Suffolk 2019

Contents

About this course

This course is an introduction R, RStudio and statistics. Our primary aims are to get comfortable working with R and to be able to prepare, manipulate, analyse and visualise data.

Slides day 1

1 Introduction to R and RStudio

1.1 Learning objectives

In this session, we will have a look at R and RStudio. We will interact with both and use the various components of RStudio.

1.1.1 What is R?

R is an environment for statistical computing and graphics. RStudio is an editor or integrated development environment (IDE) that makes working with R much more comfortable.

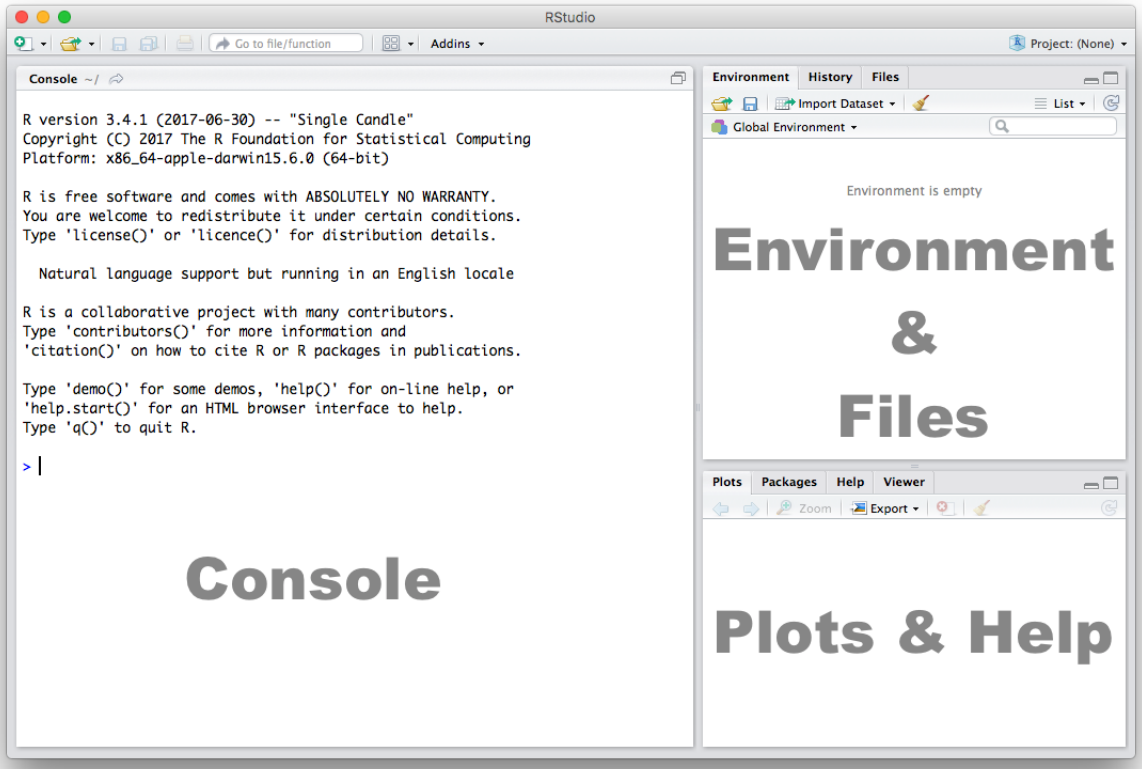
To install R and RStudio on your computer, download both from the following sources:

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

Keep both R and RStudio up to date. That means go online and check for newer versions. In case there are new versions, download those and re-install.

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:

```
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 (/).

```
10 - 4
```

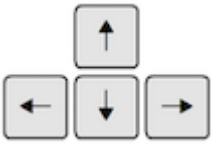
```
[1] 6
```

```
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

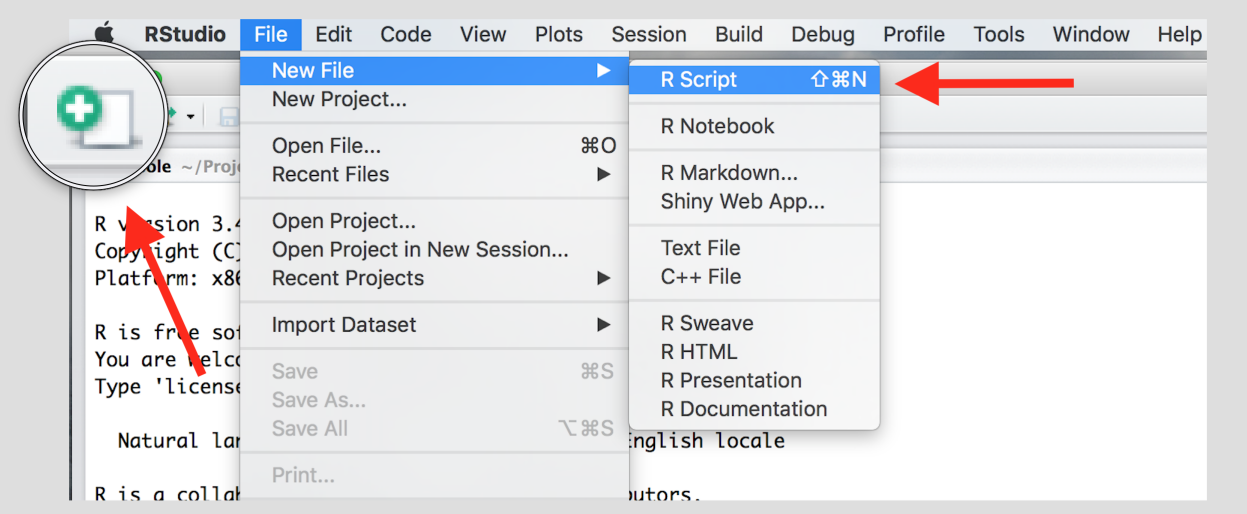
---

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!

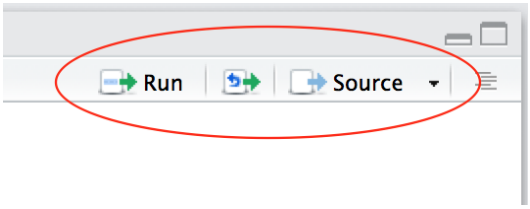
1.1.4 Scripts

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

2 R-syntax, data structures and types

2.1 Seminar

In this session we introduce R-syntax, and data types.

2.1.1 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.

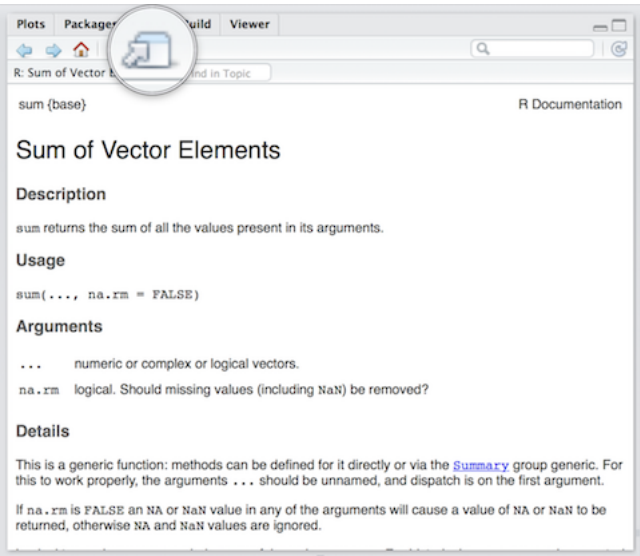
2.1.2 Getting Help

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

2.1.3 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:

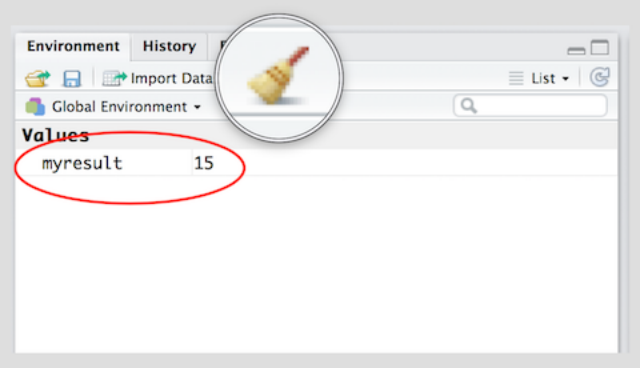
```
myresult <- sum(1, 4, 10)
```

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	

2.1.4 Vectors and subsetting

A vector is one dimensional. It can contain one element in which case it is also called a scalar or many elements. We can add and multiply vectors. Think of a vector as a row or column in your excel spreadsheet.

To create a vector, we use the `c()` function, where c stands for collect. We start by creating a numeric vector.

```
vec1 <- c(10, 47, 99, 34, 21)
```

Creating a character vector works in the same way. We need to use quotation marks to indicate that the data type is textual data.

```
vec2 <- c("Emilia", "Martin", "Agatha", "James", "Luke", "Jacques")
```

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

```
length(vec1)
```

```
[1] 5
```

```
length(vec2)
```

```
[1] 6
```

We need one coordinate to identify a unique element in a vector. For instance, we may be interested in the first element of the vector only. We use square brackets `[]` to access a specific element. The number in square brackets is the vector element that we wish to see.

```
vec1[1]
```

```
[1] 10
```

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

```
vec1[-1]
```

```
[1] 47 99 34 21
```

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

```
vec1[2:4]
```

```
[1] 47 99 34
```

We can access non-adjacent elements bu using the collect function `c()`.

```
vec1[c(2,5)]
```

```
[1] 47 21
```

Finally, we combine the `length()` function with the square brackets to access the last element in our vector.

```
vec1[ length(vec1) ]
```

```
[1] 21
```

2.1.5 Matrices

A matrix has two dimensions and stores data of the same type, e.g. numbers or text but never both. A matrix is always rectangular. Think of it as your excel spreadsheet - essentially, it is a data table.

We create a matrix using the `matrix()` function. We need to provide the following arguments:

```
mat1 <- matrix(
  data = c(99, 17, 19, 49, 88, 54),
  nrow = 2,
  ncol = 3,
  byrow = TRUE
)
```

Argument	Description
data	the data in the matrix
nrow	number of rows
ncol	number of columns
byrow	TRUE = matrix is filled rowwise

To display the matrix, we simply call the object by its name (in this case mat1).

```
mat1
```

```
     [,1] [,2] [,3]
[1,]   99   17   19
[2,]   49   88   54
```

To access a unique element in a matrix, we need 2 coordinates. First, a row coordinate and second, a column coordinate. We use square brackets and separate the coordinates with a comma `[ , ]`. The row coordinate goes before the comma and the column coordinate after.

We can access the the second row and third column like so:

```
mat1[2, 3]
```

```
[1] 54
```

To display an entire column, we specify the column we want to display and leave the row coordinate empty like so:

```
# display the 2nd column
mat1[ , 2]
```

```
[1] 17 88
```

Similarly, to display the entire second row, we specify the row coordinate but leave the column coordinate empty.

```
mat1[2, ]
```

```
[1] 49 88 54
```

2.1.6 Arrays

Arrays are similar to matrices but can contain more dimensions. You can think of an array as stacking multiple matrices. Generally, we refer to the rows, columns and layers in array. Let's create an array with 2 rows, 3 columns and 4 layers using the `array()` function.

```
arr1 <- array(
  data = c(1:24),
  dim = c(2, 3, 4)
)
```

To display the object, we call it by its name.

```
arr1
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

We can subset an array using the square brackets `[]`. To access a single element we need as many coordinates as our object has dimensions. Let's check the number of dimensions in our object first.

