Statistics 1

Contents

About this course

This course is an introduction to data science. We have three primary aims. First, to introduce you to the logic of quantitative research design. Second, to familiarise you with statistical models that scientists and policy-makers use to answer social science questions. Third, to help you acquire the necessary skills to conduct your own quantitative research projects. No prior statistical knowledge is assumed. We will use the statistical software R and RStudio on top.

Syllabus

Moodle

Piazza

Chapter 1

Introduction: Measurement, Central Tendency, Dispersion, Validity, Reliability

1.1 Seminar

In this seminar session, we introduce working with R. We illustrate some basic functionality and help you familiarise yourself with the look and feel of RStudio. Measures of central tendency and dispersion are easy to calculate in R. We focus on introducing the logic of R first and then describe how central tendency and dispersion are calculated in the end of the seminar.

1.1.1 Getting Started

Install R and RStudio on your computer by downloading them from the following sources:

- Download R from The Comprehensive R Archive Network (CRAN)
- Download RStudio from RStudio.com

1.1.2 RStudio

Let's get acquainted with R. When you start RStudio for the first time, you'll see three panes:

1.1.3 Console

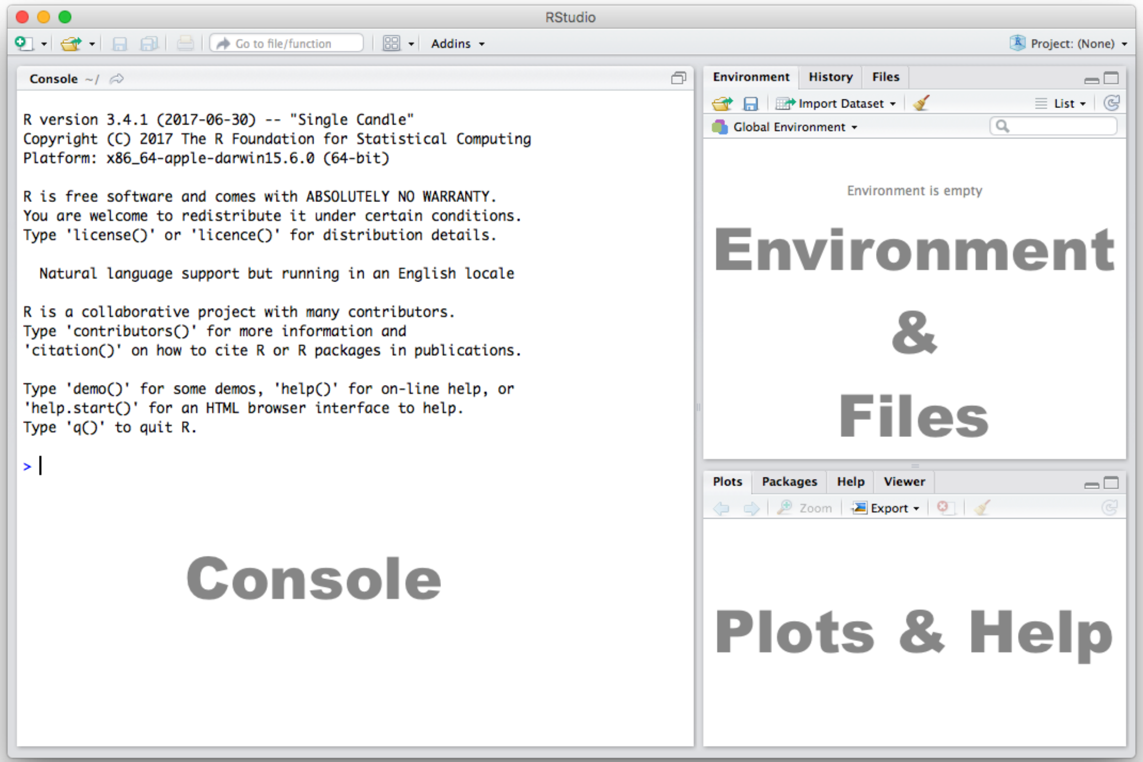
The Console in RStudio is the simplest way to interact with R. You can type some code at the Console and when you press ENTER, R will run that code. Depending on what you type, you may see some output in the Console or if you make a mistake, you may get a warning or an error message.

Let's familiarize ourselves with the console by using R as a simple calculator:

```r
2 + 4
```

```
[1] 6
```

Now that we know how to use the `+` sign for addition, let's try some other mathematical operations such as subtraction (`-`), multiplication (`*`), and division (`/`).



```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

> |
```

Figure 1.1:

```
10 - 4
```

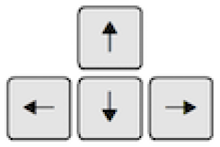
```
[1] 6
```

```
5 * 3
```

```
[1] 15
```

```
7 / 2
```

```
[1] 3.5
```



You can use the cursor or arrow keys on your keyboard to edit your code at the console:- Use the UP and DOWN keys to re-run something without typing it again- Use the LEFT and RIGHT keys to edit

Take a few minutes to play around at the console and try different things out. Don't worry if you make a mistake, you can't break anything easily!

1.1.4 Functions

Functions are a set of instructions that carry out a specific task. Functions often require some input and generate some output. For example, instead of using the `+` operator for addition, we can use the `sum` function to add two or more numbers.

```
sum(1, 4, 10)
```

```
[1] 15
```

In the example above, `1, 4, 10` are the inputs and 15 is the output. A function always requires the use of parenthesis or round brackets `()`. Inputs to the function are called **arguments** and go inside the brackets. The output of a function is displayed on the screen but we can also have the option of saving the result of the output. More on this later.

1.1.5 Getting Help

Another useful function in R is `help` which we can use to display online documentation. For example, if we wanted to know how to use the `sum` function, we could type `help(sum)` and look at the online documentation.

```
help(sum)
```

The question mark `?` can also be used as a shortcut to access online help.

```
?sum
```

Use the toolbar button shown in the picture above to expand and display the help in a new window.

Help pages for functions in R follow a consistent layout generally include these sections:

Description	A brief description of the function
Usage	The complete syntax or grammar including all arguments (inputs)
Arguments	Explanation of each argument

Details	Any relevant details about the function and its arguments
Value	The output value of the function
Examples	Example of how to use the function

1.1.6 The Assignment Operator

Now we know how to provide inputs to a function using parenthesis or round brackets `()`, but what about the output of a function?

We use the assignment operator `<-` for creating or updating objects. If we wanted to save the result of adding `sum(1, 4, 10)`, we would do the following:

```
myresult <- sum(1, 4, 10)
```

The line above creates a new object called `myresult` in our environment and saves the result of the `sum(1, 4, 10)` in it. To see what's in `myresult`, just type it at the console:

```
myresult
```

```
[1] 15
```

Take a look at the **Environment** pane in RStudio and you'll see `myresult` there.

To delete all objects from the environment, you can use the **broom** button as shown in the picture above.

We called our object `myresult` but we can call it anything as long as we follow a few simple rules. Object names can contain upper or lower case letters (`A-Z`, `a-z`), numbers (`0-9`), underscores (`_`) or a dot (`.`) but all object names must start with a letter. Choose names that are descriptive and easy to type.

Good Object Names	Bad Object Names
result	a
myresult	x1
my.result	this.name.is.just.too.long
my_result	
data1	

1.1.7 Sequences

We often need to create sequences when manipulating data. For instance, you might want to perform an operation on the first 10 rows of a dataset so we need a way to select the range we're interested in.

There are two ways to create a sequence. Let's try to create a sequence of numbers from 1 to 10 using the two methods:

1. Using the colon `:` operator. If you're familiar with spreadsheets then you might've already used `:` to select cells, for example `A1:A20`. In R, you can use the `:` to create a sequence in a similar fashion:

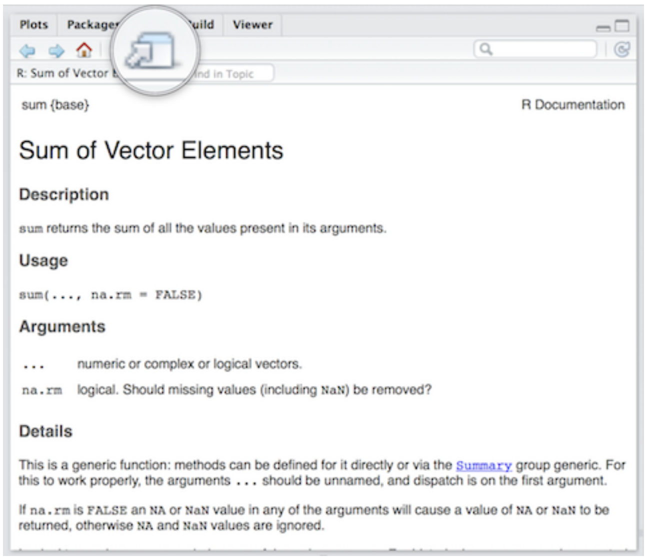
```
1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

1. Using the `seq` function we get the exact same result:

```
seq(from = 1, to = 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```



sum {base} R Documentation

Sum of Vector Elements

Description

sum returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

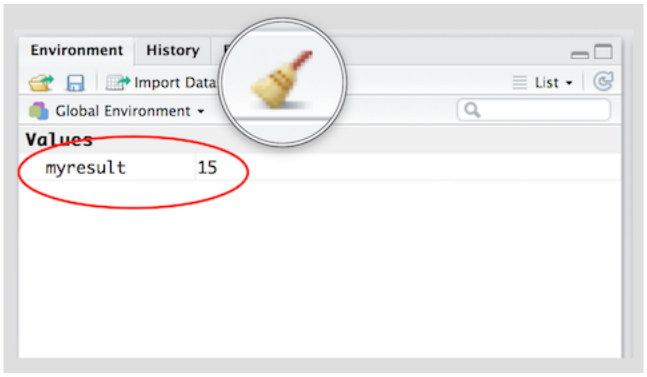
`...`	numeric or complex or logical vectors.
`na.rm`	logical. Should missing values (including `NaN`) be removed?

Details

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` or `NaN` value in any of the arguments will cause a value of `NA` or `NaN` to be returned, otherwise `NA` and `NaN` values are ignored.

Figure 1.2:



```
myresult        15
```

Figure 1.3:

The `seq` function has a number of options which control how the sequence is generated. For example to create a sequence from 0 to 100 in increments of `5`, we can use the optional `by` argument. Notice how we wrote `by = 5` as the third argument. It is a common practice to specify the name of argument when the argument is optional. The arguments `from` and `to` are not optional, se we can write `seq(0, 100, by = 5)` instead of `seq(from = 0, to = 100, by = 5)`. Both, are valid ways of achieving the same outcome. You can code whichever way you like. We recommend to write code such that you make it easy for your future self and others to read and understand the code.

```
seq(from = 0, to = 100, by = 5)
```

```
 [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80
[18]  85  90  95 100
```

Another common use of the `seq` function is to create a sequence of a specific length. Here, we create a sequence from 0 to 100 with length 9, i.e., the result is a vector with 9 elements.

```
seq(from = 0, to = 100, length.out =  9)
```

```
[1]   0.0  12.5  25.0  37.5  50.0  62.5  75.0  87.5 100.0
```

Now it's your turn:

- Create a sequence of **odd** numbers between 0 and 100 and save it in an object called `odd_numbers`

```
odd_numbers <- seq(1, 100, 2)
```

- Next, display `odd_numbers` on the console to verify that you did it correctly

```
odd_numbers
```

```
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
[24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
[47] 93 95 97 99
```

- What do the numbers in square brackets `[ ]` mean? Look at the number of values displayed in each line to find out the answer.
- Use the `length` function to find out how many values are in the object `odd_numbers`.
  - HINT: Try `help(length)` and look at the examples section at the end of the help screen.

```
length(odd_numbers)
```

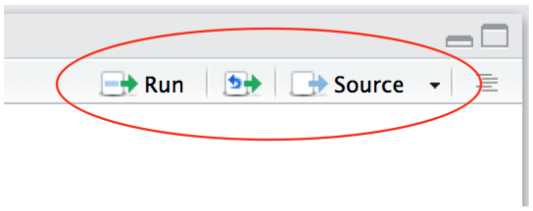
```
[1] 50
```

1.1.8 Scripts

The Console is great for simple tasks but if you're working on a project you would mostly likely want to save your work in some sort of a document or a file. Scripts in R are just plain text files that contain R code. You can edit a script just like you would edit a file in any word processing or note-taking application.

Create a new script using the menu or the toolbar button as shown below.

Once you've created a script, it is generally a good idea to give it a meaningful name and save it immediately. For our first session save your script as **seminar1.R**



Familiarize yourself with the script window in RStudio, and especially the two buttons labeled **Run** and **Source**

There are a few different ways to run your code from a script.

One line at a time	Place the cursor on the line you want to run and hit CTRL-ENTER or use the **Run** button
Multiple lines	Select the lines you want to run and hit CTRL-ENTER or use the **Run** button
Entire script	Use the **Source** button

1.1.9 Central Tendency

The appropriate measure of central tendency depends on the level of measurement of the variable. To recap:

Level of measurement	Appropriate measure of central tendency
Continuous	arithmetic mean (or average)
Ordered	median (or the central observation)
Nominal	mode (the most frequent value)

1.1.9.1 Mean

We calculate the average grade on our eleven homework assignments in statistics 1. We create our vector of 11 (fake) grades first using the `c()` function, where `c` stands for collect or concatenate.

```r
hw.grades <- c(80, 90, 85, 71, 69, 85, 83, 88, 99, 81, 92)
```

We now take the sum of the grades.

```r
sum.hw.grades <- sum(hw.grades)
```

We also take the number of grades

```r
number.hw.grades <- length(hw.grades)
```

The mean is the sum of grades over the number of grades.

```r
sum.hw.grades / number.hw.grades
```

```
[1] 83.90909
```

R provides us with an even easier way to do the same with a function called `mean()`.

```r
mean(hw.grades)
```

```
[1] 83.90909
```

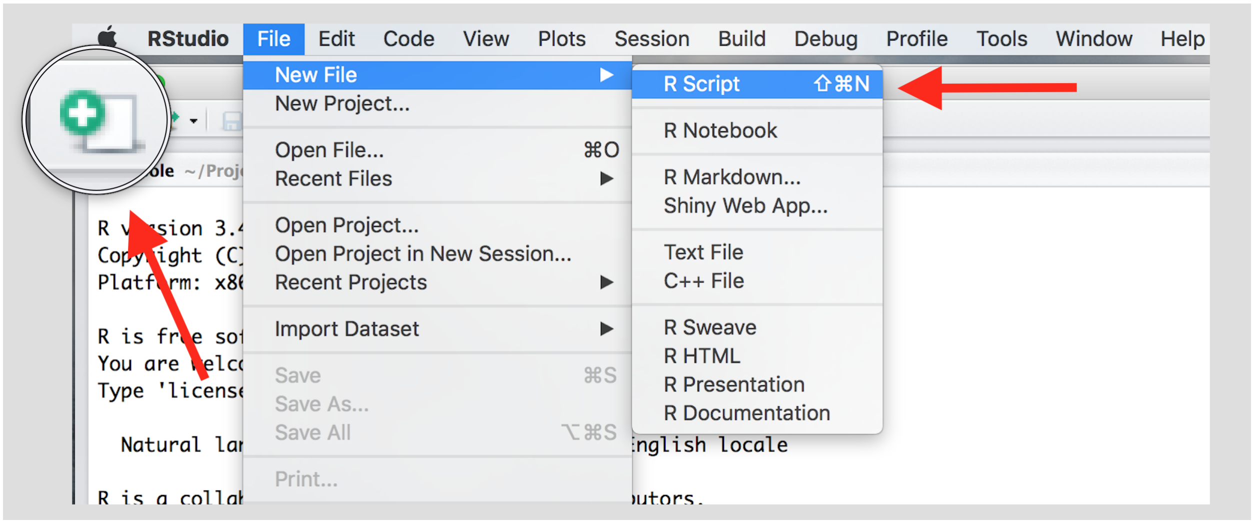


Figure 1.4:

1.1.9.2 Median

The median is the appropriate measure of central tendency for ordinal variables. Ordinal means that there is a rank ordering but not equally spaced intervals between values of the variable. Education is a common example. In education, more education is better. But the difference between primary school and secondary school is not the same as the difference between secondary school and an undergraduate degree.

