for future reference.

Running the test

We can run the Pearson’s correlation as follows:

```
cor.test(birth_data$Birth_weight, birth_data$Mother_weight_pre_preg,
na.action=na.omit, method="pearson")
```

Don’t forget to include the ‘na.action’ argument – if you don’t have any missing values, it simply won’t do anything, so you’re always better off leaving it in just in case. You should see the output below returned:

```
	Pearson's product-moment correlation

data:  birth_data$Birth_weight and birth_data$Mother_weight_pre_preg
t = 21.471, df = 25420, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1213685 0.1455161
sample estimates:
      cor
0.1334621
```

The third line down reports the T-value, degrees of freedom and p-value, while the last value 'cor' is the correlation coefficient. So here we have a significant, positive correlation between babies' birth weights and the weights of their mothers before pregnancy, although not a particularly strong one. This should not be surprising given the scatterplot we examined a minute ago. The square of the correlation coefficient (R^2) tells us the proportion of variance in the dependent variable explained by the independent variable:

```
0.1334621^2
```

Which gives us 0.02. Put another way, 98% of the variance in babies' weights at birth is *not* explained by mothers' pre-pregnancy weights!

Scatterplots & lines of best fit

The results of Pearson's correlations can be illustrated intuitively with a scatterplot and a line of best fit. First, we can create a scatterplot using the code below:

```
plot(Birth_weight~Mother_weight_pre_preg, birth_data,
     las=1,
     pch=19,
     col=rgb(0,0,0,0.25), # set point colour to transparent grey
     ylab="Birth-weight (grams)",
     xlab="Pre-pregnancy weight (lbs)")
```

Then we can add a line of best fit using the function `abline`. The command below will add a line to the plot that goes through, or close to, as many points as possible. It works this out by constructing a linear model, which we'll cover in tomorrow's sessions. As before with similar plotting commands, the line will be pasted on top of the existing plot:

```
abline(lm(Birth_weight~Mother_weight_pre_preg, birth_data),
       col="blue", # set line colour
       lwd=2) # set line thickness
```

Non-parametric alternative

We can run the Spearman's correlation using the same command as above, but replacing "pearson" with "spearman" in the 'method' argument:

```
cor.test(birth_data$Birth_weight, birth_data$Mother_weight_pre_preg,
na.action=na.omit, method="spearman")
```

And we will see the following output:

```
      Spearman's rank correlation rho

data:  birth_data$Birth_weight and birth_data$Mother_weight_pre_preg
S = 2.2836e+12, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.1660391
```

```
Warning message:
In cor.test.default(birth_data$Birth_weight,
birth_data$Mother_weight_pre_preg,  :
  Cannot compute exact p-value with ties
```

As with the Pearson's correlation, the third line down tells us about the significance of the correlation, while the last one (reporting Spearman's *rho*) reports its strength. Reassuringly both tests give us pretty much the same answer: mothers who were heavier before pregnancy tend to give birth to heavier babies, but this relationship is not particularly strong.

The warning message is telling us that we have ties in our data. This occurs because the Spearman's correlation is based on ranked data, so observations with the same value will be assigned the same (i.e. 'tied') rank. Unless you have a large number of tied ranks, this usually isn't a major problem[15].

Session 4: Questions and exercises

1. Is there a significant association between infant sex and low birth weight?

2. Is gestation length approximately normally distributed?
3. Does birth weight vary as much for male as it does for female infants?

4. Are male babies significantly heavier than female babies, on average?

5. What test should you use to find out whether gestation time varies according to whether mothers were heavy smokers, light smokers or did not smoke at all during the first trimester?

6. What is the Pearson's correlation coefficient for babies' birth weight and number of daily cigarettes smoked in the first trimester?

7. What percentage of variation in babies' birth weight does smoking in the first trimester explain?

8. Based on a scatterplot of babies' birth weights against daily cigarette smoking, which assumption(s) of the Pearson's correlation do you suspect the data violate?

9. Based on a scatterplot of babies' birth weights against gestation length, which assumption(s) of the Pearson's correlation do you suspect the data violate?

10. What test do you think would be most appropriate to investigate the relationship between mother's level of education and mother age at birth? Mother's education takes values from 1 to 9, where 1 = 8th grade or less, 2 = 9th-12th grade, 3 = high school, 4 = college credit, 5 = associate degree, 6 = bachelor's degree, 7 = master's degree, 8 = doctorate and 9 = unknown.

[15] If you do have lots of ties, you might be better off using another alternative non-parametric correlation called **Kendall's tau**.

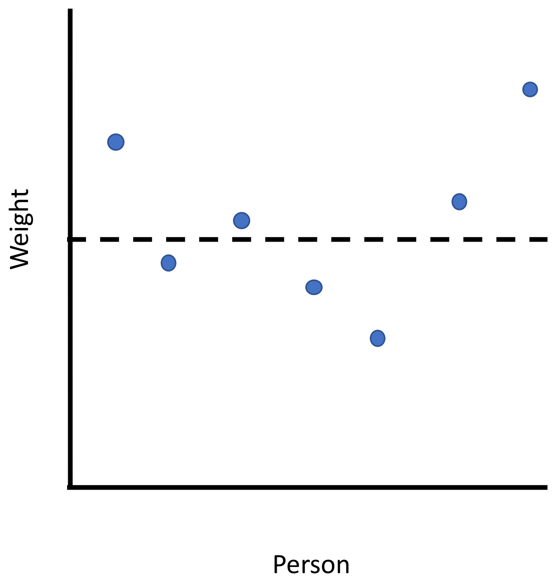
Session 5: introduction to generalized linear models part 1

In sessions 5 and 6 we will be introducing the **generalized linear model** (GLM) – a very powerful and highly flexible statistical technique with innumerable useful applications. In session 5 we will introduce linear models, leaving explanation of the 'generalized' part of GLM until session 6.

Models & prediction

In general, the goal of statistical analysis is to create a **model** – an approximation of reality. As the term suggests, models are not supposed to perfectly replicate reality but rather describe some element of it in a useful way, so that we can make predictions. To illustrate this issue you will often hear statisticians say something like "*all models are wrong, but some are useful*"[16].

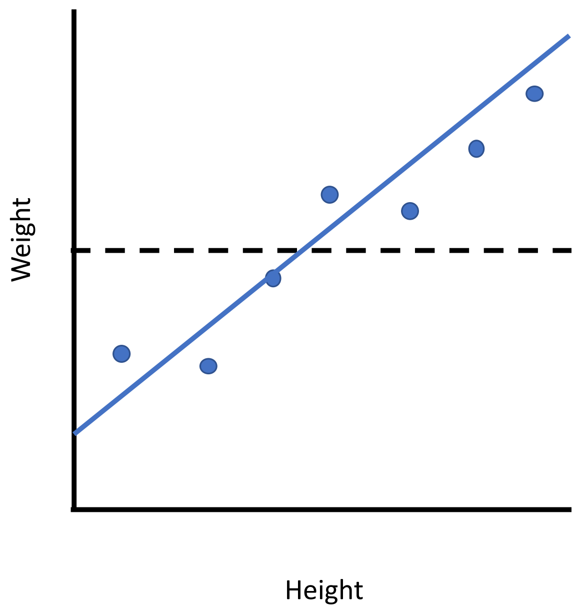
The idea of a 'statistical model' can sound quite complex but it need not be – in fact, even a statistic as simple as a mean is a statistical model. Let's imagine, for example, that we have measured the weights of a small sample of people, and plotted them out on the graph below:



The dashed line represents the sample mean. As you can see, no single individual in the sample actually weighs the same as the mean, and some people are substantially heavier or lighter than the mean. Therefore, the mean is an abstraction – an approximate rather than perfect description of the sample – i.e. a statistical model. Knowing the mean of a sample would not allow us to perfectly predict any single person's weight within that sample, but we would be able to have a better guess at their weight than if we had no model at all.

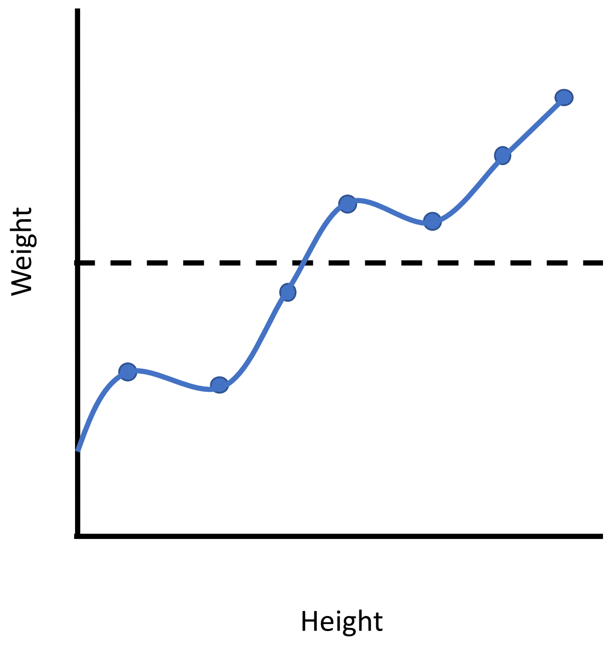
Of course, we can probably do a better job of predicting the data than just describing it with a mean. Continuing with the current example, a person's weight probably depends on many factors such as diet, ancestry, health, exercise, sex and so on, so we can probably create a much better model to explain variation in weight. One of the best predictors of weight is probably height – i.e. taller people are generally heavier. If we were to re-arrange the points on the plot above so that they are in height order from shortest to tallest along the X-axis, it might look as follows:

[16] See Box (1976), Science and Statistics, *Journal of the American Statistical Association*, 71: 791-799

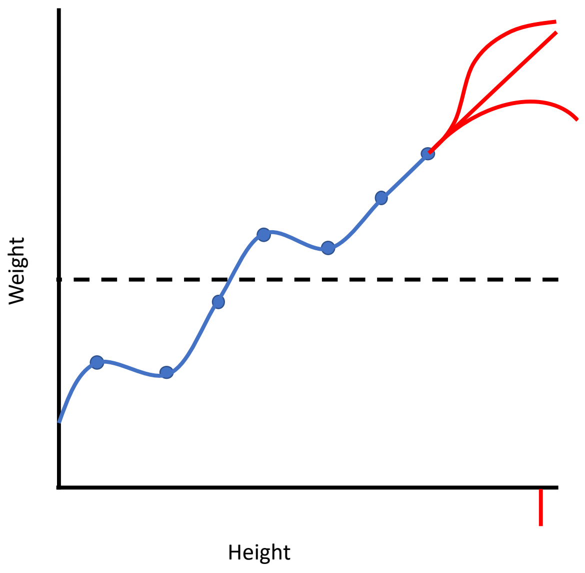


The dashed black line, again, represents the mean weight in the sample, while the solid blue line represents a new model which assumes that weight increases with height. Both of these lines represent different **linear models** – i.e. straight lines that attempt to describe patterns in the data. We can now see that a person's weight can to a large extent be predicted by their height, and that this model seems to do a better job of describing the data than the mean alone. Neither model, however, is perfect – we can see that some points fall below and some above the blue line, meaning that some people are heavier or lighter than you would expect simply given their height. This is because weight must be explained by many other factors than height alone – two people of the same height but with different compositions of muscle versus fat would be expected to have very different weights, for example.