```
dim(arr1)
```

```
[1] 2 3 4
```

The `dim()` function informs us that we have 3 dimensions. The first is of length 2, the second of length 3 and the fourth of length 4.

Access the second column of the third layer on your own.

```
arr1[ , 2, 3]
```

```
[1] 15 16
```

3 Data import (from csv, txt, and excel)

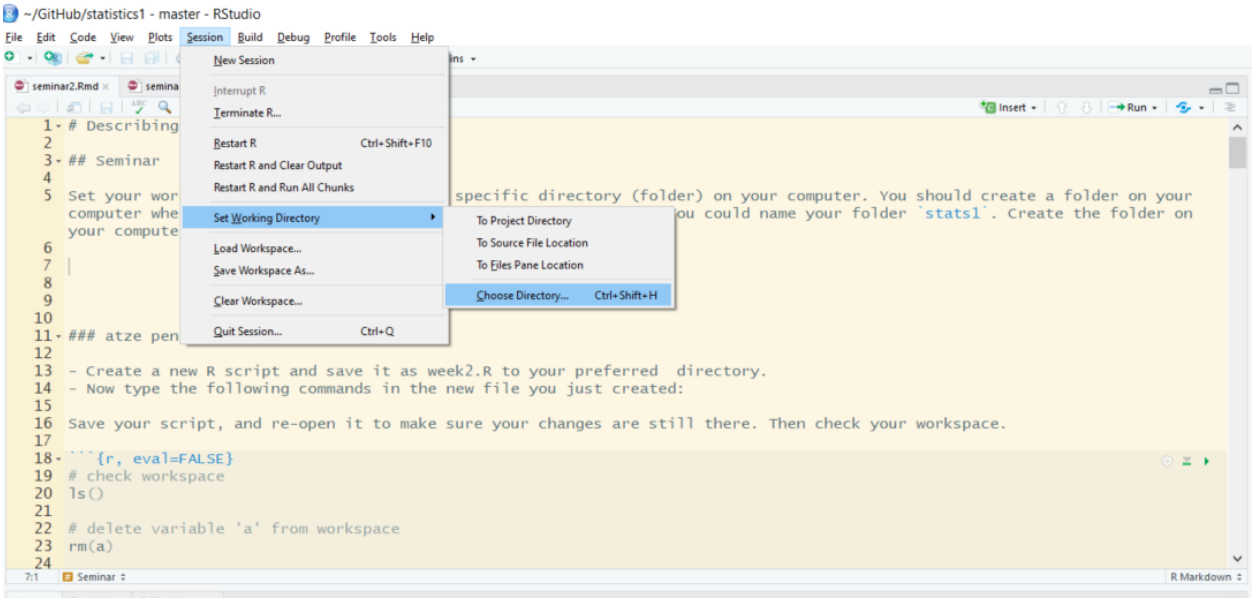
3.1 Seminar

In this section, we will load data in csv, txt, excel and R format. We will learn how to check and set our working directory.

3.1.1 Setting up

We set our working directory. R operates in specific directory (folder) on our computer. We create a folder on where we save our scripts. We name the folder `suffolk2019`. 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 day1.R to your `suffolk2019` directory.

At the beginning of each new script, we want to clear the workspace. The workspace is stored in working memory on our computer. If we do not clear it for a new script, it becomes too full over time. Our computer will slow down and it will become difficult for us to know which objects are stored in working memory.

We check the contents of our workspace like so:

```
# check workspace
ls()
```

To remove a specific object, we use the `rm()` function which stands for remove. Within the round brackets, we put the name of the object we want to remove. We could remove the object `a` like so:

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

At the beginning of each script, we should always clear the entire workspace. We can do so in the following way:

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

You can also clear text from the console window. To do so press Crtl+l on Winowds or Command+l on Mac.

3.1.2 Loading data

Data comes in different file formats such as `.txt`, `.csv`, `.xlsx`, `.RData`, `.dta` and many more. To know the file type of a file right click on it and view preferences (in Windows explorer or Mac finder).

R can load files coming in many different file formats. To find out how to import a file coming in a specific format, it is usually a good idea to the google "R load file_format".

3.1.3 Importing a dataset in `.csv` format

One of the most common file types is `.csv` which means comma separated values. Columns are separated by commas and rows by line breaks.

The dataset's name is "non_western_immigrants.csv". To load it, we use the `read.csv()` function.

```r
dat1 <- read.csv("non_western_immigrants.csv")
```

3.1.4 Importing a dataset in Excel (xlsx) format

Another common file format is Microsoft's Excel `xlsx` format. We will load a dataset in this format now. To do so, we will need to install a package first. Packages are additional functions that we can add to R. A package is like an app on our phones.

We install the `readxl` package using `install.packages("readxl")`.

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

We only need to install a package once. It does not hurt to do it more often though, because every time we install, it will install the most recent version of the package.

Once a package is installed, we need to load it using the `library()` function.

```r
library(readxl)
```

To load the excel file, we can now use the `read_excel()` function that is included in the `readxl` library. We need to provide the following arguments to the function:

Argument	Description
path	Filename of excel sheet
sheet	Sheet number to import

Now, let's load the file:

```r
dat2 <- read_excel("non_western_immigrants.xlsx", sheet = 1)
```

3.1.5 Importing a dataset in RData format

The native file format of R is called `.RData`. To load files saved in this format, we use the `load()` function like so:

```
load(file = "non_western_immigrants.RData")
```

Notice that we usually need to assign the object we load to using the `<-` operator. The `load()` function is an exception where we do not need to do this.

3.1.6 Importing a dataset in .txt format.

Loading a dataset that comes in .txt format requires some additional information. The format is a text format and we need to know how the columns are separated. Usually it is enough to open the file in a word processor such as notepad to see how this is done. The most common ways to separate columns is by using commas or tabs but other separators such as for instance semicolons are sometimes also used.

In our example, columns are separated by semicolons. We use the `read.table()` function and provide the following arguments:

Argument	Description
file	Filename of excel sheet
sep	the symbol that separates columns
header	whether the first row contains variable names or not

```
dat3 <- read.table(file = "non_western_immigrants.txt", sep =";", header = TRUE)
```

4 Creating, amending, exporting data frames

4.1 Seminar

4.1.1 Creating data frames

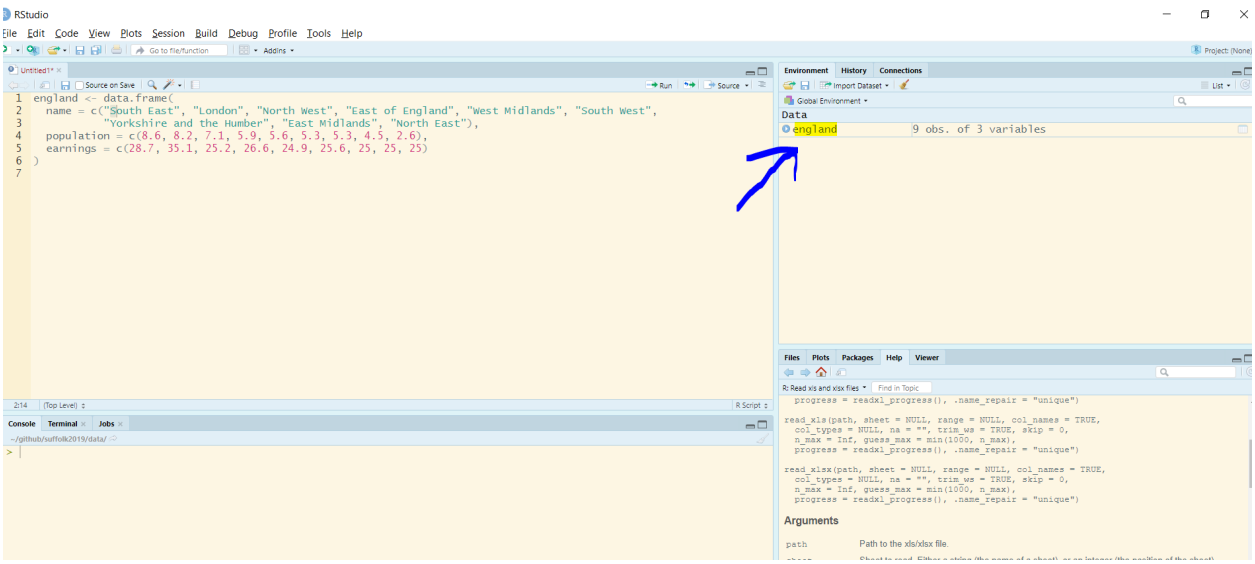
A data frame is an object that holds data in a tabular format similar to how spreadsheets work. Variables are generally kept in columns and observations are in rows. Data frames are similar to matrices but they can store vectors of different types (e.g. numbers and text).

We start by creating a data frame with the `data.frame()` function. We will give each column a name (a variable name) followed by the `=` operator and the respective vector of data that we want to assign to that column.

```
england <- data.frame(
  name = c("South East", "London", "North West", "East of England", "West Midlands", "South West",
           "Yorkshire and the Humber", "East Midlands", "North East"),
  population = c(8.6, 8.2, 7.1, 5.9, 5.6, 5.3, 5.3, 4.5, 2.6),
  earnings = c(28.7, 35.1, 25.2, 26.6, 24.9, 25.6, 25, 25, 25)
)
```

4.1.2 Working with data frames

we can display the entire dataset in spreadsheet view by clicking on the object name in the environment window.



Alternatively, you can call the object name to display the dataset in the console window. Let's do so:

```
england
```

```
                    name population earnings
1             South East        8.6     28.7
2                 London        8.2     35.1
3             North West        7.1     25.2
4          East of England        5.9     26.6
5            West Midlands        5.6     24.9
6             South West        5.3     25.6
7 Yorkshire and the Humber        5.3     25.0
8            East Midlands        4.5     25.0
9             North East        2.6     25.0
```

Often, datasets are too long to be viewed to in the console window. It is a good idea to look at the first couple of rows of a datasets to get an overview of its contents. We use the square brackets `[]` to view the first five rows and all columns.

```
england[1:5, ]
```

```
            name population earnings
1     South East        8.6     28.7
2         London        8.2     35.1
3     North West        7.1     25.2
4 East of England        5.9     26.6
5   West Midlands        5.6     24.9
```

v Columns in a dataframe have names. We will often need to know the name of a column/variable to access it. We use the `names()` function to view all variable names in a dataframe.

```
names(england)
```

```
[1] "name"       "population" "earnings"
```

We can access the earnings variable in multiple ways. First, we can use the `$` operator. We write the name of the dataset object, followed by the `$`, followed by the variable name like so:

```
england$earnings
```

```
[1] 28.7 35.1 25.2 26.6 24.9 25.6 25.0 25.0 25.0
```

We can also use the square brackets to access the earnings column.

```
england[, "earnings" ]
```

```
[1] 28.7 35.1 25.2 26.6 24.9 25.6 25.0 25.0 25.0
```

The square brackets are sometimes preferred because we could access multiple columns at once like so:

```
england[, c("name", "earnings") ]
```

```
                     name earnings
1               South East     28.7
2                   London     35.1
3               North West     25.2
4          East of England     26.6
5            West Midlands     24.9
6               South West     25.6
7 Yorkshire and the Humber     25.0
8            East Midlands     25.0
9               North East     25.0
```

Variables come in different types such as numbers, text, logical (true/false). We need to know the type of a variable because the type affects statistical analysis. We use the `str()` function to check the type of each variable in our dataset.