Let's generate a fake example with 100 people. We use numbers to code different levels of education.

Code	Meaning	Frequency in our data
0	no education	1
1	primary school	5
2	secondary school	55
3	undergraduate degree	20
4	postgraduate degree	10
5	doctorate	9

We introduce a new function to create a vector. The function `rep()`, replicates elements of a vector. Its arguments are the item `x` to be replicated and the number of `times` to replicate. Below, we create the variable education with the frequency of education level indicated above. Note that the arguments `x` and `times` do not have to be written out.

```
edu <- c( rep(x=0, times=1), rep(x=1, times=5), rep(x=2, times=55),
          rep(x=3, times=20), rep(4,10), rep(5,9) )
```

The median level of education is the level where 50 percent of the observations have a lower or equal level of education and 50 percent have a higher or equal level of education. That means that the median splits the data in half.

We use the `median()` function for finding the median.

```
median(edu)
```

```
[1] 2
```

The median level of education is secondary school.

1.1.9.3 Mode

The mode is the appropriate measure of central tendency if the level of measurement is nominal. Nominal means that there is no ordering implicit in the values that a variable takes on. We create data from 1000 (fake) voters in the United Kingdom who each express their preference on remaining in or leaving the European

```
table(stay)
```

```
stay
  0   1
509 491
```

The mode is leaving the EU because the number of 'leavers' (0) is greater than the number of 'remainers' (1).

1.1.10 Dispersion

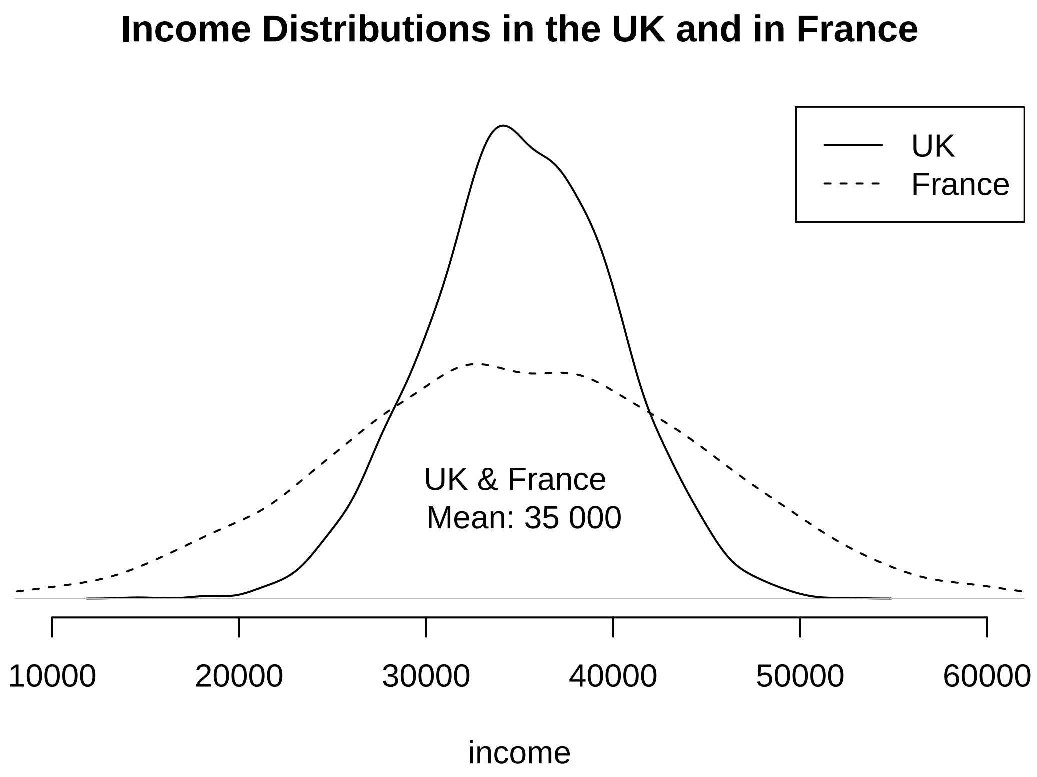
The appropriate measure of dispersion depends on the level of measurement of the variable we wish to describe.

Level of measurement	Appropriate measure of dispersion
Continuous	variance and/or standard deviation
Ordered	range or interquartile range
Nominal	proportion in each category

1.1.10.1 Variance and standard deviation

Both the variance and the standard deviation tell by how much an average realisation of a variable differs from the mean of that variable. Let's assume that our variable is income in the UK. Let's assume that its mean is 35 000 per year. We also assume that the average deviation from 35 000 is 5 000. If we ask 100 people in the UK at random about their income, we get 100 different answers. If we average the differences betweeen the 100 answers and 35 000, we would get 5 000. Suppose that the average income in France is also 35 000 per year but the average deviation is 10 000 instead. This would imply that income is more equally distributed in the UK than in France.

Dispersion is important to describe data as this example illustrates. Although, mean income in our hypothetical example is the same in France and the UK, the distribution is tighter in the UK. The figure below illustrates our example:



The variance gives us an idea about the variability of data. The formula for the variance in the population is

$$\frac{\sum_{i=1}^{n}(x_i - \mu_x)^2}{n}$$

The formula for the variance in a sample adjusts for sampling variability, i.e., uncertainty about how well our sample reflects the population by subtracting 1 in the denominator. Subtracting 1 will have next to no effect if n is large but the effect increases the smaller n. The smaller n, the larger the sample variance. The intuition is, that in smaller samples, we are less certain that our sample reflects the population. We, therefore, adjust variability of the data upwards. The formula is

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$

Notice the different notation for the mean in the two formulas. We write μ_x for the mean of x in the population and $\bar{x}$ for the mean of x in the sample. Notation is, however, unfortunately not always consistent.

Take a minute to think your way through the formula. There are 4 setps: (1), In the numerator, we subtract the mean of x from some realisation of x. (2), We square the deviations from the mean because we want positive numbers only. (3) We sum the squared deviations. (4) We divide the sum by $(n-1)$. Below we show this for the homework example. In the last row, we add a 5th step. We take the square root in order to return to the orginial units of the homework grades.

Obs	Var	Dev. from mean	Squared dev. from mean
i	grade	$x_i - \bar{x}$	$(x_i - \bar{x})^2$
1	80	-3.9090909	15.2809917

Obs	Var	Dev. from mean	Squared dev. from mean
2	90	6.0909091	37.0991736
3	85	1.0909091	1.1900826
4	71	-12.9090909	166.6446281
5	69	-14.9090909	222.2809917
6	85	1.0909091	1.1900826
7	83	-0.9090909	0.8264463
8	88	4.0909091	16.7355372
9	99	15.0909091	227.7355372
10	81	-2.9090909	8.4628099
11	92	8.0909091	65.4628099
$\sum_{i=1}^{n}$			762.9090909
$\div n-1$			76.2909091
$\surd$			8.7344667

Our first grade (80) is below the mean (83.9090909). The sum is, thus, negative. Our second grade (90) is above the mean, so that the sum is positive. Both are deviations from the mean (think of them as distances). Our sum shall reflect the total sum of distances and distances must be positive. Hence, we square the distances from the mean. Having done this for all eleven observations, we sum the squared distances. Dividing by 10 (with the sample adjustment), gives us the average squared deviation. This is the variance. The units of the variance—squared deviations—are somewhat awkward. We return to this in a moment.

We take the variance in R by using the `var()` function. By default `var()` takes the sample variance.

```r
var(hw.grades)
```

```
[1] 76.29091
```

The average squared difference form our mean grade is 76.2909091. But what does that mean? We would like to get rid of the square in our units. That's what the standard deviation does. The standard deviation is the square root over the variance.

$$\sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

We get the average deviation from our mean grade (83.9090909) with the `sd()` function.

```r
sd(hw.grades)
```

```
[1] 8.734467
```

The standard deviation is much more intuitive than the variance because its units are the same as the units of the variable we are interested in. "Why teach us about this awful variance then", you ask. Mathematically, we have to compute the variance before getting the standard deviation. We recommend that you use the standard deviation to describe the variability of your continuous data.

Note: We used the sample variance and sample standard deviation formulas. If the eleven assignments represent the population, we would use the population variance formula. Whether the 11 cases represent a sample or the population depends on what we want to know. If we want learn about all students' assignments or future assignments, the 11 cases are a sample.

1.1.10.2 Range and interquartile range

The proper measure of dispersion of an ordinal variable is the range or the interquartile range. The interquartile range is usually the preferred measure because the range is strongly affected by outlying cases.

Let's take the range first. We get back to our education example. In R, we use the `range()` function to compute the range.

```r
range(edu)
```

```
[1] 0 5
```

Our data ranges from no education all the way to those with a doctorate. However, no education is not a common value. Only one person in our sample did not have any education. The interquartile range is the range from the 25th to the 75th percentiles, i.e., it contains the central 50 percent of the distribution.

The 25th percentile is the value of education that 25 percent or fewer people have (when we order education from lowest to highest). We use the `quantile()` function in R to get percentiles. The function takes two arguments: `x` is the data vector and `probs` is the percentile.

```r
quantile(edu, 0.25) # 25th percentile
```

```
25% 
  2 
```

```r
quantile(edu, 0.75) # 75th percentile
```

```
75% 
  3 
```

Therefore, the interquartile range is from 2, secondary school to 3, undergraduate degree.

1.1.10.3 Proportion in each category

To describe the distribution of our nominal variable, support for remaining in the European Union, we use the proportions in each category.

Recall, that we looked at the frequency table to determine the mode:

```r
table(stay)
```

```
stay
  0   1 
509 491 
```

To get the proportions in each category, we divide the values in the table, i.e., 509 and 491, by the sum of the table, i.e., 1000.

```r
table(stay) / sum(table(stay))
```

```
stay
    0     1 
0.509 0.491 
```

1.1.11 Exercises

1. Create a script and call it assignment01. Save your script.
2. Download this cheat-sheet and go over it. You won't understand most of it right a away. But it will become a useful resource. Look at it often.
3. Calculate the square root of 1369 using the `sqrt()` function.
4. Square the number 13 using the ^ operator.
5. What is the result of summing all numbers from 1 to 100?

We take a sample of yearly income in Berlin. The values that we got are: 19395, 22698, 40587, 25705, 26292, 42150, 29609, 12349, 18131, 20543, 37240, 28598, 29007, 26106, 19441, 42869, 29978, 5333, 32013, 20272, 14321, 22820, 14739, 17711, 18749.

6. Create the variable `income` with the values form our Berlin sample in R.
7. Describe Berlin income using the appropriate measures of central tendency and dispersion.
8. Compute the average deviation without using the `sd()` function.

Take a look at the Sunday Question (who would you vote for if the general election were next Sunday?) by following this link Sunday Question Germany. You should be able to translate the website into English by right clicking in your browser and clicking "Translate to English."

9. What is the level of measurement of the variable in the Sunday Question?
10. Take the most recent poll and describe what you see in terms of central tendency and dispersion.
11. Save your script, which should now include the answers to all the exercises.
12. Source your script, i.e. run the entire script without error message. Clean your script if you get error messages.

1.2 Solutions

1.2.1 Exercise 3

Calculate the square root of 1369 using the `sqrt()` function.

```r
sqrt(1369)
```

```
[1] 37
```

1.2.2 Exercise 4

Square the number 13 using the ^ operator.

```r
13^2
```

```
[1] 169
```

1.2.3 Exercise 5

What is the result of summing all numbers from 1 to 100?

```r
# sequence of numbers from 1 to 100 in steps of 1
numbers_1_to_100 <- seq(from = 1, to = 100, by = 1)
# sum over the vector
result <- sum(numbers_1_to_100)
# print the result
result
```

```
[1] 5050
```

The result is 5050.

1.2.4 Exercise 6

Create the variable *income* with the values form our Berlin sample in R.

```r
# create the income variable using the c() function
income <- c(
  19395, 22698, 40587, 25705, 26292, 42150, 29609, 12349, 18131,
  20543, 37240, 28598, 29007, 26106, 19441, 42869, 29978, 5333,
  32013, 20272, 14321, 22820, 14739, 17711, 18749
)
```

1.2.5 Exercise 7

Describe Berlin income using the appropriate measures of central tendency and dispersion.

We use the mean for the central tendency of *income*. The variable is interval scaled and the mean is the appropriate measure of central tendency for interval scaled variables. Our *income* variable is also normally distributed. Income distributions in most countries are right skewed. Therefore, the central tendency of income is often described using the median.

When asked, e.g., in an exam, to describe the central tendency of an interval scaled variable, use the mean. You can also use the median if you tell us why.

```r
# central tendency of income
mean(income)
```

```
[1] 24666.24
```

```r
# dispersion
sd(income)
```

```
[1] 9467.383
```

Average income in our Berlin sample is 24666.24. The average difference from that value is 9467.38.

1.2.6 Exercise 8

Compute the average deviation without using the sd() function.