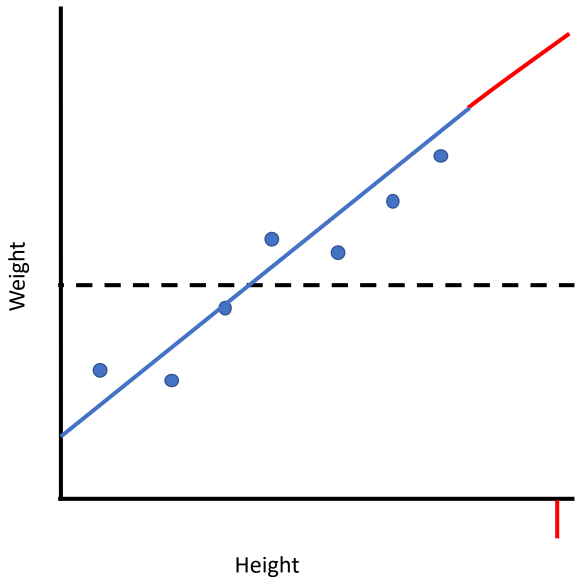
Theoretically, there might be more complex models than straight lines that would do a much better job of describing the data. For example, let's consider the non-linear model below:



Since the line travels through every single point, this model *perfectly* describes the relationship between height and weight in the sample, so our model is no longer an approximation but an exact description of reality. Seemingly paradoxically, however, this more complex model is not necessarily a *better* model, because it now does a very poor job of predicting reality outside of this specific sample. Imagine that we wanted to use it to estimate the weight of an additional person, who was taller than everybody else in the sample (indicated by the red line on the X-axis below). Based on the model, we really have no idea how weight would continue to increase with height outside of the sample range – the extended red lines represent just three of many possibilities:



In contrast, assuming a simpler linear relationship does allow us to make some predictions about a person's height based on their weight[17]. Even though we won't be able to guess it perfectly, we will probably do a better job than if we relied just on the mean, or on the more complex model:

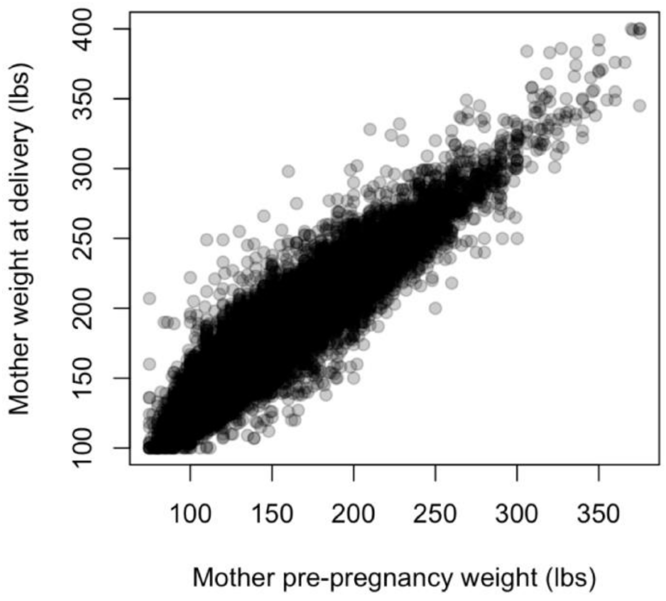


[17] A note of caution: using a model to predict data *outside* of the range of the sample is potentially risky and a contentious issue in statistics. This is because your sample inevitably doesn't perfectly capture the full complexities of the population from which it came. You should be especially careful when considering extrapolating far beyond the data. For example, people's weight might not continue to increase linearly with height as we approach heights of seven feet or more, due to possible health complications associated with extreme height.

Linear regression

In this section, we will translate the abstract ideas covered in the previous section to an example from the current dataset, covering the mechanics of linear models in more detail. Here we will use linear regression to explore the relationship between mothers' weights before pregnancy and their weights at delivery.

First, let's start simply by examining the data with a scatterplot. Use what you have learned so far to produce a plot something like the one below:



As you would expect, this plot suggests that mothers' weight at delivery is strongly positively correlated with her weight prior to pregnancy. We could characterise this relationship using a correlation – try running a Pearson's correlation for these two variables now, to produce the output below:

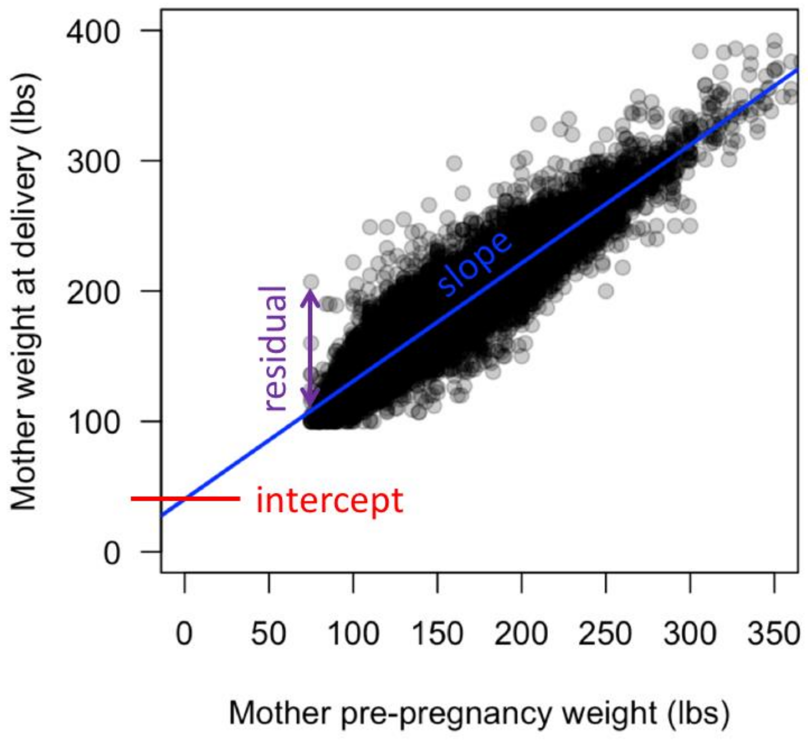
```
      Pearson's product-moment correlation

data:  birth_data$Mother_weight_delivery and 
birth_data$Mother_weight_pre_preg
t = 402.05, df = 25420, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9278847 0.9312261
sample estimates:
      cor 
0.9295745
```

Unsurprisingly, the output above confirms the correlation is strong and positive (r=0.93), and significant. Squaring the correlation coefficient tells us that 86% of the variation in mothers' weight at delivery is explained by her weight prior to pregnancy, leaving 14% of the variation to be explained by other factors (perhaps related to her diet, lifestyle, health, genetics and so on).

If all we wanted to know was the direction and strength of the relationship between the two variables, we could just use correlation and leave it at that. However, **linear regression models** can tell us much more than this, describing *how* one variable increases or decreases with another in more detail. Further, we can include multiple predictor variables (categorical and/or continuous) to model more complex relationships and figure out which variables explain the most variation in the dependent variable.

Components of a regression model

A linear regression model describes the relationship between variables using the following components: **slopes, intercepts** and **residual variation**. For example, if we fitted a linear model to describe the relationship between mothers' delivery weight and pre-pregnancy weight it might look like this (note that here the axes have been scaled to include zero):



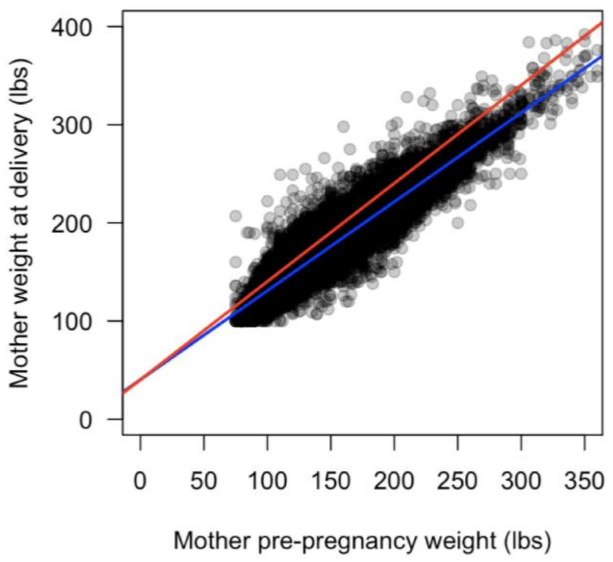
The **slope** describes the angle of the line, the **intercept** is the point at which the line crosses the Y axis when X is at zero and the **residual variation** is the distance between the points and the line – i.e. the 'left over' variation that is not explained by the model. We assume that this remaining variation is random and normally distributed.

Here, the slope crosses the Y-axis at around 40lbs when X is zero. We expect the intercept to be greater than zero in this case because mothers must gain weight throughout pregnancy! If we took this at face value, it would mean that the model predicts a mother who weighs zero lbs before pregnancy would weigh 40lbs at the point of delivery (you see now the dangers of extrapolating a model too far beyond the observed data…).

The slope is calculated as the *change in Y divided by the change in X.* So, if mothers' weight at delivery increases by 1 pound for every additional 1 pound she weighs before pregnancy, the change in Y over the change in X is 1/1 = 1. Therefore, we have a slope of 1, meaning that the two variables are increasing in *direct proportion to one another.* Together with the intercept, this would mean that mothers' weight at delivery is simply her pre-pregnancy weight plus an additional 40lbs.

