PHCM9795 Foundations of Biostatistics

Notes for R

Term 2, 2022

Contents

Contents **1**

1 Introduction to R and RStudio **3**
Learning outcomes . . . 3
1.1 Introduction . . . 3
1.2 R vs RStudio . . . 3
1.3 Installing R and RSudio . . . 4
1.4 A simple R analysis . . . 6
1.5 The RStudio environment . . . 7
1.6 Some R basics . . . 9
1.7 Part 2: Obtaining basic descriptive statistics . . . 13
1.8 Defining categorical variables as factors . . . 20
Part 3: Creating other types of graphs . . . 21

2 Probability and probability distributions: R notes **27**
2.1 Importing data into Stata . . . 27
2.2 Checking your data for errors in Stata . . . 28
2.3 Overlaying a Normal curve on a histogram . . . 30
2.4 ggplot2 . . . 32
2.5 Descriptive statistics for checking normality . . . 32
2.6 Importing Excel data into Stata . . . 33
2.7 Generating new variables . . . 33
2.8 Summarising data by another variable . . . 35
2.9 Recoding data . . . 36
2.10 Computing binomial probabilities using R . . . 37
2.11 Computing probabilities from a Normal distribution . . . 38

3 Precision: R notes **39**
3.1 Calculating a 95% confidence interval of a mean . . . 39

4 Hypothesis testing **41**
4.1 One sample t-test . . . 41

5 Comparing two means **43**
5.1 Checking data for the independent samples t-test . . . 43
5.2 Independent samples t-test . . . 48
5.3 Checking the assumptions for a Paired t-test . . . 48
5.4 Paired t-Test . . . 49

6 Proportions **51**
6.1 95% confidence intervals for proportions . . . 51
6.2 Significance test for single proportion . . . 51
6.3 Computing a relative risk and its 95% confidence interval . . . 52
6.4 Computing an odds ratio and its 95%CI . . . 54

7 Testing proportions **57**
7.1 Pearson's chi-squared test . . . 57
7.2 Chi-squared test for tables larger than 2-by-2 . . . 59
7.3 McNemar's test for paired proportions . . . 61

8 Correlation and simple linear regression **63**
8.1 Creating a scatter plot . . . 63
8.2 Calculating a correlation coefficient . . . 66
8.3 Fitting a simple linear regression model . . . 66
8.4 Plotting residuals from a simple linear regression . . . 67

9 Analysing non-normal data **71**
9.1 Transforming non-normally distributed variables . . . 71
9.2 Wilcoxon ranked-sum test . . . 72
9.3 Wilcoxon matched-pairs signed-rank test . . . 73
9.4 Estimating rank correlation coefficients . . . 74

10 Sample size **75**
10.1 Sample size calculation for two independent samples t-test . . . 75
10.2 Sample size calculation for difference between two independent proportions . . . 76
10.3 Sample size calculation with a relative risk . . . 77
10.4 Sample size calculation with an odds ratio . . . 77

Bibliography **79**

Module 1

Introduction to R and RStudio

INCLUDE:

- decide on skim vs summary vs jmv::describe
- case sensitive
- how to get help (online, google, etc)
- functions that use (data=, var=) vs functions that use an object (i.e. data$var)
- how to specify a column from a dataframe
- don't give up!
- tidyverse?

Learning outcomes

By the end of this Module, you will be able to:

- understand the difference between R and RStudio
- navigate the RStudio interface
- input and import data into R
- use R to summarise data
- perform basic data transformations
- assign variable and value labels
- understand the difference between saving R data and saving R output
- copy R output to a standard word processing package

1.1 Introduction

"R is a language and environment for statistical computing and graphics." [https://www.r-project.org/about.html]. It is an open-source programming language, used mainly for statistics. It is increasingly used in health research, as well as in other fields such as econometrics and social science. The aim of these notes is to introduce the R language within the RStudio environment, and to introduce the commands and procedures that are directly relevant to this course. There is so much more to R than we can cover in these notes. Relevant information will be provided throughout the course, and we will provide further references that you can explore if you are interested.

1.2 R vs RStudio

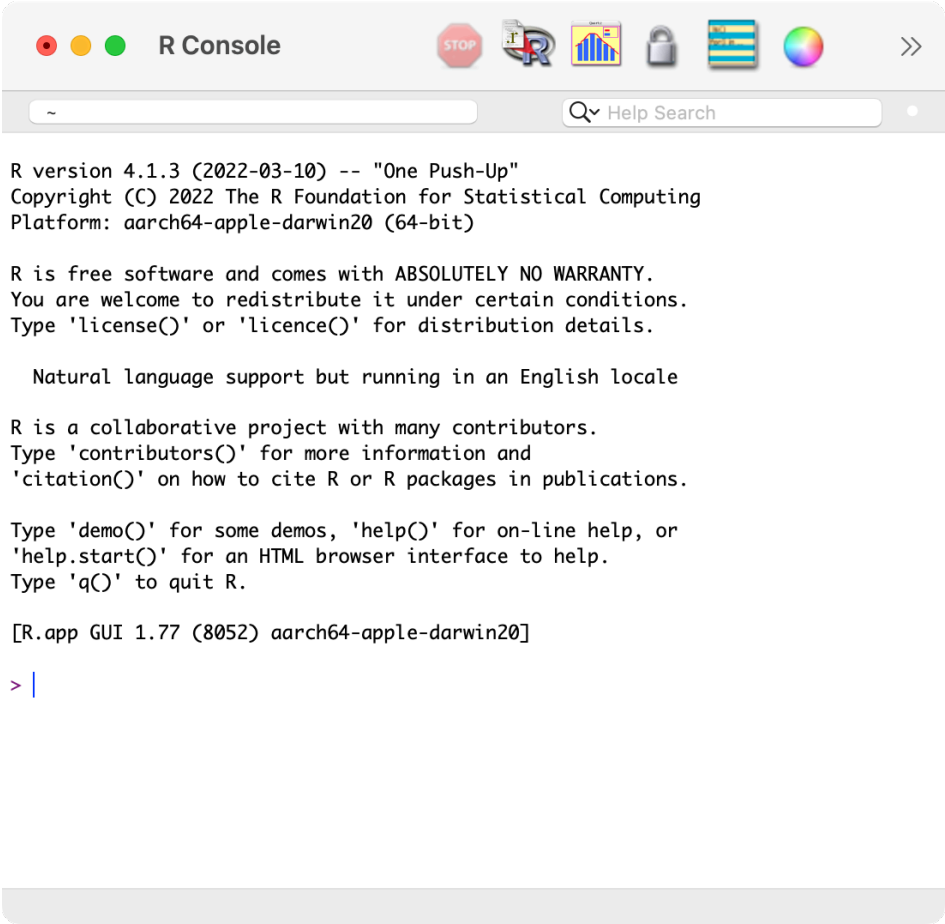
At its heart, R is a programming language. When you install R on your computer, you are installing the language and its resources, as well as a very basic interface for using R. You can write and run R code using R, but we don't recommend it.

RStudio is an “Integrated Development Environment” that runs R while also providing useful tools to help you as you’re writing code and analysing data. Think of R as the engine which does the work, and RStudio as the wrapper which provides a more user-friendly way to interact with R.

1.3 Installing R and RSudio

To install R on your computer:

1. Download the R installer:
   a. for Windows:
   b. for MacOS:
2. Install R by running the installer and following the installation instructions. The default settings are fine. **Note for macOS:** if you are running macOS 10.8 or later, you will need to install an additional application called XQuartz, which is available at https://www.xquartz.org/. Download the latest installer (XQuartz-2.8.1.dmg as of April 2022), and install it in the usual way.
3. Open the R program. You should see a screen as below:



Near the bottom of the R screen, you will find the ">" symbol which represents the command line. If you type `1 + 2` into the command line and then hit enter you should get:

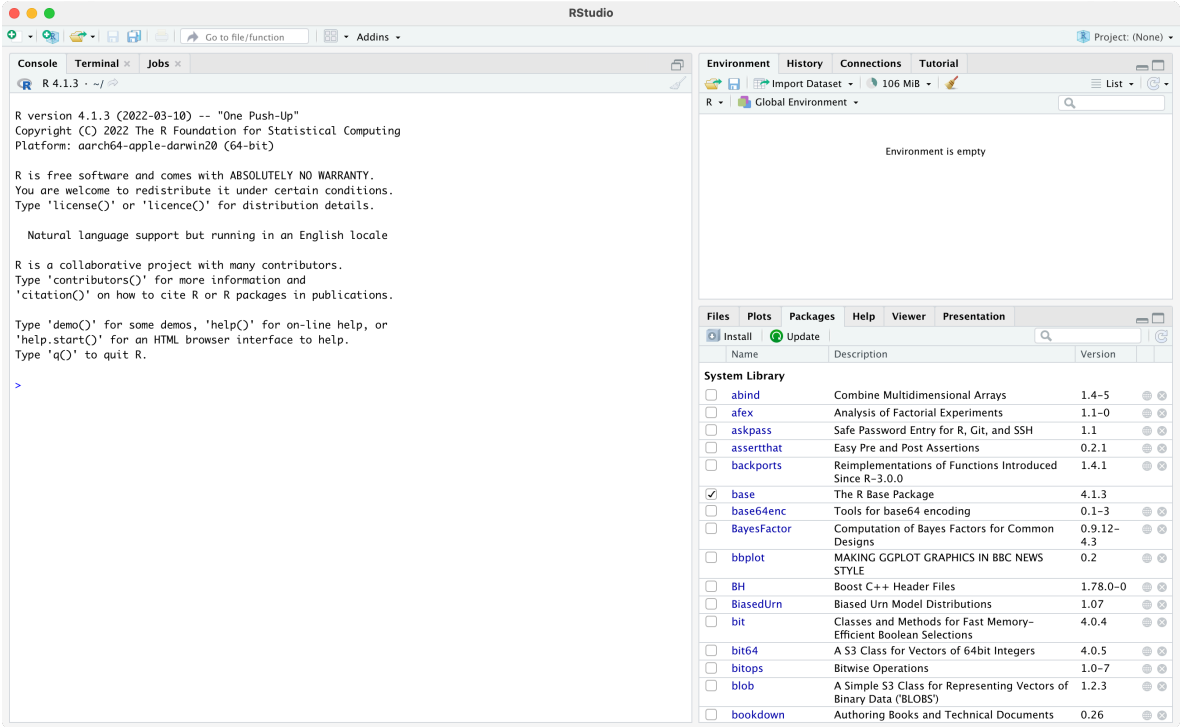
```
[1] 3
```

This is R performing your calculation, with the `[1]` indicating that the solution to `1 + 2` is a vector of size 1. We will talk about vectors later.

At this point, close R - we will not interact with R like this in the future. [HOW TO CLOSE R]

To install RStudio on your computer:

1. Make sure you have already installed R, and verified that it is working.
2. Download the RStudio desktop installer at: https://www.rstudio.com/products/rstudio/download. Ensure that you select the RStudio Desktop (Free) installer in the first column.
3. Install RStudio by running the installer and following the installation instructions. The default settings are fine.
4. Open RStudio, which will appear as below:



Locate the command line symbol ">" at the bottom of the left-hand panel. Type `1 + 2` into the command line and hit enter, and you will see:

```
[1] 3
```

This confirms that RStudio is running correctly, and calling the R language to correctly calculate the sum between 1 and 2!

RStudio currently comprises three window panes, and we will discuss these later.

1.4 A simple R analysis

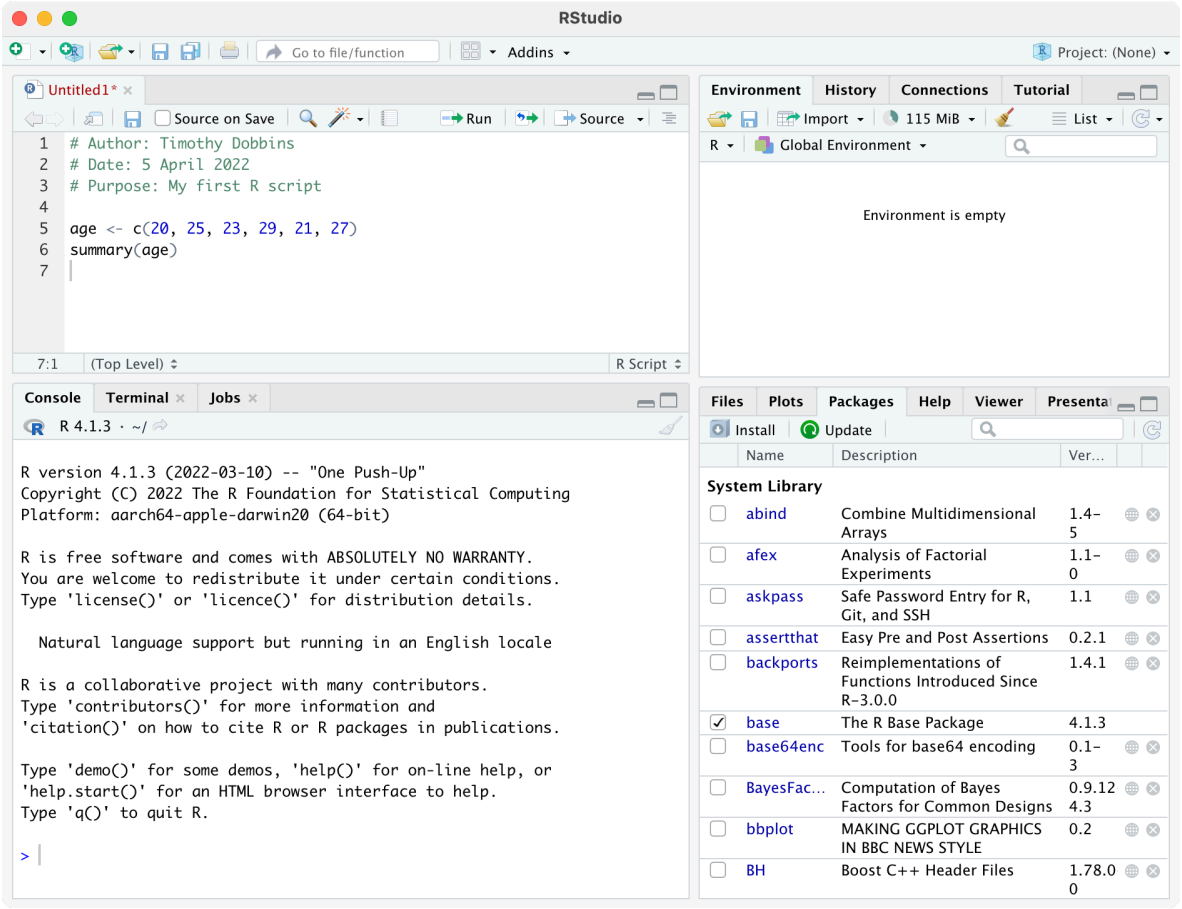
In this very brief section, we will introduce R by calculating the average of six ages.

To begin, open a new R Script by choosing **File > New file > R Script** . A script (or a program) is a collection of commands that are sequentially processed by R. You can also type Ctrl+Shift+N in Windows, or Command+Shift+N in MacOS to open a new script in RStudio, or click the **New File** button at the top of the RStudio window.

You should now see four window panes, as below. In the top-left window, type the following (replacing my name with yours, and including today's date):

```
# Author: Timothy Dobbins
# Date: 5 April 2022
# Purpose: My first R script

age <- c(20, 25, 23, 29, 21, 27)
summary(age)
```

Your screen should look something like:



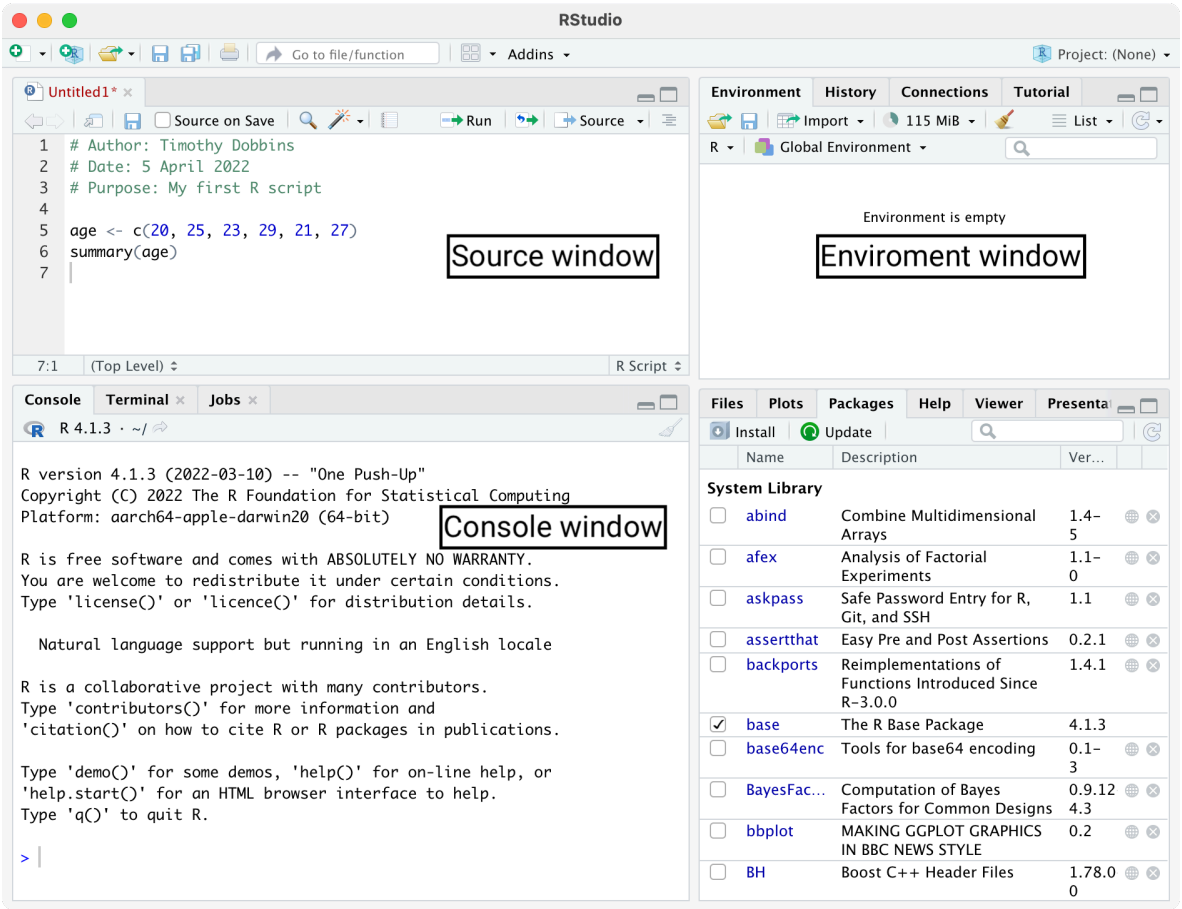
To run your script, choose **Code > Run Region > Run All**. You will see your code appear in the bottom-right window, with the following output:

```
> # Author: Timothy Dobbins
> # Date: 5 April 2022
> # Purpose: My first R script
>
> age <- c(20, 25, 23, 29, 21, 27)

> summary(age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  20.00   21.50   24.00   24.17   26.50   29.00
```

We will explain the key parts of this script later, but for now, you have entered six ages and calculated the mean age (along with five other summary statistics).

1.5 The RStudio environment

Now that we have seen a simple example of how to use R within RStudio, let's describe the RStudio environment. Let's assume that you have opened a new script editor, and you have four windows as below:



The **Source** window is where you will write and edit your R scripts. The R script can be saved by clicking on File -> Save As or by clicking on the symbol of a floppy disk at the top of the script. The file will have an extension of .R, for example name_of_script.R. Give it a meaningful title and remember to periodically save as you go.

In RStudio, the name of the script will be black when it has been saved, and will change to red if you have any unsaved changes.

The **Console** window, at the bottom left, contains the command line which is indicated with the symbol >. You can type commands here, but anything executed directly from the console is not saved and therefore is lost when the session ends (when you exit RStudio). You should always run your commands from a script file which you can save and use again later. When you run commands from a script, the output and any notes/errors are shown in the console. The Terminal and Jobs tabs will not be used in this course.

The **Environment** window at the top-right shows a list of objects that have been created during your session. When you close your RStudio session these objects will disappear. We will not use the History or Connections tabs in this course.

The bottom right corner contains some useful tabs, in particular the **Help** tab. When you are troubleshooting errors or learning how to use a function, the Help tab should be the first place you visit. Here you can search the help documents for all the packages you have installed. Whenever you create plots in R, these will be shown in the **Plots** tab. The **Packages** tab contains a list of installed packages and indicates which ones are currently in use (we will learn about packages later). Packages which are loaded, i.e. in use, are indicated with a tick. Some packages are in use by default when you begin a new session. You can access information about a package by clicking on its name. The **Files** tab provides a shortcut to access your files. The Viewer tab will not be used in this course.

1.6 Some R basics

While we use R as a statistics package, R is a programming language. In order to use R effectively, we need to define some basics.

1.6.1 Objects

If you do some reading about R, you may learn that R is an "object-oriented programming language". When we enter or import data into R, we are asking R to create **objects** from our data. These objects can be manipulated and transformed by **functions**, to obtain useful insights from our data.

Objects in R are created using the **assignment operator**. The most common form of the assignment operator looks like an arrow: `<-` and is typed as the `<` and `-` symbols. The simplest way of reading `<-` is as the words "is defined as". Note that it possible to use `->` and even `=` as assignment operators, but their use is less frequent.

Let's see an example:

```
x <- 42
```

This command creates a new object called `x`, which is defined as the number 42 (or in words, "x is defined as 42"). Running this command gives no output in the console, but the new object appears in the top-right **Environment** panel. We can view the object in the console by typing its name:

```
# Print the object x
x
#> [1] 42
```

Now we see the contents of `x` in the console.

This example is rather trivial, and we rarely assign objects of just one value. We'll see a more realistic example soon.

1.6.2 Data structures

There are two main data structures we will use in the course: **vectors** and **data frames**. A **vector** is a combination of data values, all of the same type. For example, our six ages that we entered earlier is a vector. You could think of a vector as a column of data (even though R prints vectors as rows!) And technically, even an object with only one value is a vector, a vector of size 1.

The easiest way of creating a vector in R is by using the `c()` function, where c stands for 'combine'. In our previous Simple Analysis in R (Section 1.4), we wrote the command:

```
age <- c(20, 25, 23, 29, 21, 27)
```

This command created a new object called `age`, and *combined* the six values of age into one vector.

Just as having a vector of size 1 is unusual, having just one column of data to analyse is also pretty unusual. The other structure we will describe here is a **data frame** which is essentially a collection of vectors, each of the same size. You could think of a data frame as being like a spreadsheet, with columns representing variables, and rows representing observations.

There are other structures in R, such as matrices and lists, which we won't discuss in this course.

1.6.3 Functions

If objects are the nouns of R, functions are the verbs. Essentially, functions transform objects. Functions can transform your data into summary statistics, graphical summaries or analysis results. For example, we used the `summary()` function to display summary statistics for our six ages.