We do this in several steps. First, we compute the mean.

```r
mean.income <- sum(income) / length(income)

# let's print the mean
mean.income
```

```
[1] 24666.24
```

Second, we take the differences between each individual realisation of income and the mean of *income*. The result must be a vector with the same amount of elements as the *income* vector.

```r
# individual differences between each realisation of income and the mean of income
diffs.from.mean <- income - mean.income

# let's print the vector of differences
diffs.from.mean
```

```
 [1]  -5271.24  -1968.24  15920.76   1038.76   1625.76  17483.76   4942.76
 [8] -12317.24  -6535.24  -4123.24  12573.76   3931.76   4340.76   1439.76
[15]  -5225.24  18202.76   5311.76 -19333.24   7346.76  -4394.24 -10345.24
[22]  -1846.24  -9927.24  -6955.24  -5917.24
```

You may be surprised that this works. After all, *income* is a vector with 25 elements and *mean.income* is a scalar (only one value). R treats all variables as vectors. It notices that *mean.income* is a shorter vector than *income*. The former has 1 element and the latter 25. The vector *mean.income* is recycled, so that it has the same length as *income* where each element is the same: the mean of *income*. If you did not understand this don't worry. The important thing is that it works.

Our next step is to square the differences from the mean.

```
# square each element in the diffs.from.mean vector
squared.diffs.from.mean <- diffs.from.mean^2

# print the squared vecto
squared.diffs.from.mean
```

```
 [1]  27785971   3873969 253470599   1079022   2643096 305681864  24430876
 [8] 151714401  42709362  17001108 158099441  15458737  18842197   2072909
[15]  27303133 331340472  28214794 373774169  53974882  19309345 107023991
[22]   3408602  98550094  48375363  35013729
```

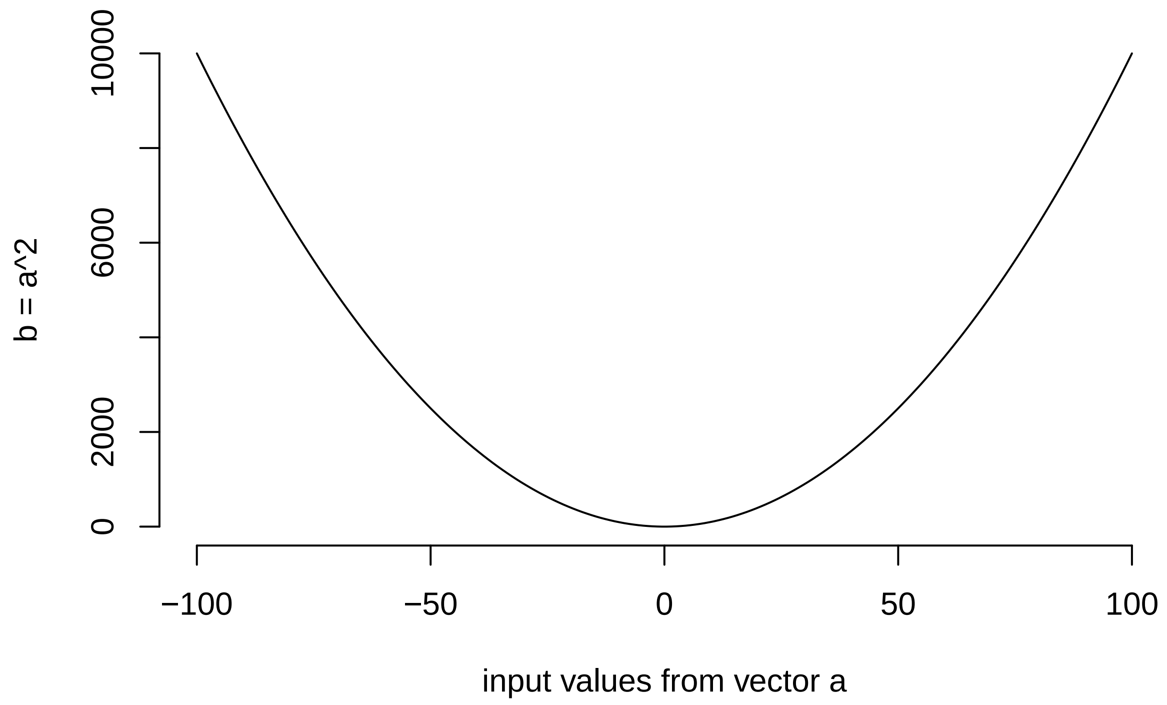
We squared each individual element in the vector. Therefore, our new variable *squared.diffs.from.mean* still has 25 elements.

Squaring a value does two things. First, all values in our vector have become positive. Second, the marginal increase increases with distance, i.e., values that are close to the mean are only somewhat larger whereas values that are further from the mean become way larger. To see this, lets plot the square (we haven't shown you the plot function yet, but we will do this next seminar).

```
# a vector of x values from negative 100 to positive 100
a <- seq(from = -100, to = 100, length.out = 200)

# the square of that vector
b <- a^2

# we plot the input vector a against b, where b is on the y-axis
plot(
  x = a, # x-axis values
  y = b, # y-axis values
  bty = "n", # no border around plot
  type = "l", # connect individual dots to a line
  xlab = "input values from vector a", # x axis label
  ylab = "b = a^2" # y axis label
)
```



In this plot, you should see that the slope of the line increases, the further we are from 0. We are taking individual differences from the mean. Hence, if a value is exactly at the mean, the difference is zero. The further, the value is from the mean (in any direction), the larger the output value.

We will sum over the individual elements in the next step. Hence, values that are further from the mean have a larger impact on the sum than values that are closer to the mean.

In the next step, we take the sum over our squared deviations from the mean

```
# sum over squared deviations vector
sum.of.squared.deviations <- sum(squared.diffs.from.mean)

# print the sum
sum.of.squared.deviations
```

```
[1] 2151152127
```

By summing over all elements of a vector, we end up with a scalar. The sum is 2151152126.56.

We divide the sum of squared deviations by $n - 1$. Recall, that n is the number of observations (elements in the vector) and -1 is our sample adjustment.

```
# get the variance
var.income <- sum.of.squared.deviations / ( length(income) - 1 )

# print the variance
var.income
```

```
[1] 89631339
```

The squared average deviation from mean income is 89631338.61.

In the last step, we take the square root over the variance to return to our original units of income.

```r
# get the standard deviation
sqrt(var.income)
```

```
[1] 9467.383
```

The average deviation from mean income in Berlin (24666.24) is 9467.38.

1.2.7 Exercise 9

What is the level of measurement of the variable in the Sunday Question?

The variable measures vote choice. The answers are categories, the parties, without any specific ordering. The level of measurement is called categorical or nominal.

1.2.8 Exercise 10

Take the most recent poll and describe what you see in terms of central tendency and dispersion.

The most recent poll was carried out by Infratest/dimap on Thursday, 6 September. The most common value, the mode, is the appropriate measure of central tendency. Christian Democrat (CDU/CSU) is the modal category. Dispersion of a categorical variable is the proportion in each category which we see displayed on the website:

Party	Proportion
CDU/CSU	0.29
SPD	0.18
GREEN	0.14
FDP	0.08
THE LEFT	0.10
AFD	0.16
other	0.05

Chapter 2

Research Design, Counterfactuals, Forming Hypotheses

2.1 Seminar

In today's seminar, we work with data frames (datasets). We will create our own dataset, we subset datasets (access elements, rows and variables). We load our first dataset into R. We also visualise data using the `plot()` function. Finally, we estimate a treatment effect in R—our first inference.

2.1.1 setting up

We set our working directory. R operates in specific directory (folder) on our computer. We create a folder on our computer where we save our scripts for our statistics 1 class. We name the folder `stats1`. Let's create the folder on our computers now (in finder on Mac and explorer on Windows).

Now, we set our working directory to the folder, we just created like so:

Create a new R script and save it as week2.R to your `stats1` directory. Now type the following commands in the new file you just created:

```
# Create a numeric and a character variable
a <- 5 # numeric
a <- "five" # character
```

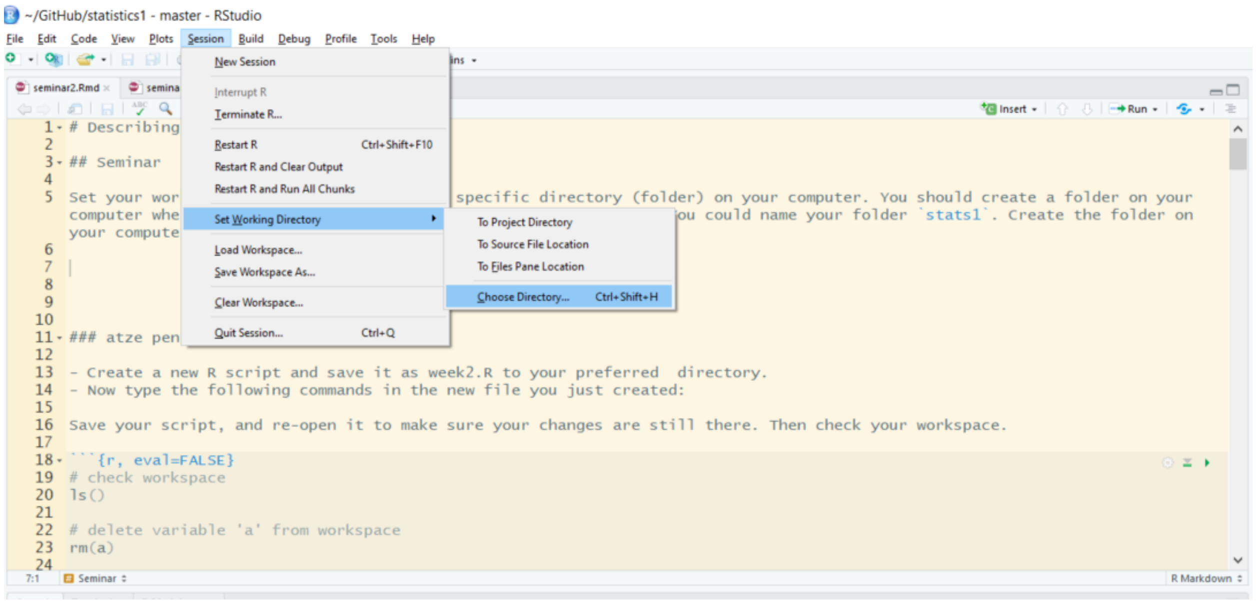
Save your script, and re-open it to make sure your changes are still there. Then check your workspace.

```
# check workspace
ls()

# delete variable 'a' from workspace
rm(a)

# delete everything from workspace
rm( list = ls() )

# to clear console window press Crtl+l on Win or Command+l on Mac
```



~~~
# Describing
## Seminar

Set your wor                                   specific directory (folder) on your computer. You should create a folder on your
computer whe                                   ou could name your folder `stats1`. Create the folder on
your compute



### atze pen

- Create a new R script and save it as week2.R to your preferred  directory.
- Now type the following commands in the new file you just created:

Save your script, and re-open it to make sure your changes are still there. Then check your workspace.

```{r, eval=FALSE}
# check workspace
ls()

# delete variable 'a' from workspace
rm(a)
~~~

Figure 2.1:

2.1.2 vectors and subsetting

Last week we have already worked with vectors. We created a sequence for example. This week, we learn about subsetting (accessing specific elements of our vector).

We create a vector using the c() function, where c stands for collect.

```r
# Create a vector
my.vector <- c(10,7,99,34,0,5) # a vector
my.vector
```

```
[1] 10  7 99 34  0  5
```

Let's see how many elements our vector contains using the length() function.

```r
length(my.vector) # how many elements?
```

```
[1] 6
```

Next, we access the first element in our vector. We use square brackets to access a specific element. The number in the square brackets is the vector element that we access

```r
# subsetting
my.vector[1] # 1st vector element
```

```
[1] 10
```

To access all elements except the first element, we use the - operator.

```r
my.vector[-1] # all elements but the 1st
```

```
[1]  7 99 34  0  5
```

We can access elements 2 to 4 by using the colon.

```r
my.vector[2:4] # the 2nd to the 4th elements
```

```
[1]  7 99 34
```

We can access two specific non-adjacent elements, by using the collect function `c()`.

```
my.vector[c(2,5)] # 2nd and 5th element
```

```
[1] 7 0
```

No, we combine the `length()` function with the square brackets to access the last element in our vector.

```
my.vector[length(my.vector)] # the last element
```

```
[1] 5
```

2.1.3 data frames

A data frame is an object that holds data in a tabular format similar to how spreadsheets work. Variables are generally kept in columns and observations are in rows.

Before we work with ready-made data, we create a small dataset ourselves. It contains the populations of the sixteen German states. We start with a vector that contains the names of those states. We call the variable *state*. Our variable shall contain text instead of numbers. In R jargon, this is a character variable, sometimes referred to as a string. Using quotes, we indicate that the variable type is character. We use the `c()` function to create the vector.

```
# create a character vector containing state names
state <- c(
  "North Rhine-Westphalia",
  "Bavaria",
  "Baden-Wurttemberg",
  "Lower Saxony",
  "Hesse",
  "Saxony",
  "Rhineland-Palatinate",
  "Berlin",
  "Schleswig-Holstein",
  "Brandenburg",
  "Saxony-Anhalt",
  "Thuringia",
  "Hamburg",
  "Mecklenburg-Vorpommern",
  "Saarland",
  "Bremen"
  )
```

Now, we create a second variable for the populations. This is a numeric vector, so we do not use the quotes.

```
population <- c(
  17865516,
  12843514,
  10879618,
  7926599,
  6176172,
  4084851,
  4052803,
  3670622,
  2858714,
  2484826,
  2245470,
  2170714,
  1787408,
  1612362,
  995597,
  671489
)
```

Now with both vectors created, we combine them into a dataframe. We put our vectors in and give them names. In this case the variable names in the dataset correspond to our vector names. The name goes in front of the equal sign and the vector object name, after.

```
popdata <- data.frame(
  state = state,
  population = population
  )
```

You should see the new data frame object in your global environment window. You can view the dataset in the spreadsheet form that we are all used to by clicking on the oject name.

We can see the names of variables in our dataset with the names function

```
names(popdata)
```

```
[1] "state"      "population"
```

Let's check the variable types in our data using the `str()` function.

```
str(popdata)
```

```
'data.frame':   16 obs. of  2 variables:
 $ state     : Factor w/ 16 levels "Baden-Wurttemberg",..: 10 2 1 8 7 13 11 3 15 4 ...
 $ population: num  17865516 12843514 10879618 7926599 6176172 ...
```

The variable *state* is a factor variable. R has turned the character variable into a categorical variable automatically. The variable *population* is numeric. These variable types differ. We can calculate with numeric variables only.

Often we want to access certain observations (rows) or certain columns (variables) or a combination of the two without looking at the entire dataset all at once. We can use square brackets to subset data frames. In square brackets we put a row and a column coordinate separated by a comma. The row coordinate goes first and the column coordinate second. So `popdata[10, 2]` returns the 10th row and second column of the data frame. If we leave the column coordinate empty this means we would like all columns. So, `popdata[10,]` returns the 10th row of the dataset. If we leave the row coordinate empty, R returns the entire column. `popdata[,2]` returns the second column of the dataset.

We can look at the first five rows of a dataset to get a better understanding of it with the colon in brackets like so: `popdata[1:5,]`. We could display the second and fifth columns of the dataset by using the `c()` function in brackets like so: `popdata[, c(2,5)]`.

It's your turn. Display all columns of the popdata dataset and show rows 10 to 15. Next display all columns of the dataset and rows 4 and 7.