However, if mothers' weight at delivery increased by 2 pounds for every 1 extra pound she weighs before pregnancy, the slope is 2/1 = 2, meaning that the Y variable is increasing faster than the X variable. Vice versa, if mothers' weight at delivery increased by only 0.5 pounds per extra pound of pre-pregnancy weight, the slope would be 0.5/1 = 0.5, meaning Y increases more slowly than X. So a slope of >1 means disproportionately *fast* increases in Y compared to X, while <1 means disproportionately *slow* increases in Y with X. (Note that the preceding explanation assumes Y and X are measured *in the same units* which is not always the case).

Below (in red) shows you what the relationship between mothers' delivery and pre-pregnancy weights would be if they increased in direct proportion to one another:



The blue slope represents a line of best fit, which we can now see has a slope of <1 given that it has a shallower angle of increase compared to the red line. This means that mothers who are heavier to start with do not gain as much weight during pregnancy compared to those who are lighter to start with.

Let's put this into explicit mathematical terms now to make sure we really get the idea. A regression model with one predictor variable is based on the following equation:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

The equation above says that any given value of the dependent variable Y (the 'ith' observation of Y) is equal to the intercept (α) plus the corresponding value of an independent variable X (the 'ith' observation of X) multiplied by the slope (β), plus some residual variance, which we assume is random (ε). Linear regression uses a mathematical technique called *ordinary least squares* (OLS) to find the values of α and β that minimise the total amount of residual variance, and thus find the line of best fit - i.e. the line that passes through (or close to) as many of the points as possible[18].

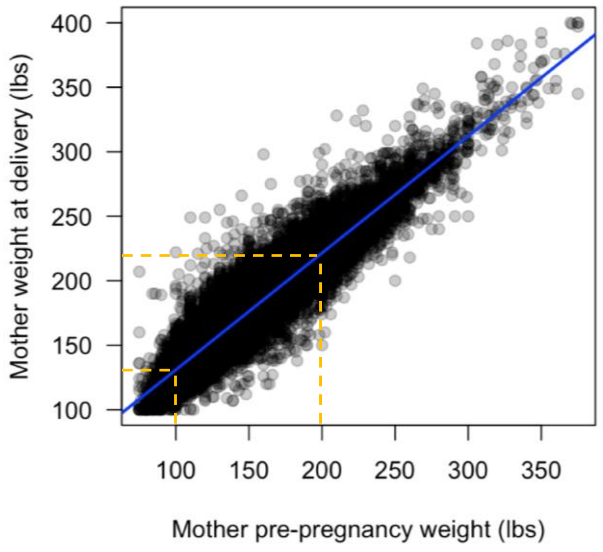
We can plug some values from the data into the equation to make this even more explicit. Let's say for example we want to predict how much someone weighs at the point of delivery if they initially weighted 100lbs before pregnancy. If the intercept of the blue line was 40 and the slope 0.9, then we anticipate that this person weighs:

```
40+100*0.9
```

= 130 lbs at delivery, give or take some random error. If on the other hand they weighted 200lbs before pregnancy, at the point of delivery they would weigh:

```
40+200*0.9
```

= 220lbs. These two predictions are illustrated graphically on the plot below:



Now we can see more clearly what the model tells us – mothers who are lighter before pregnancy gain proportionally more weight than those who are heavier before pregnancy. Thus, we gain much more detailed information about how variables relate to one another from a linear model compared with a correlation.

Simple regression

In this section we'll walk through the process of running and interpreting a simple linear model in R. First, we'll return to the relationship between babies' birth weights and mothers' pre-pregnancy weights, which we investigated in session 4 using correlations.

[18] You don't necessarily need to understand the underlying maths for OLS for a good working knowledge of linear models, but if you want to find out more you can find step-by-step explanations in many statistical textbooks, e.g. see Stinerock (2018), *Statistics with R: a Beginner's Guide*, Sage (p. 309).

Continuous predictors

Fitting a linear model in R requires two lines of code – first, one command to create an object containing the results of the linear model, below called 'bw_model1':

```
bw_model1<-lm(Birth_weight~Mother_weight_pre_preg, birth_data)
```

Then, a command using the `summary` function on the model object to retrieve the results:

```
summary(bw_model1)
```

The output will return quite a bit of information, but first let's just focus on the coefficients table:

```
Coefficients:
                        Estimate Std. Error t value Pr(>|t|)
(Intercept)            2853.2125    13.4374  212.33   <2e-16 ***
Mother_weight_pre_preg    1.8594     0.0866   21.47   <2e-16 ***
```

The first column, Estimate, contains the values of the intercept (2853.21) and the slope (1.86) estimated by the model. The second column contains standard errors of these estimates, the third the test statistics and the final column the p-values. In both cases, R is comparing the estimates against the null hypothesis of zero. So in this case, we have both an intercept and a slope that are significantly greater than zero. Since the slope estimate is positive, we know that Y increases with X, while if it had been negative, it would have meant that Y decreases with X (like a negative correlation). We aren't usually interested in whether the intercept significantly differs from zero, so we can often just ignore this p-value. The significance value for the slope, however, is important as this is what tells us if we have a significant relationship between the dependent and independent variables.

We can make more sense of what these estimates mean if we interpret them together with a plot. Use the commands below to create a scatterplot of the data and add the line of best fit from the model. As before, we'll set both the axes to extend to zero to make interpretation a bit clearer:

```
plot(Birth_weight~Mother_weight_pre_preg, birth_data, col=rgb(0,0,0,0.25),
pch=19, ylab="Birth weight (grams)", xlab="Mother pre-pregnancy weight
(lbs)",  las=1, ylim=c(0, 5000), xlim=c(0, 350))

abline(bw_model1, col="blue", lwd=2)
```

You should see that the point at which the line crosses the Y-axis when X is zero corresponds to the intercept estimate in the model output (~2850g). You'll also see that the slope is positive (i.e. Y increases with X), but fairly shallow. You might find this odd given that the slope is >1 – shouldn't this mean we have a steep slope, with Y increasing faster than X? However, this slope is slightly challenging to interpret because the Y and X variables are measured in *different units*: birth weight in grams and mothers' weight in pounds. The value of the slope (1.86) means that for every additional pound of pre-pregnancy weight, the model predicts a baby weighs an additional 1.86 grams at birth. Plugging in these values to the regression equation, this means the model predicts that mothers weighing 100lb before pregnancy should have babies weighing:

```
2853.2125+1.8594*100
```

i.e. ~3040g, while mothers at the heavier end of the scale, weighing 300lb before pregnancy, are anticipated to have babies weighing:

```
2853.2125+1.8594*300
```

Around 3400g. Taken together, the plots and model predictions suggest that while there is a significant relationship between mothers' pre-pregnancy weight and babies' birth weights, it is a pretty weak one. Mothers with weights differing by as much as 200lbs have babies that differ in weight by only 300g. Remember again that almost *any* deviation from the null hypothesis is likely to be significant with very large sample sizes. This exercise illustrates the importance of a holistic interpretation of statistical results – you should always consider the 'real world' significance of the relationship, not just the p-values!

The second important part of the output is the second to last line, which contains the model R^2:

```
Multiple R-squared:  0.01781, Adjusted R-squared:  0.01777
```

R reports two R-squared values: 'multiple' and 'adjusted' R-squared. 'Multiple' R-squared means the same thing for linear models as it does for correlations – it tells us the proportion of variation in the dependent variable that is explained by the independent variable, so it is a measure of model 'fit'. Adjusted R-squared is slightly more complex, and is more relevant to multiple regression – we'll come back to this in session 6, so you can just ignore it for now.

In this case, the R^2 tells us that our model explains only ~2% of the variation in babies' birth weight. This illustrates an important point to bear in mind when interpreting the results of regressions – OLS always finds *a* line of best fit, but this does not mean that it necessarily finds a particularly good model! This lines up with how the data look on the scatterplot – the points are very widely spread in a 'cloud' and no obvious linear relationship is evident. So, we can conclude that while babies' birth weight does increase with mothers' pre-pregnancy weight, it is not a particularly strong relationship.

Finally, before we are satisfied with our interpretation of the results we should check to see if the data violate any assumptions of the linear model. You might be wondering why we did not check to see if the data were normally distributed before running the model, as we did for correlations – this is because linear regression actually assumes not that the *data* are normally distributed, but rather that the *residual variation* is normally distributed[19]. Linear regression also assumes that the residuals are **homoscedastic** – i.e. they have a similar amount of variation across their range.

We can usually tell if assumptions of linear regressions are violated just by visually examining plots of the residuals called 'diagnostic plots'. In R, we can create diagnostic plots by applying the `plot` function to an object containing the results of a linear model. Doing so will produce four separate plots, and so you will be prompted to hit the return key on the keyboard to cycle through them[20].

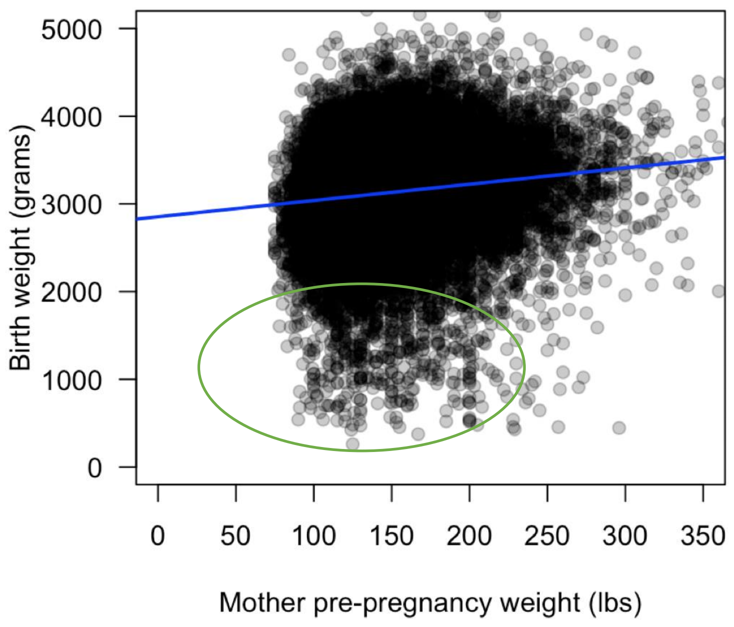
```
plot(bw_model1)
```

[19] Saying that, normally distributed data do often lend themselves to normally distributed residuals, so you will likely encounter problems with the residuals if you have very skewed data, particularly if your dependent variable is skewed.