R functions are specified by their arguments (or inputs). The arguments that can be supplied for each function can be inspected by examining the help notes for that function. To obtain help for a function, we can submit `help(summary)` (or equivalently `?summary()`) in the console, or we can use the **help** tab in the bottom-right window of RStudio. For example, the first part of the help notes for `summary` appear as:

summary {base} R Documentation

Object Summaries

Description

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular `methods` which depend on the `class` of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits, quantile.type = 7)
## S3 method for class 'data.frame'
summary(object, maxsum = 7,
       digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor'
summary(object, maxsum = 100, ...)
```

The help notes in R can be quite cryptic, but **Usage** section details what values should be provided for the function to run. Here, `summary` requires an object to be specified. In our case, we specified `age`, which is our object defined as the vector of six ages.

Most help pages also include some examples of how you might use the function. These can be found at the very bottom of the help page.

Examples

Run examples

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)

lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

The `summary` function is quite simple, in that it only requires one input, the object to be summarised. More complex functions might require a number of inputs. For example, the help notes for the `descriptives()` function in the `jmv` package show a large number of inputs can be specified:

descriptives {jmv} R Documentation

Descriptives

Description

Descriptives are an assortment of summarising statistics, and visualizations which allow exploring the shape and distribution of data. It is good practice to explore your data with descriptives before proceeding to more formal tests.

Usage

```
descriptives(data, vars, splitBy = NULL, freq = FALSE,
  desc = "columns", hist = FALSE, dens = FALSE, bar = FALSE,
  barCounts = FALSE, box = FALSE, violin = FALSE, dot = FALSE,
  dotType = "jitter", boxMean = FALSE, boxLabelOutliers = TRUE,
  qq = FALSE, n = TRUE, missing = TRUE, mean = TRUE,
  median = TRUE, mode = FALSE, sum = FALSE, sd = TRUE,
  variance = FALSE, range = FALSE, min = TRUE, max = TRUE,
  se = FALSE, ci = FALSE, ciWidth = 95, iqr = FALSE,
  skew = FALSE, kurt = FALSE, sw = FALSE, pcEqGr = FALSE,
  pcNEqGr = 4, pc = FALSE, pcValues = "25,50,75", formula)
```

There are two things to note here. First, notice that the first two inputs are listed with no = symbol, but all other inputs are listed with = symbols (with values provided after the = symbol). This means that everything apart from `data` and `vars` have **default** values. We are free to not include values for these inputs if we are happy with the defaults provided. For example, by default the variance is not calculated (as `variance = FALSE`). To obtain the variance as well as the standard deviation, we can change this default to `variance = TRUE`:

```r
# Only the standard deviation is provided as the measure of variability
descriptives(data=pbc, vars=age)

# Additionally request the variance to be calculated
descriptives(data=pbc, vars=age, variance=TRUE)
```

Second, for functions with multiple inputs, we can specify the input name and its value, or we can specify the inputs **in the order listed in the Usage section**. So the following are equivalent:

```r
# We can specify that the dataset to be summarised is pbc,
#   and the variable to summarise is age:
descriptives(data=pbc, vars=age)

# We can omit the input name, as long as we keep the inputs in the correct order -
#   that is, dataset first, variable second:
descriptives(pbc, age)

# We can change the order of the inputs, as long as we specify the input name:
descriptives(vars=age, data=pbc)
```

In this course, we will usually provide all the input names, even when they are not required.

1.6.4 Packages

A **package** is a collection of functions, documentation (and sometimes datasets) that extend the capabilities of R. Packages have been written by R users to be freely distributed and used by others. R packages can be obtained from many sources, but the most common source is CRAN: the Comprehensive R Archive Network.

A useful way of thinking about R is that R is like a smartphone, with packages being like apps which are downloaded from CRAN (similar to an app-store). When you first install R, it comes with a basic set of packages (apps) installed. You can do a lot of things with these basic packages, but sometimes you might want to do things differently (you might prefer Firefox as your browser), or you may want to perform some analyses that can't be done using the default packages. In these cases, you can install a package.

Like installing an app on a smartphone, you only need to *install* a package once. But each time you want to use the package, you need to *load* the package into R. This is similar to running the app on your phone. The analogy falls down a bit in that we usually load more than one package in an R script - but we only load the packages we need for that R session.

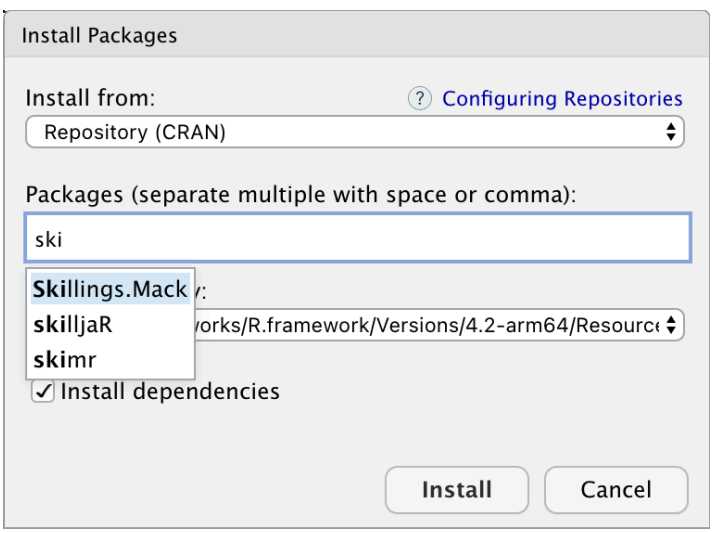
1.6.5 How to install a package

There are a couple of ways to install a package. You can use the `install.packages()` function if you know the exact name of the package. Let's use an example of installing the `skimr` package, which gives a very nice, high-level overview of any data frame. We can install `skimr` by typing the following into the console:

```
install.packages("skimr")
```

Note the use of the quotation marks.

Alternatively, RStudio offers a graphical way of installing packages that can be accessed via **Tools > Install Packages**, or via the **Install** button at the top of the **Packages** tab in the bottom-right window. You can begin typing the name of the package in the dialog box that appears, and RStudio will use predictive text to offer possible packages:



While writing code is usually the recommended way to use R, installing packages is an exception. Using the graphical interface is perfectly fine, because you only need to install a package once.

1.6.6 How to load a package

When you begin a new session in RStudio, i.e. when you open RStudio, only certain core packages are automatically loaded. You can use the `library()` function to load a package that you has previously been installed. For example, now that we have installed `skimr`, we need to load it before we can use it:

```
library(skimr)
```

Note that quotation marks are not required for the `library()` function (although they can be included if you really like quotation marks!).

1.7 Part 2: Obtaining basic descriptive statistics

In this exercise, we will analyse data to complete a descriptive table from a research study. The data come from a study in primary biliary cirrhosis, a condition of the liver, from Therneau and Grambsch [2010], Modeling Survival Data: Extending the Cox Model. By the end of this exercise, we will have completed the following table.

Table 1.1: Summary of 418 participants from the PBC study (Therneau and Grambsch, 2000)

Characteristic		**Summary**
Age (years)		Mean (SD) or Median [IQR]
Sex	Male	n (%)
	Female	n (%)
AST* (U/ml)		Mean (SD) or Median [IQR]
Serum bilirubin		Mean (SD) or Median [IQR]
Stage	I	n (%)
	II	n (%)
	III	n (%)
	IIIV	n (%)
Vital status at study end	Alive: no transplant	n (%)
	Alive: transplant	n (%)
	Deceased	n (%)

* asparate aminotransferase

This table is available in Table1.docx, saved on Moodle.

1.7.1 Opening a data file

Typing data directly into R is not common; we usually open data that have been previously saved. There are two useful packages for importing data into R: `haven` (for data that have been saved by Stata, SAS or SPSS) and `readxl` (for data saved by Microsoft Excel). Additionally, the `labelled` package is useful in working with data that have been labelled in Stata. Here, we will open a dataset that has been stored as a Stata data file (which has the .dta suffix):

1 - If necessary, install the `haven` and `readxl` packages. As mentioned earlier, packages only need to be installed if they have not been installed earlier.

```
install.packages("haven")
install.packages("readxl")
```

2 - Locate the data set called pbc.dta on Moodle. Click the file to download it, and then save it in a folder you will be able to locate later - for example, your OneDrive folder. The description of this dataset (i.e. the metadata) have been saved as a plain text file: pbc_info.txt. Locate the file and filepath of pbc.dta.

3 - In R, use the `read_dta()` function to read the Stata data into new object called `pbc`. Remember that we need to load the `haven` and `labelled` packages into R:

```
library(haven)
library(labelled)
library(skimr)

pbc <- read_dta("data/examples/pbc.dta")
```

4 - We now re-assign the pbc object by using the `unlabelled()` function from the `labelled` package:

```
pbc <- unlabelled(pbc)
```

Note that we can combine the `unlabelled()` and `read_dta()` functions together, to complete this process in one line:

```
pbc <- unlabelled(read_dta("data/examples/pbc.dta"))
```

5 - We can now use the `summary()` function to examine the pbc dataset.

```
summary(pbc)
#>        id             time          status
#>  Min.   :  1.0   Min.   :  41   Min.   :0.0000
#>  1st Qu.:105.2   1st Qu.:1093   1st Qu.:0.0000
#>  Median :209.5   Median :1730   Median :0.0000
#>  Mean   :209.5   Mean   :1918   Mean   :0.8301
#>  3rd Qu.:313.8   3rd Qu.:2614   3rd Qu.:2.0000
#>  Max.   :418.0   Max.   :4795   Max.   :2.0000
#>
#>       trt             age             sex
#>  Min.   :1.000   Min.   :26.28   Min.   :1.000
#>  1st Qu.:1.000   1st Qu.:42.83   1st Qu.:2.000
#>  Median :1.000   Median :51.00   Median :2.000
#>  Mean   :1.494   Mean   :50.74   Mean   :1.895
#>  3rd Qu.:2.000   3rd Qu.:58.24   3rd Qu.:2.000
#>  Max.   :2.000   Max.   :78.44   Max.   :2.000
#>  NA's   :106
#>     ascites            hepato          spiders
#>  Min.   :0.00000   Min.   :0.0000   Min.   :0.0000
#>  1st Qu.:0.00000   1st Qu.:0.0000   1st Qu.:0.0000
#>  Median :0.00000   Median :1.0000   Median :0.0000
#>  Mean   :0.07692   Mean   :0.5128   Mean   :0.2885
#>  3rd Qu.:0.00000   3rd Qu.:1.0000   3rd Qu.:1.0000
```

```
#>  Max.   :1.00000   Max.   :1.0000   Max.   :1.0000
#>  NA's   :106       NA's   :106      NA's   :106
#>      edema             bili             chol
#>  Min.   :0.0000   Min.   : 0.300   Min.   : 120.0
#>  1st Qu.:0.0000   1st Qu.: 0.800   1st Qu.: 249.5
#>  Median :0.0000   Median : 1.400   Median : 309.5
#>  Mean   :0.1005   Mean   : 3.221   Mean   : 369.5
#>  3rd Qu.:0.0000   3rd Qu.: 3.400   3rd Qu.: 400.0
#>  Max.   :1.0000   Max.   :28.000   Max.   :1775.0
#>                                    NA's   :134
#>     albumin          copper          alkphos
#>  Min.   :1.960   Min.   :  4.00   Min.   :  289.0
#>  1st Qu.:3.243   1st Qu.: 41.25   1st Qu.:  871.5
#>  Median :3.530   Median : 73.00   Median : 1259.0
#>  Mean   :3.497   Mean   : 97.65   Mean   : 1982.7
#>  3rd Qu.:3.770   3rd Qu.:123.00   3rd Qu.: 1980.0
#>  Max.   :4.640   Max.   :588.00   Max.   :13862.4
#>                  NA's   :108      NA's   :106
#>       ast              trig           platelet
#>  Min.   : 26.35   Min.   : 33.00   Min.   : 62.0
#>  1st Qu.: 80.60   1st Qu.: 84.25   1st Qu.:188.5
#>  Median :114.70   Median :108.00   Median :251.0
#>  Mean   :122.56   Mean   :124.70   Mean   :257.0
#>  3rd Qu.:151.90   3rd Qu.:151.00   3rd Qu.:318.0
#>  Max.   :457.25   Max.   :598.00   Max.   :721.0
#>  NA's   :106      NA's   :136      NA's   :11
#>     protime          stage
#>  Min.   : 9.00   Min.   :1.000
#>  1st Qu.:10.00   1st Qu.:2.000
#>  Median :10.60   Median :3.000
#>  Mean   :10.73   Mean   :3.024
#>  3rd Qu.:11.10   3rd Qu.:4.000
#>  Max.   :18.00   Max.   :4.000
#>  NA's   :2       NA's   :6
```

An alternative to the `summary()` function is the `skim()` function in the `skimr` package, which produces summary statistics as well as rudimentary histograms:

```
skim(pbc)
```

```
── Data Summary ────────────────────────
                           Values
Name                       pbc
Number of rows             418
Number of columns          20
_______________________
Column type frequency:
  numeric                  20
_______________________
Group variables            None

── Variable type: numeric ──────────────────────────────────────────────────────────────────────────────────────────────────────────────
   skim_variable n_missing complete_rate      mean        sd     p0     p25     p50     p75     p100 hist
 1 id                    0         1      210.      121.      1     105.    210.    314.     418
 2 time                  0         1     1918.     1105.     41    1093.   1730    2614.    4795
 3 status                0         1        0.830     0.956   0       0       0       2        2
 4 trt                 106         0.746    1.49      0.501   1       1       1       2        2
 5 age                   0         1       50.7      10.4    26.3    42.8    51.0    58.2     78.4
 6 sex                   0         1        1.89      0.307   1       2       2       2        2
 7 ascites             106         0.746    0.0769    0.267   0       0       0       0        1
 8 hepato              106         0.746    0.513     0.501   0       0       1       1        1
 9 spiders             106         0.746    0.288     0.454   0       0       0       1        1
10 edema                 0         1        0.100     0.253   0       0       0       0        1
11 bili                  0         1        3.22      4.41    0.3     0.8     1.4     3.4     28
12 chol                134         0.679  370.      232.    120     250.    310.    400     1775
13 albumin               0         1        3.50      0.425   1.96    3.24    3.53    3.77     4.64
14 copper              108         0.742   97.6      85.6     4      41.2    73     123      588
15 alkphos             106         0.746 1983.     2140.    289     872.   1259    1980    13862.
16 ast                 106         0.746  123.       56.7    26.4    80.6   115.    152.     457.
17 trig                136         0.675  125.       65.1    33      84.2   108     151      598
18 platelet             11         0.974  257.       98.3    62     188.    251     318      721
19 protime               2         0.995   10.7       1.02    9      10      10.6    11.1     18
20 stage                 6         0.986    3.02      0.882   1       2       3       4        4
```

1.7.2 Summarising continuous variables

One of the most flexible functions for summarising continuous variables is the `descriptives()` function from the `jmv` package. The function is specified as `descriptives(data=, vars=)` where:

- `data` specifies the dataframe to be analysed
- `vars` specifies the variable(s) of interest, with multiple variables combined using the `c()` function

We can summarise the three continuous variables in the pbc data: age, AST and serum bilirubin, as shown below.

```
library(jmv)

descriptives(data=pbc, vars=c(age, ast, bili))
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                          age         ast         bili
#>
#>    N                      418         312          418
#>    Missing                  0         106            0
#>    Mean              50.74155    122.5563     3.220813
#>    Median            51.00068    114.7000     1.400000
#>    Standard deviation 10.44721   56.69952     4.407506
#>    Minimum           26.27789    26.35000    0.3000000
```

```
#>    Maximum                78.43943    457.2500     28.00000
#>
```

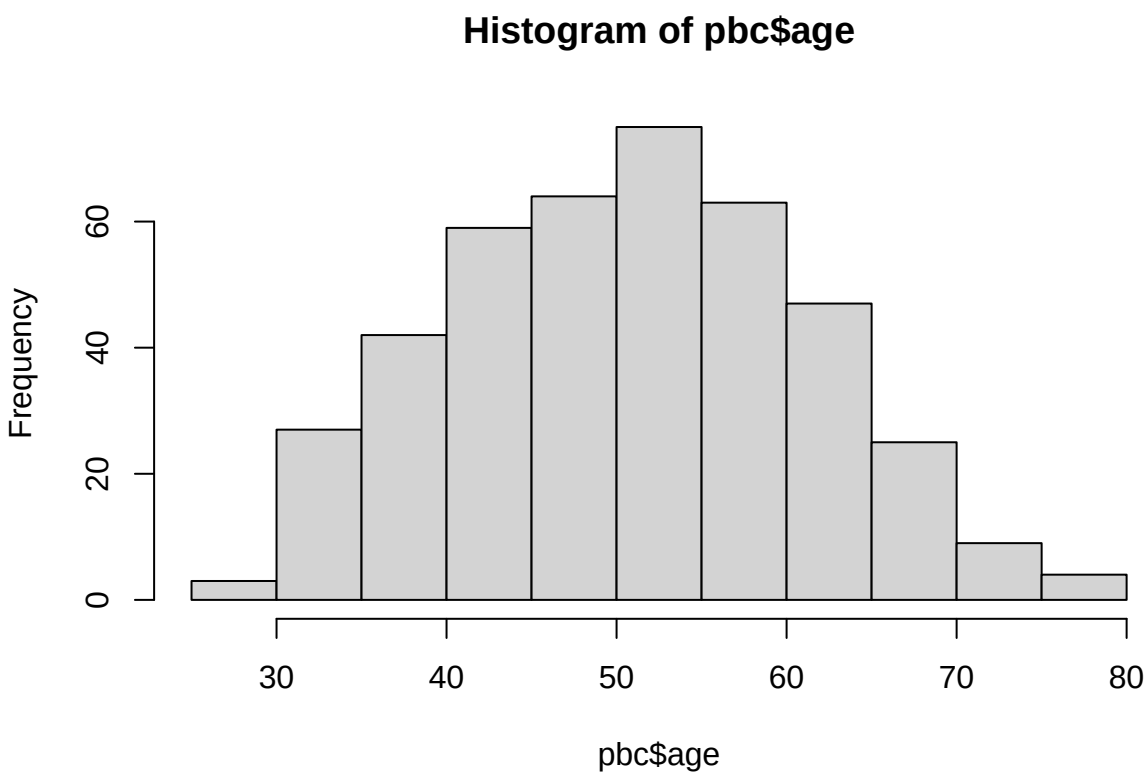
By default, the `descriptives` function presents the mean, median, standard deviation, minimum and maximum. We can request additional statistics, such as the quartiles (which are called the percentiles, or pc, in the descriptives function):

```
descriptives(data=pbc, vars=c(age, ast, bili), pc=TRUE)
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                          age         ast         bili
#>
#>    N                       418         312          418
#>    Missing                   0         106            0
#>    Mean                 50.74155    122.5563     3.220813
#>    Median               51.00068    114.7000     1.400000
#>    Standard deviation   10.44721    56.69952     4.407506
#>    Minimum              26.27789    26.35000    0.3000000
#>    Maximum              78.43943    457.2500     28.00000
#>    25th percentile      42.83231    80.60000    0.8000000
#>    50th percentile      51.00068    114.7000     1.400000
#>    75th percentile      58.24093    151.9000     3.400000
#>
```

1.7.3 Producing a histogram

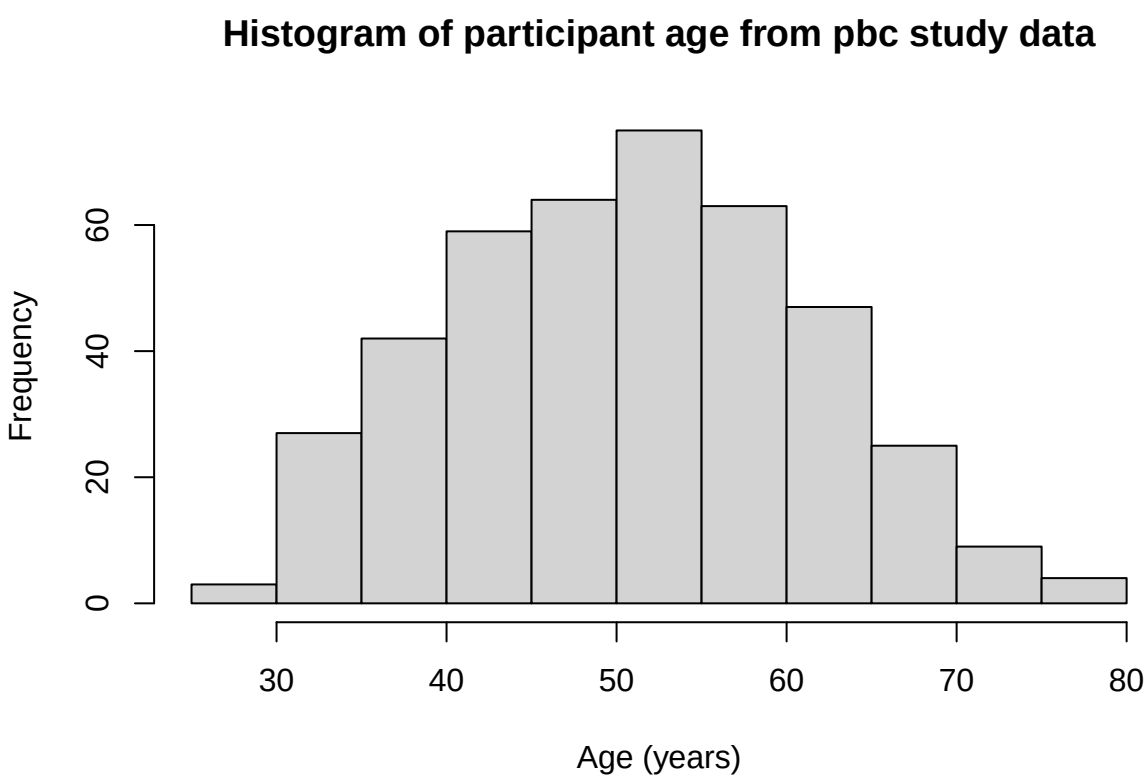
We can use the `hist()` function to produce a histogram, specifying the dataframe to use and the variable to be plotted as `dataframe$variable`:

```
hist(pbc$age)
```



The histogram function does a remakarbly good job of choosing cutpoints and binwidths, and these rarely need to be changed. However, the labelling of the histogram should be improved by using `xlab=` and `main=` to assign labels for the x-axis and overall title respectively:

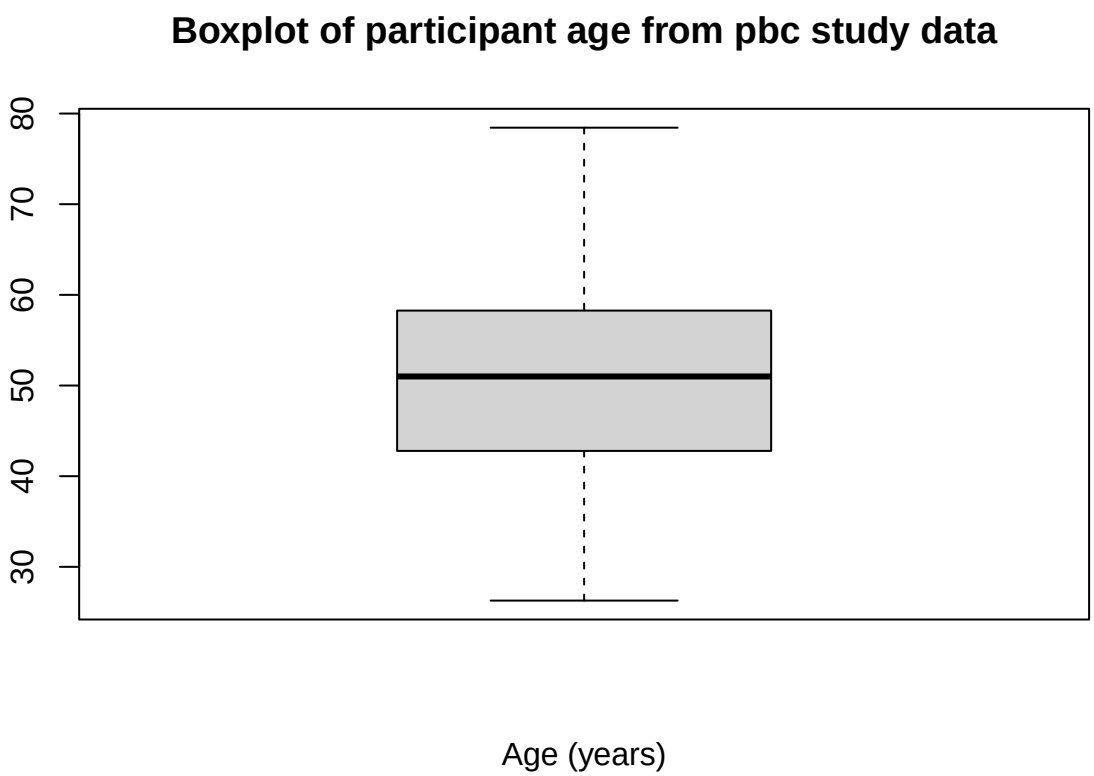
```
hist(pbc$age, xlab="Age (years)", main="Histogram of participant age from pbc study data")
```



1.7.4 Producing a boxplot

The `boxplot` function is used to produce boxplots, again specifying the dataframe to use and the variable to be plotted as `dataframe$variable`. Labels can be applied in the same way as the histogram:

```
boxplot(pbc$age, xlab="Age (years)", main="Boxplot of participant age from pbc study data")
```



1.7.5 Producing a one-way frequency table

We have three categorical variables to summarise in Table 1: sex, stage and vital status. These variables are best summarised using one-way frequency tables.