```
str(england)
```

```
'data.frame':   9 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25
```

The first variable in our dataset is a factor variable. Factors are categorical variables. Categories are mutually exclusive but they do not imply an ordering. For instance, "East of England" is not more or less than "West Midlands". The variables population and earnings are both numeric variables.

```
str(england)
```

```
'data.frame':   9 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25
```

4.1.3 Amending data frames

Amending data sets usually involves adding rows or columns or removing rows or columns. We start by adding a new variable to our dataset which contains the percent of the population on income support.

To create a new variable we simply assign a new vector to the dataframe object like so:

```
england$pct_on_support <- c(3, 5.3, 5.3, 3.5, 5.1, 3.3, 5.2, 4.2, 6.1)
```

We call the dataframe object to view our changes.

```
england
```

```
                     name population earnings pct_on_support
1               South East        8.6     28.7            3.0
2                   London        8.2     35.1            5.3
```

```
3               North West        7.1     25.2            5.3
4          East of England        5.9     26.6            3.5
5            West Midlands        5.6     24.9            5.1
6               South West        5.3     25.6            3.3
7 Yorkshire and the Humber        5.3     25.0            5.2
8            East Midlands        4.5     25.0            4.2
9               North East        2.6     25.0            6.1
```

We can delete the variable we just created by assigning `NULL` to it.

```
england$pct_on_support <- NULL
```

Let's view our most recent changes.

```
england
```

```
                     name population earnings
1               South East        8.6     28.7
2                   London        8.2     35.1
3               North West        7.1     25.2
4          East of England        5.9     26.6
5            West Midlands        5.6     24.9
6               South West        5.3     25.6
7 Yorkshire and the Humber        5.3     25.0
8            East Midlands        4.5     25.0
9               North East        2.6     25.0
```

Adding a new row to a dataset means adding an observation. Let's add Brittany to our dataset. We need to fill in a value for each variable. If, we do not know a value, we declare it as missing. Missings are `NA` for numeric variables and `""` for character variables. We use the `rbind()` function (row bind) to add a row to our dataset.

```
england <- rbind(england,  c("Brittany", 4.5, NA) )
```

```
Warning in `[<-.factor`(`*tmp*`, ri, value = "Brittany"): invalid factor
level, NA generated
```

Let's examine our dataframe.

```
england
```

```
                     name population earnings
1               South East        8.6     28.7
2                   London        8.2     35.1
3               North West        7.1     25.2
4          East of England        5.9     26.6
5            West Midlands        5.6     24.9
6               South West        5.3     25.6
7  Yorkshire and the Humber        5.3       25
8            East Midlands        4.5       25
9               North East        2.6       25
10                    <NA>        4.5     <NA>
```

We were not allowed to enter "Brittany" as a value for the name variable. This is because the variable is a factor. While it is possible to add new levels to a factor (categories such as "Midlands" are called levels), it involves a bit more advanced programming. We will solve this problem later in the type coercion section.

4.1.4 Saving data frames

Datasets can be exported in many different file formats. We recommend exporting files as ".csv" files because csv is a very common file type. Such files can be handled by all statistical packages including Microsoft's Excel. We need to provide five arguments.

Argument	Description
x	The name of the object
file	The file name
sep	The symbol that separates columns
col.names	= TRUE saves the variable names (recommended)
row.names	= FALSE omits the row names (recommended)

```
write.table(x = england, file = "england.csv", sep = ",", col.names = TRUE, row.names = FALSE)
```

5 Type coercion

5.1 Seminar

We often need to change the type of a variable. This can be necessary to clean data or because we add a new level to a factor like in the previous section on amending data frames. We begin by loading the england dataset that we created previously.

```
england <- read.csv(file = "england.csv", sep = ",", header = TRUE)
```

Recall that we could not add "Brittany" as a name to our variable. This was because the variable name is stored as a factor and "Brittany" was a new category. The easiest way to add the name "Brittany" is to convert the name variable into a character variable.

5.1.1 Coerce a factor to character

Let's check our england dataset's variable types. .

```
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 9 levels "East Midlands",..: 6 3 5 2 8 7 9 1 4 NA
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

The variable "name" is a factor variable and needs to be converted to a character variable. We coerce the variable into a different type using the `as.character()` function.

```
england$name <- as.character(england$name)
```

Let's inspect the variable types of our dataset again:

```
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : chr  "South East" "London" "North West" "East of England" ...
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

The variable name is now a character variable. We now change the last value of the name variable into Brittany.

```
england$name[ length(england$name) ] <- "Brittany"
```

Let's inspect our change.

```
england
```

```
                      name population earnings
1                South East        8.6     28.7
2                    London        8.2     35.1
3                North West        7.1     25.2
4           East of England        5.9     26.6
5             West Midlands        5.6     24.9
6                South West        5.3     25.6
7  Yorkshire and the Humber        5.3     25.0
8             East Midlands        4.5     25.0
9                North East        2.6     25.0
10                 Brittany        4.5       NA
```

5.1.2 Coerce a character variable into a factor variable

We can now easily convert the variable type back from character into a factor with the `as.factor()` function like so:

```
england$name <- as.factor(england$name)
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 4.5
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

5.1.3 Coerce a character variable into a numeric variable

In R missing vlues are called `NA` for numeric types and `""` for character types. When we use load third party data, the coding often differs. This can happen, for instance, due to data entry errors.

We simulate a data entry error by changing the population value of Brittany to a character value.

```
england$population[ length(england$population) ] <- "mistake"
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
 $ population: chr  "8.6" "8.2" "7.1" "5.9" ...
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

Notice, that the variable population is now a character vector instead of a numeric vector. Whenever numbers and text are mixed, R will automatically treat the vector as a character vector.

We can convert the population variable back into a numeric variable using the `as.numeric()` function. All values that are not recognised as numbers will be changed to `NA`.

```
england$population <- as.numeric(england$population)
```

```
Warning: NAs introduced by coercion
```

```
str(england)
```

```
'data.frame':   10 obs. of  3 variables:
 $ name      : Factor w/ 10 levels "Brittany","East Midlands",..: 7 4 6 3 9 8 10 2 5 1
```

```
 $ population: num  8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6 NA
 $ earnings  : num  28.7 35.1 25.2 26.6 24.9 25.6 25 25 25 NA
```

```
england
```

```
                      name population earnings
1                South East        8.6     28.7
2                    London        8.2     35.1
3                North West        7.1     25.2
4           East of England        5.9     26.6
5             West Midlands        5.6     24.9
6                South West        5.3     25.6
7  Yorkshire and the Humber        5.3     25.0
8             East Midlands        4.5     25.0
9                North East        2.6     25.0
10                 Brittany         NA       NA
```

6 Loops and conditions

6.1 Seminar

In this section, we introduce loops and conditional statements. Loops are generally useful, when we want to carry out the same operation over and over. Conditions are logical statements that are evaluated and if the statement is true a different operation is carried out than if the statement is false. We may, for instance, be interested in the average crime rate in our sample but only if the respondents are female. To do so, we need conditional statements.

6.1.1 For loops

Loops are useful when we need to carry out similar operations repeatedly. A for loop is an easy way to do this. We will create a simple "for loop" like so:

```
for (idx in 1:7){
  print( idx )
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
```

In the above code, the `for()` function initiates the loop. We loop from 1 through 7 and `idx` takes the values from 1 to 7 iteratively. We can write code within the curly braces. Here, we simply print the current loop iteration.

We can create nested loops, i.e. loops within loops like so:

```
# first loop
for (idx in 1:7){

  # second loop
  for (idx2 in 1: 7){

    if (idx < idx2) print("idx1 is smaller than idx2")
```

```
    else print("idx1 is larger than idx2")

  } # end of second loop
} # end of first loop
```

```
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is smaller than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
[1] "idx1 is larger than idx2"
```

While this is a toy example, it illustrates how we could do pairwise comparisons between observations in our dataset. We also added an if/else condition using the functions `if()` and `else`. If the if statement is TRUE the first statement is printed and if not the second statement is printed.

6.1.2 Conditions

To illustrate the use of conditions we load the non-western foreigners dataset.

```
dat1 <- read.csv("non_western_immigrants.csv")
```

Variable Name	Description
IMMBRIT	Out of every 100 people in Britain, how many do you think are immigrants from Non-western countries?
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
Cons, Lab, SNP, Ukip, BNP, GP, party.other	Party self-identification
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

The variable `over.estimate` is equal to 1 if respondents over estimate the number of non-western immigrants in Britain. We will evaluate whether women are more or less likely to over estimate. First, we use the `mean()` function to assess the overall average value.

```
mean(dat1$over.estimate)
```

```
[1] 0.7235462
```

A mean of 0.72 indicates that 72% of the 1049 respondents in the dataset over estimate the number of non-western immigrants. To assess whether the number is larger among men than women, we need conditional statements.

We first take the mean of "over.estimate" for men:

```
mean( dat1$over.estimate[ dat1$RSex==1 ] )
```

```
[1] 0.6527197
```

Here, we used square brackets to subset the data. The subset that we evaluate is described by the logical statement `dat1$RSex==1`. The `==` operator is a logical equal that is true if a condition is fulfilled and false otherwise. In this case, it is true if the variable "RSex" is 1 which stands for men.

Take the mean of over.estimate for women on your own.

```
mean( dat1$over.estimate[ dat1$RSex== 2] )
```

```
[1] 0.7828371
```

It turns out, that females in our sample over estimate the number of non-western immigrants more. Whether the difference in our sample is systematic, i.e. whether it would hold in the population as well is a matter that we will return to.

Here we have taken two conditional means and compared them. Doing so is the first step towards statistical inference.

6.1.3 The ifelse() function

Categorical variables such as "RSex" are usually coded 0/1 and the variable name usually refers to the category that is 1. "RSex" is a bad variable name because it is not clear whehter the values 1 and 2 refer to males or females.

We will create a new variable called "female" that is equal to 1 if the respondent is female and 0 otherwise. We do so using the `ifelse()` function. The function first evaluates a logical condition and subsequently carries out one operation if the statement is true (yes) and another if the statement is false (no).

```
dat1$female <- ifelse( dat1$RSex == 2, yes = 1, no = 0 )
```

Let's check whetehr we correctly converted the variable using the `table()` function which produces a frequency table.

```
table(dat1$RSex)
```

```
  1   2
478 571
```

```
table(dat1$female)
```

```
  0   1
478 571
```

7 Visualising data

7.1 Seminar

In this section, we will learn how to visualise data which is an important step towards understanding relationships better.

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

Let's load the dataset.