[20] Alternatively, if you want to see all four plots in the same window, enter `par(mfrow=c(2,2))` before the plotting command. Here the numbers refer to the number of rows (first number) and columns (second number) in the plotting window. To switch back to one plot per window, enter `par(mfrow=c(1,1))` before your next plot.

The first two plots are particularly important, so we'll focus just on these today. The first one helps you check the residuals for homoscedasticity, and the second one for normality[21]. The first plot shows the predicted values of the outcome variable on the X-axis against the model residuals on the Y-axis. If the residuals are homoscedastic, this plot should resemble a random cloud (or 'starry night') with no pattern in the variation of the data from left to right. However, we do see a pattern here - there is less variation for higher residual values (on the right) than there is for lower residual values (on the left), indicating heteroscedasticity.

Next, in the top right-hand corner we have a qqplot (which we've encountered before) of standardized residuals against a theoretical normal distribution. If our residuals were perfectly normally distributed, all the points would fall on the dotted line, so we can see that we do have an issue here. Compared with a normal distribution, we have far fewer values at the lower end of the scale. This makes sense if we look back at the scatterplot and line of best fit – notice how many values we have with very large negative residuals (circled below). These are babies with substantially lower birth weights than expected given their mothers' body weights, most likely representing premature babies. This means that we will have lots more large, negative residuals than we should do if the residuals were normally distributed:



In reality, judging to what extent deviations from assumptions affect results is a bit of a subtle art. If you have seriously biased residuals, you should probably consider either a) transforming the data or b) using a non-linear model instead (two examples of which we will cover in the next session). The current case is a good example of a particularly tricky one – there are clearly some deviations but a better alternative to a linear model is not immediately obvious. Therefore in this situation the use of a linear model might be justifiable, but we should be very careful when interpreting the results and particularly when generalizing from our results to those from other samples (who may not show the same quirks as the current sample). A further option would be to try to create a smarter, more complex model, including more than one predictor – e.g. perhaps if we included gestation time

[21] When it comes to your own analyses, outliers and non-linearity are two further issues you may want to investigate.

together with mothers' pre pregnancy weight, we could better predict babies' weights at birth. We'll cover these models in the section on multiple regression, later on.

Categorical predictors

The major advantage of linear models is their flexibility. For instance, we aren't limited to continuous predictor variables – we can use categorical variables as predictors too. In this section, we will use a linear model to explore whether there is a significant sex difference in babies' birth weights. So our dependent variable is birth weight, and our independent variable is infant sex (here coded on two levels, F=female and M=male).

Before we jump into running the model in R, it may help to explain how categorical predictors work mathematically in linear models. As before, the model is based on the following equation:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Where as before, Y is the value of the dependent variable (here birth weight), α is the intercept, β is the slope, x is the value of the independent variable and ε is the residual variation. Since we have a categorical predictor variable, we don't really have an intercept and slope in the same way as we did for the previous example using a continuous predictor – rather, the model will basically fit two intercepts, one for each group. The value of the dependent variable X in the current dataset is represented by characters ("F" and "M" for infant sex), but behind the scenes R will convert these to numbers (0 and 1) when running the model. By default, R assigns factor levels to numbers based on alphabetical order, so here F = 0 and M = 1. For observations where X=0 (the female infants in this case), therefore, the value of Y is equal to the intercept (α), which is the mean birth weight for females. For observations where X=1 (the male infants), the value of Y is estimated as the intercept plus some value (β), so β is the difference between the mean birth-weight for males and the mean birth-weight for females.

If this isn't making sense yet, it hopefully will do once we put it into practice. Let's now fit the model in R, using the same syntax as we did for the model with a continuous predictor:

```
bw_model2<-lm(Birth_weight~Infant_sex, birth_data)
```

And then summarise the results using `summary`:

```
summary(bw_model2)
```

So that we produce the following output:

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 3080.989      4.880  631.29   <2e-16 ***
Infant_sexM   97.641      6.809   14.34   <2e-16 ***
```

In the first column (Estimate), we have the model intercept, which here represents the mean weight of the female infants (~3081 grams), and below this we have the difference between the intercept for male infants and the intercept for female infants (~98g). So, this output tells us that the model estimates male infants weigh about 100g more at birth than female infants do, and that the effect of infant sex is statistically significant (see the p-value in the bottom right corner of the table).

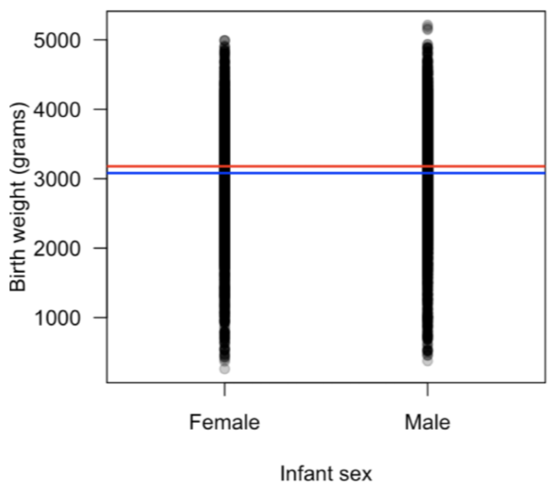
Now we can use some plotting commands to create a graphical representation of this model. First use the commands below to plot out the data and add a custom X-axis:

```
plot(Birth_weight~as.numeric(Infant_sex), birth_data, xlim=c(0.5, 2.5),
col=rgb(0,0,0,0.25), pch=19, las=1, ylab="Birth weight (grams)",
xlab="Infant sex", xaxt='n')

axis(side=1, at=c(1,2), labels=c("Female", "Male"))
```

Next, use the `abline` line function to add two lines to the plot – one representing the intercept for females (i.e. the mean birth weight for female infants) and one for the males, taken from the coefficient table above:

```
abline(h=3080.989, col="blue", lwd=2) # intercept for females

abline(h=3080.989+97.641, col="red", lwd=2) # intercept for males
```

In the commands above, the 'h' argument stands for 'horizontal', so the value of h indicates the position of a horizontal line on the Y-axis. You should end up with the following plot:



While this might look a bit odd, just like when we had a continuous predictor, we have created a linear model, because this relationship between the variables can be represented using straight lines. And just like before, we can plug in the values from our model into the regression equation to make predictions about the value of the dependent variable based on the value of the independent variable.

The model says that a female infant weighs, on average, the value of the intercept plus the value of the 'slope' (which here is a flat line) multiplied by zero:

```
3080.989+0*97.641
```

i.e. 3080.989 grams, plus or minus some random error, while a male infant weighs, on average, the value of the intercept plus the value of the 'slope' multiplied by one:

```
3080.989+1*97.641
```

i.e. 3178.63 grams. Clearly, based on the plot there is a *lot* of residual error and a huge amount of overlap between the birth weights of female and male babies, so while significant statistically, sex actually does not look like a very important predictor of babies' weight at birth.

You might be wondering why you should not just simply use a T-test or ANOVA to test for differences in means between groups. In fact, it does not matter either way because these tests are special cases of linear regression, and thus produce the same results! You can prove this to yourself by comparing the results of the linear model we have just run with that of a T-test on the same data:

```
t.test(Birth_weight~Infant_sex, birth_data)
```

```
      Welch Two Sample t-test

data:  Birth_weight by Infant_sex
t = -14.354, df = 25857, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -110.97421  -84.30782
sample estimates:
mean in group F mean in group M 
       3080.989        3178.630 
```

Looking at the last line of the output, you will see that the mean for the females is the exact same value as the intercept estimated in the linear model (3080.989), and the mean for the males (3178.630) is the same as the intercept plus the slope estimated in the linear model (3080.989+97.641=3178.63).

The same is true of correlations. Earlier on, when we ran the linear model for birth weight against mothers' pre-pregnancy weight, we obtained an R^2 value of 0.018. When we used correlation for the same variables, we got a correlation coefficient of 0.133. If we square this value to calculate R^2 based on the correlation, we arrive at the exact same result: 0.018.

The huge advantage of linear models over T-tests, ANOVAs and correlations is their flexibility. Linear models allow you to include multiple predictor variables (both categorical and continuous) in order to represent much more complex relationships between variables and explain much more of the variation in dependent variables, as we'll see in the next sections.

Want to add beta weights to your output for publication? Check out: Appendix: How to get betas in regression (and make beautiful regression tables)

Session 5: Questions and exercises

1. Imagine that you ran a simple linear regression with one continuous independent variable, and the model reports a slope of -1. Both variables are measured using the same units. How should you interpret this result?

2. Do smoking mothers have a lower pre-pregnancy weight than non-smoking mothers? Use a linear model to find out.

3. What is the value of the slope for a model in which number of cigarettes smoked per day in the first trimester predicts babies' weight at birth? What does this mean?

4. The coefficient table below comes from a model predicting babies' birth weights from the number of weeks' gestation. Based on these results, how much should a baby born at 39 weeks weigh?

```
Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)     -3382.552     51.180  -66.09   <2e-16 ***
Gestation_weeks   170.390      1.337  127.44   <2e-16 ***
```

5. The coefficient table below comes from a model predicting mother's age at delivery from fathers' age at delivery. According to these results, is the age difference between mothers and fathers greater for younger or older couples?

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 11.063861   0.117446    94.2   <2e-16 ***
Father_age   0.534340   0.003776   141.5   <2e-16 ***
```

6. The results below come from a model in which the categorical variable 'smoker' (Y=yes, N=no) is the independent variable and mother's age at birth is the dependent variable. How old are mothers who smoke during pregnancy at birth, on average, compared with those who do not smoke during pregnancy?