```
library(summarytools)
#>
#> Attaching package: 'summarytools'
#> The following object is masked from 'package:tibble':
#>
#>     view
#> The following objects are masked from 'package:huxtable':
#>
#>     label, label<-

freq(pbc$sex)
#> Frequencies
#> pbc$sex
#> Type: Numeric
#>
#>                Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
#> ----------- ------ --------- -------------- --------- --------------
```

```
#>           1     44     10.53          10.53     10.53          10.53
#>           2    374     89.47         100.00     89.47         100.00
#>        <NA>      0                               0.00         100.00
#>       Total    418    100.00         100.00    100.00         100.00
```

1.8 Defining categorical variables as factors

You will notice that the table above, in its current form, is uninterpretable as the 1 and 2 categories are not labelled. In this course, all variables including categorical variables tend to be numerically coded. To define a categorical variable as such in R, we define it as a **factor** using the `factor` function:

```
factor(variable=, levels=, labels=)
```

We specify:

- `levels`: the values the categorical variable uses can take
- `labels`: the labels corresponding to each of the levels (*entered in the same order as the levels*)

To define our variable sex as a factor, we use:

```
pbc$sex <- factor(pbc$sex, levels=c(1, 2), labels=c("Male", "Female"))
```

We can confirm the coding by re-running a frequency table:

```
freq(pbc$sex)
#> Frequencies
#> pbc$sex
#> Type: Factor
#>
#>                Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
#> ------------ ------ --------- -------------- --------- --------------
#>         Male     44     10.53          10.53     10.53          10.53
#>       Female    374     89.47         100.00     89.47         100.00
#>         <NA>      0                               0.00         100.00
#>        Total    418    100.00         100.00    100.00         100.00
```

Task: define Stage and Vital Status as factors, and produce one-way frequency tables.

1.8.1 Copying output from R [UPDATE]

It is important to note that saving data in Stata will not save your output. Stata data and output are completely separate to one another. The easiest way to retain the output of your analyses is to copy the output into a word processor package (e.g. Microsoft Word) before closing Stata. Once Stata is closed, all the output (that is, all your hard work!) is lost.

To copy output from Stata, you can select the output and choose Edit > Copy. This will copy the output as plain text for pasting into a Word document. As this is a table, you can also Copy table or Copy table as HTML. For this course, we recommend that you Copy table as HTML for pasting into Word. Whichever way you do it, you will need to make sure you reformat the table and relabel your header row and column properly for your assignments as described in Module 1. Alternatively, you can copy with the Copy table option for pasting into an Excel worksheet and reformat your table in Excel before pasting into Word.

Copying output from Stata can get a little complicated to explain. We have included a video on Moodle to summarise the different ways output can be copied.

Task: complete Table 1 using the output generated in this exercise. You should decide on whether to present continuous variables by their means or medians, and present the most appropriate measure of spread. Include footnotes to indicate if any variables contain missing observations.

Part 3: Creating other types of graphs

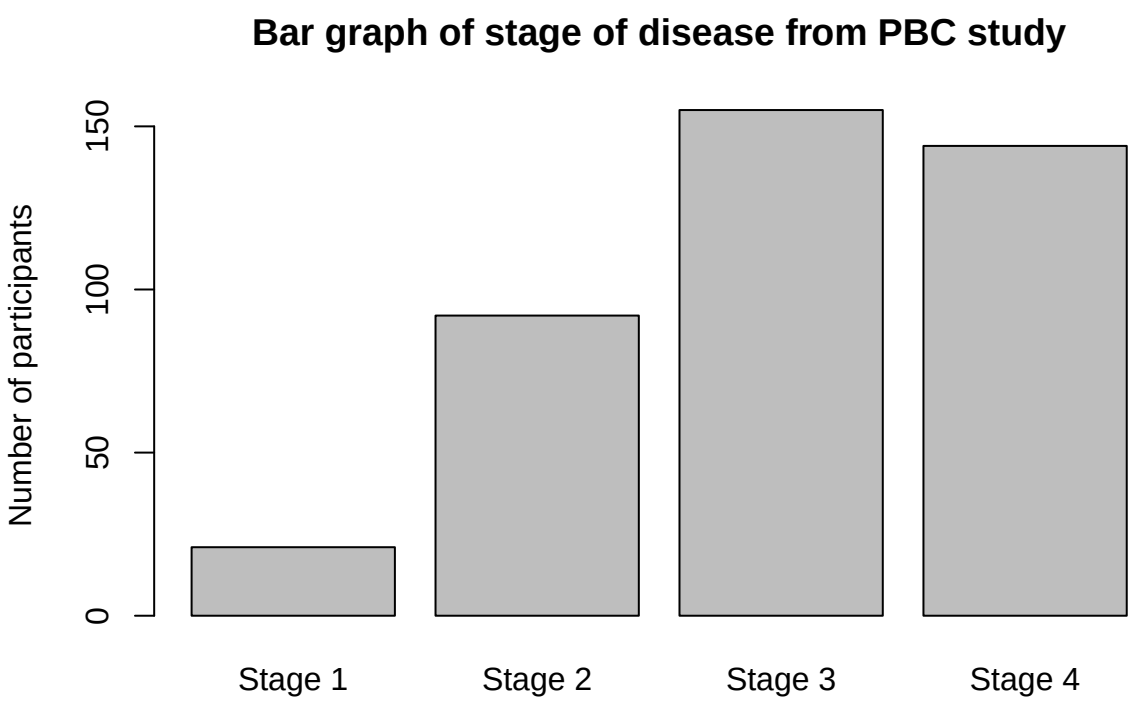
1.8.2 Bar graphs

Here we will create the bar chart shown in Figure 1.1 using the `pbc.dta` dataset. The x-axis of this graph will be the stage of disease, and the y-axis will show the number of participants in each category.

1.8.2.1 Simple bar graph

For most of our bar graphs, we will be plotting frequencies, so we choose **Graph of frequencies within categories**

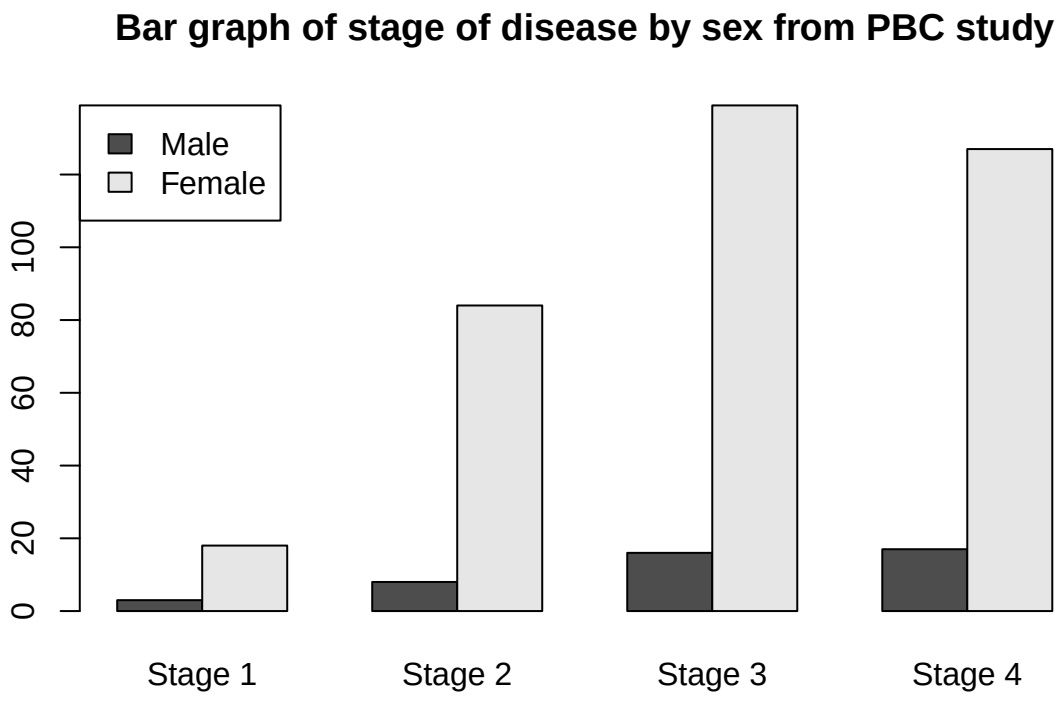
```
# Convert stage into a factpr
pbc$stage <- factor(pbc$stage, levels=c(1,2,3,4), labels=c("Stage 1", "Stage 2", "Stage 3", "Sta

plot(pbc$stage, main="Bar graph of stage of disease from PBC study", ylab="Number of participant
```



1.8.3 Clustered bar graph

To create a clustered bar chart as shown in Figure 1.2:

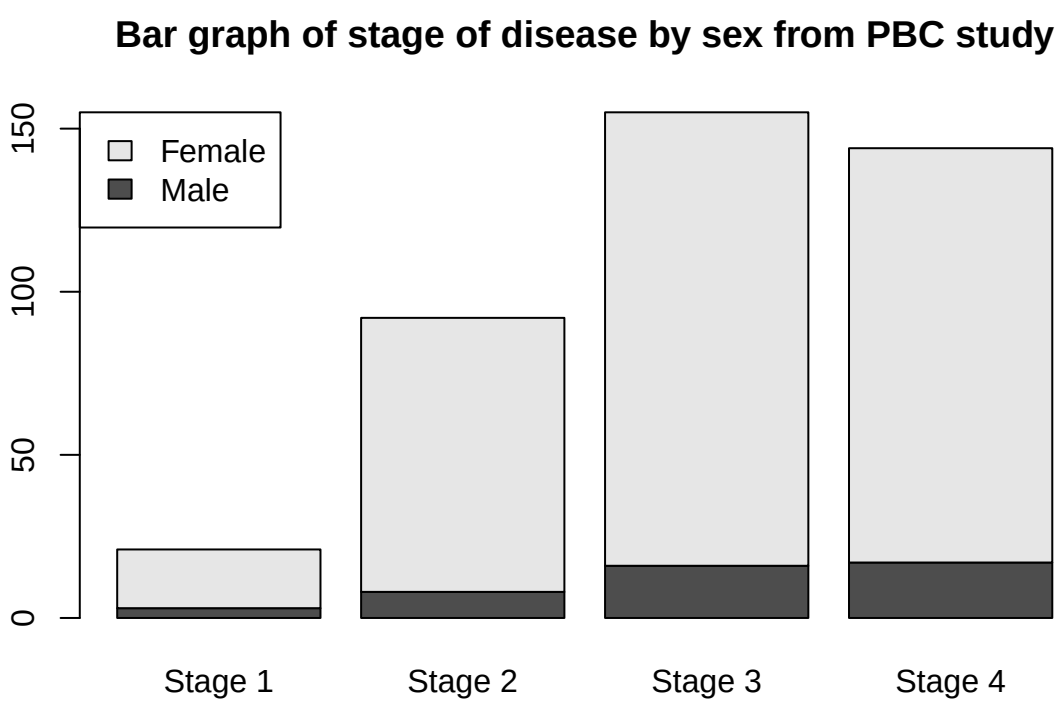
```
counts <- table(pbc$sex, pbc$stage)
barplot(counts, main="Bar graph of stage of disease by sex from PBC study",
        legend = rownames(counts), beside=TRUE, args.legend = list(x = "topleft"))
```



1.8.4 Stacked bar graph

To create a stacked bar chart shown in Figure 1.4, bring up the **Bar chart** dialog box, go to the **Options** tab and tick **Stack bars on y variables**.

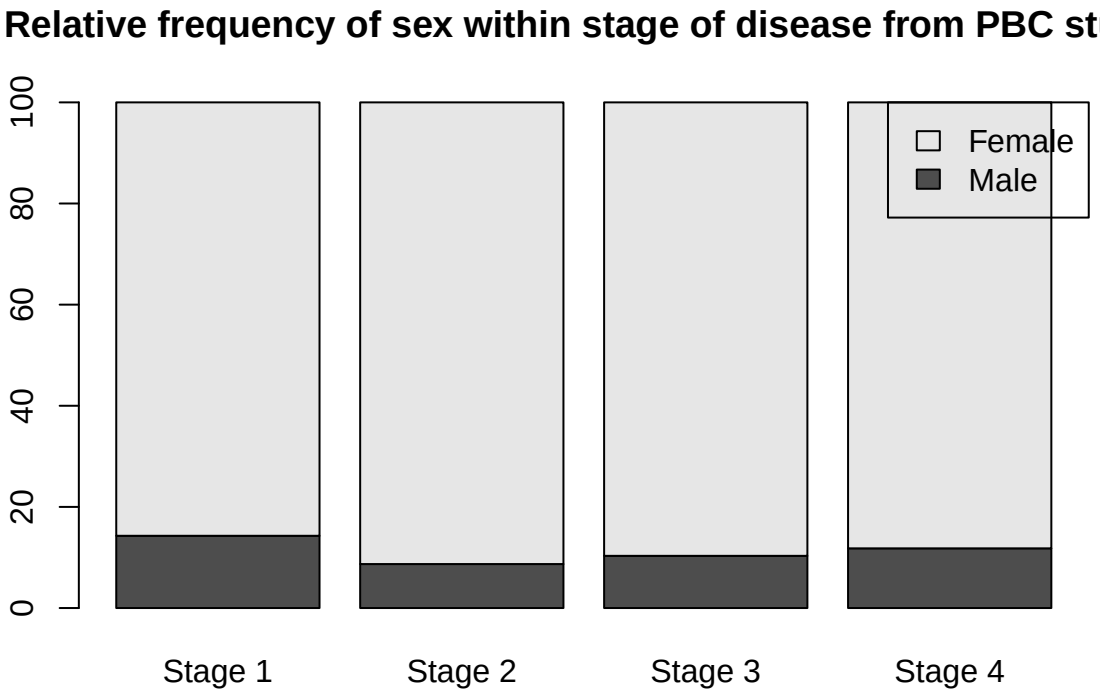
```
barplot(counts, main="Bar graph of stage of disease by sex from PBC study",
        legend = rownames(counts), beside=FALSE, args.legend = list(x = "topleft"))
```



1.8.5 Stacked bar graph of relative frequencies

If one wants to compare the sex distribution across the stage categories, it would be convenient if all the bars have the same height (100%). To generate such a bar chart in Stata, tick **Base bar heights on percentages** in the **Options** tab of the **Bar charts** dialog box. Change the y-axis title in the **Y axis** tab to `Percentage of students in each age group`.

```
percent <- prop.table(counts, margin=2)*100
percent
#>
#>            Stage 1   Stage 2   Stage 3   Stage 4
#>   Male   14.285714  8.695652 10.322581 11.805556
#>   Female 85.714286 91.304348 89.677419 88.194444

barplot(percent, main="Relative frequency of sex within stage of disease from PBC study",
        legend = rownames(counts), beside=FALSE, args.legend = list(x = "topright"))
```



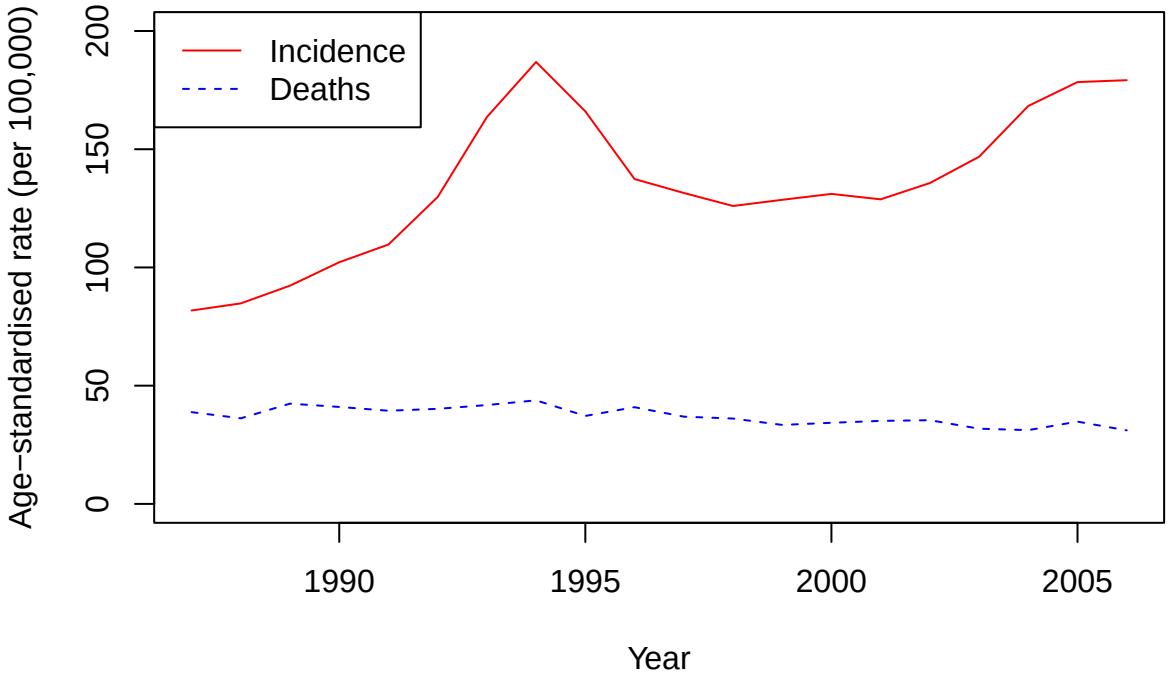
1.8.6 Creating line graphs

To demonstrate the graphing of aggregate data with Stata, we use the data on new cases and deaths from prostate cancer in males in NSW. This data has been entered into Stata as `Example_1.2.dta`.