```
dat1 <- read.csv("non_western_immigrants.csv", stringsAsFactors = FALSE)
```

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(dat1)
```

```
[1] 1049   13
```

So, the `dim()` function tells us that we have data from 1049 respondents with 13 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(dat1, n = 10)
```

```
   IMMBRIT over.estimate RSex RAge Househld paper WWWhourspW religious
1        1             0    1   50        2     0          1         0
2       50             1    2   18        3     0          4         0
3       50             1    2   60        1     0          1         0
4       15             1    2   77        2     1          2         1
5       20             1    2   67        1     0          1         1
6       30             1    1   30        4     1         14         0
7       60             1    2   56        2     0          5         1
8        7             0    1   49        1     1          8         0
9       30             1    1   40        4     0          3         1
10       2             0    1   61        3     1          0         1
   employMonths urban health.good HHInc party_self
1            72     4           1    13          2
2            72     4           2     3          7
3           456     3           3     9          7
4            72     1           3     8          7
5            72     3           3     9          7
6            72     1           2     9          7
7           180     1           2    13          3
8           156     4           2    14          7
9           264     2           2    11          3
10           72     1           3     8          1
```

Finally, let's look at summary statistics of our dataset.

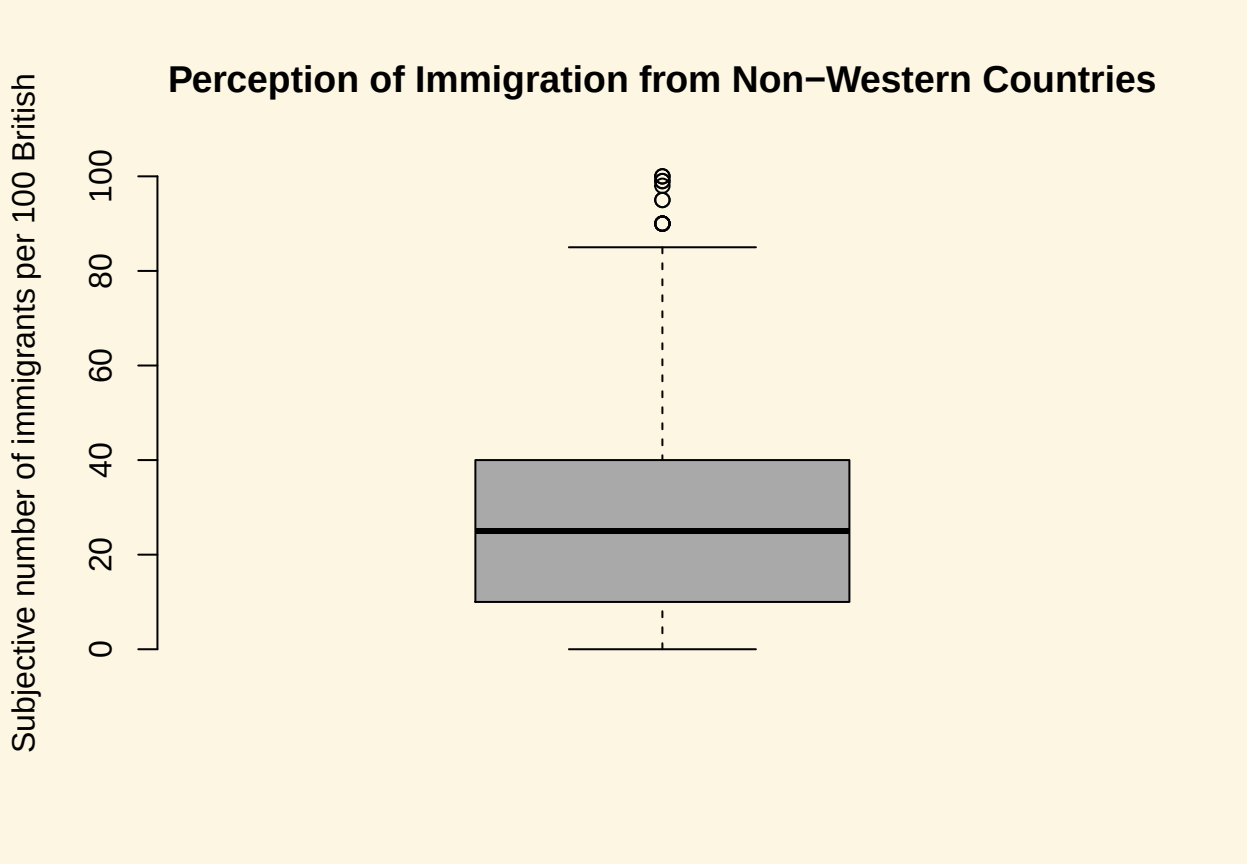
```
summary(dat1)
```

```
    IMMBRIT       over.estimate          RSex            RAge
 Min.   :  0.00   Min.   :0.0000   Min.   :1.000   Min.   :17.00
 1st Qu.: 10.00   1st Qu.:0.0000   1st Qu.:1.000   1st Qu.:36.00
 Median : 25.00   Median :1.0000   Median :2.000   Median :49.00
 Mean   : 29.03   Mean   :0.7235   Mean   :1.544   Mean   :49.75
 3rd Qu.: 40.00   3rd Qu.:1.0000   3rd Qu.:2.000   3rd Qu.:62.00
 Max.   :100.00   Max.   :1.0000   Max.   :2.000   Max.   :99.00
    Househld         paper          WWWhourspW        religious
 Min.   :1.000   Min.   :0.0000   Min.   :  0.000   Min.   :0.0000
 1st Qu.:1.000   1st Qu.:0.0000   1st Qu.:  0.000   1st Qu.:0.0000
 Median :2.000   Median :0.0000   Median :  2.000   Median :0.0000
 Mean   :2.392   Mean   :0.4538   Mean   :  5.251   Mean   :0.4929
 3rd Qu.:3.000   3rd Qu.:1.0000   3rd Qu.:  7.000   3rd Qu.:1.0000
 Max.   :8.000   Max.   :1.0000   Max.   :100.000   Max.   :1.0000
  employMonths        urban        health.good        HHInc
 Min.   :  1.00   Min.   :1.000   Min.   :0.000   Min.   : 1.000
 1st Qu.: 72.00   1st Qu.:2.000   1st Qu.:2.000   1st Qu.: 6.000
 Median : 72.00   Median :3.000   Median :2.000   Median : 9.000
 Mean   : 86.56   Mean   :2.568   Mean   :2.044   Mean   : 9.586
 3rd Qu.: 72.00   3rd Qu.:3.000   3rd Qu.:3.000   3rd Qu.:13.000
 Max.   :600.00   Max.   :4.000   Max.   :3.000   Max.   :17.000
   party_self
 Min.   :1.000
 1st Qu.:1.000
 Median :2.000
 Mean   :3.825
 3rd Qu.:7.000
 Max.   :7.000
```

7.1.1 Plots

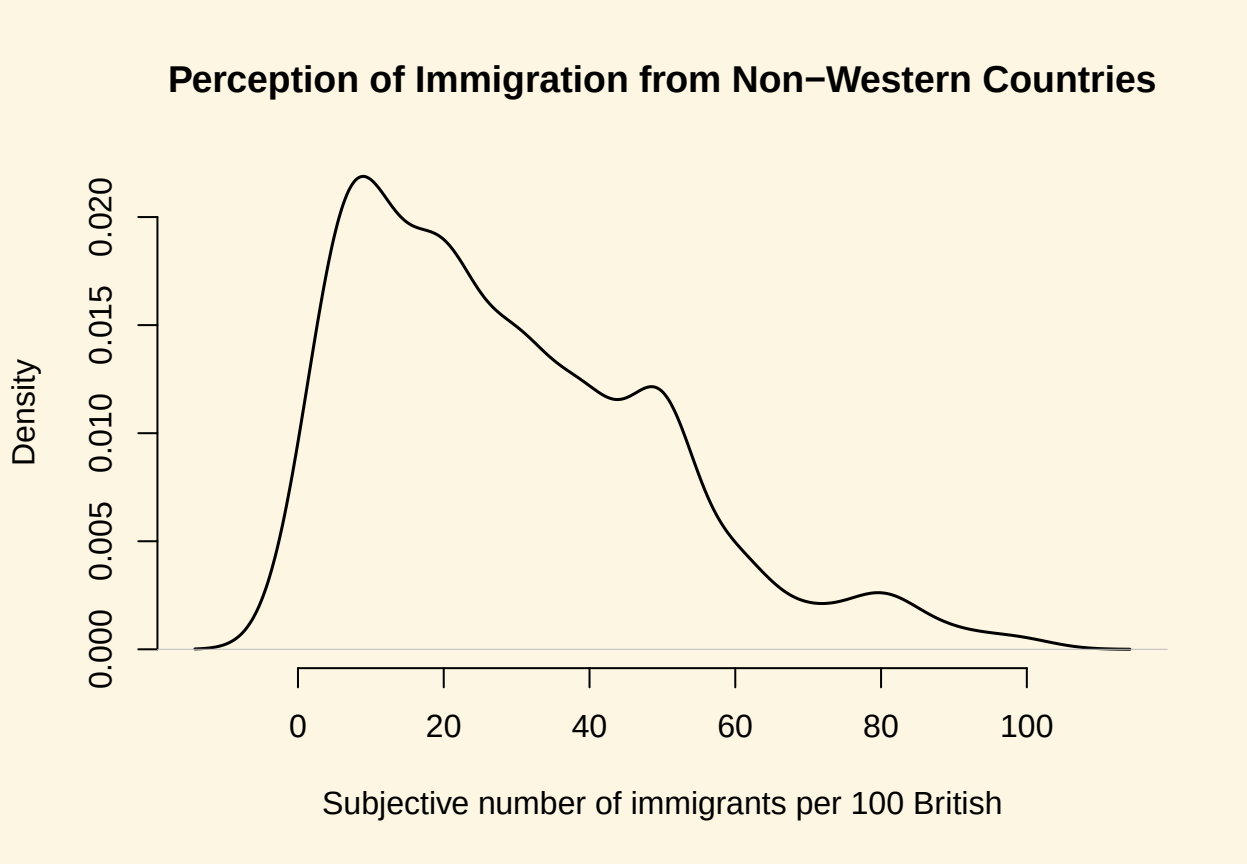
We can visualize the data with the help of a boxplot, so let's see how the perception of the number of immigrants is distributed.

```
# how good are we at guessing immigration
boxplot(
  dat1$IMMBRIT,
  main = "Perception of Immigration from Non-Western Countries",
  ylab = "Subjective number of immigrants per 100 British",
  frame.plot = FALSE, col = "darkgray"
  )
```



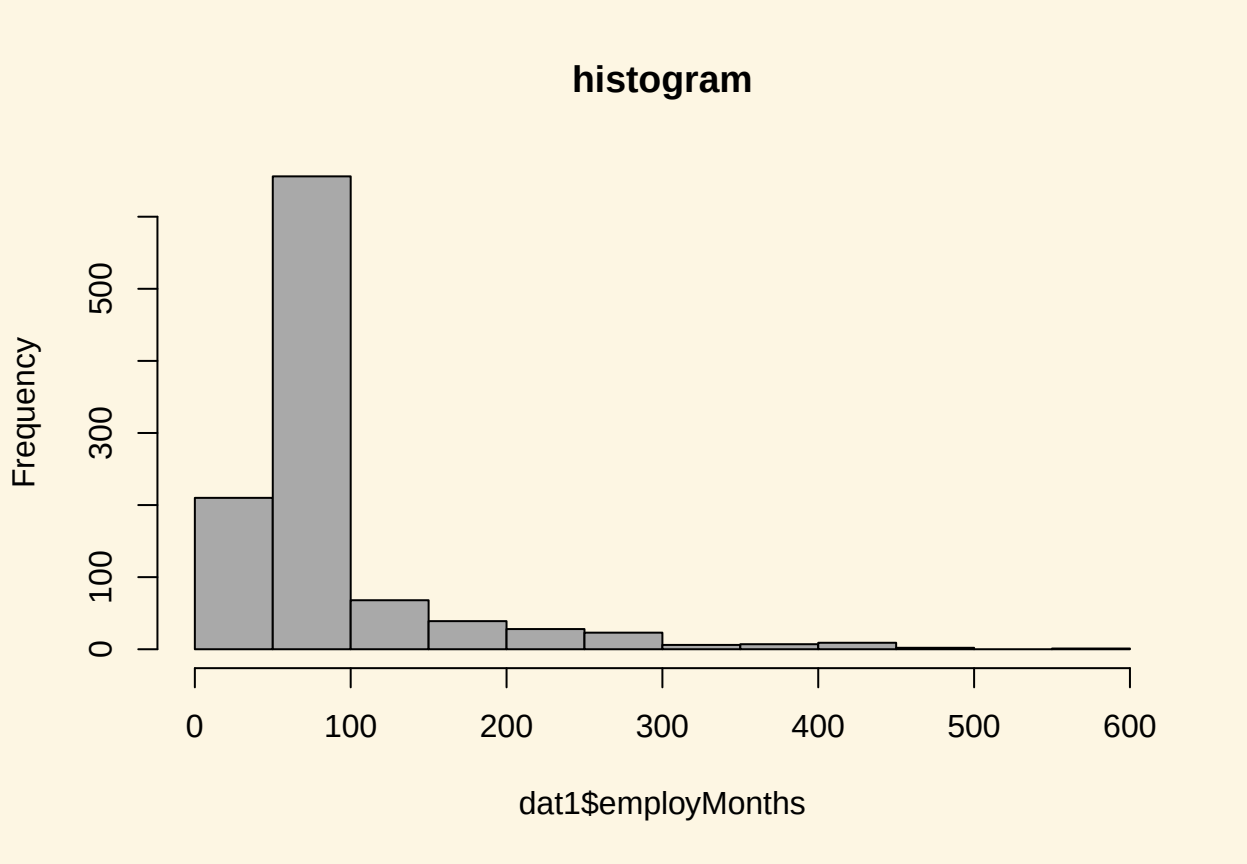
Notice how the lower whisker is much shorter than the upper one. The distribution is right skewed. The right tail (higher values) is a lot longer. We can see this beter using a density plot. We combine R's `denisty()` function with the `plot()` function.