```
popdata[10:15, ] # elements in 10th to 15th row, all columns
```

```
                    state population
10            Brandenburg    2484826
11          Saxony-Anhalt    2245470
12              Thuringia    2170714
13                Hamburg    1787408
14 Mecklenburg-Vorpommern    1612362
```

```
15                Saarland     995597
```

```
popdata[c(4, 7), ] # elements in 4th and 7th row, all column
```

```
                 state population
4         Lower Saxony    7926599
7 Rhineland-Palatinate    4052803
```

In order to access individual columns of a data frame we can also use the dollar sign `$`. For example, let's see how to access the `population` column.

```
popdata$population
```

```
 [1] 17865516 12843514 10879618  7926599  6176172  4084851  4052803
 [8]  3670622  2858714  2484826  2245470  2170714  1787408  1612362
[15]   995597   671489
```

Now, access the state column.

```
popdata$state
```

```
 [1] North Rhine-Westphalia Bavaria                Baden-Wurttemberg
 [4] Lower Saxony           Hesse                  Saxony
 [7] Rhineland-Palatinate   Berlin                 Schleswig-Holstein
[10] Brandenburg            Saxony-Anhalt          Thuringia
[13] Hamburg                Mecklenburg-Vorpommern Saarland
[16] Bremen
16 Levels: Baden-Wurttemberg Bavaria Berlin Brandenburg Bremen ... Thuringia
```

2.1.4 Loading data

Before you load the dataset into R, you first download it and save it locally in your `Stats1` folder. Download the data here.

We often load existing data sets into R for analysis. Data come in many different file formats such as `.csv`, `.tab`, `.dta`, etc. Today we will load a dataset which is stored in R's native file format: `.RData`. The function to load data from this file format is called: `load()`. If you managed to set your working directory correctly just now (`setwd("~/Stats1")`), then you should just be able to run the line of code below.

We load the dataset with the `load()` function:

```
# load perception of non-western foreigners data
load("BSAS_manip.RData")
```

The non-western foreingers data is about the subjective perception of immigrants from non-western countries. The perception of immigrants from a context that is not similar to the one's own ,is often used as a proxy for racism. Whether this is a fair measure or not is debatable but let's examine the data from a survey carried out in Britain.

Let's check the codebook of our data.

Variable	Description
IMMBRIT	Out of every 100 people in Britain, how many do you think are immigrants from non-western countr
over.estimate	1 if estimate is higher than 10.7%.
RSex	1 = male, 2 = female
RAge	Age of respondent
Househld	Number of people living in respondent's household
party identification	1 = Conservatives, 2 = Labour, 3 = SNP, 4 = Greens, 5 = Ukip, 6 = BNP, 7 = other
paper	Do you normally read any daily morning newspaper 3+ times/week?
WWWhourspW	How many hours WWW per week?
religious	Do you regard yourself as belonging to any particular religion?
employMonths	How many mnths w. present employer?
urban	Population density, 4 categories (highest density is 4, lowest is 1)
health.good	How is your health in general for someone of your age? (0: bad, 1: fair, 2: fairly good, 3: good)
HHInc	Income bands for household, high number = high HH income

We can look at the variable names in our data with the `names()` function.

The `dim()` function can be used to find out the dimensions of the dataset (dimension 1 = rows, dimension 2 = columns).

```
dim(data2)
```

```
[1] 1049   19
```

So, the `dim()` function tells us that we have data from 1049 respondents with 19 variables for each respondent.

Let's take a quick peek at the first 10 observations to see what the dataset looks like. By default the `head()` function returns the first 6 rows, but let's tell it to return the first 10 rows instead.

```
head(data2, n = 10)
```

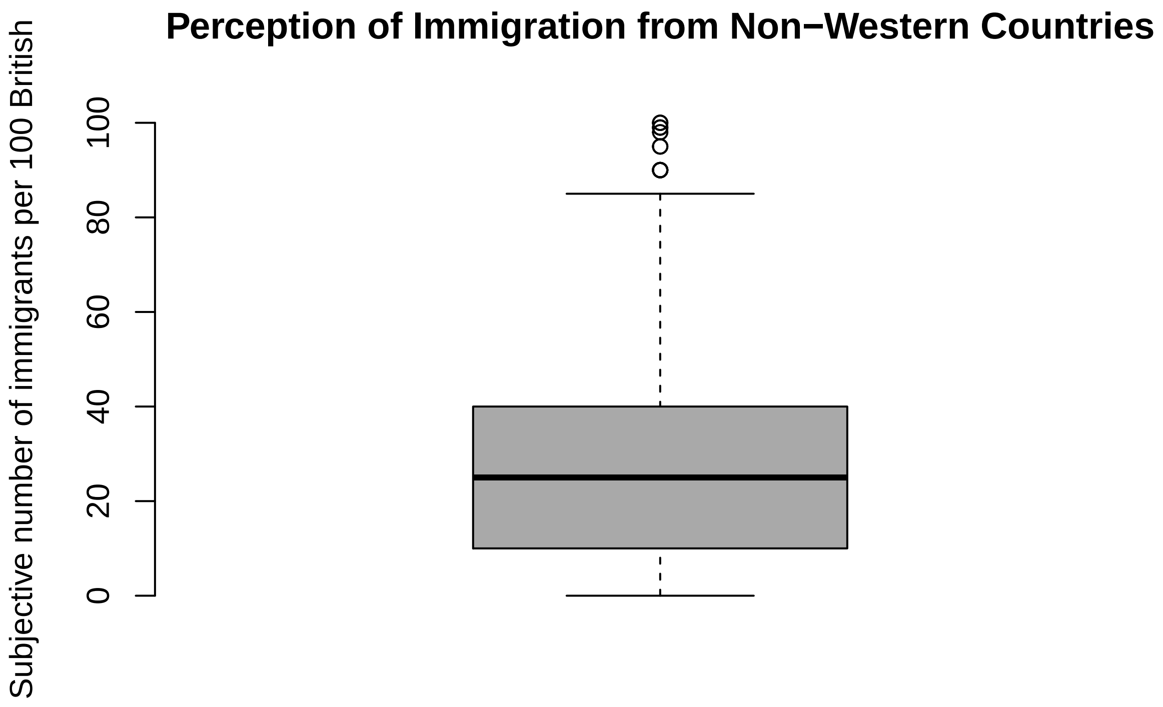
```
   IMMBRIT over.estimate RSex RAge Househld Cons Lab SNP Ukip BNP GP
1        1             0    1   50        2    0   1   0    0   0  0
2       50             1    2   18        3    0   0   0    0   0  0
3       50             1    2   60        1    0   0   0    0   0  0
4       15             1    2   77        2    0   0   0    0   0  0
5       20             1    2   67        1    0   0   0    0   0  0
6       30             1    1   30        4    0   0   0    0   0  0
7       60             1    2   56        2    0   0   1    0   0  0
8        7             0    1   49        1    0   0   0    0   0  0
9       30             1    1   40        4    0   0   1    0   0  0
10       2             0    1   61        3    1   0   0    0   0  0
   party.other paper WWWhourspW religious employMonths urban health.good
1            0     0          1         0           72     4           1
2            1     0          4         0           72     4           2
3            1     0          1         0          456     3           3
4            1     1          2         1           72     1           3
5            1     0          1         1           72     3           3
6            1     1         14         0           72     1           2
7            0     0          5         1          180     1           2
8            1     1          8         0          156     4           2
9            0     0          3         1          264     2           2
10           0     1          0         1           72     1           3
   HHInc
1     13
2      3
3      9
```

```
4       8
5       9
6       9
7      13
8      14
9      11
10      8
```

2.1.5 Plots

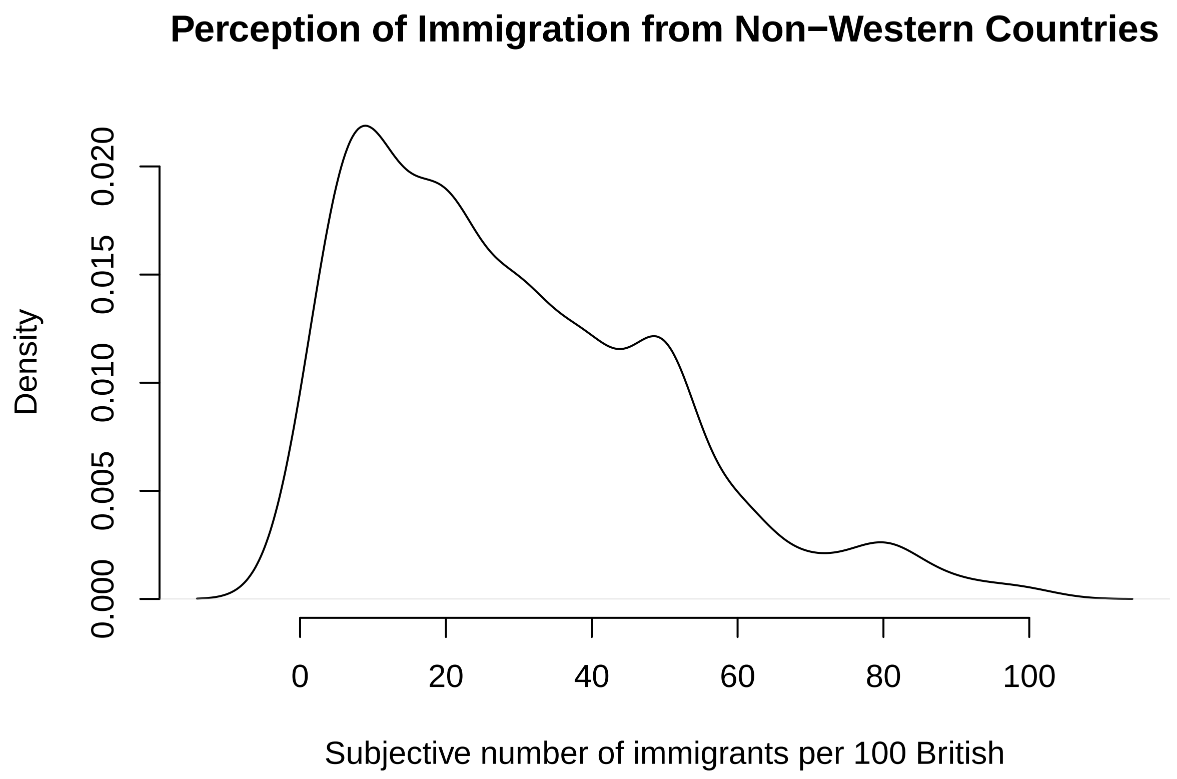
We can visualize the data with the help of a boxplot, so let's see how the perception of the number of immigrants is distributed.

```
# how good are we at guessing immigration
boxplot(
  data2$IMMBRIT,
  main = "Perception of Immigration from Non-Western Countries",
  ylab = "Subjective number of immigrants per 100 British",
  frame.plot = FALSE, col = "darkgray"
  )
```



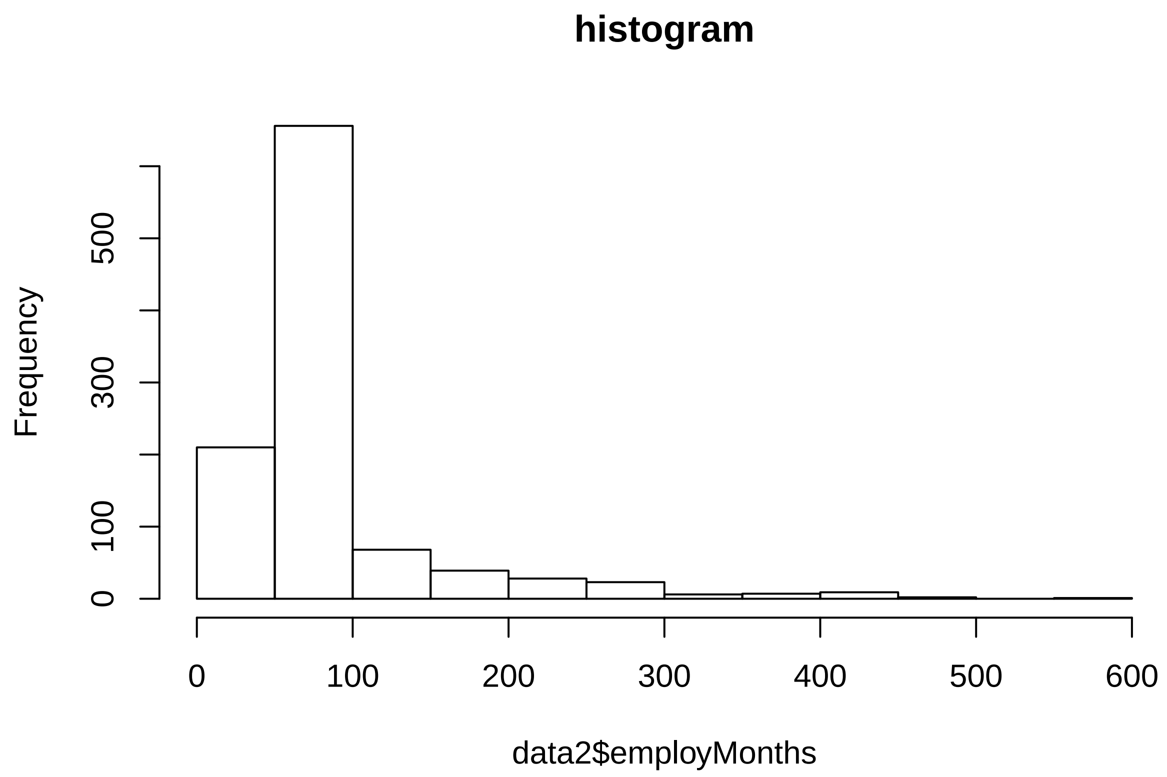
Notice how the lower whisker is much shorter than the upper one. The distribution is right skewed. The right tail (higher values) is a lot longer. We can see this beter using a density plot. We combine R's `denisty()` function with the `plot()` function.

```
plot(
  density(data2$IMMBRIT),
  bty = "n",
  main = "Perception of Immigration from Non-Western Countries",
  xlab = "Subjective number of immigrants per 100 British"
  )
```



We can also plot histograms using the `hist()` function.