```
cancer <- read_stata("data/examples/Example_1.2.dta")
summary(cancer)
#>       year          ncases        ndeaths
#>  Min.   :1987   Min.   :1567   Min.   : 645.0
#>  1st Qu.:1992   1st Qu.:2804   1st Qu.: 788.2
#>  Median :1996   Median :3790   Median : 868.0
#>  Mean   :1996   Mean   :3719   Mean   : 855.0
#>  3rd Qu.:2001   3rd Qu.:4403   3rd Qu.: 921.0
#>  Max.   :2006   Max.   :6158   Max.   :1044.0
#>      rcases         rdeaths
```

```
#>  Min.   : 81.8   Min.   :31.10
#>  1st Qu.:121.9   1st Qu.:34.67
#>  Median :131.3   Median :36.55
#>  Mean   :135.4   Mean   :37.09
#>  3rd Qu.:164.2   3rd Qu.:40.38
#>  Max.   :186.9   Max.   :43.80

plot(cancer$year, cancer$rcases, type="l", col = "red", xlab = "Year", ylab = "Age-standardised rate
```


```
# Change scale
plot(cancer$year, cancer$rcases, type="l", col = "red", xlab = "Year", ylab = "Age-standardised rate

# Add a second line
lines(cancer$year, cancer$rdeaths, col = "blue", type = "l", lty = 2)

# Add a legend to the plot
legend("topleft", legend=c("Incidence", "Deaths"),
       col=c("red", "blue"), lty = 1:2)
```


Incidence
Deaths
Age−standardised rate (per 100,000)
0
50
100
150
200
1990
1995
2000
2005
Year

Module 2

Probability and probability distributions: R notes

2.1 Importing data into Stata

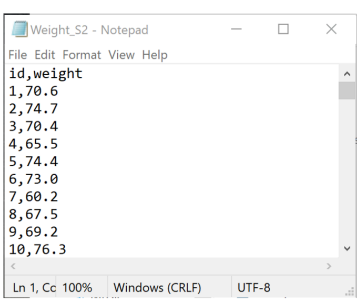
We have described previously how to import data that have been saved as Stata .dta files. It is quite common to have data saved in other file types, such as Microsoft Excel, or plain text files. In this section, we will demonstrate how to import data from other packages into R.

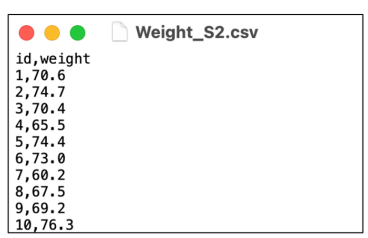
2.1.1 Importing plain text data into Stata

A `csv` file, or a "comma separated variables" file is commonly used to store data. These files have a very simple structure: they are plain text files, where data are separated by commas. csv files have the advantage that, as they are plain text files, they can be opened by a large number of programs (such as Notepad in Windows, TextEdit in MacOS, Microsoft Excel - even Microsoft Word). While they can be opened by Microsoft Excel, they can be opened by many other programs: the csv file can be thought of as the lingua-franca of data.

In this demonstration, we will use data on the weight of 1000 people entered in a csv file called `weight_s2.csv` available on Moodle.

To confirm that the file is readable by any text editor, here are the first ten lines of the file, opened in Notepad on Microsoft Windows, and TextEdit on MacOS.





We can use the `read.csv` function:

```
library(tidyverse)
#> -- Attaching packages ------------------- tidyverse 1.3.1 --
#> v ggplot2 3.3.5     v purrr   0.3.4
#> v tibble  3.1.6     v dplyr   1.0.8
#> v tidyr   1.2.0     v stringr 1.4.0
```

```
#> v readr   2.1.2     v forcats 0.5.1
#> -- Conflicts ------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(jmv)

weights <- read.csv("data/examples/Weight_s2.csv")
```

Here, the `read.csv` function has the default that the first row of the dataset contains the variable names. If your data do not have column names, you can use `header=FALSE` in the function.

Note: there is an alternative function `read_csv` which is part of the `readr` package (a component of the `tidyverse`). Some would argue that the `read_csv` function is more appropriate to use because of an issue known as `strings.as.factors`. The `strings.as.factors` default was removed in R Version 4.0.0, so it is less important which of the two functions you use to import a `.csv` file. More information about this issue can be found here and here.

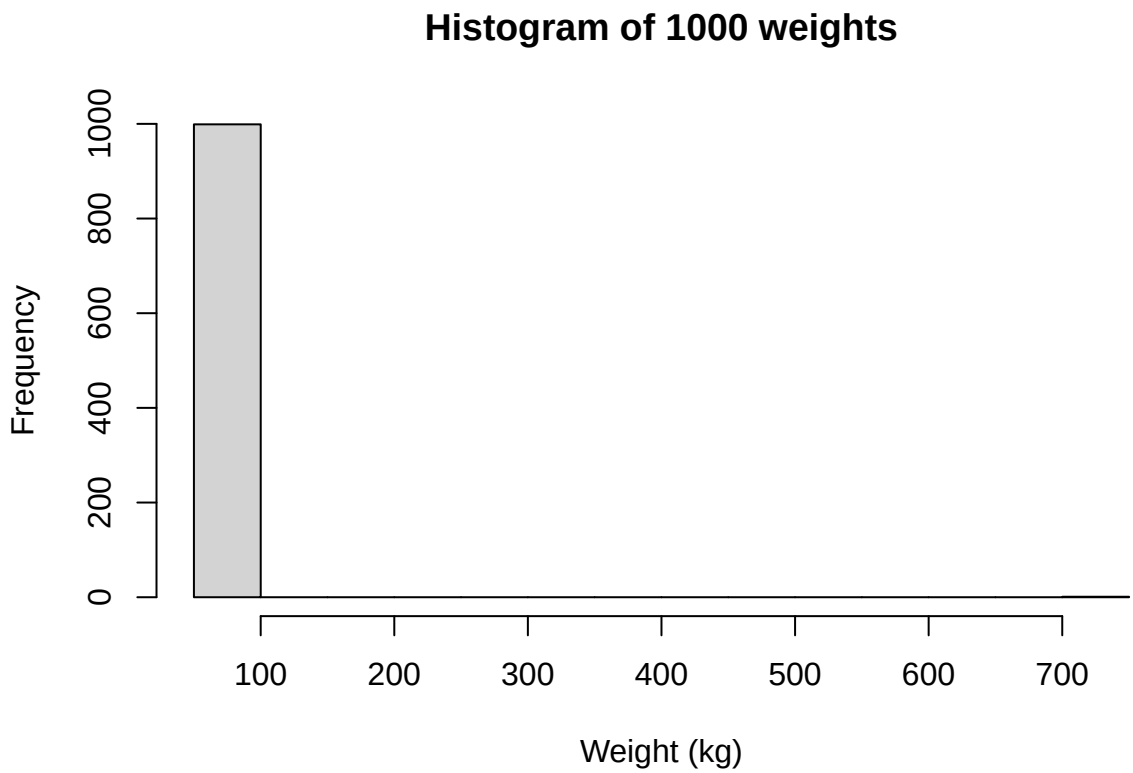
2.2 Checking your data for errors in Stata

Before you start describing and analysing your data, it is important to make sure that no errors have been made during the data entry process. Basically, you are looking for values that are outside the range of possible or plausible values for that variable.

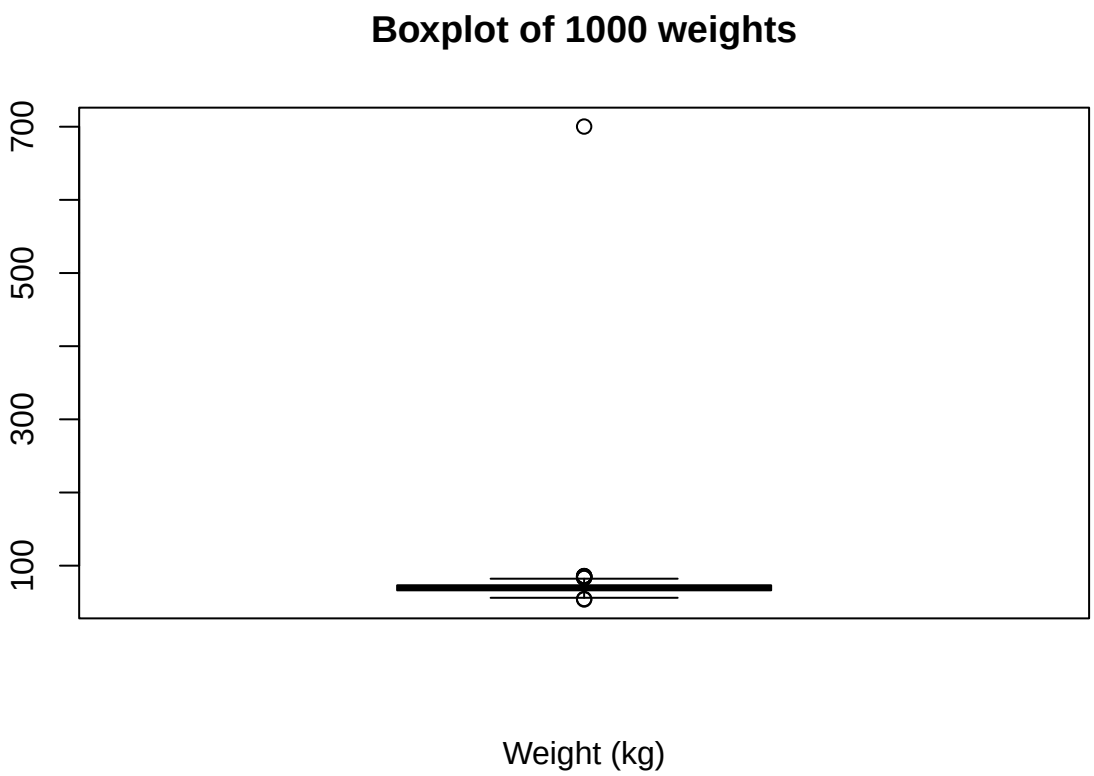
If an error is found, the best method for correcting the error is to go back to the original data e.g. the hard copy questionnaire, to obtain the original value, entering the correct value into R If the original data is not available or the original data is also incorrect, the erroneous value is often excluded from the dataset.

For continuous variables, the easiest methods are to examine a boxplot and histogram. For example, a boxplot and histogram for the weight variable we just imported appear as:

```
hist(weights$weight, xlab="Weight (kg)", main="Histogram of 1000 weights")
```




```
boxplot(weights$weight, xlab="Weight (kg)", main="Boxplot of 1000 weights")
```



There is a clear outlying point shown in the boxplot. Although not obvious, the same point is shown in the histogram as a bar around 700 with a very short height.

We can identify any outlying observations in the dataset using the `filter` function, loaded with the `tidyverse`. You will need to decide if these values are a data entry error or are biologically plausible. If an extreme value or "outlier", is biologically plausible, it should be included in all analyses.

For example, to list any observations from the `weights` dataset with a weight larger than 200:

```
dplyr::filter(weights, weight>200)
#>   id weight
#> 1 58  700.2
```

We see that there is a very high value of 700.2kg. A value as high as 700kg is likely to be a data entry error (e.g. error in entering an extra zero) and is not a plausible weight value. Here, **you should check your original data**.

You might find that the original weight was recorded in medical records as 70.2kg. You can change this in R by writing code.

Note: many statistical packages, including Stata, will allow you to view a spreadsheet version of your data and edit values in that spreadsheet. This is not best practice, as corrected observations may revert to their original values depending on whether the edited data have been saved or not. By using code-based recoding, the changes will be reproduced the next time the code is run.

We will use an `ifelse` statement to recode the incorrect weight of 700.2kg into 70.2kg. The form of the `ifelse` statement is as follows:

```
ifelse(test, value_if_true, value_if_false)
```

Our code will create a new column (called `weight_clean`) in the `weights` dataframe. We will test whether `weight` is equal to 700.2; if this is true, we will assign `weight_clean` to be 70.2, otherwise `weight_clean` will equal the value of `weight`. Putting it all together:

```
weights$weight_clean = ifelse(weights$weight==700.2, 70.2, weights$weight)
```

Note: if an extreme value lies within the range of biological plausibility it should not be removed from analysis.

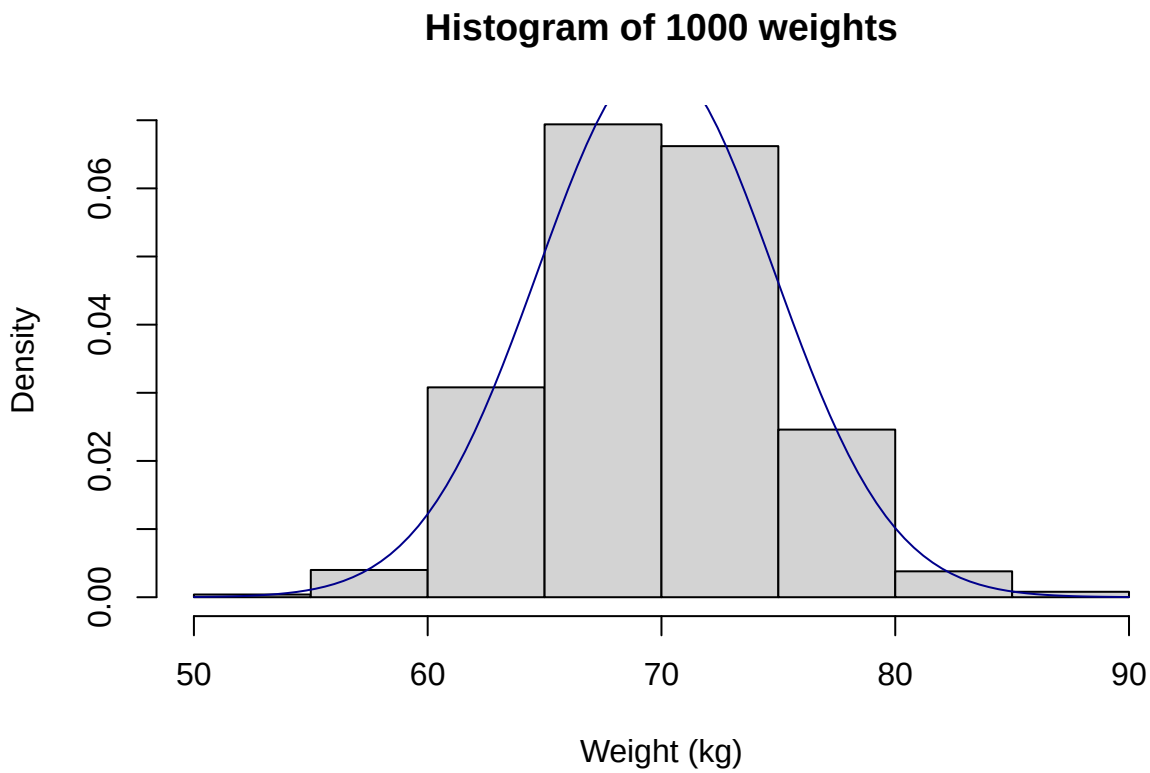
Once you have checked your data for errors, you are ready to start analysing your data.

2.3 Overlaying a Normal curve on a histogram

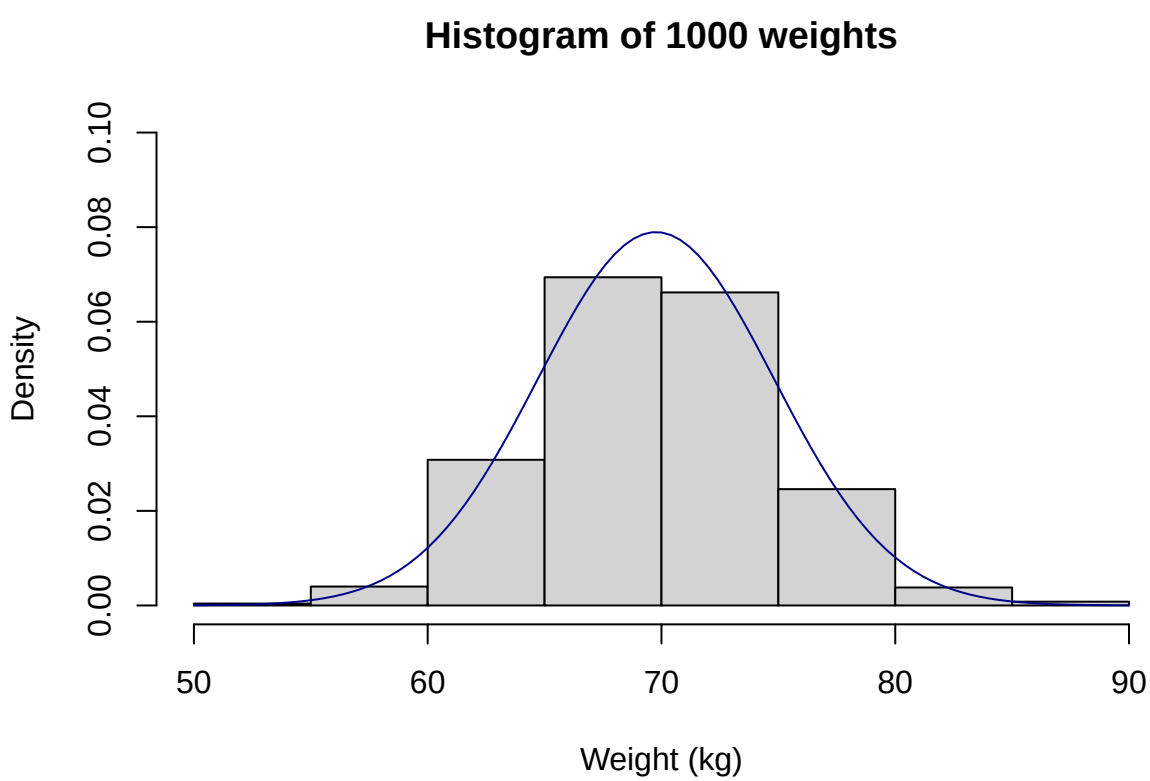
It can be useful to produce a histogram with an overlayed Normal curve to assess whether our sample appears approximately Normally distributed.

2.3.1 Base graphics

```
hist(weights$weight_clean, xlab="Weight (kg)", main="Histogram of 1000 weights", probability = TRUE)
curve(dnorm(x, mean=mean(weights$weight_clean), sd=sd(weights$weight_clean)), col="darkblue", add=TRU
```

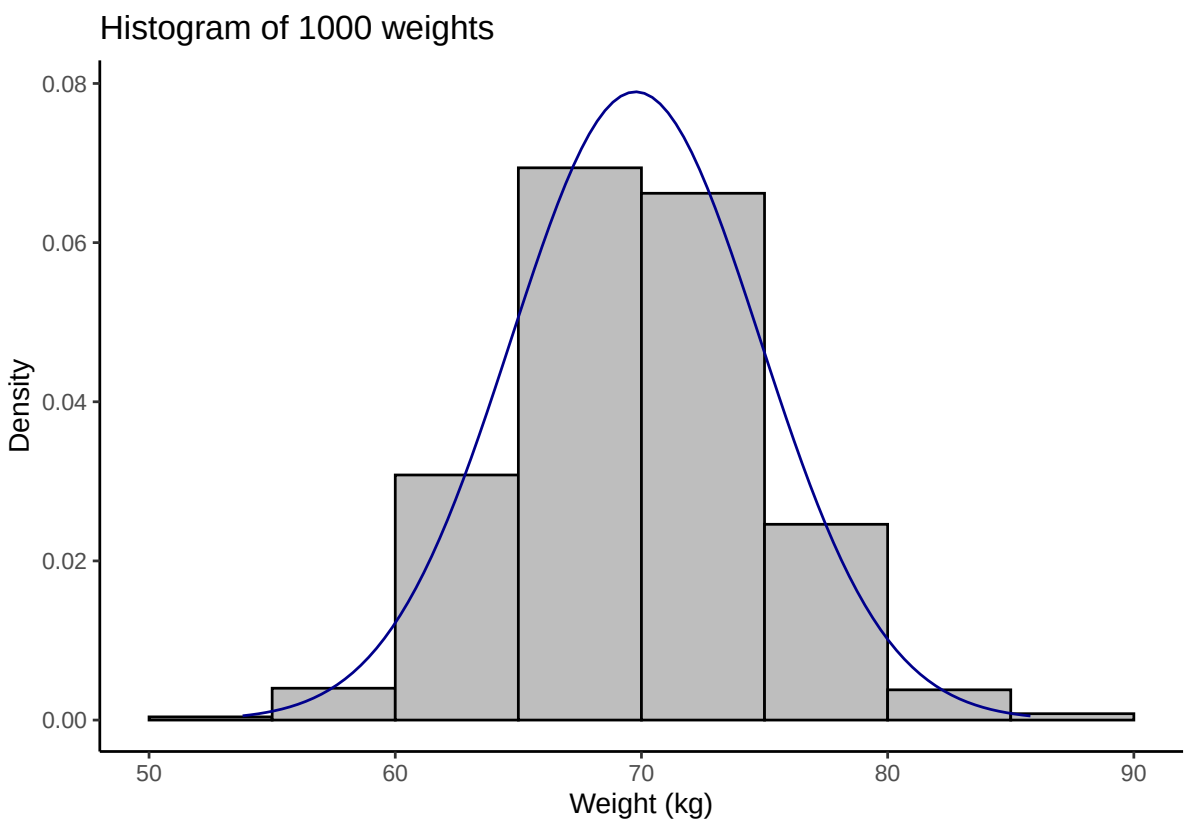



```
hist(weights$weight_clean, xlab="Weight (kg)", main="Histogram of 1000 weights", probability = T
curve(dnorm(x, mean=mean(weights$weight_clean), sd=sd(weights$weight_clean)), col="darkblue", ad
```



2.4 ggplot2

```
ggplot(weights, aes(x=weight_clean)) + geom_histogram(aes(y=..density..), breaks=seq(50, 90, 5), colo
  stat_function(fun = dnorm,
                args = list(mean = mean(weights$weight_clean),
                            sd = sd(weights$weight_clean)),
                colour="darkblue") +
  theme_classic() +
  labs(title="Histogram of 1000 weights", x="Weight (kg)", y="Density")
```



2.5 Descriptive statistics for checking normality

All the descriptive statistics including Skewness and Kurtosis discussed in this module can be obtained using the `descriptives` function from the `jmv` package. In particular, skewness and kurtosis can be requested over and above the default statistics (`skew=TRUE, kurt=TRUE`):

```
descriptives(data=weights, vars=weight_clean, skew=TRUE, kurt=TRUE)
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                            weight_clean
#>
#>    N                               1000
#>    Missing                            0
#>    Mean                        69.76450
```

```
#>    Median                     69.80000
#>    Standard deviation         5.052676
#>    Minimum                    53.80000
#>    Maximum                    85.80000
#>    Skewness                 0.07360659
#>    Std. error skewness      0.07734382
#>    Kurtosis                 0.05418774
#>    Std. error kurtosis       0.1545343
#>
```

2.6 Importing Excel data into Stata

Another common type of file that data are stored in is a Microsoft Excel file (.xls or .xlsx). In this demonstration, we will import a selection of records from a large health survey, stored in the file `health-survey.xlsx`.

The health survey data contains 1140 records, comprising:

- sex: 1 = respondent identifies as male; 2 = respondent identifies as female
- height: height in meters
- weight: weight in kilograms

To import data from Microsoft Excel, we can use the `read_excel()` function in the `readxl` package.