```
plot(
  density(dat1$IMMBRIT),
  bty = "n",
  lwd = 1.5,
  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( dat1$employMonths, main = "histogram", col = "darkgray")
```



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 first create the binary variables *Cons* and *Lab* for (Conservatives and Labour respectively) using the `ifelse()` function.

```
dat1$Cons <- ifelse(dat1$party_self == 1, yes = 1, no = 0)
dat1$Lab <- ifelse(dat1$party_self == 2, yes = 1, no = 0)
```

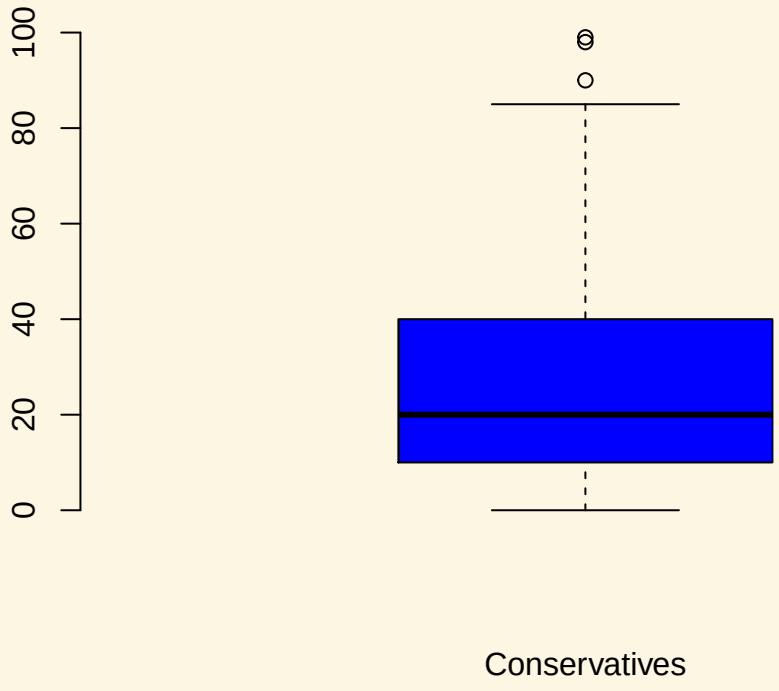
We have a look at the variable *Cons* using the `table()` function first.

```
table(dat1$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.

```
# boxplot of immbrit for those observations where Cons is 1
boxplot(
  dat1$IMMBRIT[dat1$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )
```

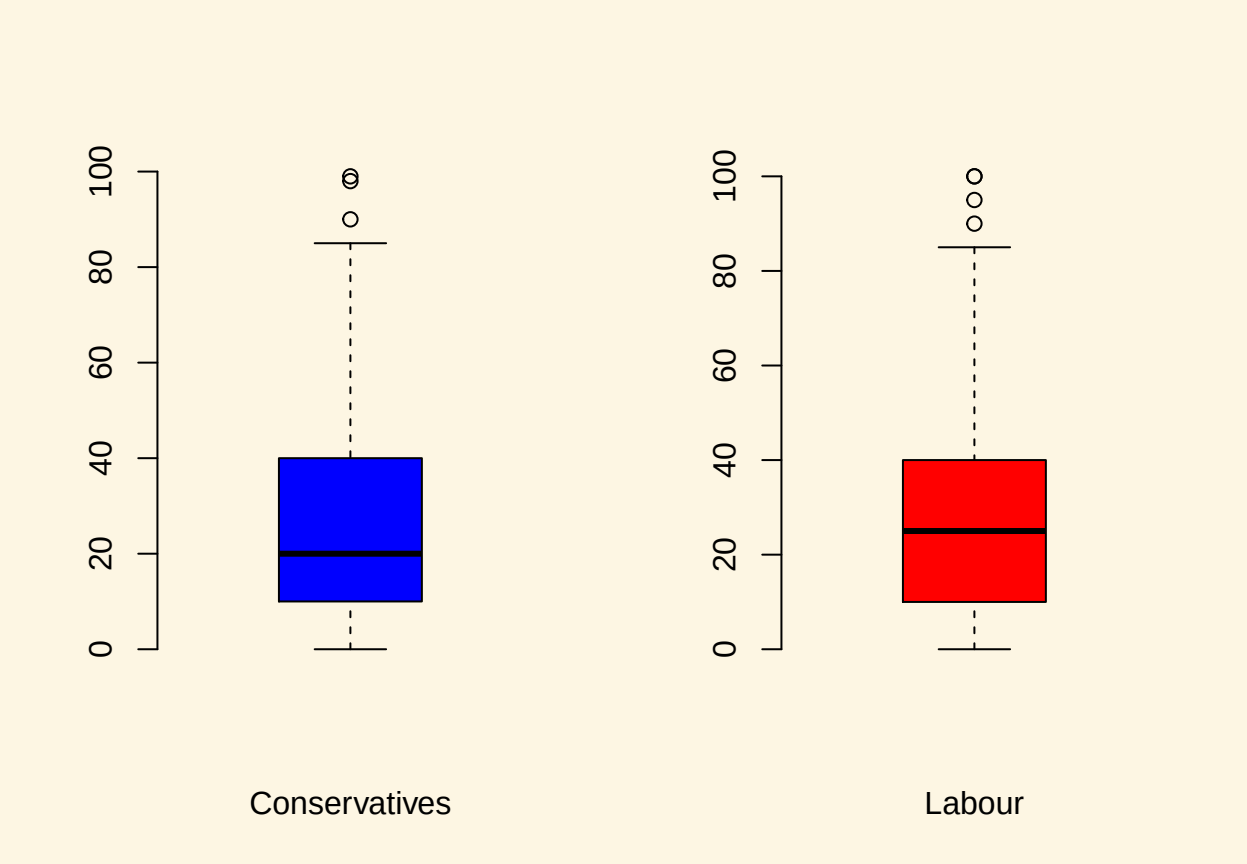


We would now like to compare the distribution of the perception fo Conservatives to the distribution among Labour respondents. We can subset the data just like we did for the Conservative Party. In addtion, we want to plot the two plots next to each other, i.e., they should be in the same plot. We can 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(
  dat1$IMMBRIT[dat1$Cons==1],
  frame.plot = FALSE,
  xlab = "Conservatives",
  col = "blue"
  )

# plot 2
boxplot(
  dat1$IMMBRIT[dat1$Lab==1],
  frame.plot = FALSE,
  xlab = "Labour",
  col = "red"
  )
```



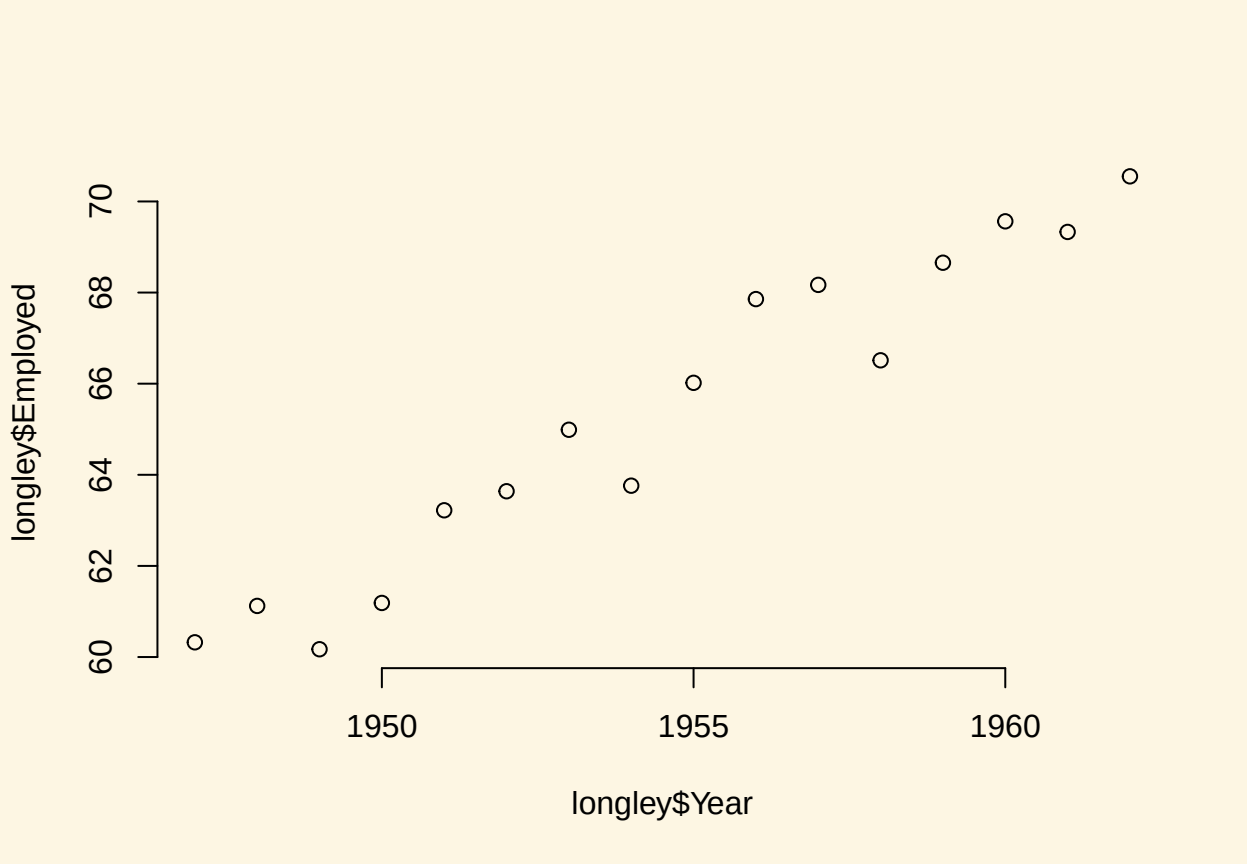
It is very hard to spot differences. The distributions are similar. The median for Labour respondents is larger which mean that the central Labour respondent over-estimates immigration more than the central Conservative respondent.

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.