```
# histogram
hist( data2$employMonths, main = "histogram")
```



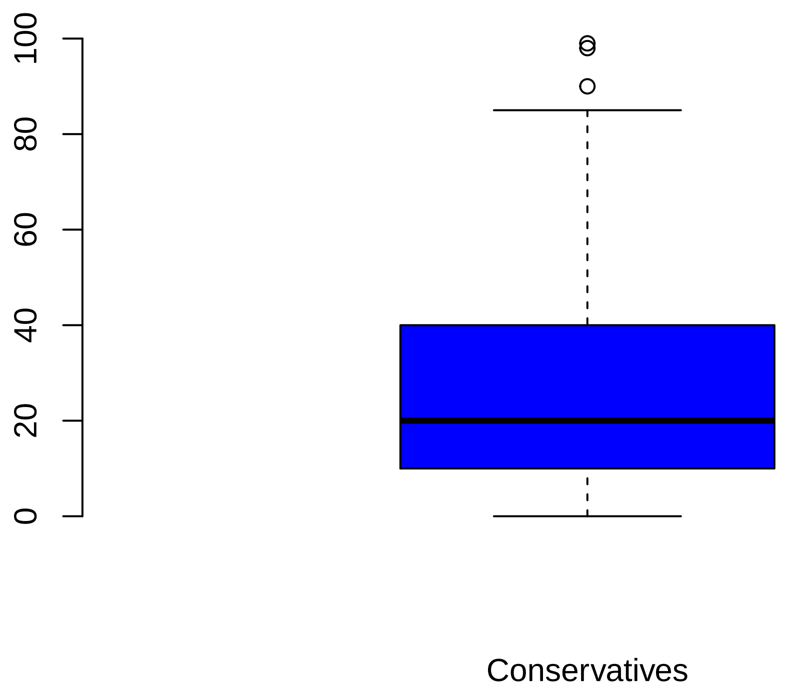
It is plausible that perception of immigration from Non-Western countries is related to party affiliation. In our dataset, we have a some party affiliation dummies (binary variables). We can use square brackets to subset our data such that we produce a boxplot only for members of the Conservative Party. We have a look at the variable *Cons* using the `table()` function first.

```r
table(data2$Cons)
```

```
  0   1 
765 284 
```

In our data, 284 respondents associate with the Conservative party and 765 do not. We create a boxplot of *IMMBRIT* but only for members of the Conservative Party. We do so by using the square brackets to subset our data.

```r
# boxplot of immbrit for those observations where Cons is 1
boxplot(
  data2$IMMBRIT[data2$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )
```

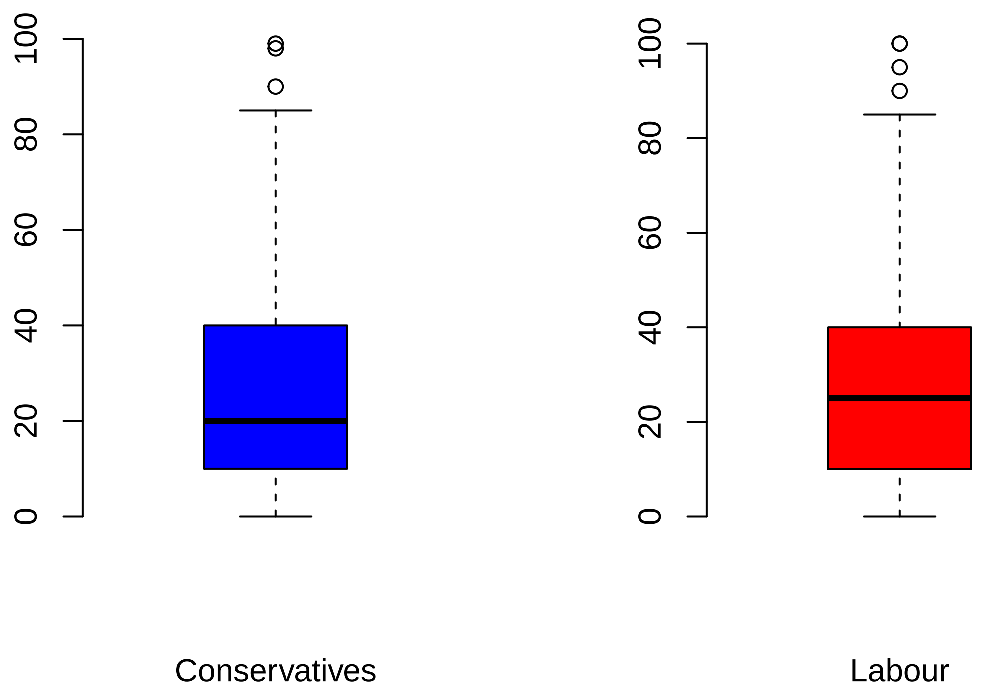


We would now like to compare the distribution of the perception fo Conservatives to the distribution among Labour respondents. We can subset the data just like we did for the Conservative Party. In addtion, we want to plot the two plots next to each other, i.e., they should be in the same plot. We can achieve this with the `par()` function and the `mfrow` argument. This will spilt the plot window into rows and columns. We want 2 columns to plot 2 boxplots next to each other.

```
# split plot window into 1 row and 2 columns
par(mfrow = c(1,2))

# plot 1
boxplot(
  data2$IMMBRIT[data2$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )

# plot 2
boxplot(
  data2$IMMBRIT[data2$Lab==1],
  frame.plot = FALSE,
  xlab = "Labour",
  col = "red"
  )
```



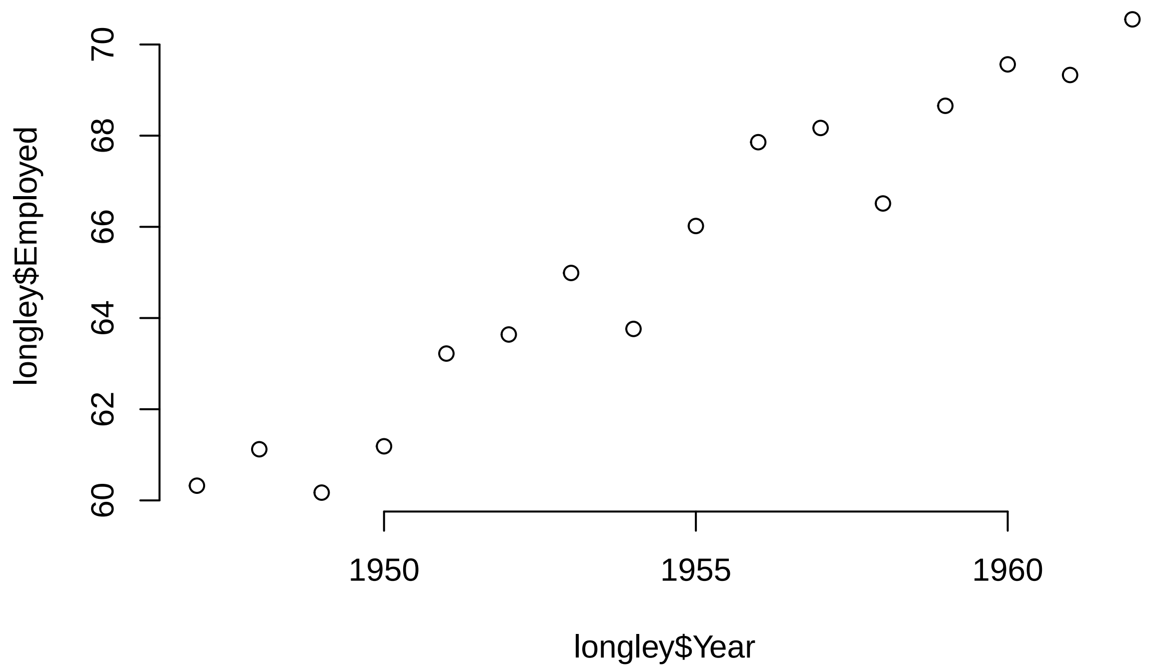
It is very hard to spot differences. The distributions are similar. The median for Labour respondents is larger which mean that the central Labour respondent over-estimates immigration more than the central Conservative respondent.

You can play around with the non-western foreigners data on your own time. We now turn to a dataset that is integrated in R already. It is called `longley`. Use the `help()` function to see what this dataset is about.

```r
help(longley)
```

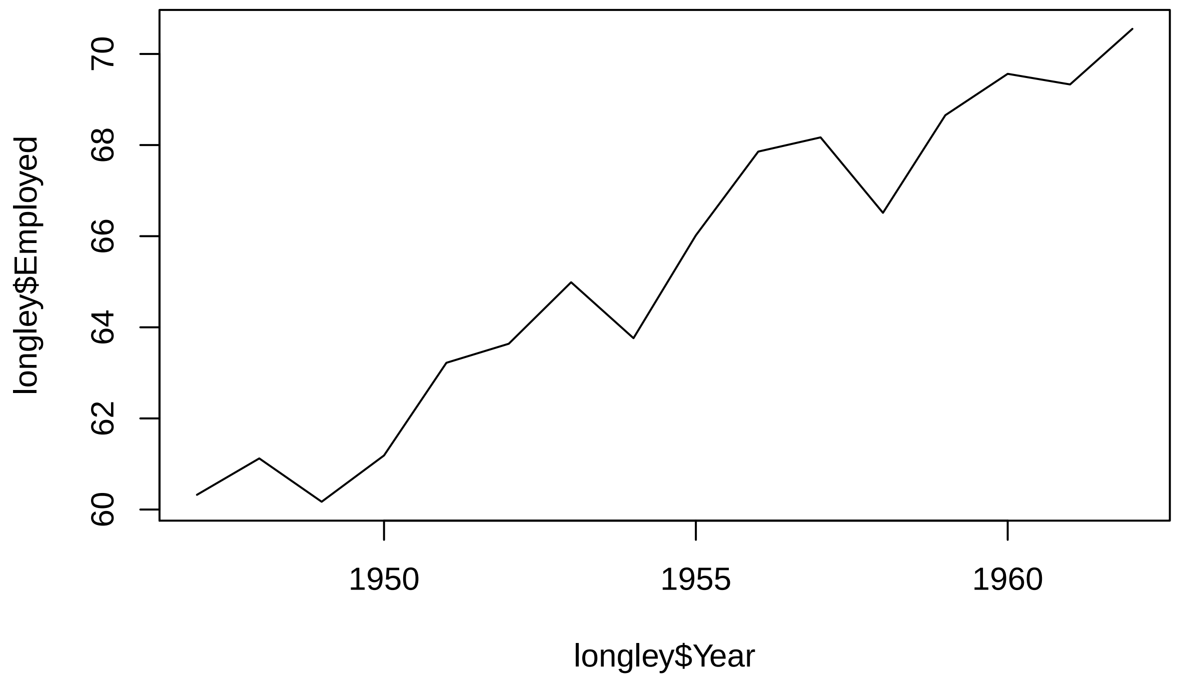
Let's create a scatterplot with the `Year` variable on the x-axis and `Employed` on the y-axis.

```r
plot(x = longley$Year, # x-axis variable
     y = longley$Employed, # y-axis variable
     bty = "n" # no box around the plot
     )
```



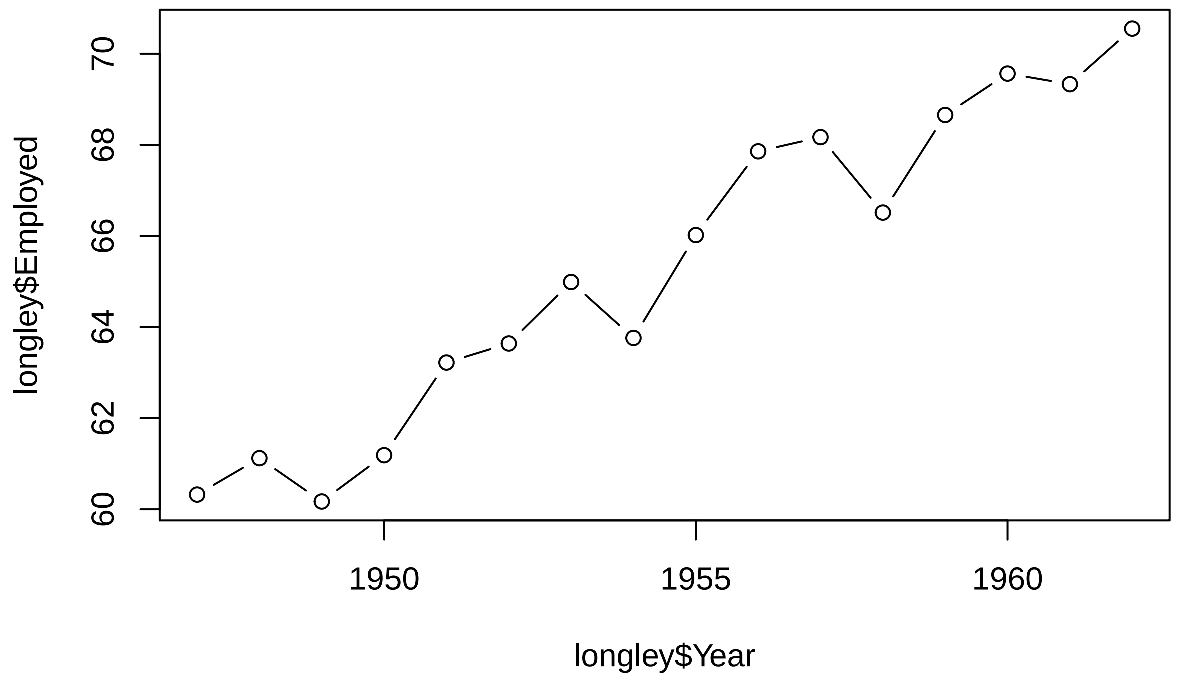
To create a line plot instead, we use the same function with one additional argument `type = "l"`.

```
plot(longley$Year, longley$Employed, type = "l")
```



Create a plot that includes both points and lines.

```
plot(longley$Year, longley$Employed, type = "b")
```



2.1.6 Average Treatment Effect

In the lecture, we estimated the average treatment effect on a small example. We will do this again here. Recall, that the average treatment effect is the difference between two means.

Let's suppose, associating with right-wing parties causes people to over-estimate the number of non-western foreigners. Our treatment variable is whether a respondent assoicates with the UK Independence Party. It is 1 if that is the case and 0 otherwise. Let's inspect the variable *Ukip*.

```
table(data2$Ukip)
```

```

   0    1 
1018   31 
```

31 respondents identify with Ukip.

The average treatment effect, as we learned, would be the difference between the mean outcomes for those who received the treament minus the mean for those who did not reicive the treatment.

We have all the tools to solve the problem. Let's take the mean of the treated group first.

```
mean.y.treated <- mean(data2$IMMBRIT[data2$Ukip == 1])
mean.y.treated
```

```
[1] 24.29032
```

The double equal sign `==` is a logical operator and means "is equal to". R returns true or false depending on whether the respondent does identify with Ukip or not. The mean of *IMMBRIT* is then computed only for respondents who accociate with Ukip.

Let's take the mean of the second group, the untreated group.