```
library(readxl)

survey <- read_excel("data/examples/health-survey.xlsx")
summary(survey)
#>       sex           height          weight
#>  Min.   :1.00   Min.   :1.220   Min.   : 22.70
#>  1st Qu.:1.00   1st Qu.:1.630   1st Qu.: 68.00
#>  Median :2.00   Median :1.700   Median : 79.40
#>  Mean   :1.55   Mean   :1.698   Mean   : 81.19
#>  3rd Qu.:2.00   3rd Qu.:1.780   3rd Qu.: 90.70
#>  Max.   :2.00   Max.   :2.010   Max.   :213.20
```

We can see that sex has been entered as a numeric variable. We should transform it into a factor so that we can assign labels to each category:

```
survey$sex <- factor(survey$sex, level=c(1,2), labels=c("Male", "Female"))

summary(survey$sex)
#>   Male Female
#>    513    627
```

We also note that height looks like it has been entered as meters, and weight as kilograms.

2.7 Generating new variables

Our health survey data contains information on height and weight. We often summarise body size using BMI: body mass index which is calculated as: $\frac{\text{weight (kg)}}{(\text{height (m)})^2}$

We can create a new column in our dataframe in many ways, and we will present two alternatives.

2.7.1 Base R

A new column can be generated using the following approach:

```
dataframe$new_column = <formula>
```

For example:

```
survey$bmi = survey$weight / (survey$height^2)
```

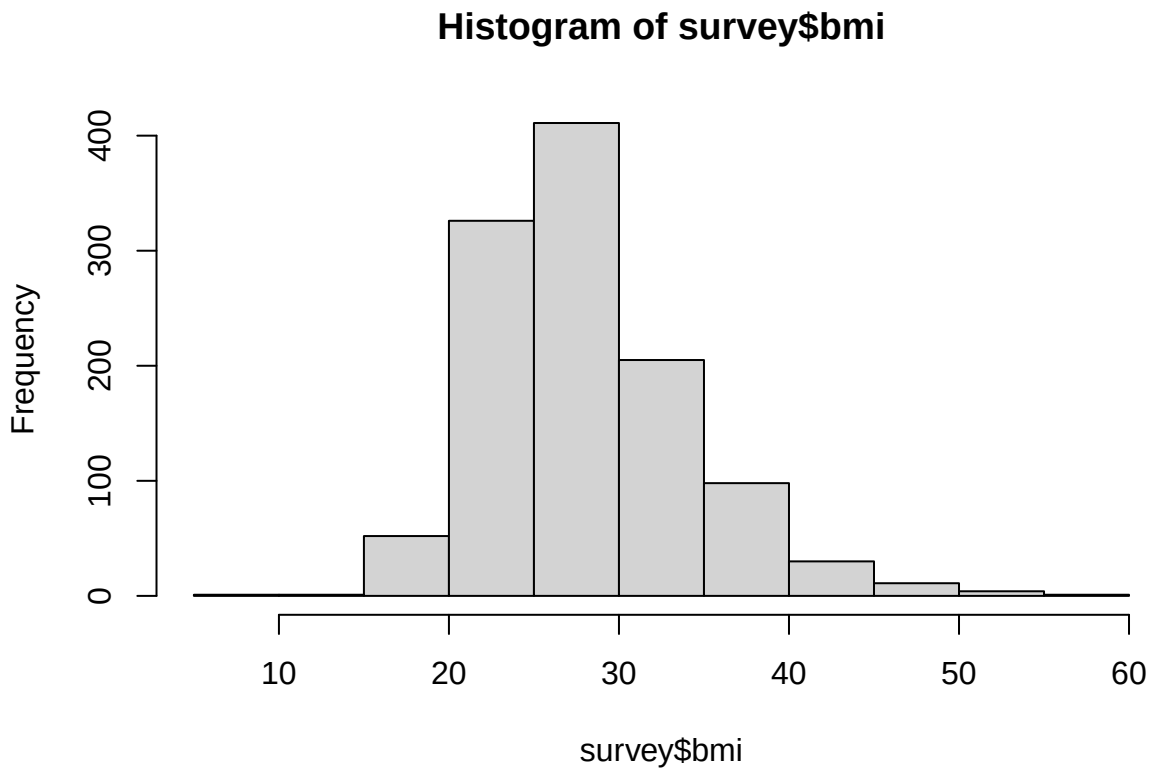
2.7.2 tidyverse

Using the tidyverse approach, we use the `mutate` command to change (or create) a column of data within the dataframe:

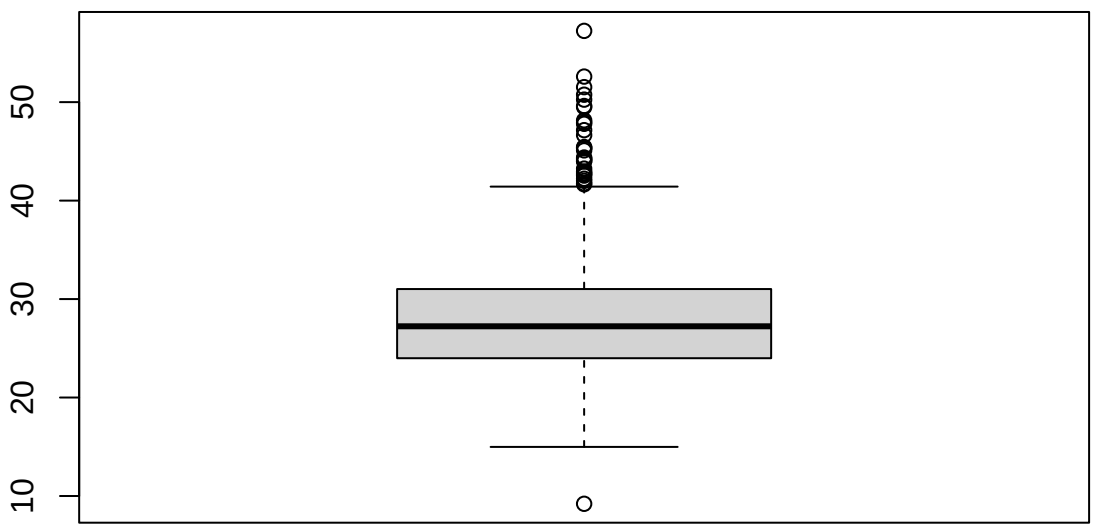
```
survey <- survey %>%
  mutate(bmi = weight / (height^2))
```

We should check the construction of the new variable by examining some records, and examining a histogram and boxplot:

```
head(survey)
#> # A tibble: 6 x 4
#>   sex    height weight   bmi
#>   <fct>   <dbl>  <dbl> <dbl>
#> 1 Male     1.63   81.7  30.8
#> 2 Male     1.63   68    25.6
#> 3 Male     1.85   97.1  28.4
#> 4 Male     1.78   89.8  28.3
#> 5 Male     1.73   70.3  23.5
#> 6 Female   1.57   85.7  34.8
tail(survey)
#> # A tibble: 6 x 4
#>   sex    height weight   bmi
#>   <fct>   <dbl>  <dbl> <dbl>
#> 1 Female   1.65   95.7  35.2
#> 2 Male     1.8    79.4  24.5
#> 3 Female   1.73   83    27.7
#> 4 Female   1.57   61.2  24.8
#> 5 Male     1.7    73    25.3
#> 6 Female   1.55   91.2  38.0

hist(survey$bmi)
```




```
boxplot(survey$bmi)
```



In the general population, BMI ranges between about 15 to 30. It appears that BMI has been correctly generated in this example. We should investigate the very low and some of the very high values of BMI, but this will be left for another time.

2.8 Summarising data by another variable

We will often want to calculate the same summary statistics by another variable. For example, we might want to calculate summary statistics for BMI for males and females separately. We can do this in in the `descriptives` function by defining sex as a `splitBy` variable:

```
descriptives(data=survey, vars=bmi, splitBy = sex)
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                          sex       bmi
#>
#>    N                     Male        513
#>                          Female      627
#>    Missing               Male          0
#>                          Female        0
#>    Mean                  Male      28.29561
#>                          Female    27.81434
#>    Median                Male      27.39592
#>                          Female    26.66667
#>    Standard deviation    Male      5.204975
#>                          Female    6.380523
#>    Minimum               Male      16.47519
#>                          Female    9.209299
#>    Maximum               Male      57.23644
#>                          Female    52.59516
#>
```

[PLOTS BY VARIABLES]

2.9 Recoding data

One task that is common in statistical computing is to recode variables. For example, we might want to group some categories of a categorical variable, or to present a continuous variable in a categorical way.

In this example, we can recode BMI into the following categories as suggested by the World Health Organisation [footnote]:

- Underweight: BMI < 18.5
- Normal weight: 18.5 $\leq$ BMI < 25
- Pre-obesity: 25 $\leq$ BMI < 30
- Obesity Class I: 30 $\leq$ BMI < 35
- Obesity Class II: 35 $\leq$ BMI < 40
- Obesity Class III: BMI $\geq$ 40

The quickest way to recode a continuous variable into categories is to use the `cut` command which takes a continuous variable, and "cuts" it into groups based on the specified "cutpoints":

```
survey$bmi_cat <- cut(survey$bmi, c(0, 18.5, 25, 30, 35, 40, 100))
```

Notice that lower (BMI=0) and upper (BMI=100) bounds have been specified, as both a lower and upper limit must be defined for each group.

If we examine the new `bmi_cat` variable:

```
summary(survey$bmi_cat)
#>  (0,18.5] (18.5,25]   (25,30]   (30,35]   (35,40]  (40,100]
#>        18       362       411       205        97        47
```

we see that each group has been labelled (a, b]. This notation is equivalent to: greater than a, and less than or equal to b. The `cut` function excludes the lower limit, but includes the upper limit. Our BMI ranges have been defined to include the lower limit, and exclude the upper limit (for example, greater than or equal to 30 and less than 35).

We can specify this recoding using the `right=FALSE` option:

```
survey$bmi_cat <- cut(survey$bmi, c(0, 18.5, 25, 30, 35, 40, 100), right=FALSE)
summary(survey$bmi_cat)
#>  [0,18.5) [18.5,25)   [25,30)   [30,35)   [35,40)  [40,100)
#>        18       362       411       201       101        47
```

More complex recoding can be done using the `case_when` command in the `dplyr` package.

[INCLUDE???]

2.10 Computing binomial probabilities using R

There are two R functions that we can use to calculate probabilities based on the binomial distribution: `dbinom` and `pbinom`:

- `dbinom(x, size, prob)` gives the probability of obtaining `x` successes from `size` trials when the probability of a success on one trial is `prob`;
- `pbinom(q, size, prob)` gives the probability of obtaining `q` **or fewer** successes from `size` trials when the probability of a success on one trial is `prob`;
- `pbinom(q, size, prob, lower.tail=FALSE)` gives the probability of obtaining **more than** `q`successes from `size` trials when the probability of a success on one trial is `prob`.

To do the computation for part (a) in Worked Example 2.1, we will use the `dbinom` function with:

- *x* is the number of successes, here, the number of smokers (i.e. k=3);
- *size* is the number of trials (i.e. n=6);
- and *prob* is probability of drawing a smoker from the population, which is 19.8% (i.e. p=0.198).

Replace each of these with the appropriate number into the formula:

```
dbinom(x=3, size=6, prob=0.198)
#> [1] 0.08008454
```

To calculate the upper tail of probability in part (b), we use the `pbinom(lower.tail=FALSE)` function. Note that the `pbinom(lower.tail=FALSE)` function **does not include** `q`, so to obtain 4 or more successes, we need to enter `q=3`:

```
pbinom(q=3, size=6, prob=0.198, lower.tail=FALSE)
#> [1] 0.01635325
```

For the lower tail for part (c), we use the `pbinom` function:

```
pbinom(q=2, size=6, prob=0.198)
#> [1] 0.9035622
```

2.11 Computing probabilities from a Normal distribution

We can use the `pnorm` function to calculate probabilities from a Normal distribution:

- `pnorm(q, mean, sd)` calculates the probability of observing a value of `q` or less, from a Normal distribution with a mean of `mean` and a standard deviation of `sd`. Note that if `mean` and `sd` are not entered, they are assumed to be 0 and 1 respectively (i.e. a standard normal distribution.)
- `pnorm(q, mean, sd, lower.tail=FALSE)` calculates the probability of observing a value of `q` or more, from a Normal distribution with a mean of `mean` and a standard deviation of `sd`.

To obtain the probability of obtaining 0.5 or greater from a standard normal distribution:

```
pnorm(0.5, lower.tail=FALSE)
#> [1] 0.3085375
```

To calculate the worked example: Assume that the mean diastolic blood pressure for men is 77.9 mmHg, with a standard deviation of 11. What is the probability that a man selected at random will have high blood pressure (i.e. diastolic blood pressure ge 90)?

```
pnorm(90, mean=77.9, sd=11, lower.tail=FALSE)
#> [1] 0.1356661
```

Module 3

Precision: R notes

3.1 Calculating a 95% confidence interval of a mean

3.1.1 Individual data

To demonstrate the computation of the 95% confidence interval of a mean we have used data from `Example_1.3.dta` which contains the weights of 30 students:

```
library(haven)
library(labelled)
library(jmv)

students <- unlabelled(read_dta("data/examples/Example_1.3.dta"))

summary(students)
#>      weight         gender
#>  Min.   :60.00   Male  :16
#>  1st Qu.:67.50   Female:14
#>  Median :70.00
#>  Mean   :70.00
#>  3rd Qu.:74.38
#>  Max.   :80.00
```

The mean and its 95% confidence interval can be obtained many ways in R. One way is to use the `descriptives` function in the `jmv` package. By default, `descriptives` does not provide a confidence interval, but we can request it by specifying `ci=TRUE`:

```
descriptives(data=students, vars=weight, ci=TRUE)
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                                weight
#>
#>    N                                30
#>    Missing                           0
#>    Mean                       70.00000
#>    95% CI mean lower bound    68.19545
#>    95% CI mean upper bound    71.80455
```

```
#>    Median                      70.00000
#>    Standard deviation          5.042919
#>    Minimum                     60.00000
#>    Maximum                     80.00000
#>
```

3.1.2 Summarised data

For Worked Example 3.2 where we are given the sample mean, sample standard deviation and sample size. R does not have a built-in function to calculate a confidence interval from summarised data, but we can write our own.

Note: writing your own functions is beyond the scope of this course. You should copy and paste the code provided to do this.

```
### Copy this section
ci_mean <- function(n, mean, sd, width=0.95, digits=3){
  lcl <- mean - qt(p=(1 - (1-width)/2), df=n-1) * sd/sqrt(n)
  ucl <- mean + qt(p=(1 - (1-width)/2), df=n-1) * sd/sqrt(n)

  print(paste0(width*100, "%", " CI: ", format(round(lcl, digits=digits), nsmall = digits),
               " to ", format(round(ucl, digits=digits), nsmall = digits) ))

}
### Copy this section

ci_mean(n=30, mean=70, sd=6, width=0.95)
#> [1] "95% CI: 67.760 to 72.240"
ci_mean(n=30, mean=70, sd=6, width=0.99)
#> [1] "99% CI: 66.981 to 73.019"
```

Module 4

Hypothesis testing

4.1 One sample t-test

We will use data from `Example_4.1.dta` to demonstrate how a one-sample t-test is conducted in R.

```
library(haven)
library(labelled)

bloodpressure <- unlabelled(read_dta("data/examples/Example_4.1.dta"))

summary(bloodpressure)
#>       dbp
#>  Min.   : 24.00
#>  1st Qu.: 64.00
#>  Median : 72.00
#>  Mean   : 72.41
#>  3rd Qu.: 80.00
#>  Max.   :122.00
#>  NA's   :35
```

To test whether the mean diastolic blood pressure of the population from which the sample was drawn is equal to 71, we can use the t.test command:

```
t.test(bloodpressure$dbp, mu=71)
#>
#>  One Sample t-test
#>
#> data:  bloodpressure$dbp
#> t = 3.0725, df = 732, p-value = 0.002202
#> alternative hypothesis: true mean is not equal to 71
#> 95 percent confidence interval:
#>  71.50732 73.30305
#> sample estimates:
#> mean of x
#>  72.40518
```

The output gives a test statistic, degrees of freedom and a P values from the two-sided test. The mean of the sample is provided, as well as the 95% confidence interval.

Module 5

Comparing two means

5.1 Checking data for the independent samples t-test

5.1.1 Producing histograms and boxplots by a second variable

We can create histograms and boxplots separated by a second variable in (at least) two ways: using Base R or `ggplot2` graphics. We will demonstrate using the birthweight data in `Example_5.1.dta`.

```
library(haven)
library(labelled)
library(ggplot2)
library(jmv)

bwt <- unlabelled(read_dta("data/examples/Example_5.1.dta"))

summary(bwt)
#>     gender    birthweight
#>  Female:56   Min.   :2.750
#>  Male  :44   1st Qu.:3.257
#>              Median :3.450
#>              Mean   :3.514
#>              3rd Qu.:3.772
#>              Max.   :4.250
summary(bwt$gender)
#> Female   Male
#>     56     44
```

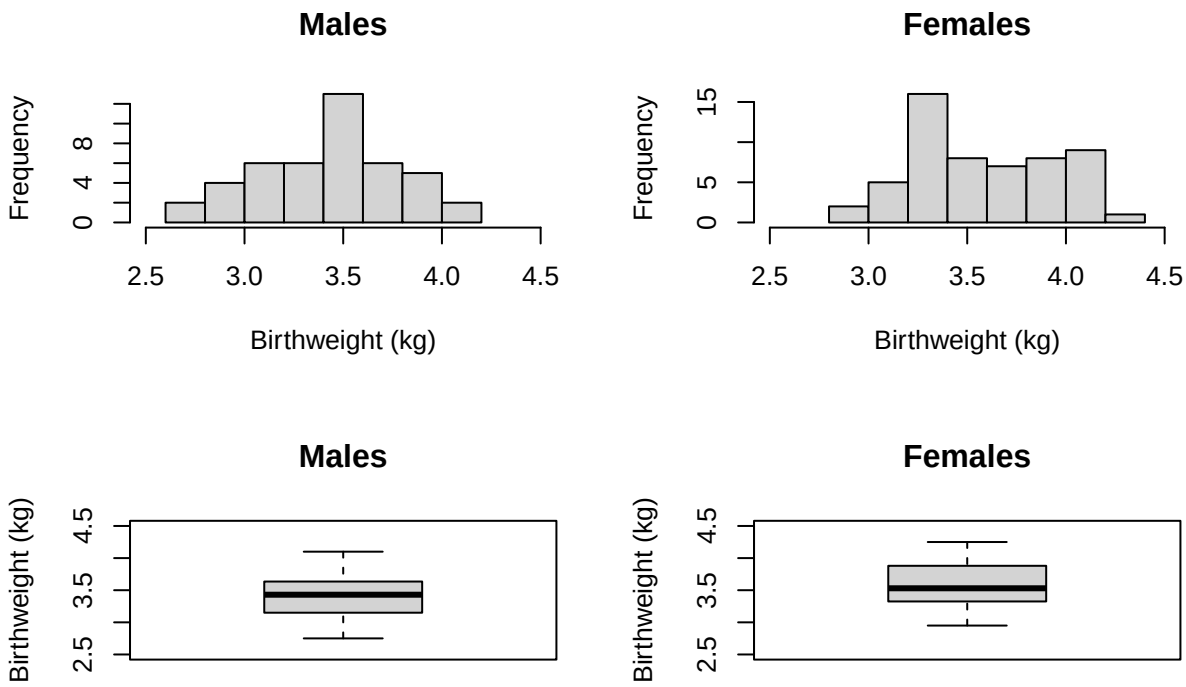
To use Base R graphics, we can create subsets of the birthweight data, subsetted for males and females separately. Note here that `gender` is a factor, so we need to select based on the factor labels, not the underlying numeric code.

```
bwt_m <- subset(bwt, bwt$gender=="Male")
bwt_f <- subset(bwt, bwt$gender=="Female")
```

We can now create hisotgrams and boxplots for males and females separately, in the usual way. To place the graphs next to each other in a single figure, we can use the `par` function. The `par` function sets the graphics parameters. Essentially, we want to tell R to split a plot window into a matrix with *nr* rows and *nc* columns, and we can decide to fill the cells by rows (`mfrow`) or columns (`mfcols`). For example, to plot four figures in a single plot, filled by rows, we use `par(mfrow=c(2,2))`.

When we are done plotting multiple graphs, we can reset the plot window by submitting `par(mfrow=c(1,1))`.

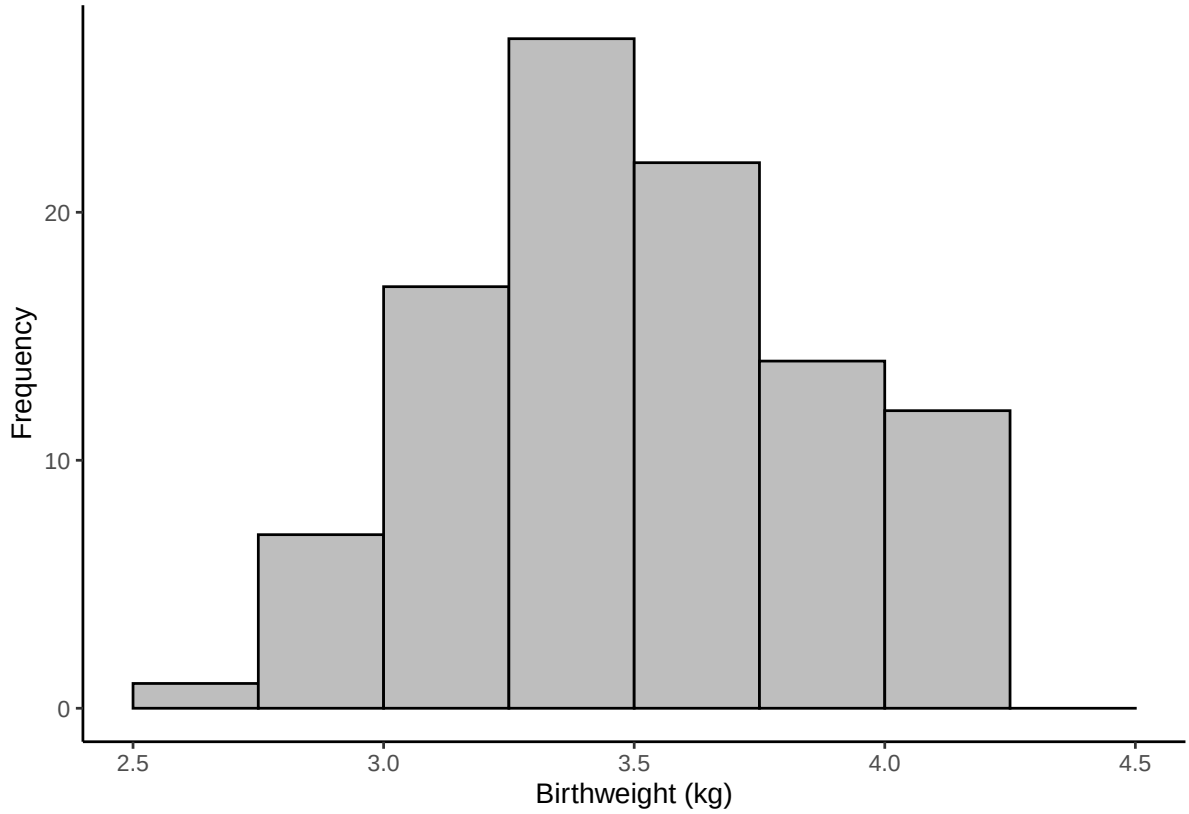
```
par(mfrow=c(2,2))
hist(bwt_m$birthweight, xlim=c(2.5, 4.5), xlab="Birthweight (kg)", main="Males")
hist(bwt_f$birthweight, xlim=c(2.5, 4.5), xlab="Birthweight (kg)", main="Females")

boxplot(bwt_m$birthweight, ylim=c(2.5, 4.5), ylab="Birthweight (kg)", main="Males")
boxplot(bwt_f$birthweight, ylim=c(2.5, 4.5), ylab="Birthweight (kg)", main="Females")
```



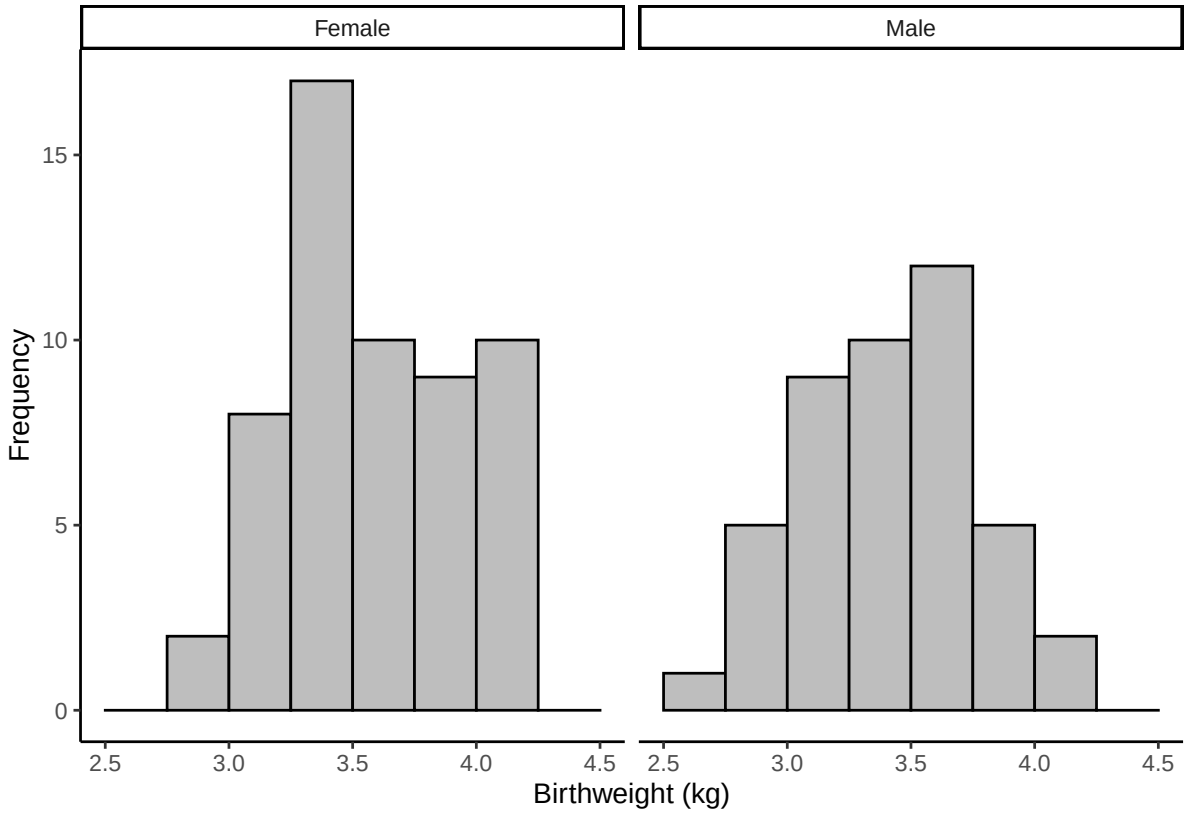

```
par(mfrow=c(1,1))
```

To produce separate histograms in `ggplot2`, we use the `facet_wrap` function to create a grid of plots. We can define the variable(s) to be plotted by in the `vars()`, and optionally, the number of rows (`nrow=`) and number of columns (`ncol=`).