```
help(longley)
```

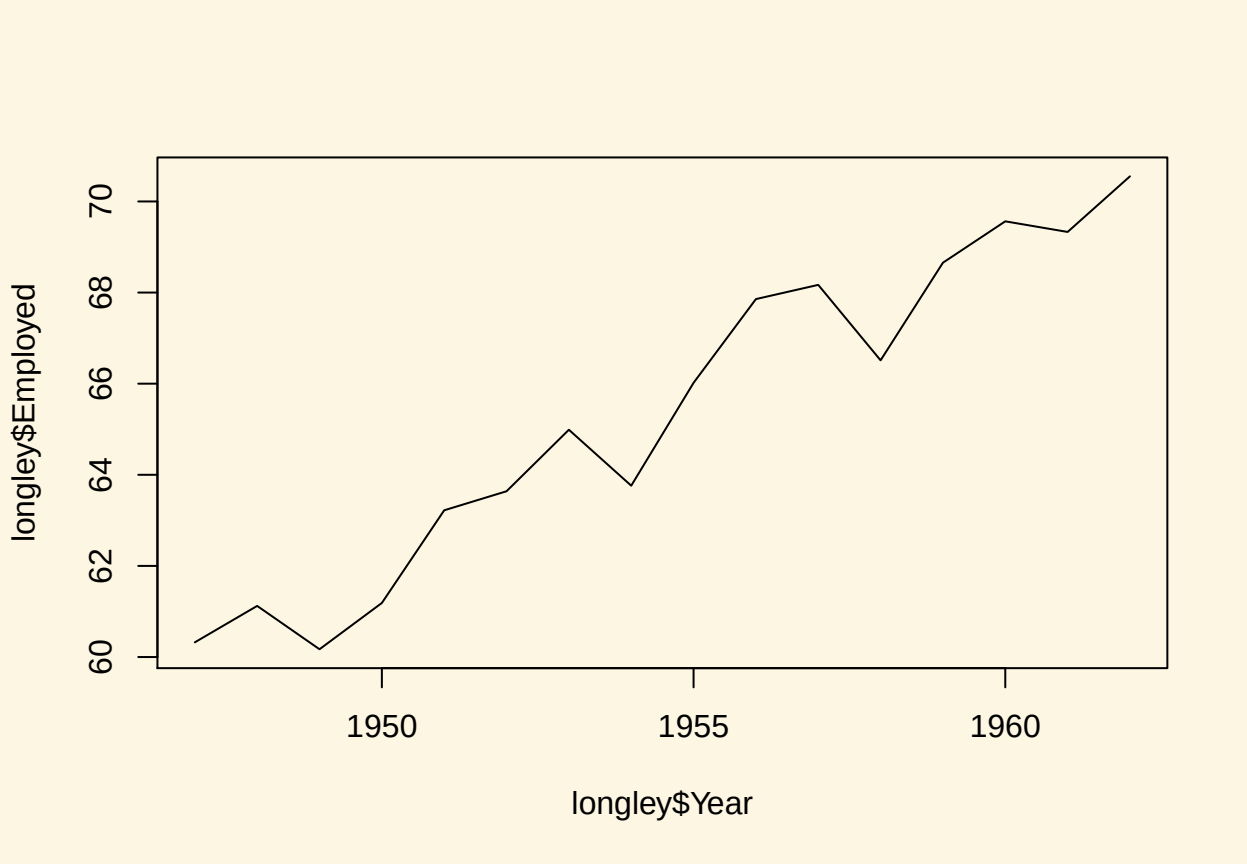
Let's create a scatterplot with the `Year` variable on the x-axis and `Employed` on the y-axis.

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



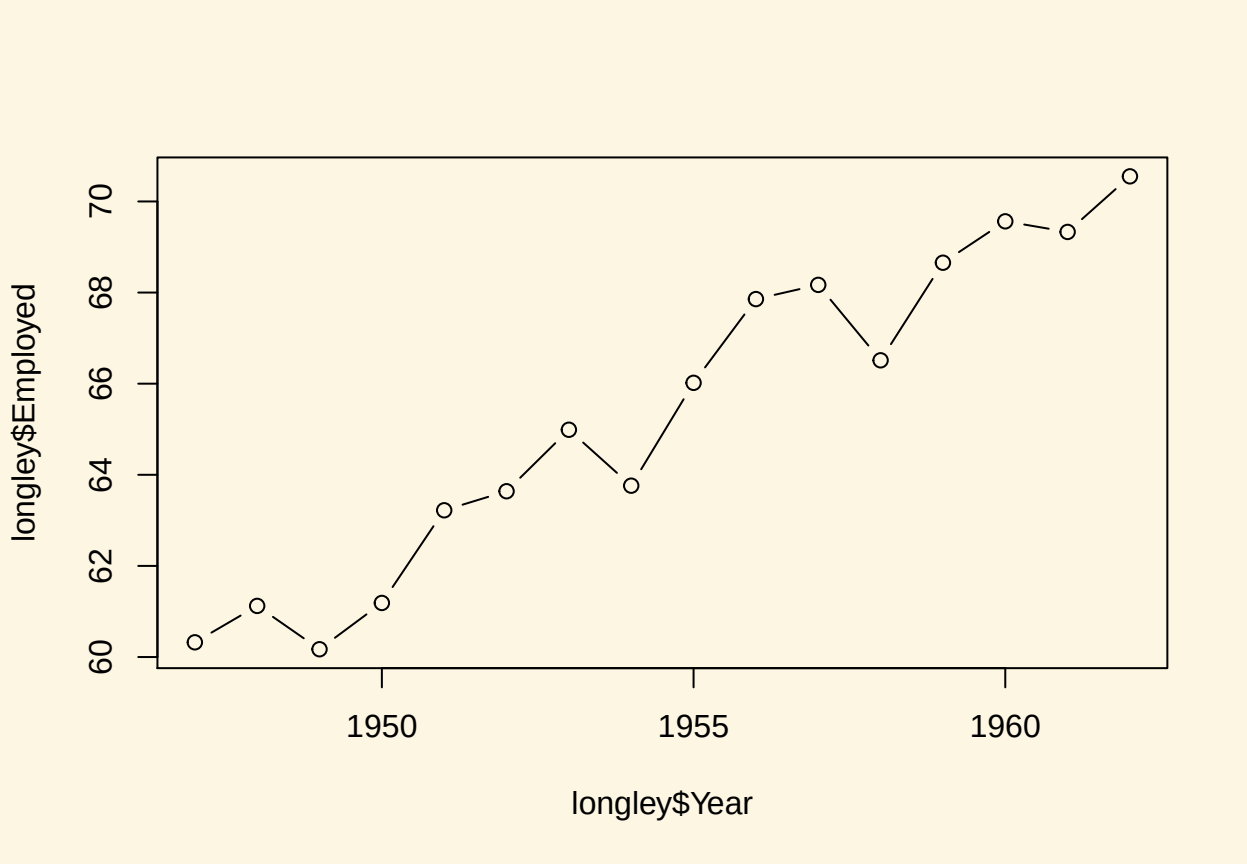
To create a line plot instead, we use the same function with one additional argument `type = "l"`.

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



Create a plot that includes both points and lines.

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



8 Correlations and differences in means

8.1 Seminar

In this session, we will cover bi-variate relationships, that is relationships between two variables. For relationships between two continuous variables, we will look at correlations and plots and for relationships between a continuous dependent variable and a binary independent variable, we will look at differences in means.

8.1.1 Sample Variance and Sample Standard Deviation

The sample variance and sample standard deviation inform us about the degree of variability of our data. Suppose, we were to roll the dice 10 times. We could then compute the mean value that we roll. The sample standard deviation measures by how much an average roll of the dice will deviate from the mean value.

We start rolling the dice, using R's random number generator and the `runif()` function. The function randomly draws numbers from a uniform distribution. In a uniform distribution each value has the same probability of being drawn. All six sides of a die should be equally likely if the die is fair. Hence, the uniform distribution.

`runif()` takes three arguments. `n` is the number of values to be drawn. `min` is the minimum value and `max` is the maximum value.

```
# random draw of 10 values from a uniform distribution
dice <- runif(n = 10, min = 1, max = 7)
dice
```

```
 [1] 2.521300 3.467514 1.369289 2.423361 4.019759 4.134584 4.994683
```

```
 [8] 6.917358 1.677068 4.005559
```

We have indeed drawn 10 numbers but they are not integers as we would like—we want to simulate a die, so the values should be 1, 2, 3, 4, 5 or 6. We will return to this in a moment but for now let's return to the randomness. Let's draw 10 numbers again:

```
# random draw of 10 values from a uniform distribution
dice2 <- runif(n = 10, min = 1, max = 7)

# first draw
dice
```

```
 [1] 2.521300 3.467514 1.369289 2.423361 4.019759 4.134584 4.994683
 [8] 6.917358 1.677068 4.005559
```

```
# second draw
dice2
```

```
 [1] 1.037233 1.183132 5.857390 2.648824 1.740200 2.385746 6.568719
 [8] 6.750616 1.130504 6.938775
```

The numbers of the first and second roll differ because we have drawn values at random. To make our results replicate and to ensure that everyone in the seminar works with the same numbers, we set R's random number generator with the `set.seed()` function. As argument we plug in some arbitrary value (it does not matter which but using a different one will lead to a different quasi-random draw).

```
# set random number generator
set.seed(123)

# random draw of 10 values from a uniform distribution
dice <- runif(n = 10, min = 1, max = 7)
dice
```

```
 [1] 2.725465 5.729831 3.453862 6.298104 6.642804 1.273339 4.168633
 [8] 6.354514 4.308610 3.739688
```

You should all have the same values. If not, run `set.seed()` again and then do the random draw once. If you do it more than once, the numbers will change. Let's see how this works:

```
# set random number generator
set.seed(123)

# 1st random draw of 10 values from a uniform distribution
dice <- runif(n = 10, min = 1, max = 7)
dice
```

```
 [1] 2.725465 5.729831 3.453862 6.298104 6.642804 1.273339 4.168633
 [8] 6.354514 4.308610 3.739688
```

```
# 2nd random draw of 10 values from a uniform distribution
dice2 <- runif(n = 10, min = 1, max = 7)
dice2
```

```
 [1] 6.741000 3.720005 5.065424 4.435800 1.617548 6.398950 2.476526
 [8] 1.252357 2.967524 6.727022
```

```
# reset random number generator
set.seed(123)

# 3rd random draw of 10 values from a uniform distribution
```

```
dice3 <- runif(n = 10, min = 1, max = 7)
dice3
```

```
 [1] 2.725465 5.729831 3.453862 6.298104 6.642804 1.273339 4.168633
 [8] 6.354514 4.308610 3.739688
```

```
# 4th random draw of 10 values from a uniform distribution
dice4 <- runif(n = 10, min = 1, max = 7)
dice4
```

```
 [1] 6.741000 3.720005 5.065424 4.435800 1.617548 6.398950 2.476526
 [8] 1.252357 2.967524 6.727022
```

As you can see, the the draws from **dice** and **dice3** are the same and the draws from **dice2** and **dice4** are the same as well. Let's make the values integers with the `as.integer()` function which simply cuts off all decimal places.