```
mean.y.untreated <- mean(data2$IMMBRIT[data2$Ukip == 0])
mean.y.untreated
```

```
[1] 29.17485
```

The treatment effect is the difference in means:

```
mean.y.treated - mean.y.untreated
```

```
[1] -4.88453
```

The result is surprising. Ukip members over-estimate the number of non-western foreigners less members of all other paries. Our claim is not quite supported by the data. We should be very careful with these results, however. We used experimental language but our data is observational. A multitude of confounders could bias our estimate of the causal effect.

2.1.7 Exercises

1. Create a script and call it assignment02. Save your script.
2. Use the `names()` function to display the variable names of the `longley` dataset.
3. Use square brackets to access the 4th column of the dataset.
4. Use the dollar sign to access the 4th column of the dataset.
5. Access the two cells from row 4 and column 1 and row 6 and column 3.
6. Using the `longley` data produce a line plot with GNP on the y-axis and population on the x-axis.
7. Use the help function to find out how to label the y-axis "wealth" and the x-axis "population".
8. Create a boxplot showing the distribution of *IMMBRIT* by each party in the data and plot these in one plot next to each other.
9. Is there a difference between women and men in terms of their subjective estimation of foreingers?
10. What is the difference between women and men?
11. Could you form a hypothesis out of the relationship that you see if any exists?
12. Save your script, which should now include the answers to all the exercises.
13. Source your script, i.e. run the entire script without error message. Clean your script if you get error messages.

2.2 Solutions

2.2.1 Exercise 2

Use the `names()` function to display the variable names of the `longley` dataset.

```
names(longley)
```

```
[1] "GNP.deflator" "GNP"          "Unemployed"   "Armed.Forces"
[5] "Population"   "Year"         "Employed"
```

2.2.2 Exercise 3

Use square brackets to access the 4th column of the dataset.

```
longley[, 4]
```

```
 [1] 159.0 145.6 161.6 165.0 309.9 359.4 354.7 335.0 304.8 285.7 279.8
[12] 263.7 255.2 251.4 257.2 282.7
```

2.2.3 Exercise 4

Use the dollar sign to access the 4th column of the dataset.

```
longley$Armed.Forces
```

```
 [1] 159.0 145.6 161.6 165.0 309.9 359.4 354.7 335.0 304.8 285.7 279.8
[12] 263.7 255.2 251.4 257.2 282.7
```

Note: There is yet another way to access the 4th column of the dataset. We can put the variable name into the square brackets using quotes like so:

```
longley[, "Armed.Forces"]
```

```
 [1] 159.0 145.6 161.6 165.0 309.9 359.4 354.7 335.0 304.8 285.7 279.8
[12] 263.7 255.2 251.4 257.2 282.7
```

2.2.4 Exercise 5

Access the two cells from row 4 and column 1 and row 6 and column 3.

```
# row 4, column 1
longley[4, 1]
```

```
[1] 89.5
```

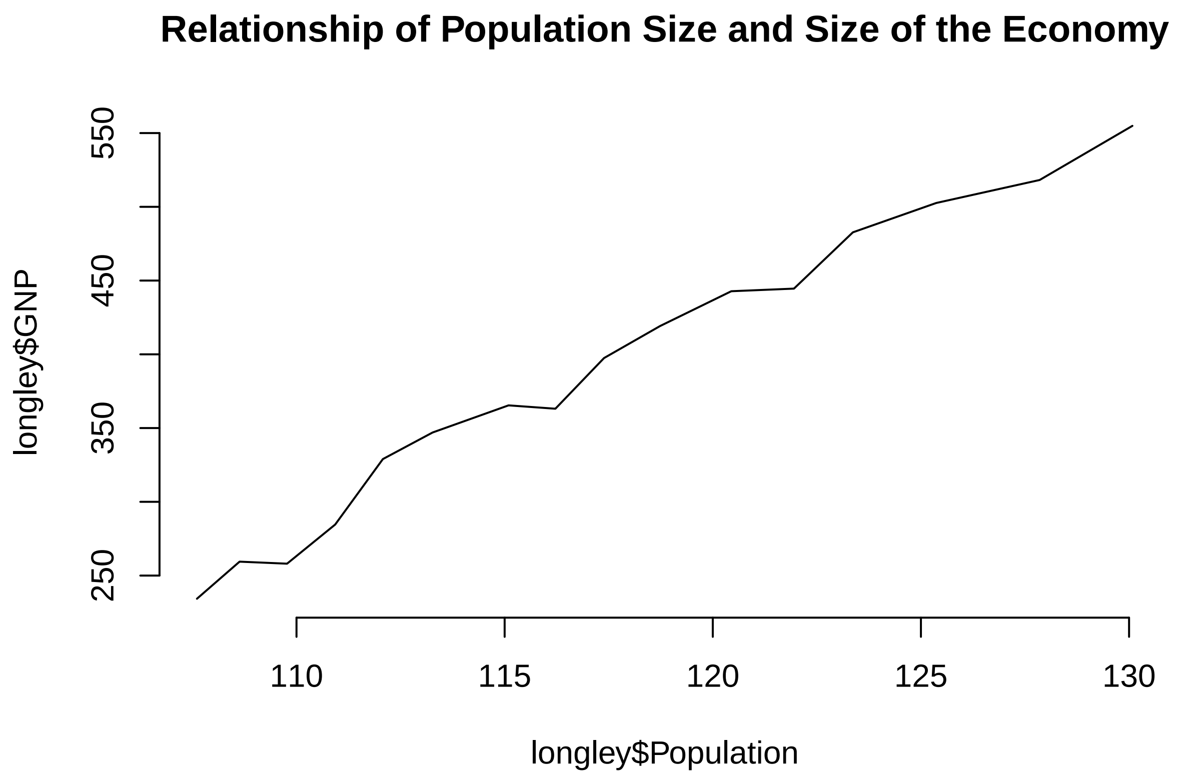
```
# row 6, column 3
longley[6, 3]
```

```
[1] 193.2
```

2.2.5 Exercise 6

Using the `longley` data produce a line plot with GNP on the y-axis and population on the x-axis.

```
plot(
  y = longley$GNP, # y-axis variable
  x = longley$Population, # x-axis variable
  type = "l", # produce a line plot
  bty = "n", # no box around our plot
  main = "Relationship of Population Size and Size of the Economy"
)
```



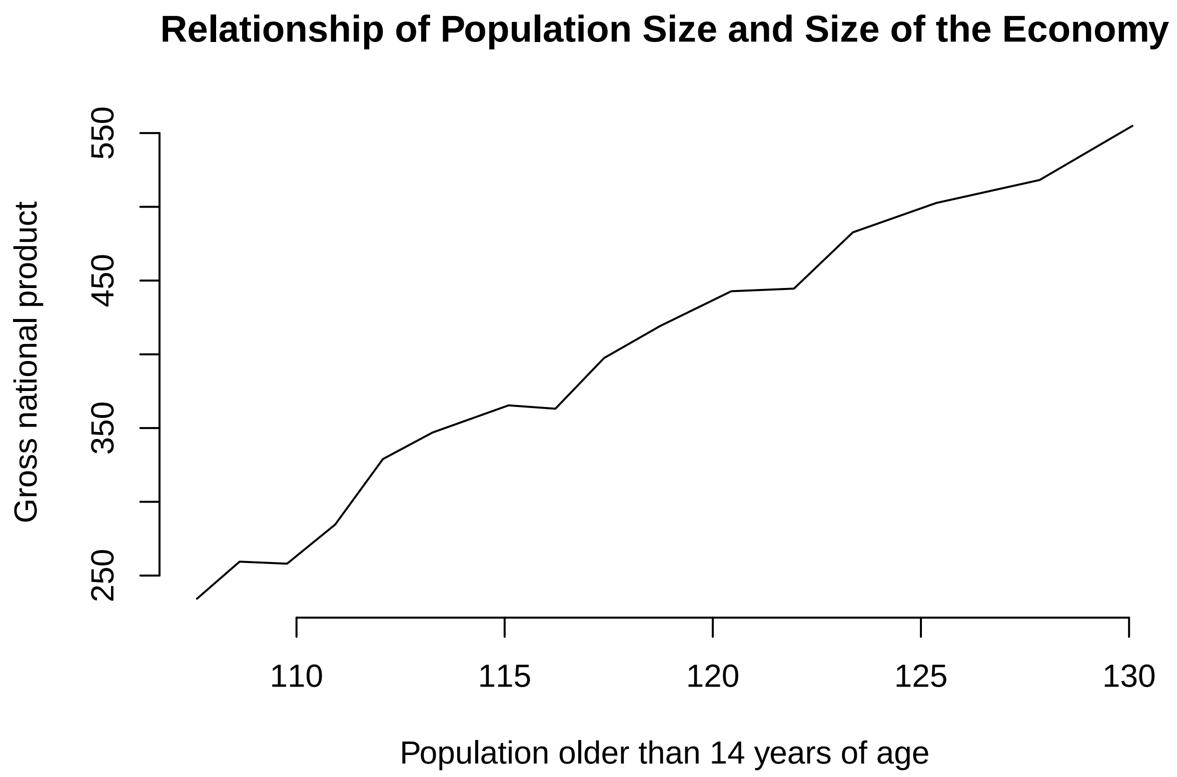
2.2.6 Exercise 7

Use the help function to find out how to label the y-axis "wealth" and the x-axis "population".

```r
?plot
```

The `?` is short for the `help()` function. We see that the `xlab` argument lets us label the x-axis and the `ylab` argument lets us label the y-axis. We do so below.

```r
plot(
  y = longley$GNP, # y-axis variable
  x = longley$Population, # x-axis variable
  type = "l", # produce a line plot
  bty = "n", # no box around our plot
  main = "Relationship of Population Size and Size of the Economy",
  xlab = "Population older than 14 years of age",
  ylab = "Gross national product"
)
```



2.2.7 Exercise 8

Create a boxplot showing the distribution of *IMMBRIT* by each party in the data and plot these in one plot next to each other.

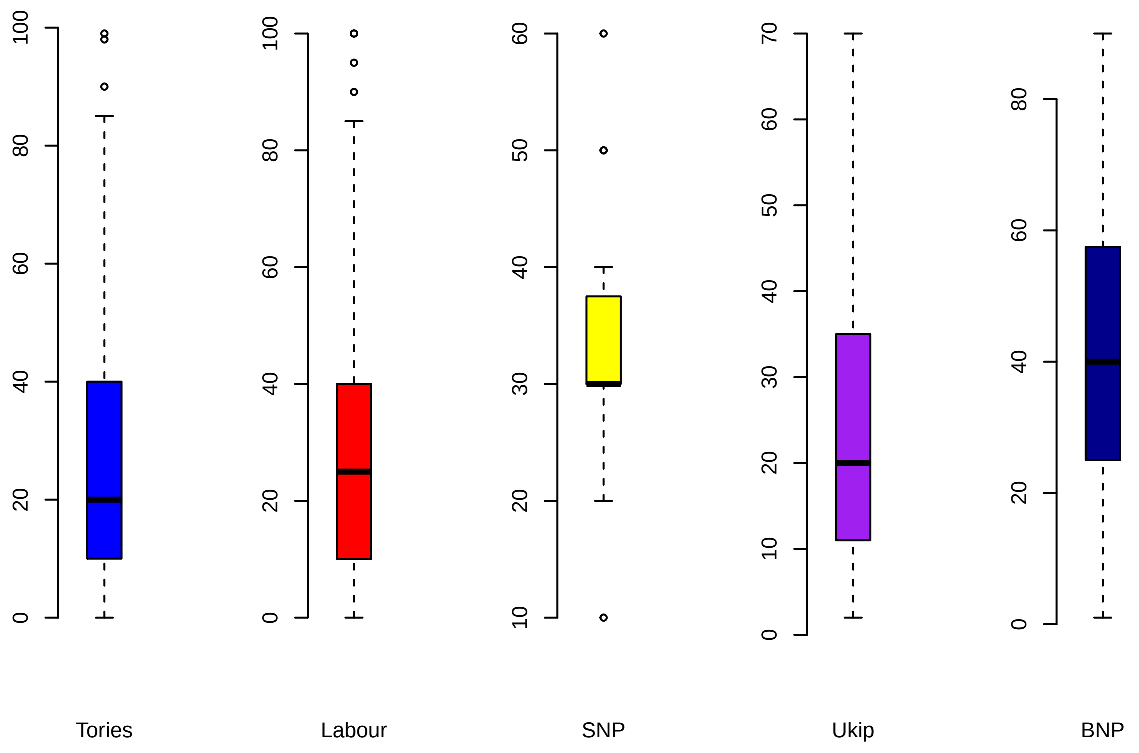
To do that, we load the non-western foreigners dataset first.

Note: You have to set your working directory that R operates in to the location of the dataset.

```r
# load perception of non-western foreigners data
load("BSAS_manip.RData")
```

We have five parties in our dataset. We plot 5 boxplots next to each other. Hence, we separate the plot window into 1 row and 5 columns.

```r
# plot window to 1 row and 5 columns
par(mfrow = c(1, 5))
boxplot(data2$IMMBRIT[ data2$Cons == 1 ], frame.plot = FALSE, col = "blue", xlab = "Tories")
boxplot(data2$IMMBRIT[ data2$Lab == 1 ], frame.plot = FALSE, col = "red", xlab = "Labour")
boxplot(data2$IMMBRIT[ data2$SNP == 1 ], frame.plot = FALSE, col = "yellow", xlab = "SNP")
boxplot(data2$IMMBRIT[ data2$Ukip == 1 ], frame.plot = FALSE, col = "purple", xlab = "Ukip")
boxplot(data2$IMMBRIT[ data2$BNP == 1 ], frame.plot = FALSE, col = "darkblue", xlab = "BNP")
```



2.2.8 Exercises 9 and 10

We combine the answer to questions 9 and 10.

Question from 9: Is there a difference between women and men in terms of their subjective estimation of foreingers?

Question from 10: What is the difference between women and men?

Women's subjective estimate is the mean of *IMMBRIT* across women and equally, men's subjective estimate is the mean of *IMMBRIT* over all men. Let's get these numbers with the mean function and the square brackets.

```
womens.mean <- mean(data2$IMMBRIT[ data2$RSex == 2 ])
womens.mean
```

```
[1] 32.79159
```

```
mens.mean <- mean(data2$IMMBRIT[ data2$RSex == 1 ])
mens.mean
```

```
[1] 24.53766
```

The difference between women and men is the difference in means. Let's take the difference between them. The difference in means is often referred to as the first difference.

```
first.difference <- womens.mean - mens.mean
first.difference
```

```
[1] 8.253937
```

Let's round that number. We don't like to see so many decimal places. You should usually present precision up to the second decimal place. We can use the `round()` function. The first argument is number to round and the second is the amount of digits.