```
# Overall histogram of birthweight
ggplot(bwt, aes(x=birthweight)) +
  geom_histogram(breaks=seq(2.5, 4.5, 0.25), colour="black", fill="grey") +
  labs(x="Birthweight (kg)", y="Frequency") +
  theme_classic()
```

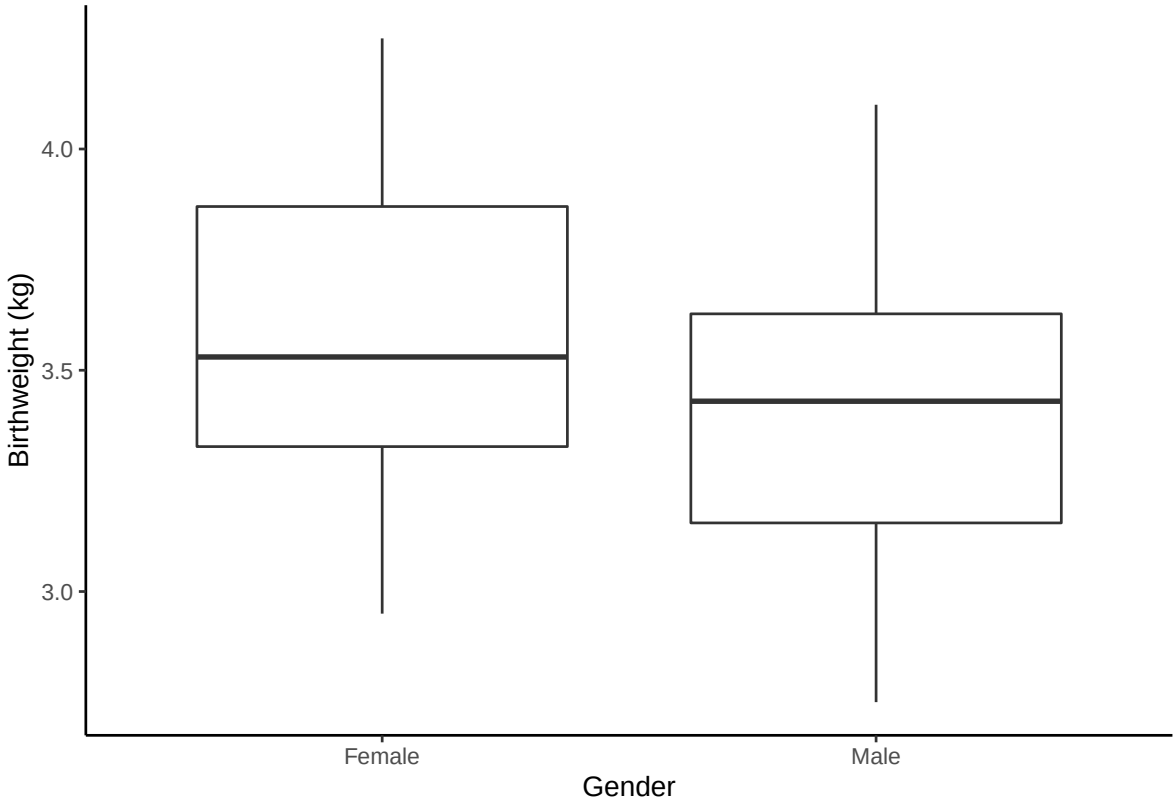



```
# Histogram by gender
ggplot(bwt, aes(x=birthweight)) +
  geom_histogram(breaks=seq(2.5, 4.5, 0.25), colour="black", fill="grey") +
  facet_wrap(vars(gender), nrow=1, ncol=2) +
  labs(x="Birthweight (kg)", y="Frequency") +
  theme_classic()
```



While it is possible to use `facet_wrap` to produce separate boxplots, we can use the fact that the boxplot allows an `x` variable to be assigned to the ggplot aesthetic. By defining `birthweight` as the `y` variable and `gender` as the `x` variable, we can produce two boxplots in the same figure:

```
ggplot(bwt, aes(x=gender, y=birthweight)) +
  geom_boxplot() +
  labs(y="Birthweight (kg)", x="Gender") +
  theme_classic()
```



5.1.2 Producing split summary statistics

The `descriptives` function within the `jmv` function allows summary statistics to be calculated within subgroups using the `splitBy` argument:

```
descriptives(data=bwt, vars=birthweight, splitBy=gender)
#>
#>  DESCRIPTIVES
#>
#>  Descriptives
#>
#>                          gender    birthweight
#>
#>    N                     Female             56
#>                          Male               44
#>    Missing               Female              0
#>                          Male                0
#>    Mean                  Female       3.587411
#>                          Male         3.421364
#>    Median                Female       3.530000
#>                          Male         3.430000
#>    Standard deviation    Female      0.3629788
#>                          Male        0.3536165
#>    Minimum               Female       2.950000
#>                          Male         2.750000
#>    Maximum               Female       4.250000
#>                          Male         4.100000
#>
```

5.2 Independent samples t-test

```r
ttestIS(data=bwt, vars=birthweight, group=gender)
```

```
INDEPENDENT SAMPLES T-TEST

Independent Samples T-Test
─────────────────────────────────────────────────────────────────────────── Statistic df p
─────────────────────────────────────────────────────────────────────────── birthweight Student's t
2.296556 98.00000 0.0237731
───────────────────────────────────────────────────────────────────────────
```

```r
ttestIS(data=bwt, vars=birthweight, group=gender, meanDiff=TRUE, ci=TRUE)
```

```
INDEPENDENT SAMPLES T-TEST

Independent Samples T-Test
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
Statistic df p Mean difference SE difference Lower Upper
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
birthweight Student's t 2.296556 98.00000 0.0237731 0.1660471 0.07230265 0.02256481
0.3095293
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

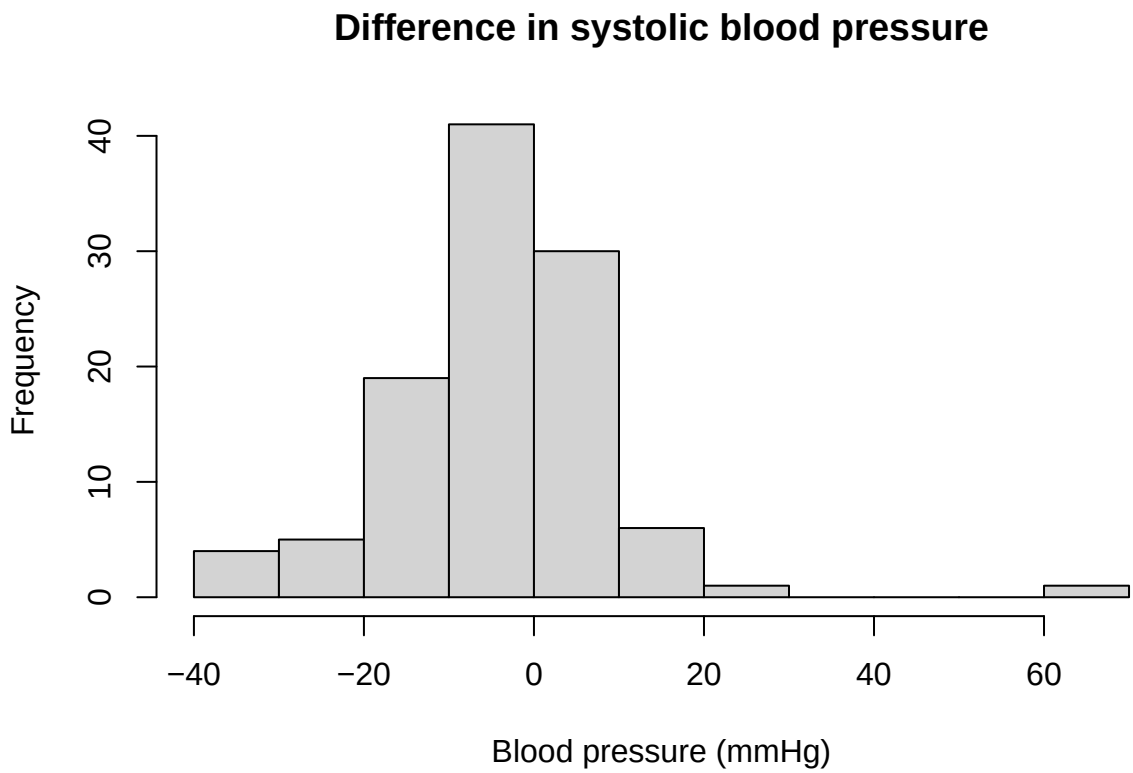
```r
ttestIS(data=bwt, vars=birthweight, group=gender, meanDiff=TRUE, ci=TRUE, welchs=TRUE)
```

```
INDEPENDENT SAMPLES T-TEST

Independent Samples T-Test
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
Statistic df p Mean difference SE difference Lower Upper
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
birthweight Student's t 2.296556 98.00000 0.0237731 0.1660471 0.07230265 0.02256481
0.3095293
Welch's t 2.303840 93.54377 0.0234458 0.1660471 0.07207403 0.02293328 0.3091609
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

5.3 Checking the assumptions for a Paired t-test

Before performing a paired t-test, you must check that the assumptions for the test have been met. Using the dataset `Example_5.2.dta` to show that the difference between the pair of measurements between the sites is normally distributed, we first need to compute a new variable of the differences and examine its histogram.

```r
sbp <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_5.2.dta")
sbp$diff = sbp$sbp_dp - sbp$sbp_tp
hist(sbp$diff, xlab="Blood pressure (mmHg)", main="Difference in systolic blood pressure")
```



5.4 Paired t-Test

To perform a paired t-test we will use the dataset `Example_5.2.dta`.

```
ttestPS(data=sbp, pairs=list(list(i1 = 'sbp_dp', i2 = 'sbp_tp')), meanDiff=TRUE, ci=TRUE)
#>
#>  PAIRED SAMPLES T-TEST
#>
#>  Paired Samples T-Test
#>
#>                                        statistic     df          p            Mean difference
#>
#>    sbp_dp    sbp_tp    Student's t    -0.9621117    106.0000    0.3381832          -1.261682
#>
```

The syntax of the ttestPS function is a little cumbersome. The `t.test` function can be used as an alternative:

```
t.test(sbp$sbp_dp, sbp$sbp_tp, paired=TRUE)
#>
#>  Paired t-test
#>
#> data:  sbp$sbp_dp and sbp$sbp_tp
#> t = -0.96211, df = 106, p-value = 0.3382
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -3.861596  1.338232
#> sample estimates:
```

```
#> mean of the differences
#>                -1.261682
```

Module 6

Proportions

6.1 95% confidence intervals for proportions

We can use the `BinomCI(x=, n=, method=)` function within the `DescTools` package to compute 95% confidence intervals for proportions. Here we specify `x`: the number of successes, `n`: the sample size, and optionally, the `method` (which defaults to Wilson's method).

```
library(DescTools)

BinomCI(x=47, n=215, method='wald')
#>            est    lwr.ci    upr.ci
#> [1,] 0.2186047 0.1633595 0.2738498
BinomCI(x=47, n=215, method='wilson')
#>            est    lwr.ci    upr.ci
#> [1,] 0.2186047 0.1685637 0.2785246
```

6.2 Significance test for single proportion

We can use the `binom.test` function to perform a significance test for a single proportion: `binom.test(x=, n=, p=)`. Here we specify `x`: the number of successes, `n`: the sample size, and `p`: the hypothesised proportion (which defaults to 0.5 if nothing is entered).

```
binom.test(x=54, n=300, p=0.2)
#>
#>  Exact binomial test
#>
#> data:  54 and 300
#> number of successes = 54, number of trials = 300,
#> p-value = 0.4273
#> alternative hypothesis: true probability of success is not equal to 0.2
#> 95 percent confidence interval:
#>  0.1382104 0.2282394
#> sample estimates:
#> probability of success
#>                   0.18
```

6.3 Computing a relative risk and its 95% confidence interval

```
library(haven)
library(labelled)
library(jmv)

drug <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_6.4.dta") %>%
  unlabelled()

head(drug)
#> # A tibble: 6 x 2
#>   group   side_effect
#>   <fct>   <fct>
#> 1 Placebo Nausea
#> 2 Placebo Nausea
#> 3 Placebo Nausea
#> 4 Placebo Nausea
#> 5 Placebo No nausea
#> 6 Placebo No nausea

table(drug$group)
#>
#> Placebo  Active
#>      50      50
table(drug$side_effect)
#>
#> No nausea    Nausea
#>        81        19
table(drug$group, drug$side_effect)
#>
#>           No nausea Nausea
#>   Placebo        46      4
#>   Active         35     15

drug$group <- relevel(drug$group, ref="Active")
drug$side_effect <- relevel(drug$side_effect, ref="Nausea")

table(drug$group)
#>
#>  Active Placebo
#>      50      50
table(drug$side_effect)
#>
#>    Nausea No nausea
#>        19        81
table(drug$group, drug$side_effect)
#>
#>           Nausea No nausea
#>   Active      15        35
#>   Placebo      4        46

contTables(data=drug, rows=group, cols=side_effect, pcRow=TRUE, relRisk = TRUE, diffProp = TRUE)
#>
#>  CONTINGENCY TABLES
```

```
#>
#>  Contingency Tables
#>
#>    group                       Nausea       No nausea    Total
#>
#>    Active     Observed               15           35           50
#>               % within row     30.00000     70.00000    100.00000
#>
#>    Placebo    Observed                4           46           50
#>               % within row      8.00000     92.00000    100.00000
#>
#>    Total      Observed               19           81          100
#>               % within row     19.00000     81.00000    100.00000
#>
#>
#>
#>  x² Tests
#>
#>          Value       df    p
#>
#>    x²    7.862248     1    0.0050478
#>    N          100
#>
#>
#>
#>  Comparative Measures
#>
#>                                   Value        Lower         Upper
#>
#>    Difference in 2 proportions    0.2200000    0.07238986    0.3676101
#>    Relative risk                   3.750000      1.337540     10.51370
#>
```

If you only have the cross-tabulated data (i.e. aggregated), you will need to enter your data into a new data frame.

```
drug_aggregated <- data.frame(
  group = c(1, 1, 0, 0),
  side_effect = c(1, 0, 1, 0),
  n = c(15, 35, 4, 46)
)

drug_aggregated$group <- factor(drug_aggregated$group, levels=c(1,0), labels=c("Active", "Placeb
drug_aggregated$side_effect <- factor(drug_aggregated$side_effect, levels=c(1,0), labels=c("Naus

drug_aggregated
#>     group side_effect  n
#> 1  Active      Nausea 15
#> 2  Active   No nausea 35
#> 3 Placebo      Nausea  4
#> 4 Placebo   No nausea 46

contTables(data=drug_aggregated, rows=group, cols=side_effect, count=n, pcRow=TRUE, relRisk = TR
#>
```

```
#>  CONTINGENCY TABLES
#>
#>  Contingency Tables
#>
#>    group                       Nausea       No nausea    Total
#>
#>    Active     Observed               15           35           50
#>               % within row     30.00000     70.00000    100.00000
#>
#>    Placebo    Observed                4           46           50
#>               % within row      8.00000     92.00000    100.00000
#>
#>    Total      Observed               19           81          100
#>               % within row     19.00000     81.00000    100.00000
#>
#>
#>
#>  x² Tests
#>
#>          Value       df    p
#>
#>    x²    7.862248     1    0.0050478
#>    N          100
#>
#>
#>
#>  Comparative Measures
#>
#>                                   Value        Lower         Upper
#>
#>    Difference in 2 proportions    0.2200000    0.07238986    0.3676101
#>    Relative risk                   3.750000      1.337540     10.51370
#>
```

6.4 Computing an odds ratio and its 95%CI

We can use the `contTables` function To obtain an odds ratio and its 95% CI, by specifying `odds=TRUE`:

```
hpv <- data.frame(
  cancer = c(1, 1, 0, 0),
  hpv = c(1, 0, 1, 0),
  n = c(57, 14, 43, 186)
)

hpv$cancer <- factor(hpv$cancer, levels=c(1,0), labels=c("Case", "Control"))
hpv$hpv <- factor(hpv$hpv, levels=c(1,0), labels=c("HPV +", "HPV -"))

hpv
#>    cancer   hpv   n
#> 1    Case HPV +  57
#> 2    Case HPV -  14
#> 3 Control HPV +  43
#> 4 Control HPV - 186
```

```
contTables(data=hpv, rows=hpv, cols=cancer, count=n, odds = TRUE)
#>
#>  CONTINGENCY TABLES
#>
#>  Contingency Tables
#>
#>    hpv      Case    Control    Total
#>
#>    HPV +      57         43      100
#>    HPV -      14        186      200
#>    Total      71        229      300
#>
#>
#>
#>  x² Tests
#>
#>          Value       df    p
#>
#>    x²    92.25660     1    < .0000001
#>    N          300
#>
#>
#>
#>  Comparative Measures
#>
#>                  Value       Lower       Upper
#>
#>    Odds ratio    17.61130    8.992580    34.49041
#>
```

Module 7

Testing proportions

7.1 Pearson's chi-squared test

7.1.1 Individual data

We will demonstrate how to use R to conduct a Pearson chi-squared test using Worked Example 7.1.

```
library(haven)
library(labelled)

nausea <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_7.1.dta") %>%
  unlabelled()

head(nausea)
#> # A tibble: 6 x 2
#>   group   side_effect
#>   <fct>   <fct>
#> 1 Placebo Nausea
#> 2 Placebo Nausea
#> 3 Placebo Nausea
#> 4 Placebo Nausea
#> 5 Placebo No nausea
#> 6 Placebo No nausea
```

These data have been labelled in Stata, and we use the `unlabelled` function to convert the labelled data into factors. We can confirm that the variables are stored as factors using the `str` function to examine the structure of the variables, and the `table` function to produce a frequency table.

```
str(nausea$group)
#>  Factor w/ 2 levels "Placebo","Active": 1 1 1 1 1 1 1 1 1 1 ...
#>  - attr(*, "label")= chr "Group"
table(nausea$group)
#>
#> Placebo  Active
#>      50      50

str(nausea$side_effect)
#>  Factor w/ 2 levels "No nausea","Nausea": 2 2 2 2 1 1 1 1 1 1 ...
#>  - attr(*, "label")= chr "Side effect"
table(nausea$side_effect)
```

```
#>
#> No nausea    Nausea
#>        81        19
```

To conduct a chi-square test on these data, we first construct a table and view the expected frequencies.

```
tab <- table(nausea$group, nausea$side_effect)
tab
#>
#>           No nausea Nausea
#>   Placebo        46      4
#>   Active         35     15

chisq.test(tab)$expected
#>
#>           No nausea Nausea
#>   Placebo      40.5    9.5
#>   Active       40.5    9.5
```

After confirming that there are no cells with small expected frequencies, we can conduct the chi-square test. By default, R conducts a chi-square test with a continuity correction. To obtain an identical result to that produced by Stata, we use the `correct=FALSE` statement.

```
chisq.test(tab)
#>
#>  Pearson's Chi-squared test with Yates' continuity
#>  correction
#>
#> data:  tab
#> X-squared = 6.4977, df = 1, p-value = 0.0108
chisq.test(tab, correct=FALSE)
#>
#>  Pearson's Chi-squared test
#>
#> data:  tab
#> X-squared = 7.8622, df = 1, p-value = 0.005048
```

The last line labelled Pearson chi2(1) reports the appropriate Chi-squared test statistic which has a value of 7.862 with 1 degree of freedom and a P value of 0.005.

```
fisher.test(tab)
#>
#>  Fisher's Exact Test for Count Data
#>
#> data:  tab
#> p-value = 0.009489
#> alternative hypothesis: true odds ratio is not equal to 1
#> 95 percent confidence interval:
#>   1.384999 21.862717
#> sample estimates:
#> odds ratio
#>   4.852862
```

7.1.2 Summarised data

When you only have the cross-tabulated data, you can enter the summarised data manually. The TextToTable function in the DescTools library is useful here:

```
library(DescTools)

text <- "
           NoNausea, Nausea
Placebo,         46, 4
ActiveDrug,      35, 15"

table <- TextToTable(text, header=TRUE, sep=",", dimnames=c("Group", "SideEffect"))

chisq.test(table)$expected
#>             SideEffect
#> Group        NoNausea Nausea
#>   Placebo        40.5    9.5
#>   ActiveDrug     40.5    9.5
chisq.test(table)
#>
#>  Pearson's Chi-squared test with Yates' continuity
#>  correction
#>
#> data:  table
#> X-squared = 6.4977, df = 1, p-value = 0.0108
chisq.test(table, correct=FALSE)
#>
#>  Pearson's Chi-squared test
#>
#> data:  table
#> X-squared = 7.8622, df = 1, p-value = 0.005048
```

7.2 Chi-squared test for tables larger than 2-by-2

Use the data in `Example_7.2.dta`. We use similar steps as described above for a 2-by-2 table.