```
# reset random number generator
set.seed(123)
# random draw of 10 numbers from a uniform distribution with minimum 1 and maximum 7
dice <- runif(10, 1, 7)
# cut off decimals places
dice <- as.integer(dice)
dice
```

```
 [1] 2 5 3 6 6 1 4 6 4 3
```

```
# frequency of dice rolls
table(dice)
```

```
dice
1 2 3 4 5 6
1 1 2 2 1 3
```

We have rolled a six relatively often. All sides should be equally likely but due to sampling variability, we have rolled the six most often. The expected value of a die is 3.5. That is:

$$1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = 3.5$$

We compute the mean in our sample and the standard deviation. Let's start with the mean. Do so yourself.

```
dice.mean <- mean(dice)
dice.mean
```

```
[1] 4
```

The sample standard deviation tells by how much an average roll of the dice differs from the estimated sample mean. Estimate the sample standard deviation on your own using the `sd()` function.

```
std.dev <- sd(dice)
std.dev
```

```
[1] 1.763834
```

An average deviation from the sample mean is 1.76.

8.1.2 T test for the sample mean

Our estimate of the mean is 4. The expected value is 3.5. Is this evidence that the die is loaded? The null hypothesis is that the die is fair. The alternative hypothesis is that the die is loaded. Run a t test using the `t.test()` function. The syntax of the function is:

`t.test(formula, mu, alt, conf)`

Lets have a look at the arguments.

Arguments	Description
`formula`	The formula describes the relationship between the dependent and independent variables, for example: • `dependent.variable ~ independent.variable`. We will do this in the t-test for the difference in means. Here, we have only one estimated mean. So, we write: • `variable.name`
`mu`	Here, we set the null hypothesis. The null hypothesis is that the true population mean is 10000. Thus, we set `mu = 10000`.
`alt`	There are two alternatives to the null hypothesis that the difference in means is zero. The difference could either be smaller or it could be larger than zero. To test against both alternatives, we set `alt = "two.sided"`.
`conf`	Here, we set the level of confidence that we want in rejecting the null hypothesis. Common confidence intervals are: 95%, 99%, and 99.9%.

```
t.test(dice, mu = 3.5, conf.level = .95, alt = "two.sided")
```

```

    One Sample t-test

data:  dice
t = 0.89642, df = 9, p-value = 0.3934
alternative hypothesis: true mean is not equal to 3.5
95 percent confidence interval:
 2.738229 5.261771
sample estimates:
mean of x
        4
```

8.1.3 Loading real data

We will load a new dataset from the Quality of Government Institute. The dataset is called *QoG2012.csv*. The codebook follows.

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 load the csv dataset yourself.

8.1.4 T-test (one-sample hypothesis test) with real data

A knowledgeable friend declares that worldwide wealth stands at exactly 10 000 US dollars per capita today. We would like to know whether she is right and tease her relentlessly if she isn't.

So, first we take the mean of the wealth variable `wdi_gdpc`.

```
mean(world.data$wdi_gdpc)
```

```
[1] NA
```

R returns `NA` because for some countries we have no reliable information on their per capita wealth. `NA` means that the data is missing. We can tell the `mean()` function to estimate the mean only for those countries we have data for, we will, therefore, ignore the countries we do not have information for.

We do so by setting the argument `na.rm` to TRUE like so: `mean(dataset_name$var_name, na.rm = TRUE)`.

```
wealth.mean <- mean(world.data$wdi_gdpc, na.rm = TRUE)
wealth.mean
```

```
[1] 10184.09
```

Wow, our friend is quite close. Substantially, the difference of our friends claim to our estimate is small but we could still find that the difference is statistically significant (it's a noticeable systematic difference).

Because we do not have information on all countries, our 10184.09 is an estimate and the true population mean – the population here would be all countries in the world – may be 10000 as our friend claims. We test this statistically.

In statistics jargon: we would like to test whether our estimate is statistically different from the 10000 figure (the null hypothesis) suggested by our friend. Put differently, we would like to know the probability that we estimate 10184.09 if the true mean of all countries is 10000. This probability is called the p value. Think of the p value of the probability that the relationship that we see, immerged by chance alone. If the p value is low, we can be reasonably confident that the observed relationship did not imerge by chance alone.

We usually compare our p value to a threshold value. That threshold value is called the alpha level and it is the tolerance for false positives. If the p value is less than this value, we speak of a statisically significant relationship. Otherwise, a relationship is insignificant which means we are not confident enought that result is not due to chance.

We must specify our alpha level before we carry out analysis. The acceptable alpha level depends on the quantity of data that you have and how accurartely data is measured. Essentially, the acceptable value is arbitaray. In social sciences the usual threshold value 0.5.

If the p value is less than 0.5, a relationship is significant in our eyes.

We now test whether our estimate is systematically different from the proprosed 10000 figure or not. We do this using the the t-test function `t.test()`.

```
t.test(world.data$wdi_gdpc, mu = 10000, alt = "two.sided")
```

```
    One Sample t-test

data:  world.data$wdi_gdpc
t = 0.19951, df = 177, p-value = 0.8421
alternative hypothesis: true mean is not equal to 10000
95 percent confidence interval:
  8363.113 12005.069
sample estimates:
mean of x
 10184.09
```

The results are similar. Therefore, we can conclude that we are unable to reject the null hypothesis suggested by our friend that the population mean is equal to `10000`. In statistics jargon, we fail to reject the null hypothesis.

Let's move on to a t-test to test the difference between two estimated means.

8.1.5 T-test (difference in means)

We are interested in whether there is a difference in income between countries that have an independent judiciary and countries that do not have an independent judiciary. Put more formally, we are interested in the difference between two conditional means. Recall that a conditional mean is the mean in a subpopulation such as the mean of income given that the country was a victim of colonialization (conditional mean 1).

The t-test is the appropriate test-statistic. Our interval-level dependent variable is `wdi_gdpc` which is GDP per capita taken from the World Development Indicators of the World Bank. Our binary independent variable is `h_j`.

Let's check the summary statistics of our dependent variable GDP per captia using the `summary()`. It returns several descriptive statistics as well as the number of NA observations (missing values). Missing values mean that we have no information on the correct value of the variable for an observation. Missing values may be missing for many different reasons. We need to be aware of missings because we cannot calculate with missings.

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

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  226.2  1768.0  5326.1 10184.1 12976.5 63686.7      16
```

We use the `which()` function to identify the row-numbers of the countries in our dataset that have free judiciaries. The code below returns the row index numbers of countries with free judiciaires.

```
which(world.data$h_j == 1)
```

```
 [1]   9  10  15  16  20  25  31  36  38  43  44  46  47  48  49  55  57
[18]  59  60  65  75  76  77  79  81  82  83  86  88  91  92  97 101 102
[35] 113 114 116 119 122 124 125 128 138 139 143 156 157 158 159 163 167
[52] 168 169 171 174 177 180 181 182 183 184 185 186 194
```

Now, all we need is to index the dataset like we did yesterday. We access the variable that we want (`wdi_gdpc`) with the dollar sign and the rows in square brackets. The code below returns the per capita wealth of the countries with a free judiciary.

```
mean( world.data$wdi_gdpc[which(world.data$h_j == 1)], na.rm = TRUE)
```

```
[1] 17826.59
```

Now, go ahead and find the mean per capita wealth of countries with controlled judiciaries yourself.

```
mean( world.data$wdi_gdpc[which(world.data$h_j == 0)], na.rm = TRUE)
```

```
[1] 5884.882
```

There is a numeric difference. However, we know that samples are subject to sampling variability. We therefore need to quantify the uncertainty that results from variable samples. To assess whether we can be reasonably sure that the difference between the estimates of wealth is not due to a strage sample but an actual difference in the population, we carry out the t test.

```
t.test(world.data$wdi_gdpc[world.data$h_j == 1], world.data$wdi_gdpc[world.data$h_j == 0],
       mu = 0, alt = "two.sided", conf = 0.95)
```

```
    Welch Two Sample t-test

data:  world.data$wdi_gdpc[world.data$h_j == 1] and world.data$wdi_gdpc[world.data$h_j == 0]
t = 6.0094, df = 98.261, p-value = 3.165e-08
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
```

```
  7998.36 15885.06
sample estimates:
mean of x mean of y
17826.591  5884.882
```

Let's interpret the results we get from `t.test()`. The first argument is the vector of wealth values conditional on a free independent judiciary. The second argument is the vector of wealth values conditional on an independent judiciary. The variable of interest is wealth - we call this the dependent variable. We call it dependent because it depends or is conditional on the value of another variable which is the judiciary. We call the conditioning variable, the independent variable.

In our example the question is: Do countries with independent judiciaries have different mean income levels than countries without independent judiciaries?

In the following line you see the t-value, the degrees of freedom and the p-value. Knowing the t-value and the degrees of freedom you can check in a table on t distributions how likely you were to observe this data, if the null-hypothesis was true. The p-value gives you this probability directly. For example, a p-value of 0.02 would mean that the probability of seeing this data given that there is no difference in incomes between countries with and without independent judiciaries *in the population*, is 2%. Here the p-value is much smaller than this: 3.165e-08 = 0.00000003156!

In the next line you see the 95% confidence interval because we specified `conf=0.95`. If you were to take 100 samples and in each you checked the means of the two groups, 95 times the difference in means would be within the interval you see there.

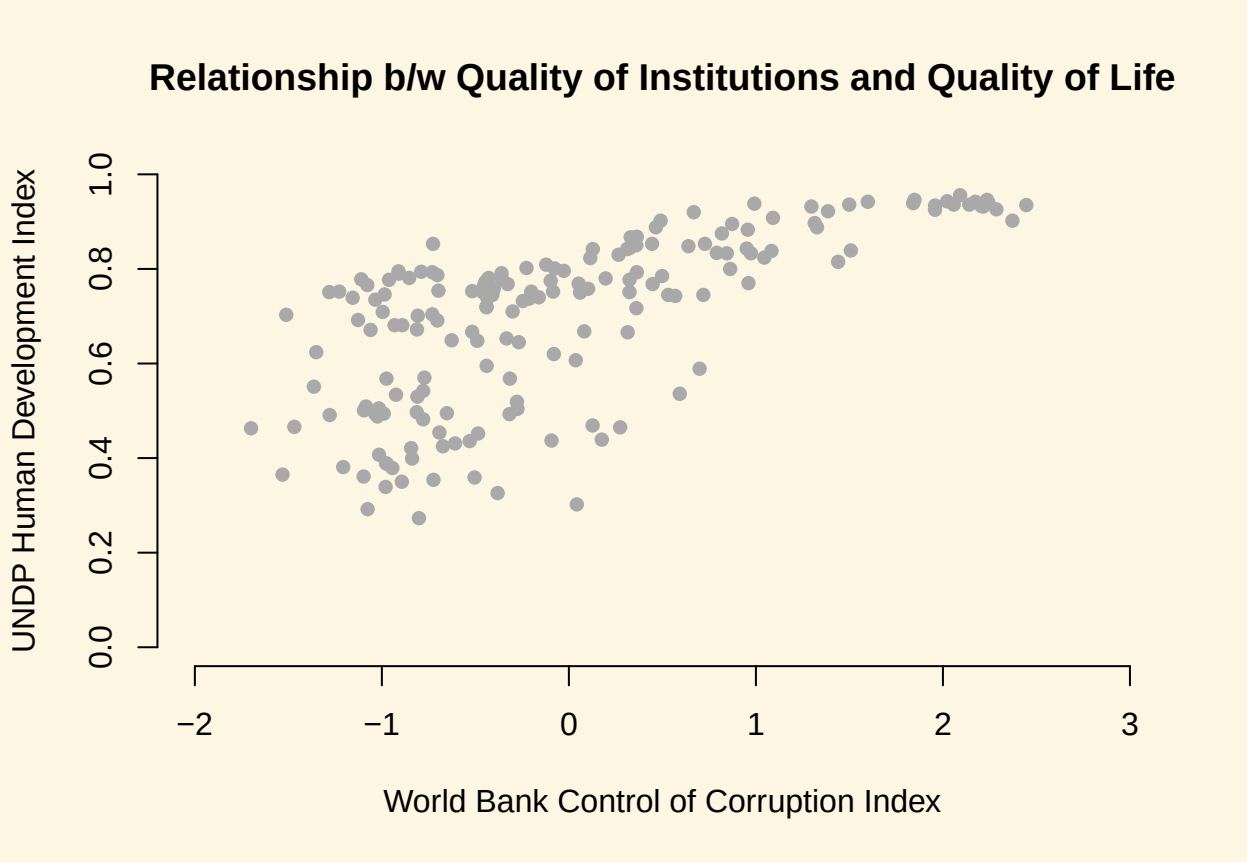
At the very bottom you see the means of the dependent variable by the two groups of the independent variable. These are the means that we estimated above. In our example, you see the mean income levels in countries were the executive has some control over the judiciary, and in countries were the judiciary is independent.