```r
round(first.difference, 2)
```

```
[1] 8.25
```

We do find a difference between men and women. On average, women's estimate of the number of non-western foreingers is 8.25 greater than men's estimate.

At this point we have established that there is a difference in our sample. Samples are subject to sampling variability. That means, we cannot yet say that the difference is systematic, i.e., British women, generally, think that there are more non-western foreingers than British men.

2.2.9 Exercises 11

Could you form a hypothesis out of the relationship that you see if any exists?

Our testable hypothesis could be: Women tend to overestimate the number of foreigners more than men. In our sample, women tend to estimate on the number of foreingers at

Chapter 3

Sampling and Distributions

3.1 Seminar

In today's seminar, we work with missing data. We will turn a numerical variable into a nominal data type. We then turn to distributions.

```
rm(list=ls())
setwd("~/PUBLG100")
```

3.1.1 Loading Dataset in CSV Format

In this seminar, we load a file in comma separated format (`.csv`). The `load()` function from last week works only for the native R file format. To load our csv-file, we use the `read.csv()` function.

Our data comes from the Quality of Government Institute. Let's have a look at the codebook:

Download the data here.

Or download the data here

Variable	Description
h_j	1 if Free Judiciary
wdi_gdpc	Per capita wealth in US dollars
undp_hdi	Human development index (higher values = higher quality of life)
wbgi_cce	Control of corruption index (higher values = more control of corruption)
wbgi_pse	Political stability index (higher values = more stable)
former_col	1 = country was a colony once
lp_lat_abst	Latitude of country's captial divided by 90

```
world.data <- read.csv("QoG2012.csv")
```

Go ahead and (1) check the dimensions of `world.data`, (2) the names of the variables of the dataset, (3) print the first six rows of the dataset. (

```
# the dimensions: rows (observations) and columns (variables)
dim(world.data)
```

```
[1] 194   7
```

```
# the variable names
names(world.data)
```

```
[1] "h_j"          "wdi_gdpc"     "undp_hdi"     "wbgi_cce"     "wbgi_pse"
[6] "former_col"  "lp_lat_abst"
```

```
# top 6 rows of the data
head(world.data)
```

```
  h_j   wdi_gdpc undp_hdi   wbgi_cce   wbgi_pse former_col lp_lat_abst
1   0   628.4074       NA -1.5453584 -1.9343837          0   0.3666667
2   0  4954.1982    0.781 -0.8538115 -0.6026081          0   0.4555556
3   0  6349.7207    0.704 -0.7301510 -1.7336243          1   0.3111111
4  NA         NA       NA  1.3267342  1.1980436          0   0.4700000
5   0  2856.7517    0.381 -1.2065741 -1.4150945          1   0.1366667
6  NA 13981.9795    0.800  0.8624368  0.7084046          1   0.1892222
```

3.1.2 Missing Values

Let's inspect the variable *h_j*. It is categorical, where 1 indicates that a country has a free judiciary. We use the `table()` function to find the frequency in each category.

```
table(world.data$h_j)
```

```

  0   1
105  64
```

We now know that 64 countries have a free judiciary and 105 countries do not.

Conceptually the variable is nominal. To see how the variable is stored in R, we can use the `str()` function.

```
str(world.data$h_j)
```

```
 int [1:194] 0 0 0 NA 0 NA 0 0 1 1 ...
```

The function returns 'int' which abbreviates 'integer', i.e., a numeric type. The function also shows us the first 10 realisations of the variable. We se zeroes and ones which are the two categories. We also see NA's which abbreviates not available. NAs are missing values. Values can be missing for different reasons. For instance, a coder may have forgotten to code whether a country had been colonised at some point in its history or the country may be new and the categories, therefore, don't apply. It is important for us that we cannot calculate with NAs.

There are different ways of dealing with NAs. We will always delete missing values. Our dataset must maintain its rectangular structure. Hence, when we delete a missing value from one variable, we delete it for the entire row of the dataset. Consider the following example.

Row	Variable1	Variable2	Variable3	Variable4
1	15	22	100	65
2	NA	17	26	75
3	27	NA	58	88
4	NA	NA	4	NA
5	75	45	71	18
6	18	16	99	91

If we delete missing values from *Variable1*, our dataset will look like this:

Row	Variable1	Variable2	Variable3	Variable4
1	15	22	100	65
3	27	NA	58	88
5	75	45	71	18
6	18	16	99	91

The new dataset is smaller than the original one. Rows 2 and 4 have been deleted. When we drop missing values from one variable in our dataset, we lose information on other variables as well. Therefore, you only want to drop missing values on variables that you are interested in. Let's drop the missing values on our variable *h_j*. We do this in several steps.

First, we introduce the `is.na()` function. We supply a vector to the function and it checks for every element, whether it is missing or not. R returns true or false. Let's use the function on our variable.

```
is.na(world.data$h_j)
```

```
  [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
 [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
 [23]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [67]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[100]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[111] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[122] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[144] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
[155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[177] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[188] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

To see the amount of missingness in the variable *h_j*, we can combine `is.na()` with the `table()` function.

```
table( is.na(world.data$h_j) )
```

```

FALSE  TRUE 
  169    25 
```

So, we have 25 missing values on *h_j*. Our dataset has 194 rows. Check your global environment to confirm this or use the `nrow()` function. That means, if we drop all missing values from *h_j*, the our dataset *world.data* will lose 25 rows.

Before we drop the missings, we introduce the `which()` function. It returns the row indexes (the rows in the dataset) where some condition is true. So if we use `which()` and `is.na()`, we get the row numbers in the *world.data* dataset where values are missing on *h_j*.

```
which( is.na( world.data$h_j ) )
```

```
 [1]   4   6  11  22  23  67 100 108 112 120 123 129 130 131 141 146 147
[18] 148 149 150 154 165 173 179 191
```

We said that our dataset will lose 25 rows. Let's use the `length()` function to confirm that this is the case.

```
length( which( is.na( world.data$h_j ) ) )
```

```
[1] 25
```

We have, indeed, identified 25 rows that we want to delete from our dataset.

The function `is.na()` returns "TRUE" if an observation is missing. We can use the `!` operator so that the function returns "TRUE" if an observation is **not** missing. The `!` means not.

Let's confirm this:

```
# true = observation is missing
table( is.na(world.data$h_j) )
```

```

FALSE  TRUE
  169    25
```

```
# true = observations is NOT missing
table( !is.na(world.data$h_j) )
```

```

FALSE  TRUE
   25   169
```

We now drop the rows with missings on *h_j* by overwriting our original dataset with a new dataset that is a copy of the old without the missings. We use the square brackets to subset our dataset.

```
world.data <- world.data[ !is.na( world.data$h_j ) , ]
```

Confirm that our new *world.data* dataset has only 169 remaining.

"But what if we want our original dataset back," you ask. We have overwritten the original. It is no longer in our work environment. We have to reload the data set from the disk.

Let's do that:

```
world.data <- read.csv("QoG2012.csv")
```

Right, we now have all observations back. This is important. Let's say we need to drop missings on a variable. We do is. If a later analysis does not involve that variable, we want all the observations back. Otherwise we would have thrown away valuable information. The smaller our dataset, the less information it contains. Less information will make it harder for us to detect systematic correlations. We have to options. Either we reload the original dataset or we create a copy of the original with a different name that we could use later on. Let's do this.

```
full.dataset <- world.data
```

Let's drop missings on *h_j* in the *world.data* dataset.

```
world.data <- world.data[ !is.na( world.data$h_j ) , ]
```

Now, if we want the full dataset back, we can overwrite *world.data* with *full.dataset*. The code would be the following:

```
world.data <- full.dataset
```

If you ran this line. Delete missings from *h_j* in *world.data* again.

This data manipulation may seem boring but it is really important that you know how to do this. Most of the work in data science is not running statistical models but data manipulation. Most of the dataset you will work with in your jobs, as a research assistant or on you dissertation won't be cleaned for you. You will have to do that work. It takes time and is sometimes frustrating. That's unfortunately the same for all of us.

3.1.3 Factor Variables

Categorical/nominal variables can be stored as numeric variables in R. However, the values do not imply an ordering or relative importance. We often store nominal variables as factor variables in R. A factor variable is a nominal data type. The advantage of turning a variable into a factor type is that we can assign labels to the categories and that R will not calculate with the values assigned to the categories.

The function `factor()` lets us turn the variable into a nominal data type. The first argument is the variable itself. The second are the category labels and the third are the levels associated with the categories. To know how those correspond, we have to scroll up and look at the codebook.

We also overwrite the original numeric variable `h_j` with our nominal copy indicated by the assignment arrow `<-`.

```
# factorize judiciary variable
world.data$h_j <- factor(world.data$h_j, labels = c("controlled", "free"), levels = c(0,1))

# frequency table of judiciary variable
table(world.data$h_j)
```

```

controlled       free 
       105         64 
```

3.1.4 Renaming Variables

We want to rename *h_j* into something more meaningful. The new name should be *free.judiciary*. We can use the `names()` function to get a vector of variable names.

```
names(world.data)
```

```
[1] "h_j"          "wdi_gdpc"     "undp_hdi"     "wbgi_cce"     "wbgi_pse"    
[6] "former_col"   "lp_lat_abst"
```

We want to change the first element of that vector. We know that we can use square brackets to subset vectors. Let's display the first element of the vector of variable names only.

```
names(world.data)[1]
```

```
[1] "h_j"
```

Now we simply change the name using the assignment arrow `<-` and our new variable names goes in quotes.

```
names(world.data)[1] <- "free.judiciary"
```

We now check the variable names to confirm that we successfully changed the name.

```
names(world.data)
```

```
[1] "free.judiciary" "wdi_gdpc"       "undp_hdi"       "wbgi_cce"      
[5] "wbgi_pse"       "former_col"     "lp_lat_abst"
```

3.1.5 Distributions

A marginal distribution is the distribution of a variable by itself. Let's look at the summary statistics of the United Nations Development Index *undp_hdi* using the `summary()` function.

```
summary(world.data$undp_hdi)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
 0.2730  0.5272  0.7455  0.6946  0.8350  0.9560       9 
```

How nice. This returns summary stats. We see the range(minimum to maximum). We see the interquartile range (1st quartile to 3rd quartile). We also see mean and median. Finally, we see the number of NAs.

Oh we forgot. We said, when we drop missing on variable, we may lose information when we work on a new variable. Let's restore our dataset *world.data* to its original state.

```r
world.data <- full.dataset
```

Now, we check the summary stats again.

```r
summary(world.data$undp_hdi)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
 0.2730  0.5390  0.7510  0.6982  0.8335  0.9560      19 
```

In the smaller dataset (where we had dropped missings from *h_j*), we had 9 missings. Now, we have 19 missings. The difference is 10. Our smaller dataset had 25 rows less than the bigger dataset. Therefore, we would have thrown away 6 good observations. That is not nothing. It's 3 percent of our data.

Let's drop missing on *undp_hdi* and rename it to *hdi*.

```r
world.data <- world.data[ which( !is.na(world.data$undp_hdi) ) , ]
```

Let's change the name.

```r
names(world.data)[3] <- "hdi"
names(world.data)
```

```
[1] "h_j"          "wdi_gdpc"     "hdi"          "wbgi_cce"     "wbgi_pse"    
[6] "former_col"  "lp_lat_abst"
```

Let's take the mean of *hdi*.

```r
hdi.mean <- mean( world.data$hdi )
hdi.mean
```

```
[1] 0.69824
```

The mean of *hdi* is the mean in the sample. We would like the mean of hdi in the population. Remember that sampling variability causes us to estimate a different mean every time we take a new sample.

We learned that the means follow a distribution if we take the mean repeatedly in different samples. In expectation the population mean is the sample mean. How certain are we about the mean. Well, we need to know how the sampling distribution looks like.

To find out we estimate the standard error of the mean. The standard error is the standard deviation of the sampling distribution. The name is not standard deviation but standard error to indicate that we are talking about the distribution of a statistic (the mean) and not a random variable.

The formula for the standard error of the mean is:

$$s_{\bar{x}} = \frac{\sigma}{\sqrt(n)}$$

The σ is the real population standard deviation of the random variable *hdi* which is unknown to us. We replace the population standard deviation with our sample estimate of it.

$$s_{\bar{x}} = \frac{s}{\sqrt(n)}$$

The standard error of the mean estimate is then

```r
se.hdi <- sd(world.data$hdi) / sqrt( nrow(world.data) )
se.hdi
```

```
[1] 0.01362411
```

Okay, so the mean is 0.69824 and the standard error of the mean is 0.0136241.

We know that the sampling distribution is approximately normal. That means that 95 percent of all observations are within 1.96 standard deviations (standard errors) of the mean.

$$\bar{x} \pm 1.96 \times s_{\bar{x}}$$

So what is that in our case?

```r
lower.bound <- hdi.mean - 1.96 * se.hdi
lower.bound
```

```
[1] 0.6715367
```

```r
upper.bound <- hdi.mean + 1.96 * se.hdi
upper.bound
```

```
[1] 0.7249432
```

That now means the following. Were we to take samples of *hdi* again and again and again, then 95 percent of the time, the mean would be in the range from 0.6715367 to 0.7249432.

What is a probability? "The long-run relative frequency," you all scream in unison. Given that definition, you can say: "With 95 percent probability, the mean is in the range 0.6715367 to 0.7249432."

Sometimes people like to former way of phrasing this relationship better than the latter. In this case you tell them: "a probability is the long-run relative frequency of an outcome."

Now, let's visualise our sampling distribution. We haven't actually taken many samples, so how could we visualise the sampling distribution? Well, we know the sampling distribution looks normal. We know that the mean is our mean estimate in the sample. And finally, we know that the standard deviation is the standard error of the mean.