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 27.02460    0.03749 720.906  < 2e-16 ***
SmokerY      1.67588    0.41700   4.019 5.86e-05 ***
```

7. How much variation in babies' birth weight is explained by mothers' age at birth?

8. Is babies' birth weight better predicted by mothers' weight before pregnancy, or mothers' height?

9. What is the value of the slope for a model in which babies' birth weight is predicted by mothers' age at birth? What does this mean?

10. Run a linear model in which gestation length is the dependent variable and mothers' age at birth is the predictor, then check the diagnostic plots. Do the data violate the assumptions of normality and homoscedasticity of residuals?

Session 6: introduction to generalized linear models part 2

Multiple regression

Multiple regression is an extension of simple regression, which allows us to include more than one independent variable. Multiple regression is based on the same equation as for simple regression, except that we include more than one coefficient, as follows:

$$y_i = \alpha + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \cdots + \varepsilon_i$$

Where β_1 is the coefficient for the independent variable x_1, $\beta_2$is the coefficient for the variable $x_2$and so on. Theoretically, you can add any number of predictor variables but you should bear in mind that the more predictors you include, the more data you will need in order to estimate effects confidently. There are no hard and fast rules about sufficient sample sizes, but various heuristics exist such as the 'one in ten' rule – this says that you should have at least 10 observations for every predictor you include in the model.

Continuous predictors

In this section we will walk through an example of multiple regression with two continuous predictor variables. Let's say that we want to model babies' birth weights based on both 1) mothers' pre-pregnancy weights and 2) number of cigarettes smoked during the 1st trimester by the mother.

Multiple regression models can be especially useful when we are trying to tease apart the effects of potentially confounding variables. So far, we have found that babies' birth weight increases with mothers' pre-pregnancy weight and decreases with number of cigarettes smoked during pregnancy, when modelling these variables individually. As far as we know, however, the negative effect of cigarette smoking on birth weight could be **confounded** by mothers'body weight – what if mothers who smoke weigh less, and have lighter babies for reasons unrelated to their smoking? If that is the case, when we put the two variables into a multiple regression model together, we ought to find that only mothers' weight has a significant effect on babies' birth weight, while the negative effect of smoking disappears or weakens substantially.

We have seen already that number of cigarettes smoked is not a normally distributed variable, but remember that what matters is whether the *residuals* are normally distributed, so we can leave checking assumptions until after we have run the model.

Use the commands below to run the model and report the results:

```
bw_multiple_model<-
lm(Birth_weight~Mother_weight_pre_preg+Daily_cigs_trim1, birth_data)

summary(bw_multiple_model)
```

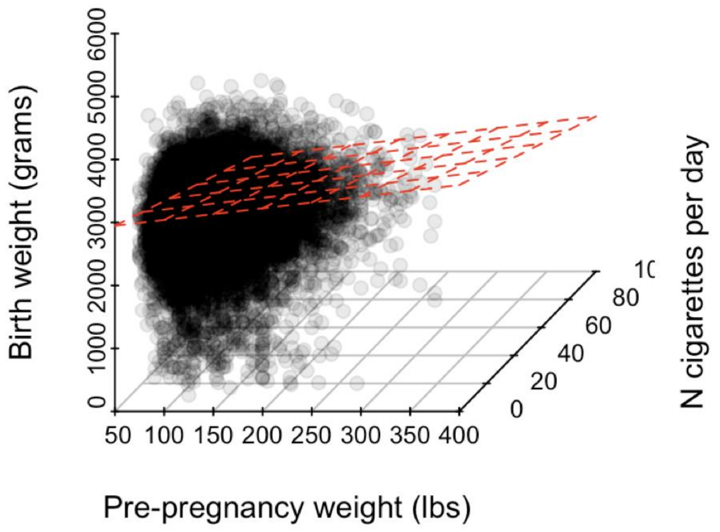
Let's start by interpreting the coefficients:

```
Coefficients:
                         Estimate Std. Error t value Pr(>|t|)    
(Intercept)            2855.23782   13.46978 211.974  < 2e-16 ***
Mother_weight_pre_preg    1.84872    0.08677  21.305  < 2e-16 ***
Daily_cigs_trim1        -11.30729    3.22345  -3.508 0.000453 ***
```

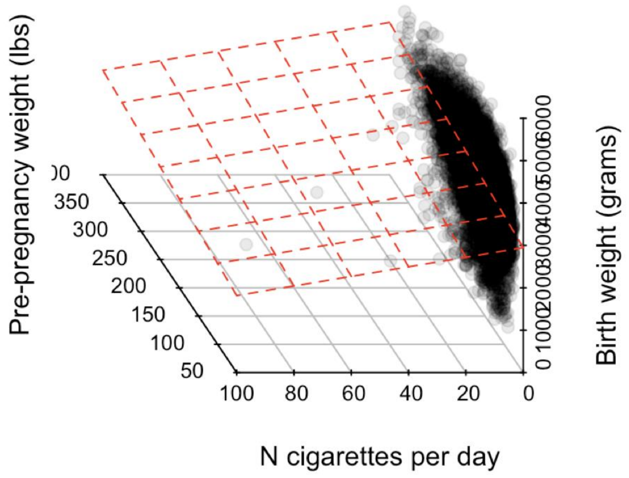
From the coefficient table above, we can see that birth weight increases with mothers' weight before pregnancy (because the slope 1.84872 is positive) while it decreases with number of cigarettes smoked during pregnancy (because the slope -11.30729 is negative). Translating these numbers back to the data, this model is telling us that for a mother of average body weight, each additional cigarette she smokes per day decreases the weight of her baby at birth by ~11 grams. For a mother who smokes an average number of cigarettes per day during pregnancy, each additional pound that she weighs before pregnancy increases the weight of her baby at birth by ~1.85 grams. Therefore, we still have a pretty robust negative effect of smoking on babies' birth weights even after accounting for the effect of mothers' weights before pregnancy. We can then conclude it is unlikely that the effect of smoking on babies' birth weight is confounded by mothers' body weight.

We could illustrate this model graphically using a scatterplot as before, but now that we have two predictors we would need to represent this in three, rather than two, dimensions. If we have more than two continuous predictors, it becomes increasingly difficult to represent relationships graphically since we cannot think in more than three dimensions! If we were to plot out the relationship between these three predictors graphically, it would look something like this[22]:

[22] If you have time, you can try out 3D scatterplots for yourself using the 'scatterplot3d' package. See the following link for a useful tutorial: http://www.sthda.com/english/wiki/scatterplot3d-3d-graphics-r-software-and-data-visualization



Since we have two predictors, the relationship between our variables now needs to be represented not just by a single unidimensional line, but by a two-dimensional plane (illustrated in red above). Unfortunately this example is not particularly clear visually, but it should help make sense of things to some extent. Looking first at the relationship between pre-pregnancy weight and birth weight, we can see the positive relationship between these variables evident in the angle of the plane, particularly at the edge of the plane closest to us. If we were to spin the plot around on the Y-axis, we'd be able to see the negative relationship between birth weight and number of cigarettes smoked per day (again most clearly at the end of the plane closest to us):



The goal of multiple regression is to tease out the *independent* effects of multiple predictor variables. Therefore, we will run into issues if our predictors are very strongly correlated with one another, a problem known as **multicollinearity**. For example, let's imagine that we wanted to use multiple regression to predict babies' birth weight based on both mothers' weight before pregnancy and mothers' weight at delivery. *A priori*, you might assume that both of these effects should have positive effects – i.e. heavier mothers, heavier babies. Look what happens, however, if we run this model:

```
bw_multiple_model2<-
lm(Birth_weight~Mother_weight_pre_preg+Mother_weight_delivery, birth_data)

summary(bw_multiple_model2)
```

If we took these results at face value, we could have to conclude that mothers' weight before pregnancy is *significantly negatively* associated with babies' birth weight, the exact opposite of what we found when we included this predictor in a simple linear model as the sole predictor, and contrary to biological common sense. What is going on here?

The reason we get this nonsensical result is that mothers' pre-pregnancy and delivery weights are highly correlated with one another (Pearson's R = 0.93), because they are measuring essentially the same thing – variation between individual mothers in their body weights. This means that the model cannot distinguish between their effects and you will get bizarre results like the one above.

One way to check for multicollinearity efficiently is to calculate **variance inflation factors** (VIFs). These are calculated based on the correlation between your independent variables and tell you to what extent the model will struggle to distinguish between the effects of related variables. You can calculate these using the function `vif` from the 'car' package. First install and/or load the car package if you need to, then feed your linear model to the `vif` function as follows:

```
vif(lm(Birth_weight~Mother_weight_pre_preg+Mother_weight_delivery,
birth_data))
```

The `vif` function will run all possible correlations between the predictor variables and report a VIF for each one separately. Since we only have two predictors here, we get the same VIF for each one: 7.36. Advice on interpreting VIFs varies between statisticians, but often values of >5 or >10 are taken as problematically high. As with all statistical cut-off values however, these are arbitrary and so should be interpreted together with a bit of common sense. A correlation as high as 0.92 between the variables, together with the weird results, should be enough to tip you off that something is amiss. If this is the case, you probably should not be including both variables in the same model.

As with simple linear regression, we can check to see if the data violate the assumptions of multiple regression by producing diagnostic plots using the following commands:

```
plot(bw_multiple_model2)
```

Based on the first two plots, we can see some evidence of bias in the residuals, although the situation does not look any worse than for the simple linear model. This illustrates that the fact that non-normally distributed predictors (in this case N cigarettes smoked per day) often have little impact over the fit of the model to the data – what typically matters much more is the distribution of the *dependent* variable.

Comparing model fit

One extremely useful application of regression is comparing models with one another in order to identify which combination of independent variables best explains variation in the dependent variable. There are many different ways to compare models and statisticians often argue over the best way to do this, but for today we will keep things simple and just introduce the concept by comparing R^2 values.

Let's imagine that we are interested in predicting variation in birth weight from two different variables: mothers' pre-pregnancy weight and gestation time. We should expect positive effects of both these variables on birth weight. Our question here is as follows: is variation in babies' birth weight best explained by either mothers' weight, gestation time or a combination of the two?

Let's start by fitting all three models and saving their results as objects (you can name them something else if you like – whatever you find least confusing!)