```
allergy <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_7.2.dta") %>
  unlabelled()

head(allergy)
#> # A tibble: 6 x 8
#>      id asthma hdmallergy catallergy infection sex
#>   <dbl> <fct>  <fct>      <fct>      <fct>     <fct>
#> 1     1 No     Yes        No         Yes       Female
#> 2     2 Yes    No         No         No        Female
#> 3     3 Yes    No         No         No        Female
#> 4     4 No     No         No         No        Male
#> 5     4 Yes    Yes        Yes        No        Female
#> 6     5 Yes    Yes        Yes        No        Female
#> # ... with 2 more variables: maternalasthma <fct>,
#> #   allergy_severity <fct>
```

```
tab_allergy <- table(allergy$allergy_severity, allergy$sex)
tab_allergy
#>
#>                     Female Male
#>   Non-allergic         150  137
#>   Slight allergy        50   70
#>   Moderate allergy      27   32
#>   Severe allergy        15   19

chisq.test(tab_allergy)$expected
#>
#>                      Female    Male
#>   Non-allergic      138.908 148.092
#>   Slight allergy     58.080  61.920
#>   Moderate allergy   28.556  30.444
#>   Severe allergy     16.456  17.544
chisq.test(tab_allergy)
#>
#>  Pearson's Chi-squared test
#>
#> data:  tab_allergy
#> X-squared = 4.3089, df = 3, p-value = 0.23

jmv::contTables(allergy, allergy_severity, sex, pcCol=TRUE)
#>
#>  CONTINGENCY TABLES
#>
#>  Contingency Tables
#>
#>    allergy_severity                        Female       Male         Total
#>
#>    Non-allergic        Observed                 150        137           287
#>                        % within column     61.98347   53.10078      57.40000
#>
#>    Slight allergy      Observed                  50         70           120
#>                        % within column     20.66116   27.13178      24.00000
#>
#>    Moderate allergy    Observed                  27         32            59
#>                        % within column     11.15702   12.40310      11.80000
#>
#>    Severe allergy      Observed                  15         19            34
#>                        % within column      6.19835    7.36434       6.80000
#>
#>    Total               Observed                 242        258           500
#>                        % within column    100.00000  100.00000     100.00000
#>
#>
#>
#>  x² Tests
#>
#>          Value       df    p
#>
#>    x²    4.308913     3    0.2299813
#>    N          500
```

```
#>
```

7.3 McNemar's test for paired proportions

To perform this test in R, we will use the dataset `Example_7.3.dta`.

```
drug <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_7.3.dta") %>%
  unlabelled()

head(drug)
#> # A tibble: 6 x 2
#>   druga drugb
#>   <fct> <fct>
#> 1 Yes   Yes
#> 2 Yes   Yes
#> 3 Yes   Yes
#> 4 Yes   Yes
#> 5 Yes   Yes
#> 6 Yes   Yes
```

Responses to each drug should be in separate variables in the dataset as shown in Table 7.2 using the `tabulate2` command (**Statistics > Summaries, tables, and tests > Frequency tables > Two-way table with measures of association**). In the tabulate2 dialog box, tick **Relative frequencies** under **Cell contents** as shown below.

```
tab_drug <- table(drug$druga, drug$drugb)
tab_drug
#>
#>       No Yes
#>   No   5  14
#>   Yes 20  21

prop.table(tab_drug)
#>
#>               No        Yes
#>   No  0.08333333 0.23333333
#>   Yes 0.33333333 0.35000000
```

To perform the McNemar's test, go to **Statistics > Epidemiology and related > Tables for epidemiologists > Matched case-control studies**. In the `mcc` dialog box, select the variable `drugb` as the **Exposed case variable** and `druga` as the **Exposed control variable** as shown below.

```
mcnemar.test(tab_drug)
#>
#>  McNemar's Chi-squared test with continuity
#>  correction
#>
#> data:  tab_drug
#> McNemar's chi-squared = 0.73529, df = 1, p-value =
#> 0.3912

epibasix::mcNemar(tab_drug)$rd
#> [1] -0.1
```

```
epibasix::mcNemar(tab_drug)$rd.CIL
#> [1] -0.3054528
epibasix::mcNemar(tab_drug)$rd.CIU
#> [1] 0.1054528
```

Module 8

Correlation and simple linear regression

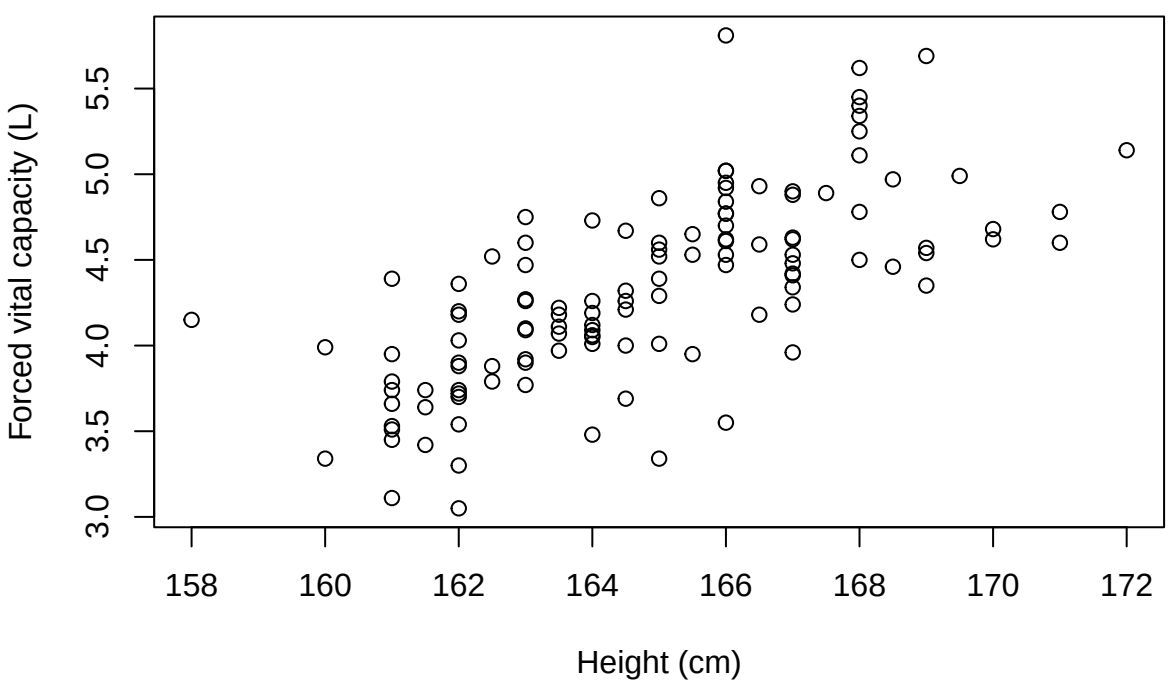
We will demonstrate using Stata for correlation and simple linear regression using the dataset `Example_8.1.dta`.

```
library(ggplot2)   # Optional, for nicer looking scatterplots
library(haven)     # For importing data

lung <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_8.1.dta")
```

8.1 Creating a scatter plot

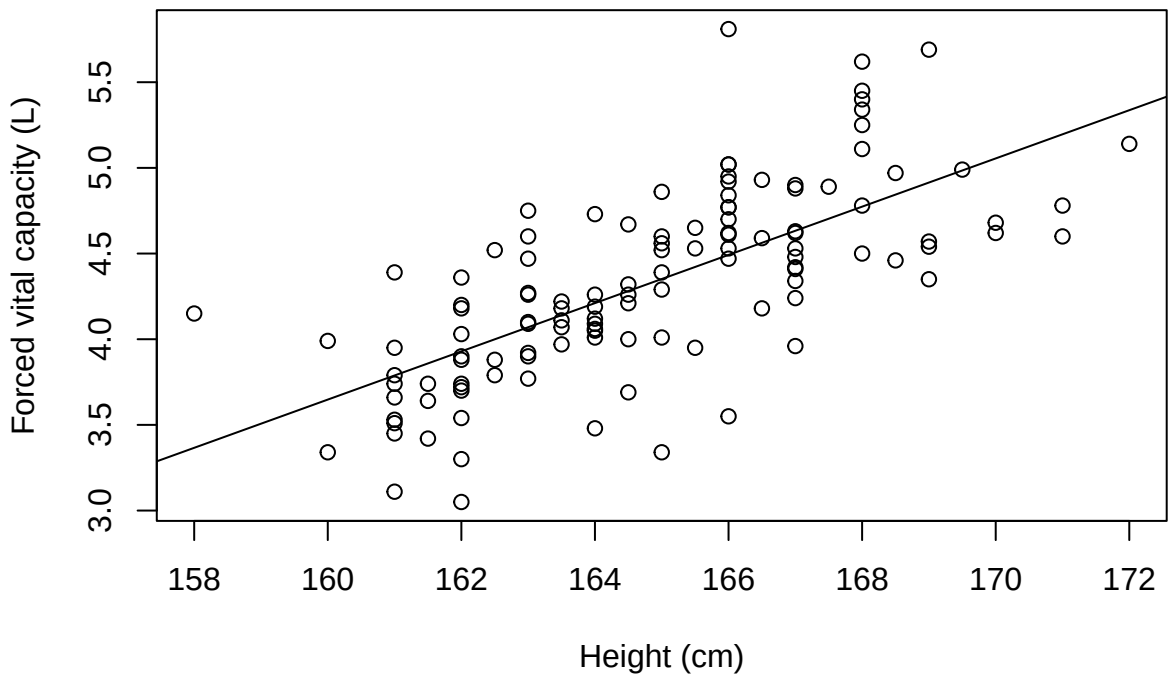
We can use the `plot` function to create a scatter plot to explore the association between height and FVC, assigning meaningful labels with the `xlab` and `ylab` commands:

```
plot(x=lung$Height, y=lung$FVC, xlab="Height (cm)", ylab="Forced vital capacity (L)")
```



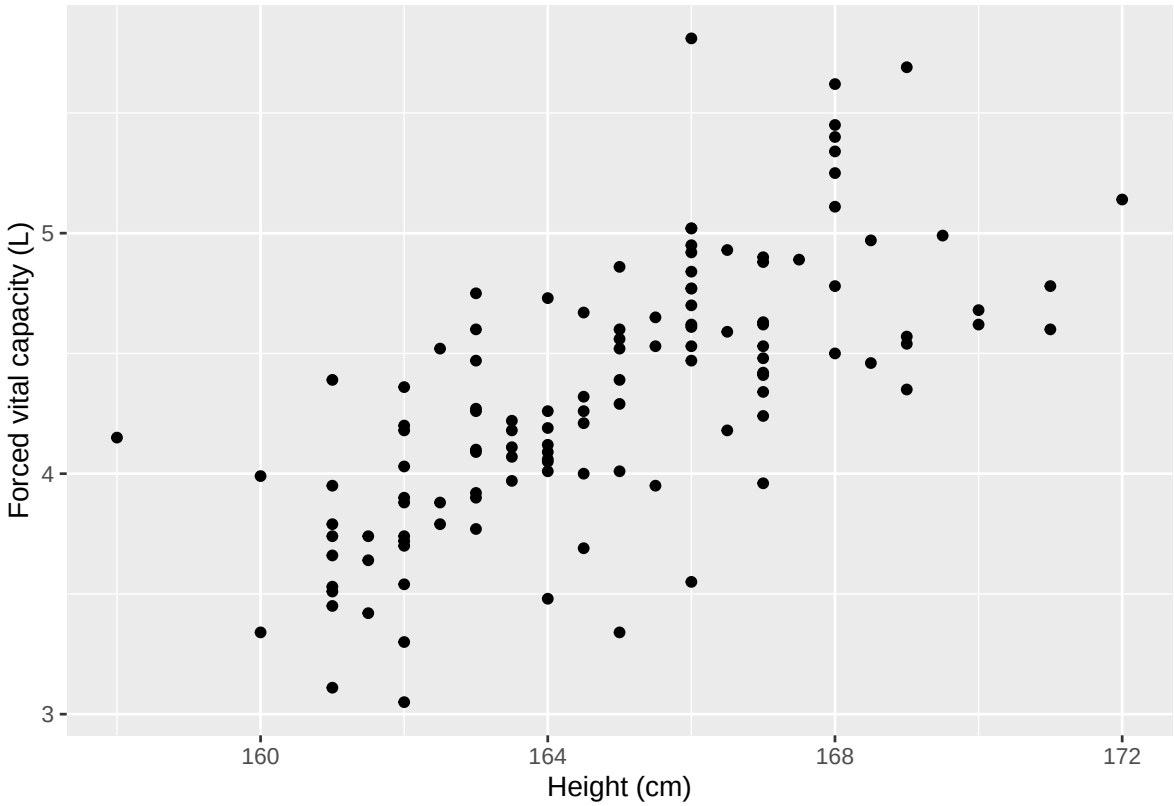
To add a fitted line, we can use the `abline()` function which adds a straight line to the plot. The equation of this straight line will be determined from the estimated regression line, which we specify with `lm(y ~ x)`. Putting this all together:

```
plot(x=lung$Height, y=lung$FVC, xlab="Height (cm)", ylab="Forced vital capacity (L)")
abline(lm(lung$FVC ~ lung$Height))
```



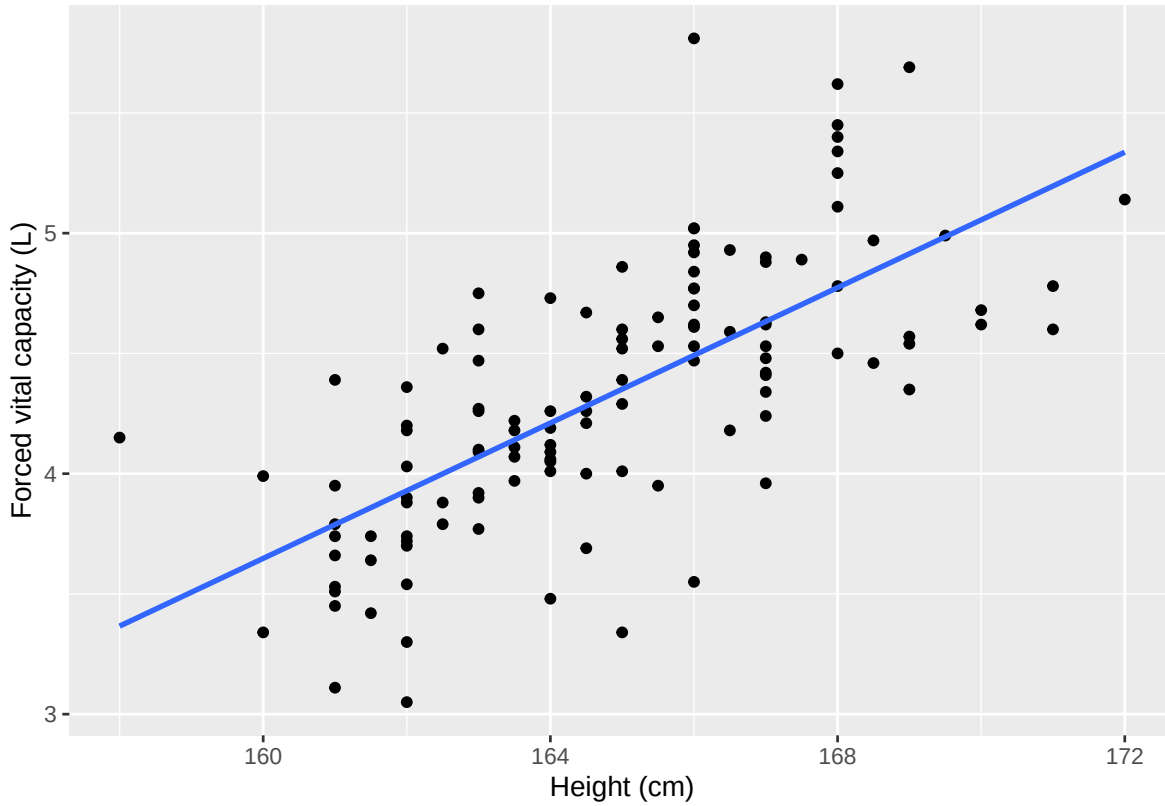
To create a scatter plot using `ggplot2`, we define the x and y aesthetics as the `Height` and `FVC`. We then specify that we want to plot points, by specifying the point geometry using `geom_point`. We can add labels in the usual way. Putting it all together:

```
ggplot(data=lung, aes(x=Height, y=FVC)) +
  geom_point() +
  labs(x="Height (cm)", y="Forced vital capacity (L)")
```



We can add an estimated regression line by adding a `geom_smooth`, specifying that the line should be based on a linear model (`lm`), and no error shading should be included (`se=FALSE`):

```
ggplot(data=lung, aes(x=Height, y=FVC)) +
  geom_point() +
  geom_smooth(method=lm, se=FALSE) +
  labs(x="Height (cm)", y="Forced vital capacity (L)")
#> `geom_smooth()` using formula 'y ~ x'
```



8.2 Calculating a correlation coefficient

We can use the `cor.test` function to calculate a Pearson's correlation coefficient:

```
cor.test(lung$Height, lung$FVC)
#>
#>  Pearson's product-moment correlation
#>
#> data:  lung$Height and lung$FVC
#> t = 10.577, df = 118, p-value < 2.2e-16
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  0.5924715 0.7794090
#> sample estimates:
#>      cor
#> 0.697628
```

8.3 Fitting a simple linear regression model

We can use the `lm` function to fit a simple linear regression model, specifying the model as `y ~ x`. Using `Example_8.1.dta`, we can quantify the relationship between FVC and height.

```
lm(FVC ~ Height, data=lung)
#>
#> Call:
#> lm(formula = FVC ~ Height, data = lung)
#>
#> Coefficients:
```

```
#> (Intercept)       Height
#>    -18.8735       0.1408
```

The default output from the `lm` function is rather sparse. We can obtain much more useful information by defining the model as an object, then using the `summary()` function:

```
model1 <- lm(FVC ~ Height, data=lung)
summary(model1)
#>
#> Call:
#> lm(formula = FVC ~ Height, data = lung)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -1.01139 -0.23643 -0.02082  0.24918  1.31786
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -18.87347    2.19365  -8.604 3.89e-14 ***
#> Height        0.14076    0.01331  10.577  < 2e-16 ***
#> ---
#> Signif. codes:
#> 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.3965 on 118 degrees of freedom
#> Multiple R-squared:  0.4867, Adjusted R-squared:  0.4823
#> F-statistic: 111.9 on 1 and 118 DF,  p-value: < 2.2e-16
```

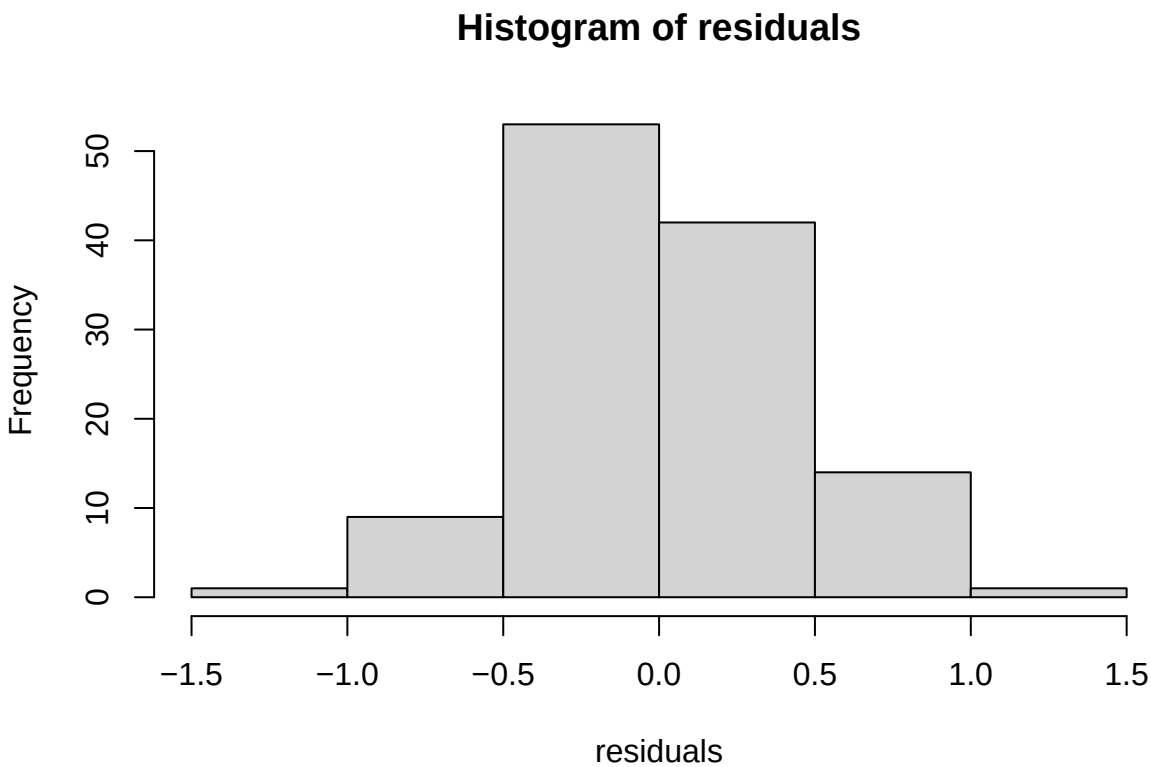
Finally, we can obtain 95% confidence intervals for the regression coefficients using the `confint` function:

```
confint(model1)
#>                   2.5 %      97.5 %
#> (Intercept) -23.2174967 -14.5294444
#> Height        0.1144042   0.1671092
```

8.4 Plotting residuals from a simple linear regression

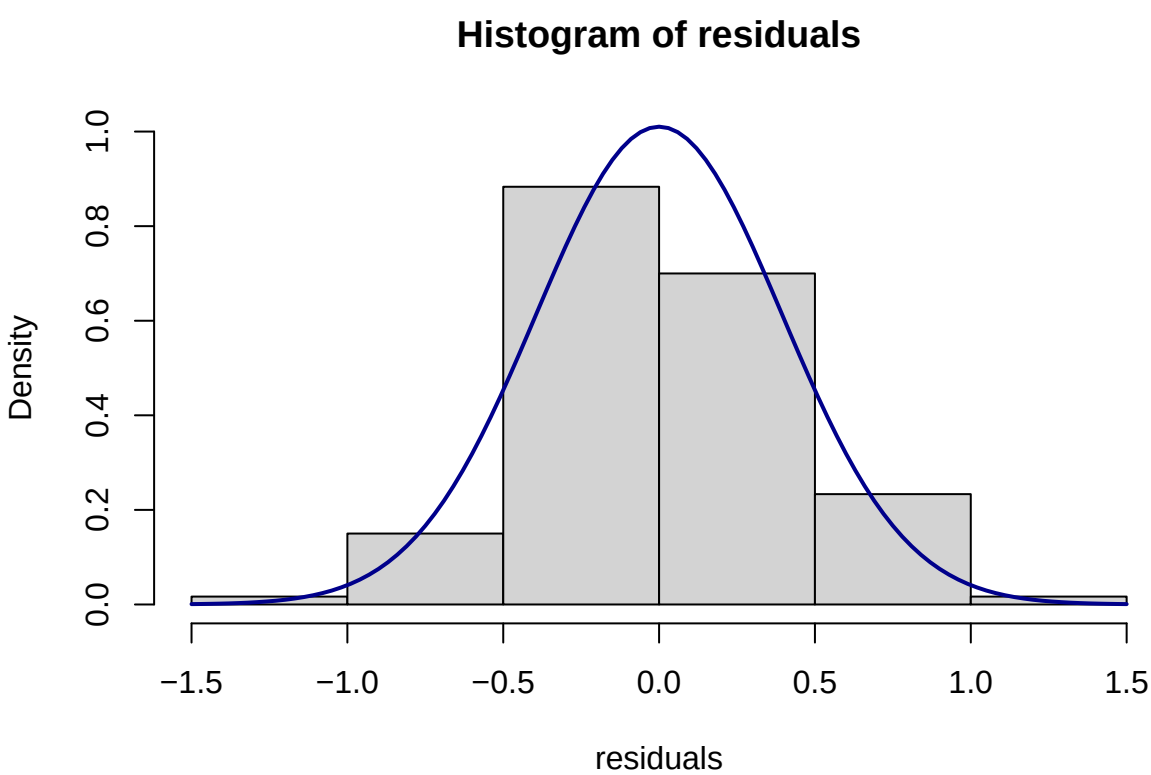
We can use the `resid` function to obtain the residuals from a saved model. These residuals can then be plotted using a histogram in the usual way:

```
residuals <- resid(model1)
hist(residuals)
```



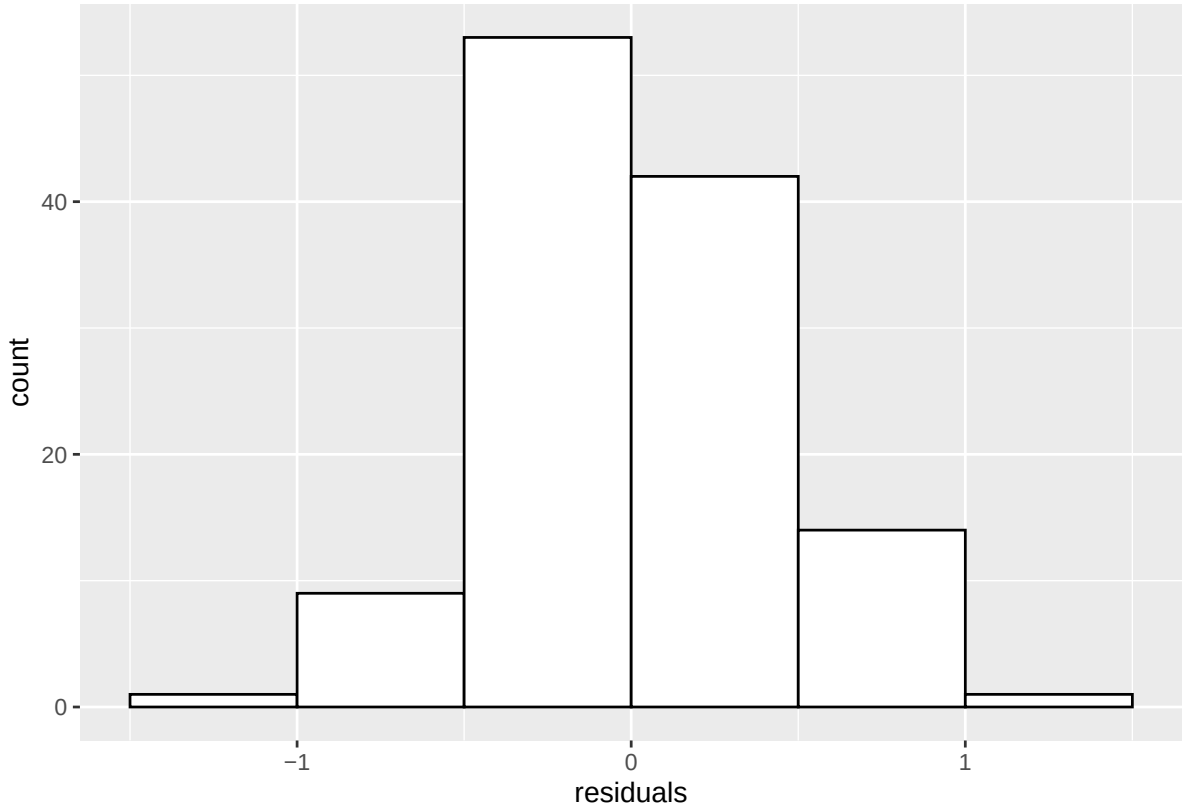
A Normal curve can be overlaid if we plot the residuals using a probability scale.

```
hist(residuals, probability = TRUE, ylim=c(0,1))
curve(dnorm(x, mean=mean(residuals), sd=sd(residuals)),
      col="darkblue", lwd=2, add=TRUE)
```

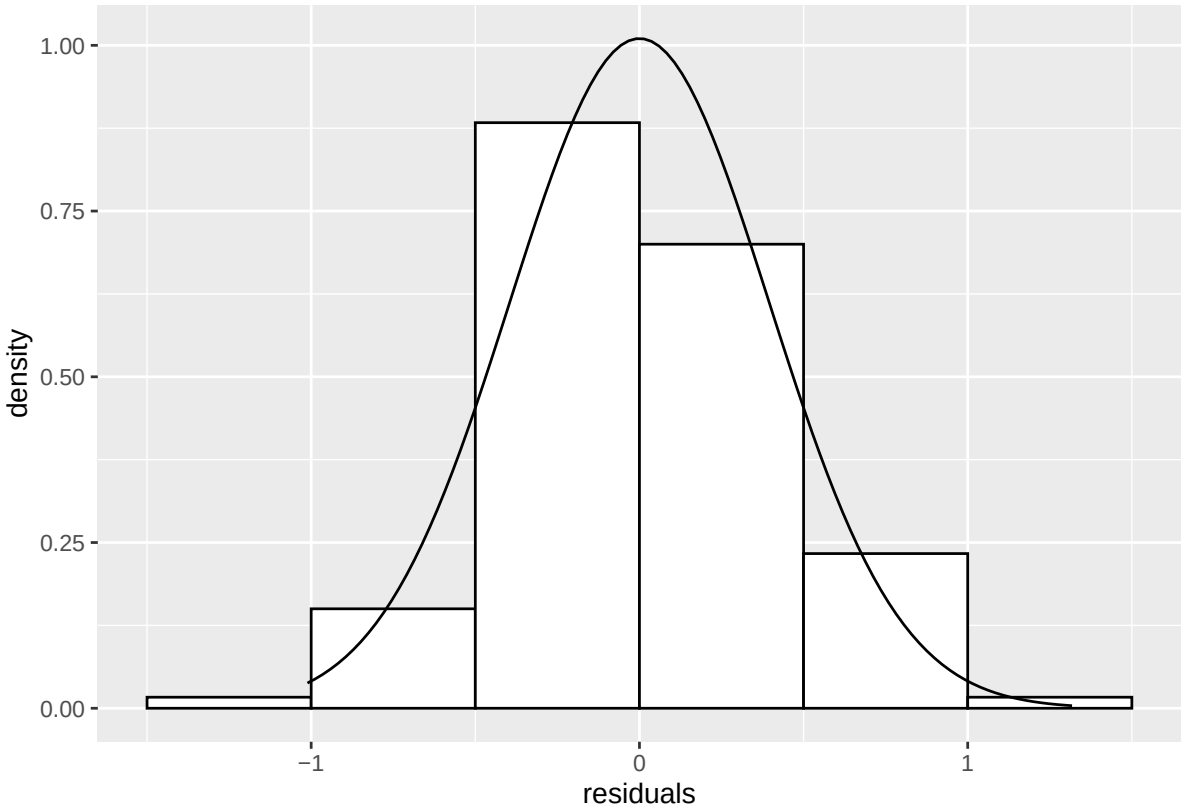


Alternatively, a `ggplot2` approach can be used, after converting the single vector of residuals into a dataframe:

```
resid <- as.data.frame(residuals)

ggplot(resid, aes(x=residuals)) +
  geom_histogram(binwidth = 0.5, boundary=-1.5, colour="black", fill="white")
```




```
ggplot(resid, aes(x=residuals)) +
  geom_histogram(aes(y = ..density..), binwidth = 0.5, boundary=-1.5, colour="black", fill="whi
  stat_function(fun = dnorm, args = list(mean = mean(resid$residuals), sd = sd(resid$residuals)
```


1.00
0.75
0.50
0.25
0.00
density
−1
0
1
residuals

Module 9

Analysing non-normal data

9.1 Transforming non-normally distributed variables

One option for dealing with a non-normally distributed varaible is to transform it into its square, square root or logarithmic value. The new transformed variable may be normally distributed and therefore a parametric test can be used. First we check the distribution of the variable for normality, e.g. by plotting a histogram.

You can calculate a new, transformed, variable using variable transformation commands. For example, to create a new column of data based on the log of length of stay using Base R:

```
library(haven)      # For importing data
library(labelled)

hospital <- unlabelled(read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_

hospital$ln_los <- log(hospital$los+1)
summary(hospital)
#>        id            gender               los
#>  Min.   : 10.00   Length:132         Min.   :  0.00
#>  1st Qu.: 42.75   Class :character   1st Qu.: 20.75
#>  Median : 75.50   Mode  :character   Median : 27.00
#>  Mean   : 75.50                      Mean   : 38.05
#>  3rd Qu.:108.25                      3rd Qu.: 42.00
#>  Max.   :141.00                      Max.   :244.00
#>  infect         surgery       ln_los
#>  No :106   Abdominal:48   Min.   :0.000
#>  Yes: 26   Cardiac  :53   1st Qu.:3.079
#>            Other    :31   Median :3.332
#>                           Mean   :3.407
#>                           3rd Qu.:3.761
#>                           Max.   :5.501
```

A tidyverse version uses the `mutate` command to create a new variable:

```
library(tidyverse)
#> -- Attaching packages ------------------- tidyverse 1.3.1 --
#> v ggplot2 3.3.5     v purrr   0.3.4
#> v tibble  3.1.6     v dplyr   1.0.8
#> v tidyr   1.2.0     v stringr 1.4.0
```

```
#> v readr   2.1.2     v forcats 0.5.1
#> -- Conflicts ---------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()

hospital <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_9.1.dta") %>%
  unlabelled()

hospital <- hospital %>%
  mutate(ln_los = log(los+1))

summary(hospital)
#>        id            gender               los
#>  Min.   : 10.00   Length:132         Min.   :  0.00
#>  1st Qu.: 42.75   Class :character   1st Qu.: 20.75
#>  Median : 75.50   Mode  :character   Median : 27.00
#>  Mean   : 75.50                      Mean   : 38.05
#>  3rd Qu.:108.25                      3rd Qu.: 42.00
#>  Max.   :141.00                      Max.   :244.00
#>  infect         surgery       ln_los
#>  No :106   Abdominal:48   Min.   :0.000
#>  Yes: 26   Cardiac  :53   1st Qu.:3.079
#>            Other    :31   Median :3.332
#>                           Mean   :3.407
#>                           3rd Qu.:3.761
#>                           Max.   :5.501
```

You can now check whether this logged variable is normally distributed as described in Module 2, for example by plotting a histogram as shown in Figure 9.2.

To obtain the back-transformed mean shown in Output 9.1, we can use the `exp` command:

```
exp(3.407232)
#> [1] 30.18159
```

If your transformed variable is approximately normally distributed, you can apply parametric tests such as the t-test. In the Worked Example 9.1 dataset, the variable `infect` (presence of nosocomial infection) is a binary categorical variable. To test the hypothesis that patients with nosocomial infection have a different length of stay to patients without infection, you can conduct a t-test on the `ln_los` variable. You will need to back transform your mean values, as shown in Worked Example 9.1 in the course notes when reporting your results.

9.2 Wilcoxon ranked-sum test

We use the `wilcox.test` function to perform the Wilcoxon ranked-sum test:

```
wilcox.test(continuous_variable ~ group_variable, data=df, correct=FALSE)
```

The Wilcoxon ranked-sum test will be demonstrated using the length of stay data in `Example_9.1.dta`. Here, out continuous variable is `los` and the grouping variable is `infect`.

```
wilcox.test(los ~ infect, data=hospital, correct=FALSE)
#>
#>  Wilcoxon rank sum test
#>
#> data:  los by infect
#> W = 949, p-value = 0.01402
#> alternative hypothesis: true location shift is not equal to 0
```

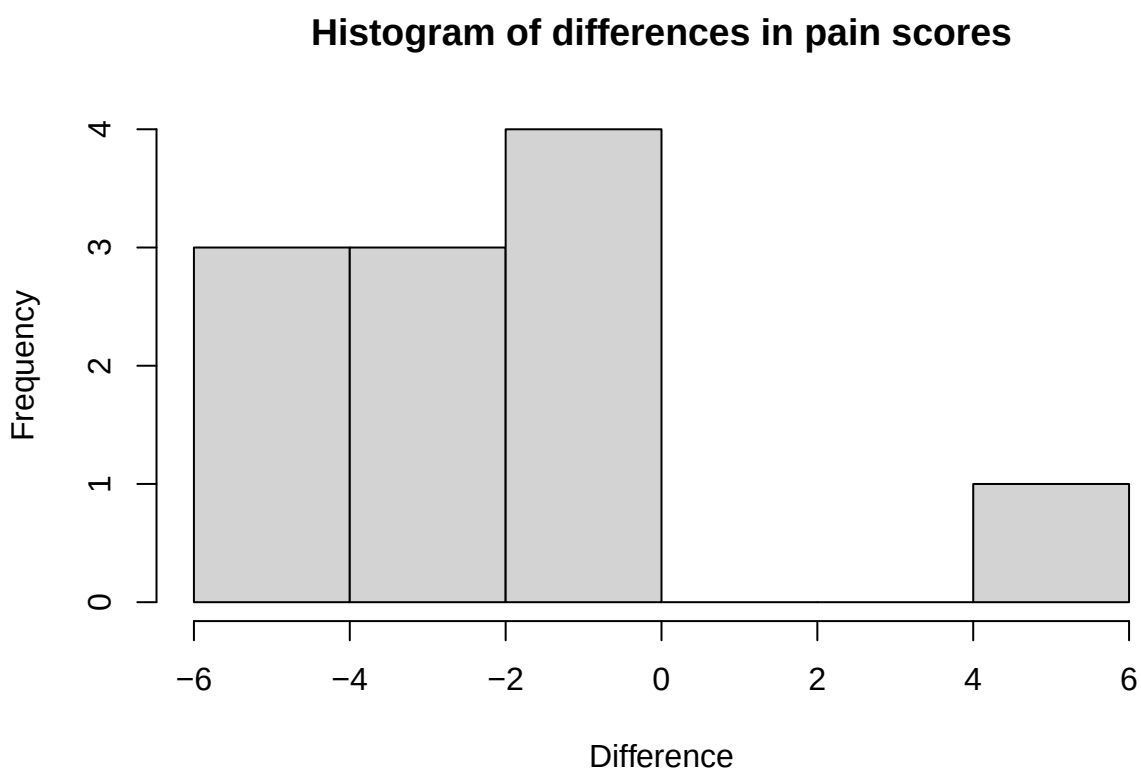
9.3 Wilcoxon matched-pairs signed-rank test

The `wilcox.test` function can also be used to conduct the Wilcoxon matched-pairs signed-rank test. The specification of the variables is a little different, in that each variable is specified as `dataframe$variable`:

```
wilcox.test(df$continuous_variable_1, df$continuous_variable_1, paired=TRUE)
```

We will demonstrate using the dataset on the arthritis drug cross-over trial (`Example_9.2.dta`). Like the paired t-test the paired data need to be in separate columns.

```
arthritis <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_9.2.dta")
  unlabelled()

arthritis$difference = arthritis$drug_1 - arthritis$drug_2

hist(arthritis$difference, xlab="Difference", main="Histogram of differences in pain scores")
```



```
wilcox.test(arthritis$drug_1, arthritis$drug_2, paired=TRUE)
#> Warning in wilcox.test.default(arthritis$drug_1,
#> arthritis$drug_2, paired = TRUE): cannot compute exact p-
#> value with ties
#>
#>  Wilcoxon signed rank test with continuity correction
#>
#> data:  arthritis$drug_1 and arthritis$drug_2
#> V = 10.5, p-value = 0.04898
#> alternative hypothesis: true location shift is not equal to 0
```

9.4 Estimating rank correlation coefficients

The analyses for Spearman's and Kendall's rank correlation are conducted in similar ways:

```
lung <- read_dta("/Users/td/Documents/GithubRepos/phcm9795/data/examples/Example_8.1.dta")

cor.test(lung$Height, lung$FVC, method="spearman")
#> Warning in cor.test.default(lung$Height, lung$FVC, method =
#> "spearman"): Cannot compute exact p-value with ties
#>
#>  Spearman's rank correlation rho
#>
#> data:  lung$Height and lung$FVC
#> S = 72699, p-value < 2.2e-16
#> alternative hypothesis: true rho is not equal to 0
#> sample estimates:
#>       rho
#> 0.7475566

cor.test(lung$Height, lung$FVC, method="kendall")
#>
#>  Kendall's rank correlation tau
#>
#> data:  lung$Height and lung$FVC
#> z = 8.8244, p-value < 2.2e-16
#> alternative hypothesis: true tau is not equal to 0
#> sample estimates:
#>       tau
#> 0.5609431
```

Module 10

Sample size

Many power and sample size procedures are available in the `epiR` package.

```
# If not yet installed, submit the following:
# install.packages("epiR")
library(epiR)
```

10.1 Sample size calculation for two independent samples t-test

To do the problem discussed in Worked Example 10.2, we use the `epi.sscompc` function: **Sample size, power and minimum detectable difference when comparing continuous outcomes**.

```
epi.sscompc(treat, control, n, sigma, power,
   r = 1, design = 1, sided.test = 2, nfractional = FALSE, conf.level = 0.95)
```

The first line contains parameters that we can specify, with one parameter that is to be calculated. That is, we define values for four of the five parameters, and the remaining parameter is calculated. For example, we can define the mean in the treated group, the mean in the control group, the assumed standard deviation and the desired power, and the function will calculate the required sample size. We specify the unknown value as being equal to R's missing value, `NA`.

For example, to calculate the required sample size in Worked Example 10.2, we specify:

- the assumed mean in the experimental, or treatment, group: 90mmHg
- the assumed mean in the control group: 95mmHg
- the standard deviation of blood pressure: 5mmHg
- the required power, 0.9 (representing 90%)

The values on the second line of the function are defined by default, and we can leave these as default.

Putting this all together, and specifying the sample size as unknown:

```
epi.sscompc(treat=90, control=95, n=NA, sigma=5, power=0.9)
#> $n.total
#> [1] 44
#>
#> $n.treat
#> [1] 22
```

```
#>
#> $n.control
#> [1] 22
#>
#> $power
#> [1] 0.9
#>
#> $delta
#> [1] 5
```

The results indicate that we need 22 participants in each group, or 44 in total.

We can define whether we want unequal numbers in each group by specifying `r`: the number in the treatment group divided by the number in the control group.

10.2 Sample size calculation for difference between two independent proportions

To do the problem discussed in Worked Example 10.3, we use the `epi.sscohortc` function: **Sample size, power or minimum detectable incidence risk ratio for a cohort study using individual count data**.

```
epi.sscohortc(irexp1, irexp0, pexp = NA, n = NA, power = 0.80,
   r = 1, N, design = 1, sided.test = 2, finite.correction = FALSE,
   nfractional = FALSE, conf.level = 0.95)
```

We can enter:

- irexp1: the assumed risk of the outcome in the exposed group: here 0.35
- irexp0: the assumed risk of the outcome in the unexposed group: here 0.2
- n: the total sample size, to be determined
- power: the required power: here 0.8 (representing 80%)

```
epi.sscohortc(irexp1=0.35, irexp0=0.2, n=NA, power=0.8)
#> $n.total
#> [1] 276
#>
#> $n.exp1
#> [1] 138
#>
#> $n.exp0
#> [1] 138
#>
#> $power
#> [1] 0.8
#>
#> $irr
#> [1] 1.75
#>
#> $or
#> [1] 2.153846
```

Note: It doesn't matter if you swap the proportions for the **exposed** and **unexposed** groups, i.e. the command `epi.sscohortc(irexp1=0.2, irexp0=0.35, n=NA, power=0.8)` gives the same results.

10.3 Sample size calculation with a relative risk

The `epiR` package does not have a function to estimate sample size and power directly for a relative risk, but we can use the `epi.sscohortc` function.

10.4 Sample size calculation with an odds ratio

We can use the `epi.sscc` function to calculate a sample size based on an odds ratio in a case-control study:

```
epi.sscc(OR, p1 = NA, p0, n, power, r = 1,
   phi.coef = 0, design = 1, sided.test = 2, nfractional = FALSE,
   conf.level = 0.95, method = "unmatched", fleiss = FALSE)
```

Using information from Worked Example 10.4, we specify:

- OR: the odds ratio to be detected, here 1.5
- p0: the proportion of the outcome in the controls, here 0.5
- n: the sample size, here to be calculated
- power: the required study power, here 0.9

```
epi.sscc(OR=1.5, p0=0.5, n=NA, power=0.9)
#> $n.total
#> [1] 1038
#>
#> $n.case
#> [1] 519
#>
#> $n.control
#> [1] 519
#>
#> $power
#> [1] 0.9
#>
#> $OR
#> [1] 1.5
```

Now we calculate the sample size for Worked Example 10.5:

```
epi.sscc(OR=2, p0=0.3, n=NA, power=0.9)
#> $n.total
#> [1] 376
#>
#> $n.case
#> [1] 188
#>
#> $n.control
#> [1] 188
#>
#> $power
#> [1] 0.9
#>
#> $OR
#> [1] 2
```

Here we see that we require a total of 376 participants to detect an odds ratio of 2.0 with 90% power;

```
epi.sscc(OR=2, p0=0.3, n=NA, power=0.8)
#> $n.total
#> [1] 282
#>
#> $n.case
#> [1] 141
#>
#> $n.control
#> [1] 141
#>
#> $power
#> [1] 0.8
#>
#> $OR
#> [1] 2
```

or a total of 282 participants to detect an odds ratio of 2.0 with 80% power.

Bibliography

Terry M. Therneau and Patricia M. Grambsch. *Modeling Survival Data: Extending the Cox Model*. Springer, New York Berlin Heidelberg, December 2010. ISBN 978-1-4419-3161-0.