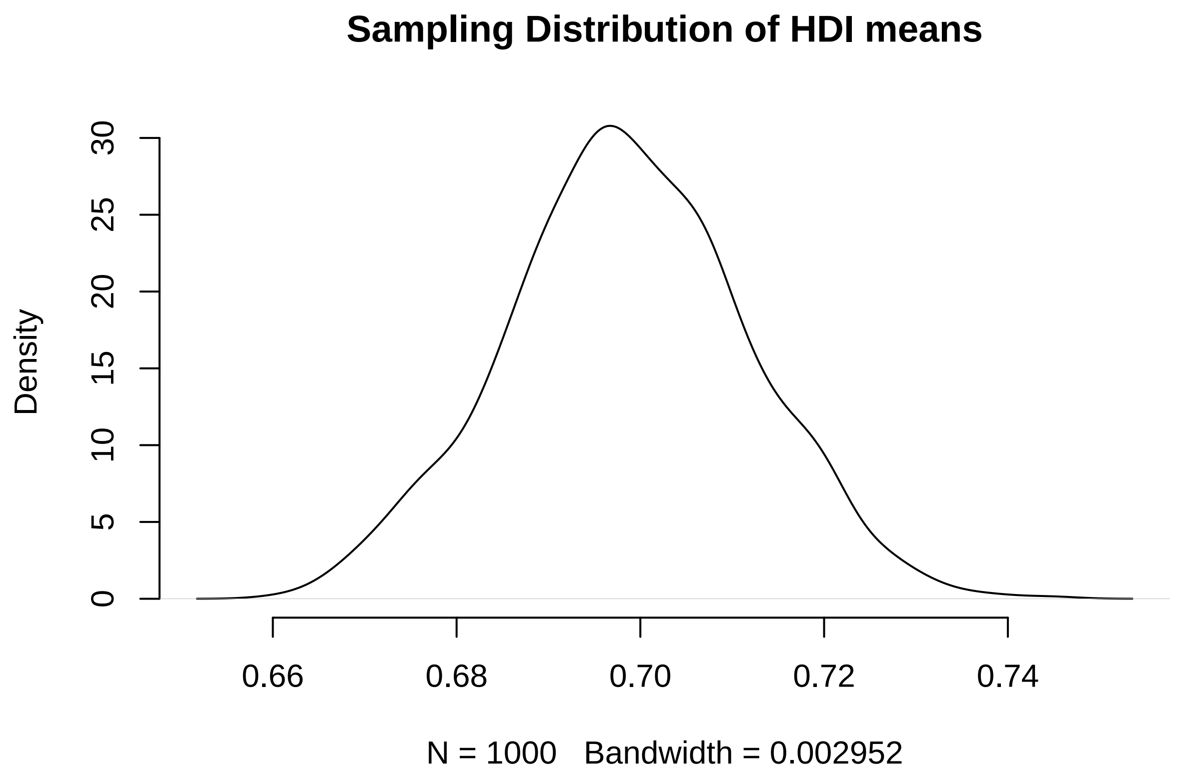
We can randomly draw values from a normal distribution with mean 0.69824 and standard deviation 0.0136241. We do this with the `rnorm()` function. It's first argument is the number of values to draw at random from the normal distribution. The second argument is the mean and the third is the standard deviation.

Recall, that a normal distribution has two parameters that characterise it completely: the mean and the standard deviation. So with those two we can draw the distribution.

```r
draw.of.hdi.means <- rnorm( 1000, mean = hdi.mean, sd = se.hdi )
```

We have just drawn 1000 mean values at random from the distribution that looks like our sampling distribution.

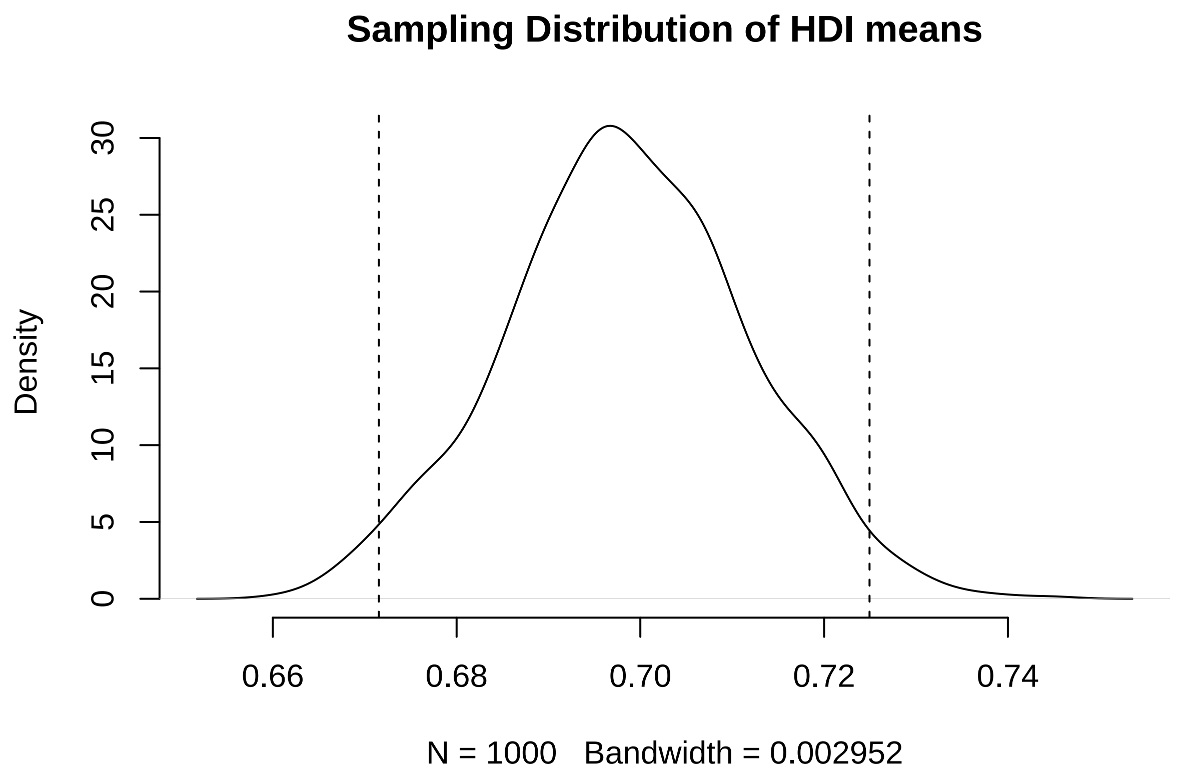
```r
plot(
 density( draw.of.hdi.means ),
 bty = "n",
 main = "Sampling Distribution of HDI means"
)
```



Beautiful Let's add the 95 percent confidence interval around our mean estimate. The confidence interval quantifies our uncertainty. We said 95 percent of the time the mean would be in the interval from 0.6715367 to 0.7249432."

```r
abline( v = lower.bound, lty = "dashed")
abline( v = upper.bound,  lty = "dashed")
```

You do not need to run the plot function again. You can just add to the plot. Check the help function of `abline()` to see what its arguments refer to.



Fantastic! You can see that values below and above our confidence interval are quite unlikely. Those values in the tails would not occur often.

Not often, but not impossible.

Let's say that we wish know the probability that we take a sample and our estimate of the mean is greater or equal 0.74. We would need to integrate over the distribution from $-\inf$ to 0.74. Fortunately R has a function that does that for us. We need the `pnorm()`. It calculates the probability of a value that is smaller or equal to the value we specify. In other words, it gives us the probability from the cumulative normal.

As the first argument `pnrom()` wants the value; 0.74 in our case. The second and third arguments are the mean and the standard deviation that characterise the normal distribution.

```
pnorm(0.74, mean = hdi.mean, sd = se.hdi)
```

```
[1] 0.9989122
```

What!? The probability to draw a mean 0.74 is 99.9 percent!? That cannot be the value is so far in the tail of the distribution.

Well, this is the cumulative probability of drawing a value that is equal to or smaller than 0.74. All probabilities sum to 1. So if we want to know the probability of drawing a value that is greater than 0.74, we subtract the probability, we just calculated, from 1.

```
1 - pnorm(0.74, mean = hdi.mean, sd = se.hdi)
```

```
[1] 0.001087785
```

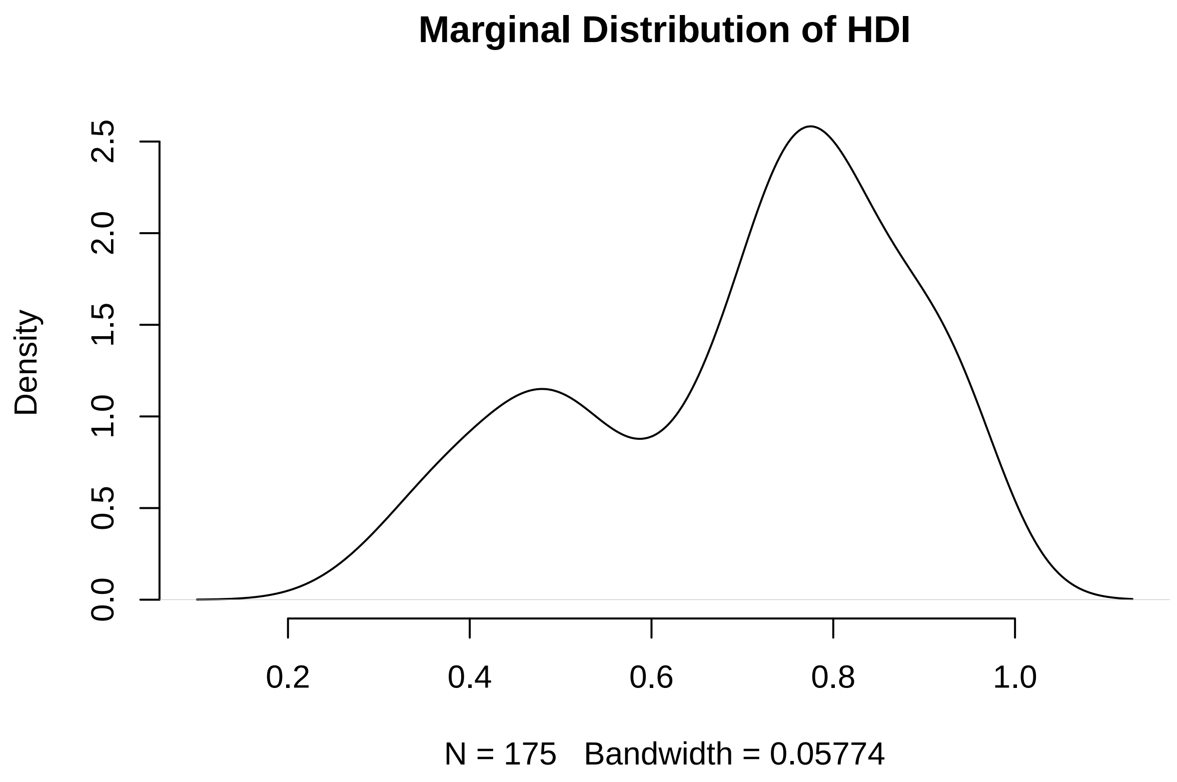
Right, so the probability of getting a mean of *hdi* in a sample is 0.1 percent.

3.1.6 Conditional Distributions

Let's look at *hdi* by *former_col*. The variable *former_col* is 1 if a country is a former colony and 0 otherwise. The variable *hdi* is continuous.

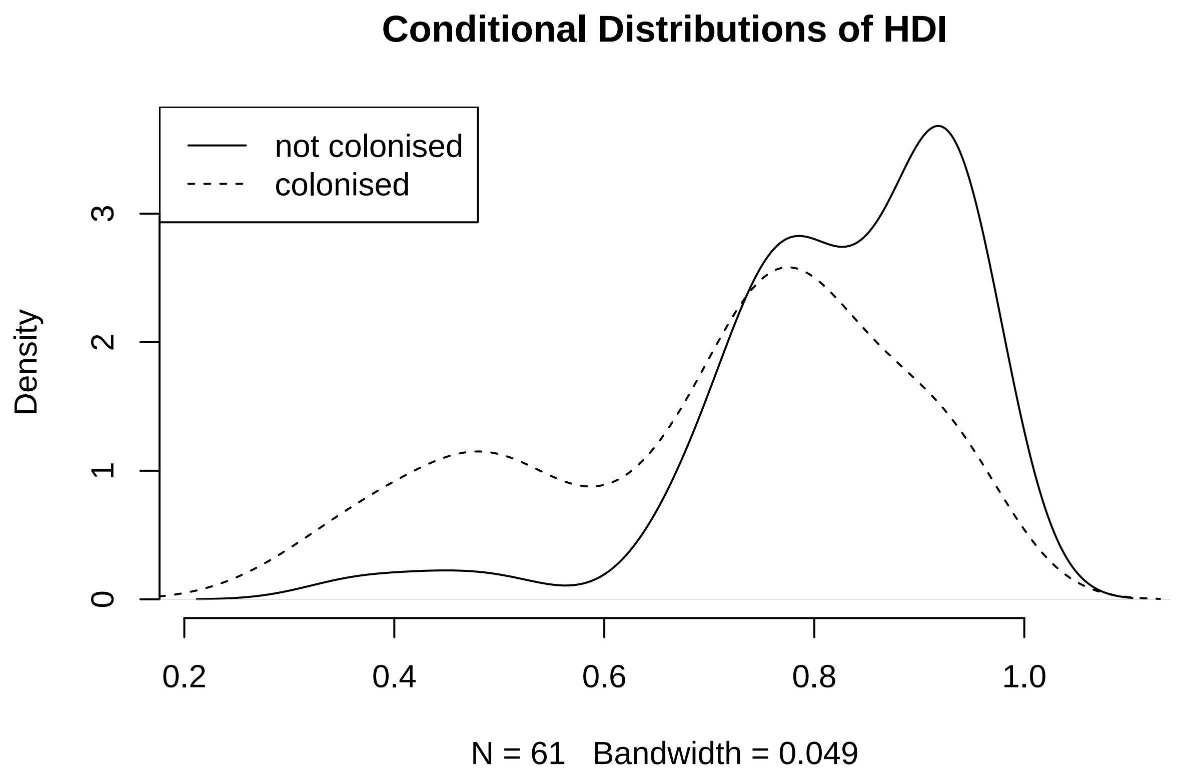
Before we start, we plot the marginal pdf of *hdi*.

```r
plot(
  density(world.data$hdi),
  bty = "n",
  main = "Marginal Distribution of HDI"
)
```



The distribution is bimodal. There is one peak at the higher development end and one peak at the lower development end. Could it be that these two peaks are conditional on whether a country was colonised or not? Let's plot the conditional distributions.

```r
plot(
  density(world.data$hdi[world.data$former_col == 0]),
  bty = "n",
  main = "Conditional Distributions of HDI"
)
lines(density(world.data$hdi), lty = "dashed")
legend("topleft", c("not colonised", "colonised"), lty = c("solid", "dashed"))
```



It's not quite like we expected. The distribution of human development of not colonised countries is shifted to right of the distribution of colonised countries and it is clearly narrower. Interestingly though, the distribution of former colonies has a greater variance. Evidently, some former colonies are doing very well and some are doing very poorly. It seems like knowing whether a country was colonised or not tells us something about its likely development but not enough. We cannot, e.g., say colonisation is the reason why countries do poorly. Probably, there are differences among types of colonial institutions that were set up by the colonisers.

Let's move on and examine the probability that a country has .8 or more on *hdi* given that it is a former colony.

We can get the cumulative probability with the `ecdf()` function. It returns the empirical cumulative distribution, i.e., the cumulative distribution of our data. We know that we can subset using square brackets. That's all we need.

```
cumulative.p <- ecdf(world.data$hdi[ world.data$former_col == 1 ])
1 - cumulative.p(.8)
```

```
[1] 0.1666667
```

Okay, the probability that a former colony has .8 or larger on the *hdi* is 16.6 percent. Go ahead figure out the probability for not former colonies on your own.

3.1.7 Exercises

1. Create a script and call it assignment03. Save your script.
2. Load the *world.data* dataset from your disk.
3.