```
BW_MW<-lm(Birth_weight~Mother_weight_pre_preg, birth_data)
BW_GW<-lm(Birth_weight~Gestation_weeks, birth_data)
BW_MW_GW<-lm(Birth_weight~Mother_weight_pre_preg+Gestation_weeks,
birth_data)
```

If you use `summary` to report the model results, you'll see that mothers' weight and gestation length both have significant, positive effects on babies' birth weight across all the models, as we would expet. To try to find out which is most important in terms of variance explained, however, we need to look at model R^2 values. The table below summarises them for you:

Model	**Multiple R^2**	**Adjusted R^2**
Mother weight only	0.01781	0.01777
Gestation time only	0.3858	0.3857
Mother weight & gestation time	0.4064	0.4064

The first pattern you will notice is that regardless of whether we look at the multiple or adjusted R^2, the model containing only mother weight as a predictor explains much less variation than the one containing gestation time only. When we include both together, we gain only an additional ~2% variance explained compared to one including only gestation time. So we can see that gestation time explains much more variance in birth weight compared with mothers' pre-pregnancy weight.

Now we can return to the difference between multiple R^2 and adjusted R^2 (although in this case, they happen to be pretty similar anyway). Whenever we add *any* additional predictor to a model, we will inevitably explain more variance in the dependent variable and R^2 will increase, even if it has only a very slight effect on the dependent variable. A model with many, many redundant parameters may therefore have a high R^2 and appear to be a good fit to the data, but might actually be pretty useless at predicting it – this is known as **overfitting**. The adjusted R^2 includes a correction for overfitting and thus should *only increase if the additional predictor improves the fit of the model more than expected by chance.* Therefore, when comparing models with R^2 it would make more sense to use the adjusted version. In this case, we can see that including mother weight does slightly increase the adjusted R^2 of the model compared to one with only gestation time, so we would be justified in concluding that this more complex model is a better fit to the data.

Both continuous & categorical predictors

In this section, we will cover how to run and interpret multiple regression models with both continuous and categorical predictors. We'll look again at modelling babies' birth weights based on smoking behaviour and mothers' pre-pregnancy weights, but here we will treat smoking as a categorical variable indicating whether mothers did or did not smoke at all during pregnancy. This kind of model is often referred to as **ANCOVA** (analysis of covariance). The simplest ANCOVA model is one in which we fit two lines for each group represented in the categorical variable (here smokers versus non smokers), each with a different intercept but the same slope.

First we'll cover how to run the model, then we will walk through interpreting the results with the aid of plots. Run the model and report the results using the commands below:

```
bw_ANCOVA<-lm(Birth_weight~Mother_weight_pre_preg+Smoker, birth_data)
```

```
summary(bw_ANCOVA)
```

```
Coefficients:
                          Estimate Std. Error t value Pr(>|t|)    
(Intercept)             2856.06953   13.47137 212.010  < 2e-16 ***
Mother_weight_pre_preg     1.84738    0.08676  21.292  < 2e-16 ***
SmokerY                 -175.86578   40.00323  -4.396 1.11e-05 ***
```

Reading across the second row in the table, we can see that as before, babies' birth weight increases with mothers' pre-pregnancy weight, smoking during pregnancy has a negative effect on babies' birth weight, and both effects are significant. Next we'll make sense of how this all fits together.

Because "N" comes before "Y" alphabetically, R has coded the non-smokers as zero behind the scenes, and treats them as the 'reference' group. Therefore, the intercept estimate in the table above (2856.06953) refers to the intercept that has been fitted for the non-smoking mothers. The value in the Estimate column for 'SmokerY' is the difference between the intercept fitted for non-smokers and the intercept fitted for smokers. Since this is negative, it is telling us that smokers give birth to lighter babies than non-smokers (as we've seen before for the simple linear model). Additionally, this model estimates a slope for mothers' pre-pregnancy weight on babies' birth weights (1.85), which is the same for both smoking and non-smoking mothers.

We can plot out these relationships using the following steps. First, we'll plot out the data just for the non-smokers as follows, using indexing to select just the non-smoking subgroup from the data:

```
plot(Birth_weight~Mother_weight_pre_preg,
birth_data[birth_data$Smoker=="N",], col=rgb(0,0,0,0.25), pch=19, las=1,
ylab="Birth weight (grams)", xlab="Mother weight pre-pregnancy (lbs)")
```

And we can add the fit line for the non-smokers using `abline`. Previously, we've just fed the name of the model object to `abline` and it has fitted the line for us automatically, but that only works for simple linear regression. When we have more complex models, we will need to feed it the values of the intercepts and slopes extracted manually from the model summary, where a=intercept and b=slope. So, for the fit line for the non-smokers we use:

```
abline(a=2856.06953, b=1.84738, col="black", lwd=2)
```

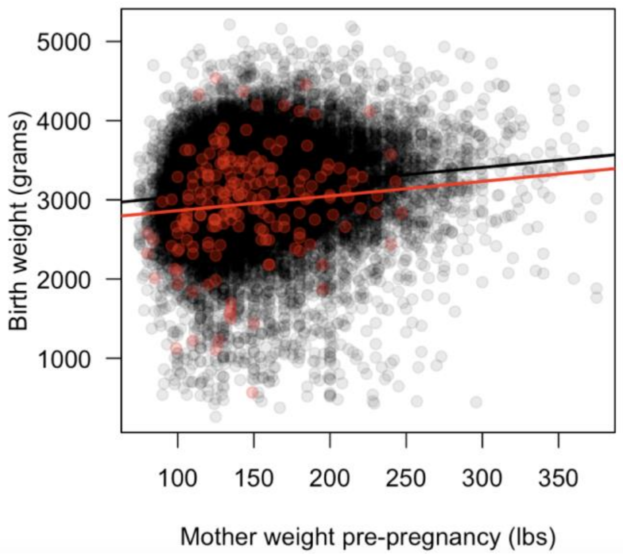
Then, we can add the data for the smokers to the existing plot using the function `points`. This takes the same arguments as `plot`, so we can use the following command to add the data for the smokers, in a different colour from the non-smokers:

```
points(Birth_weight~Mother_weight_pre_preg,
birth_data[birth_data$Smoker=="Y",], col=rgb(1,0,0,0.25), pch=19, las=1,
ylab="Birth weight (grams)", xlab="Mother weight pre-pregnancy (lbs)")
```

Now we can add the fit line for the smoking group (coloured red, to match the points), which has the same slope, but a different intercept, calculated as the intercept for the non-smokers plus the coefficent for the smokers:

```
abline(a=2856.06953+-175.86578, b=1.84738, col="red", lwd=2)
```

And you should end up with a plot that looks like the one below:



Bringing this all together, this is telling us that smokers have lighter babies even after accounting for the effect of mothers' pre-pregnancy weight on babies' birth weight. Put another way, babies are lighter *than expected given their mothers' weights* if the mother was a smoker than if she was a non-smoker. The model above says that regardless of how heavy mothers were before pregnancy, smoking during pregnancy decreases the weight of the baby by ~176 grams.

Fitting the same slope for each group assumes that smoking decreases babies' birth weight by the same amount regardless of how much mothers weighed before pregnancy. This may well not be the case, however. If we want to find out whether we are right to assume this, we can fit a more complex ANCOVA model in which the slopes are allowed to vary between the groups.

To do this in R, all we need to do is replace the plus sign separating the two variables with an asterisk (more generally, this is how we model **interactions** in R). Try this now using the commands below:

```
bw_ANCOVA2<-lm(Birth_weight~Mother_weight_pre_preg*Smoker, birth_data)

summary(bw_ANCOVA2)
```

```
Coefficients:
                                Estimate Std. Error t value Pr(>|t|)    
(Intercept)                   2857.71210   13.51031 211.521  < 2e-16 ***
Mother_weight_pre_preg           1.83644    0.08703  21.101  < 2e-16 ***
SmokerY                       -432.03337  165.61276  -2.609  0.00909 ** 
Mother_weight_pre_preg:SmokerY   1.75922    1.10366   1.594  0.11095    
```

You'll notice that we now have one additional line in the coefficients table. Reading down the rows, we have our intercept (2857.71) for the reference group (non-smokers) as before, then we have a slope (1.84), which is now the slope for the reference group. After that, we have the difference in intercepts between smokers and non-smokers (-432.03), and lastly, we have the difference in slopes between smokers and non-smokers (1.76). This last coefficient, however, is not significant (p=0.11), suggesting that the slopes are fairly similar to one another and so this more complex model is probably not necessary.

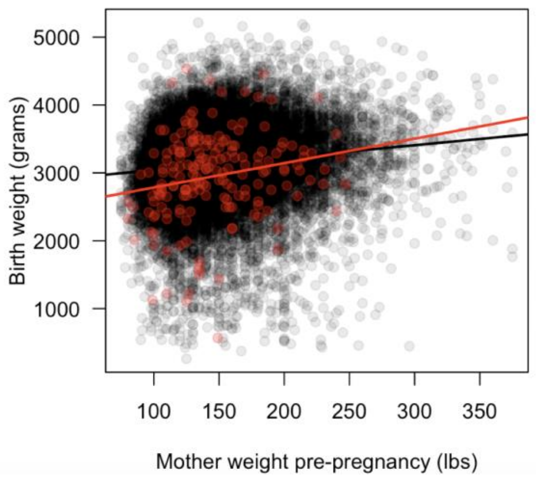
Just for learning purposes though, we'll walk through how to plot out the results of the different slopes model. As before, start by plotting out the data for the non-smokers. Then, add the fit line for this group using the reference-level intercept and slope (for the non-smokers):

```
abline(a=2857.71210, b=1.83644, col="black", lwd=2)
```

After you've added the data for the smokers, you can plot the line for this group as the intercept plus the difference in intercepts, and the slope plus the difference in slopes, as follows:

```
abline(a=2857.71210+-432.03337, b=1.83644+1.75922, col="red", lwd=2)
```

You should end up with the following plot:



Although the difference in slopes was not significant, taken at face value we can interpret this plot as follows. Lighter weight mothers who smoked during pregnancy tend to have lighter babies, although this difference diminishes as mothers get heavier. In fact, for mothers weighing more than about 250lb the pattern seems to reverse! However, we should be *very* cautious in interpreting this apparent pattern due to heteroscedasticity – we have far fewer mothers at the heavier (300lb+) end of the scale, and among these mothers' babies' birth weight is not as variable as it is at the lower end of the scale. Therefore, we can have much, much less certainty about the expected birth weights for such mothers depending on their smoking behaviour and body weight. This point nicely illustrates how violating the assumptions of regression can affect our results and interpretation.

GLM versus LM

Many statistical relationships are not as simple and linear as we might ideally wish. Linear models, as the name suggests, can only model linear relationships between variables. In contrast, ***generalized* linear models** are a generalization from the linear model to allow us to model other, more complex, types of relationships. So in fact, the linear model is a specialised case of the GLM. In these last sections we will briefly introduce two of the most common non-linear GLMs that you're likely to encounter, **logistic** and **Poisson** regression. These models are complex and we have time only for a brief foray into their basic functionality. As with all the other techniques we have encountered, I would therefore strongly recommend further reading before trying to use them on your own data, although the following exercises should help get things started.

Logistic regression

Logistic regression can be used to model a categorical dependent variable from one or more independent variables, which may be either categorical or continuous. In this section, we'll illustrate logistic regression by investigating possible predictors of smoking during pregnancy (treating smoking as a categorical variable on two levels: 'yes' and 'no'. Before we go any further, to make things easier to interpret down the line we will create a new variable for Smoker using numerical coding, where smokers = 1 and non-smokers = 0, which we'll call Smoker01, using ifelse statements:

```
birth_data$Smoker01<-ifelse(is.na(birth_data$Smoker), NA,
ifelse((birth_data$Smoker=="Y"), 1, 0))
```

Note that the above ifelse statement contains an additional, nested ifelse statement. This is necessary to deal with missing values. The command says essentially 'if the data are missing, code as NA, otherwise run another ifelse statement…'.

Logistic regression is based on the same equation we introduced that underpins linear regression:

$$y_i = \alpha + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \cdots + \varepsilon_i$$

However, since the outcome variable is now categorical, we want to predict the *probability* of Y rather than the value of Y (i.e. in this case, the probability that an individual is a smoker). This requires a slightly different equation:

$$P(Y) = \frac{1}{1 + e^{-(\alpha + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \cdots + \varepsilon_i)}}$$

P(Y) is the probability of Y occurring, *e* is the base of natural logarithms, while the brackets contain the right-hand side of the linear regression equation. Generally speaking, what we are assuming with non-linear GLM is that *some function of Y* (rather than raw values of Y) increases linearly with the predictor variables, known as a 'link function'. Each type of GLM (Poisson, logistic, etc.) has its own link function appropriate for the type of relationship it attempts to model.

By default, the function `glm` in R assumes that you wish to model a linear relationship, using the so-called 'identity link' (which simply models the raw values of Y). With linear models, we can use `glm` in exactly the same way as `lm`, and it will return the same results. Try for example comparing the results of these two models:

```
lm_test<-lm(Birth_weight~Gestation_weeks, data=birth_data)
```

```
glm_test<-glm(Birth_weight~Gestation_weeks, data=birth_data)
```

And you should see that the results are identical.

We'll now walk through a simple logistic regression model, predicting whether mothers smoked during pregnancy from a single predictor, their age at birth. We might anticipate that older mothers are more likely to have smoked during pregnancy than younger mothers, given how attitudes towards smoking have become increasingly negative over time. First, use the commands below to run the model. Note that we are using the `glm` function exactly as we did previously for the linear model, except we now include an additional argument 'family' that specifies we want to run a binomial model (i.e. a model with a categorical outcome variable with only two levels), using a logit link (the link used in logistic regression).

```
smoking_model1<-glm(Smoker01~Mother_age, data=birth_data,
family=binomial(link="logit"))

summary(smoking_model1)
```

We'll start by interpreting the coefficients:

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) -6.05483    0.32801  -18.46  < 2e-16 ***
Mother_age   0.04470    0.01117    4.00 6.33e-05 ***
```

First of all, since the coefficient for mothers' age is positive (and significant), we know that the probability of mothers smoking during pregnancy (coded as 1) increases with their age. Because we are no longer predicting linear relationships, however, we cannot interpret the coefficients from logistic regression models in the same way as those from linear regression models. We need to apply some kind of conversion in order to translate them back to the scale of the Y variable. In the case of logistic regression, we can make the results more intuitive by calculating an **odds ratio**, which tells us the increase in the odds of Y from a unit change in the predictor, i.e. in this case, how much the odds of smoking during pregnancy increase with each additional year of the mothers' age, relative to the odds of smoking given no change in her age. To calculate this, we need to take the exponential of the coefficient, using the function `exp`:

```
exp(0.04470)
```

Which gives us 1.045714. Since this is the ratio of the odds after a one unit increase in the predictor (i.e. one extra year of mothers' age) to the odds without a one unit increase in the predictor, a value over 1 means a positive effect. So, according to this model the odds of smoking during pregnancy increase, on average, by 1.05 times for each additional year of age when the mother gave birth. We can see therefore that while significant, this is not a huge effect in 'real world' terms.

Poisson regression

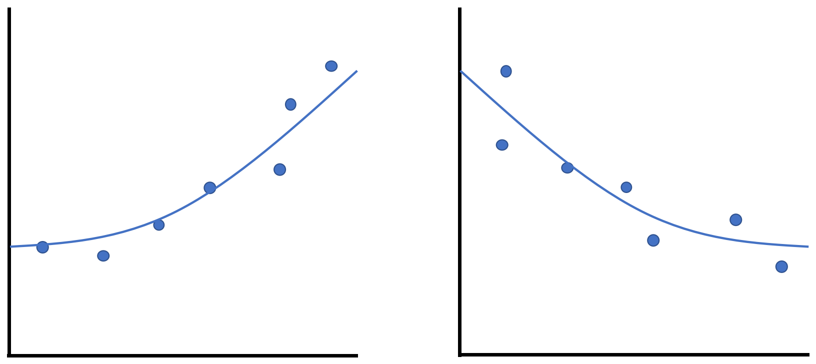
Poisson regression can be used to model dependent variables with Poisson distributions. Poisson distributions are described with a single parameter λ, which is equal to both the mean and the variance. At low values of lambda, the distribution is skewed with a long tail to the right, while as lambda increases, the distribution approaches normality. The command below will create a histogram for 1000 random observations from a Poisson distribution, setting lambda to 2. Try playing around with different values of lambda to get a feel for how this distribution works:

```
hist(rpois(n=1000, lambda=2))
```

Poisson models typically use a 'log link' function, which means they assume that the logarithm of Y (rather than its original value) increases linearly with the predictors. Putting this into the form of an equation, it means that we assume:

$$\ln(y_i) = \alpha + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \cdots + \varepsilon_i$$

This is the same as the linear regression equation, except that we are modelling the log of Y instead of the raw values of Y. This allows us to fit models with curvilinear relationships (upward curves for positive relationships and downward curves for negative relationships, as below):



Poisson regression is useful because many variables are Poisson distributed, particularly **count data**. Count data often have a characteristic right-skewed distribution, where there are many low counts and few high counts. Since count data are not truly continuous (they can only take positive integer values) they should not really be used as outcome variables in linear models. It is common practice to log-transform count data so that they approximate a normal distribution and then use linear modelling on the transformed data, but several papers have shown that this may be a bad idea[23].

One variable in the current dataset that might be Poisson distributed is the number of cigarettes smoked per day in the first trimester of pregnancy, since this is a count variable. To see if this is the case, we could plot out its distribution using a histogram, but it is so skewed that it is hard to tell what is going on:

```
hist(birth_data$Daily_cigs_trim1, breaks=30)
```

Alternatively we could check to see if the data fit with a Poisson distribution by calculating the mean and variance – they should be similar if the data are Poisson distributed:

```
mean(birth_data$Daily_cigs_trim1, na.rm=T)
```

```
var(birth_data$Daily_cigs_trim1, na.rm=T)
```

As you can see, the variance is many times greater than the mean, so the data don't really conform to a Poisson distribution. They are **over-dispersed,** meaning there is a lot more variation than there ought to be for a Poisson distribution, caused by a very long-tailed distribution in which there are a few very large values highly spread out from one another. These data are also probably **zero-inflated**, because the vast majority of mothers smoked no cigarettes at all during pregnancy. Therefore it is quite likely that the Poisson regression is **not** appropriate for these data, but just for teaching purposes we'll walk through using it anyway. You should, however, take these issues seriously if using Poisson models for your own data as results can be very different between models with different underlying assumptions about the distribution of the data.

We can use a Poisson regression to revisit the relationship between smoking during pregnancy and mothers' age at birth, but here treating smoking as count data rather than a categorical variable. We can fit the Poisson model as follows.

[23] E.g. O'Hara & Kotze (2010), Do not log-transform count data, *Methods in Ecology & Evolution* 1, 118-122 *:* https://besjournals.onlinelibrary.wiley.com/doi/10.1111/j.2041-210X.2010.00021.x

```
smoking_model2<-glm(Daily_cigs_trim1~Mother_age, data=birth_data,
family="poisson")
```

```
summary(smoking_model2)
```

And you'll see the following coefficients reported:

```
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.878604   0.115303  -33.64   <2e-16 ***
Mother_age   0.040968   0.003943   10.39   <2e-16 ***
```

Since we see a positive coefficient for mothers' age, which is significantly different from zero, we can conclude that older mothers smoked more cigarettes during pregnancy than younger mothers. To interpret the coefficients, as for the logistic regression we need to translate them back to the scale of the data. Since we are predicting the natural log of Y, we can exponentiate the coefficients (again using exp) to convert them to the data scale. This will convert the coefficients to a **rate ratio**, which tells us how many times the outcome variable is *multiplied* for every unit increase in the value of a predictor. So if we enter:

```
exp(0.040968)
```

We get 1.041819, meaning that for every additional year of mothers' age, the model predicts that she smokes 1.04 times as many cigarettes.

Session 6: Questions and exercises

1. Run a multiple regression model in which mother age and father age are both predictors of birth weight. Based on the coefficients and p-values, how should you interpret these results?

2. Run an ANCOVA model in which mothers' weight at delivery is predicted by their pre-pregnancy weight and the sex of the infant. Assume that the slopes are equal. How should you interpret the results?

3. Run a multiple regression model in which babies' birth weights are predicted by both mothers' heights and mothers' weights before pregnancy. Based on the variance inflation factors, do we have a problem with multicollinearity?

4. Based on adjusted R^2 values, which of the following predictors best explain variation in babies' birth weights: mother age, mother weight before pregnancy, or both together?

5. Based on adjusted R^2 values, how much additional variance in birth weights does daily smoking in the first trimester explain, compared to gestation time alone?

6. The results below are from an ANCOVA model predicting mother's weight at delivery from her weight before pregnancy and the sex of the infant. For which sex does the mother gain more weight during pregnancy?

```
Coefficients:
                                   Estimate Std. Error t value Pr(>|t|)
(Intercept)                       40.576282   0.500098  81.137   <2e-16
Mother_weight_pre_preg             0.901652   0.003219 280.093   <2e-16
Infant_sexM                       -0.518326   0.698777  -0.742   0.4582
Mother_weight_pre_preg:Infant_sexM  0.008825   0.004506   1.958   0.0502
```

7. Run a logistic regression model predicting the probability of smoking during pregnancy from fathers' age at birth. Based on the coefficient and p-value, is there any relationship between these variables?

8. Run a Poisson model predicting number of cigarettes smoked per day *before* pregnancy from mothers' age. What is the rate ratio for the effect of mother age on smoking?

Appendix: How to get betas in regression (and make beautiful regression tables)

Lynda Boothroyd, August 2019

Linear model functions in R all return 'unstandardized' regression estimates – this shows the amount of change in the DV in original units, for every unit of change in the IV. For instance if I run the following code

```
lm(weightkg ~ heightcm, data=data)
```

then the output will tell me how many kilos an individual is predicted to gain in weight, for every cm they gain in height.

Sometimes, however, we wish to know the standardised regression coefficient – that is how much change we predict in the DV for change in the IV, both measured in standard deviations. For instance, how many standard-deviations-worth of weight does someone gain for every standard-deviation of height? This is useful for comparing across models using different measures or populations with different distributions, although it's less easy to interpret in real-units terms.

One way to get standardised betas is to simply standardise (z-score) your variables before you run your models. However, it is also possible to get them as part of the output when creating regression tables using the tab_model function in the sjPlot package.
For instance this is code from a paper I have in progress in which participant characteristics are used to predict their preferences for weight (BMI) and shape (WHR) in female figures.

```
library(sjPlot)
modelB <- lm(valid_peakBMIpref ~  Age + Sex0male1female
                  + eth_mest + eth_misk + eth_gar + eth_creo +eth_other, data=MLMB)
modelW <- lm(validWHR_resid_beta ~  Age + Sex0male1female
                  + eth_mest + eth_misk + eth_gar + eth_creo +eth_other, data=MLMB)

tab_model(modelB, modelW,  show.ci=F, show.se=T, show.std = TRUE,
          pred.labels = pl,
          dv.labels = c("BMI preference","WHR preference slope"),
          string.est = "B",  string.se = "SE",  string.std = "β")
```

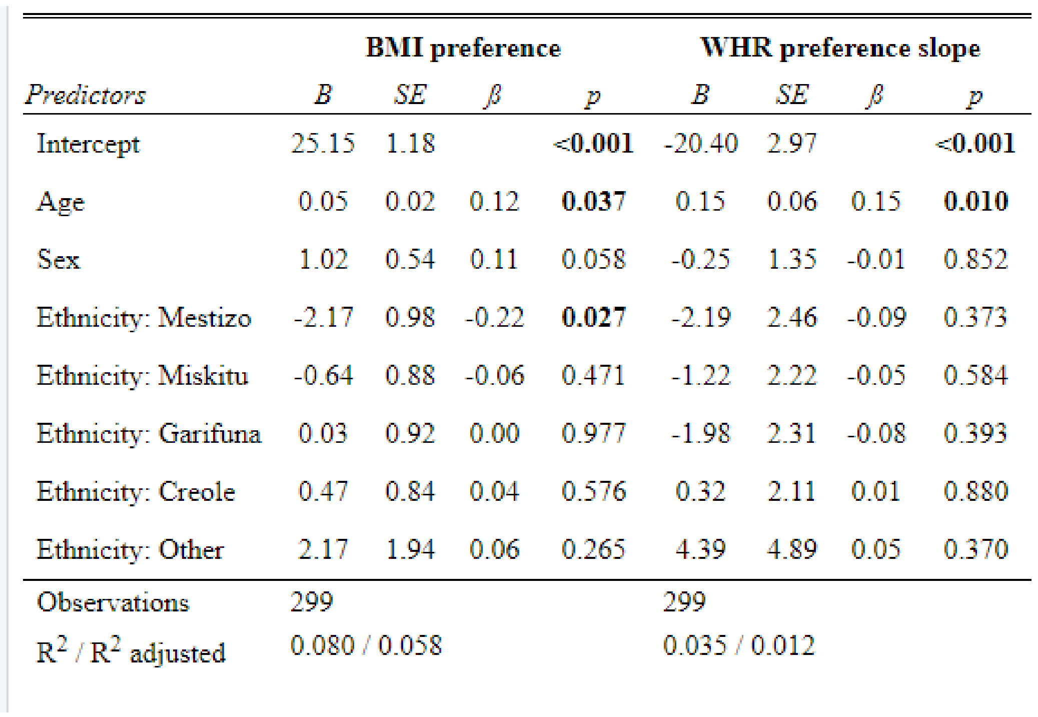
The crucical part is highlighted in yellow. The most basic way to run this is to simply indicate the models you wish tabulated and to include the show.std direction –

e.g. tab_model(model1, model2, show.std=T)

Everything else in the code above is to tweak what else it shows, set model and predictor and outcome labels, and to make it look more pretty.

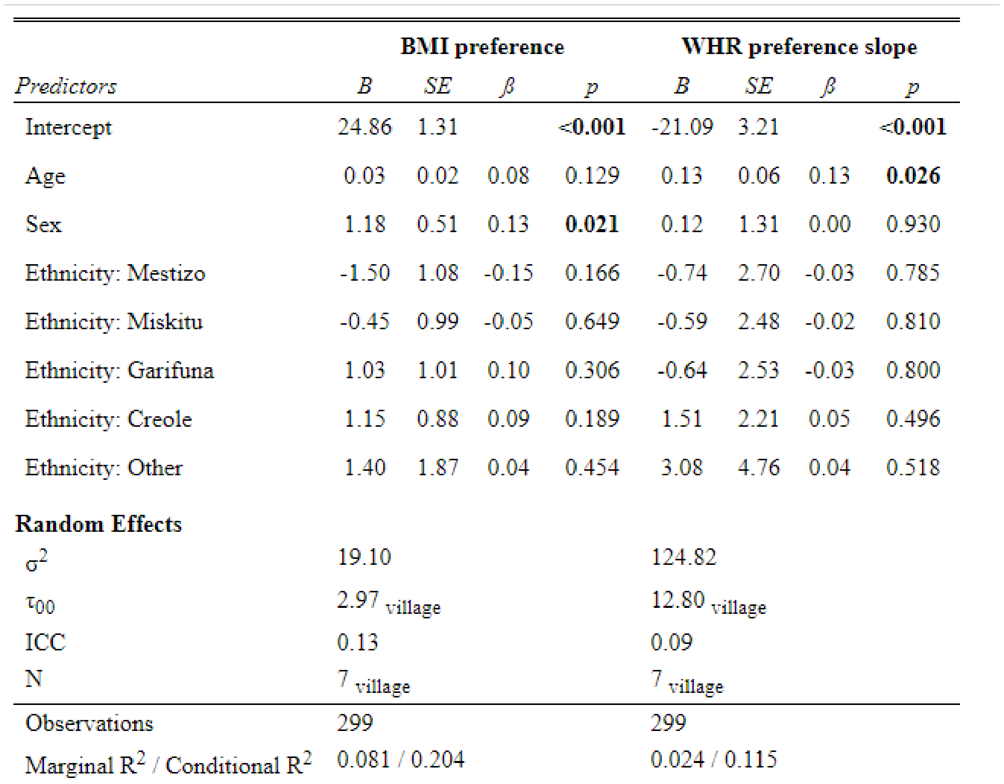
Here is the pretty table below. The package only outputs in html, but you can screencap it, or simply create a table in Word and then copy and paste the cells across.

There is more information on how to use tab_model here:
https://strengejacke.github.io/sjPlot/articles/tab_model_estimates.html



	BMI preference				WHR preference slope			
Predictors	*B*	*SE*	*ß*	*p*	*B*	*SE*	*ß*	*p*
Intercept	25.15	1.18		**<0.001**	-20.40	2.97		**<0.001**
Age	0.05	0.02	0.12	**0.037**	0.15	0.06	0.15	**0.010**
Sex	1.02	0.54	0.11	0.058	-0.25	1.35	-0.01	0.852
Ethnicity: Mestizo	-2.17	0.98	-0.22	**0.027**	-2.19	2.46	-0.09	0.373
Ethnicity: Miskitu	-0.64	0.88	-0.06	0.471	-1.22	2.22	-0.05	0.584
Ethnicity: Garifuna	0.03	0.92	0.00	0.977	-1.98	2.31	-0.08	0.393
Ethnicity: Creole	0.47	0.84	0.04	0.576	0.32	2.11	0.01	0.880
Ethnicity: Other	2.17	1.94	0.06	0.265	4.39	4.89	0.05	0.370
Observations	299				299			
R^2 / R^2 adjusted	0.080 / 0.058				0.035 / 0.012			

Tab_model will also deal with mixed effect models like this:



	BMI preference				WHR preference slope			
Predictors	*B*	*SE*	*ß*	*p*	*B*	*SE*	*ß*	*p*
Intercept	24.86	1.31		**<0.001**	-21.09	3.21		**<0.001**
Age	0.03	0.02	0.08	0.129	0.13	0.06	0.13	**0.026**
Sex	1.18	0.51	0.13	**0.021**	0.12	1.31	0.00	0.930
Ethnicity: Mestizo	-1.50	1.08	-0.15	0.166	-0.74	2.70	-0.03	0.785
Ethnicity: Miskitu	-0.45	0.99	-0.05	0.649	-0.59	2.48	-0.02	0.810
Ethnicity: Garifuna	1.03	1.01	0.10	0.306	-0.64	2.53	-0.03	0.800
Ethnicity: Creole	1.15	0.88	0.09	0.189	1.51	2.21	0.05	0.496
Ethnicity: Other	1.40	1.87	0.04	0.454	3.08	4.76	0.04	0.518
Random Effects								
σ^2	19.10				124.82			
τ_{00}	2.97 $_{village}$				12.80 $_{village}$			
ICC	0.13				0.09			
N	7 $_{village}$				7 $_{village}$			
Observations	299				299			
Marginal R^2 / Conditional R^2	0.081 / 0.204				0.024 / 0.115			

See more detail here: https://strengejacke.github.io/sjPlot/articles/tab_mixed.html