8.1.6 Relationships between continuous variables

The best way to get a sense for the relationship beetween two continuous variables is to do a scatter plot. The human development index measures the quality of life and the World Bank Control of Corruption Index is a measure for the quality if institutions. We want to evaluate whether the two variables are related at all. Here, we could form the hypothesis that better insitutions improve the quality of life of citizens.

To investigate this relationship, we construct a scatterplot.

```r
plot(
  x = world.data$wbgi_cce,
  y = world.data$undp_hdi,
  xlim = c(-2, 3),
  ylim = c(0, 1),
  frame = FALSE,
  xlab = "World Bank Control of Corruption Index",
  ylab = "UNDP Human Development Index",
  main = "Relationship b/w Quality of Institutions and Quality of Life",
  pch = 16,
  col = "darkgray"
)
```



The plot will give you a good idea whether about whether these two variables are related or not. Sometimes, the correleation coefficient is reported. The correlation coefficient is a measure of **linear** association. It can take values between -1 and +1. Where -1 is a perfect negative relationship, 0 is a no relationship and +1 is a perfect positive relationship.

While the correlation coefficient is widely used as a summary statistic. Its weakness is that it is a measure of linear association only. That means, there could be a curvilinear realtionship which we miss (for instance a u-shaped relationship). visual inspection using a scatterplot is usually better than estimating the correlation coefficient.

We estimate the correlation coefficient in the following:

```
cor(y = world.data$undp_hdi, x = world.data$wbgi_cce, use = "complete.obs")
```

```
[1] 0.6821114
```

Argument	Description
`x`	The x variable that you want to correlate.
`y`	The y variable that you want to correlate.
`use`	How R should handle missing values. `use="complete.obs"` will use only those rows where neither `x` nor `y` is missing.

The interpretation of the correlation coefficient is that there is a positive relationship. There is really no threshold value beyond or below which a correlation coefficient is large or small. This depends entirely on the context in this is where your domain knowledge would come in.

9 Regression

9.1 Seminar

In this section, we will cover regression models. We will first introduce the bivariate linear regression model. We will then move to linear models with multiple independent variables. Next, we discuss counfounders as threats to our inference. Finally, we will introduce the logistic regression model.

First, however, we install a new pacakge called `texreg` which makes it easy to produce publication quality output from our regression models. We'll discuss this package in more detail as we go along. For now let's install the package with the `install.packages("texreg")` function and then load it using `library(texreg)`.

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

We will use a dataset collected by the US census bureau that contains several socioeconomic indicators.

```
communities <- read.csv("communities.csv")
```

The dataset includes 38 variables but we're only interested in a handful at the moment.

Variable	Description
PctUnemployed	proportion of citizens in each community who are unemployed
PctNotHSGrad	proportion of citizens in each community who failed to finish high-school
population	proportion of adult population living in cities

```
  PctUnemployed PctNotHSGrad population
1          0.27         0.18       0.19
2          0.27         0.24       0.00
3          0.36         0.43       0.00
4          0.33         0.25       0.04
5          0.12         0.30       0.01
6          0.10         0.12       0.02
```

If we summarize these variables with the `summary()` function, we will see that they are both measured as proportions (they vary between 0 and 1):

```
summary(communities$PctUnemployed)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.2200  0.3200  0.3635  0.4800  1.0000
```

```
summary(communities$PctNotHSGrad)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.2300  0.3600  0.3833  0.5100  1.0000
```

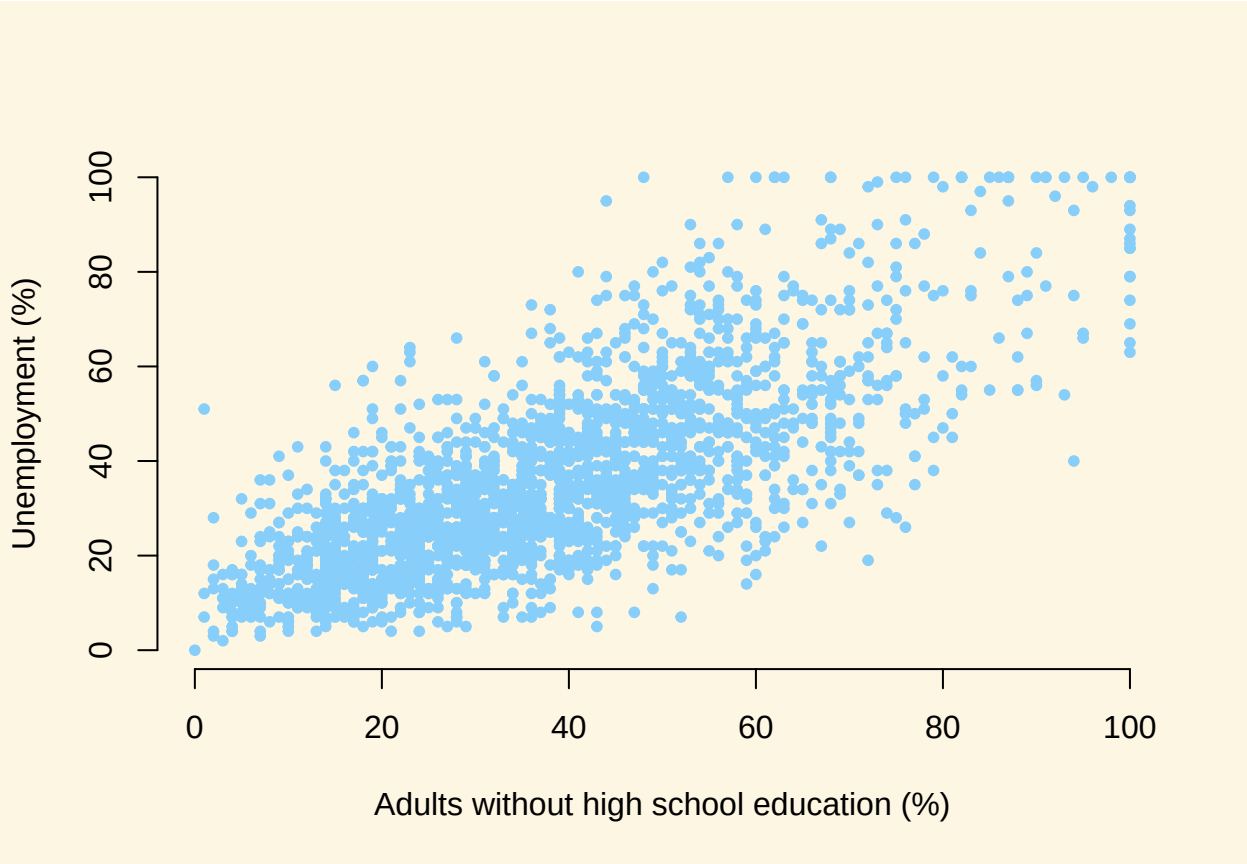
It will be a little easier to interpret the regression output if we convert these to percentages rather than proportions. We can do this with the following lines of code:

```
communities$PctUnemployed <- communities$PctUnemployed * 100
communities$PctNotHSGrad <- communities$PctNotHSGrad * 100
```

We can begin by drawing a scatterplot with the percentage of unemployed people on the y-axis and the percentage of adults without high-school education on the x-axis.

```
plot(
  x = communities$PctNotHSGrad,
  y = communities$PctUnemployed,
  xlab = "Adults without high school education (%)",
  ylab = "Unemployment (%)",
```

```
  frame.plot = FALSE,
  pch = 20,
  col = "LightSkyBlue"
)
```



From looking at the plot, what is the association between the unemployment rate and lack of high-school level education?

In order to answer that question empirically, we will run a linear regression using the `lm()` function in R. The `lm()` function needs to know a) the relationship we're trying to model and b) the dataset for our observations. The two arguments we need to provide to the `lm()` function are described below.

Argument	Description
`formula`	The `formula` describes the relationship between the dependent and independent variables, for example `dependent.variable ~ independent.variable` In our case, we'd like to model the relationship using the formula: `PctUnemployed ~ PctNotHSGrad`
`data`	This is simply the name of the dataset that contains the variable of interest. In our case, this is the merged dataset called `communities`.

For more information on how the `lm()` function works, type help(lm) in R.

```
model1 <- lm(PctUnemployed ~ PctNotHSGrad, data = communities)
```

9.1.1 Interpreting Regression Output

The `lm()` function has modeled the relationship between `PctUnemployed` and `PctNotHSGrad` and we've saved it in an object called `model1`. Let's use the `summary()` function to see what this linear model looks like.

```
summary(model1)
```

The output from `summary()` might seem overwhelming at first so let's break it down one item at a time.

```
Call:
(1) lm(formula = PctUnemployed ~ PctNotHSGrad, data = communities)

Residuals:
(2)     Min      1Q  Median      3Q     Max
-42.347  -8.499  -1.189   7.711  56.470

Coefficients:
             (3)       (4)        (5)      (6)
             Estimate  Std. Error t value  Pr(>|t|)
(Intercept)   7.89520   0.64833   12.18     <2e-16 ***
PctNotHSGrad  0.74239   0.01496   49.64     <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.52 on 1992 degrees of freedom
(7) Multiple R-squared:  0.553, Adjusted R-squared:  0.5527
F-statistic:  2464 on 1 and 1992 DF,  p-value: < 2.2e-16
```

#	Item	Description
1	*formula*	The *formula* describes the relationship between the dependent and independent variables
2	*residuals*	The differences between the observed values and the predicted values are called *residuals*.
3	*coefficients*	The *coefficients* for all the *independent* variables and the intercept. Using the *coefficients* we can write down the relationship between the *dependent* and the *independent* variables as: `PctUnemployed` = 7.8952023 + (0.7423853 * `PctNotHSGrad`) This tells us that for each unit increase in the variable `PctNotHSGrad`, the `PctUnemployed` increases by 0.7423853.
4	*standard error*	The *standard error* estimates the standard deviation of the sampling distribution of the coefficients in our model. We can think of the *standard error* as the measure of precision for the estimated coefficients.
5	*t-statistic*	The *t-statistic* is obtained by dividing the *coefficients* by the *standard error*.
6	*p-value*	The *p-value* for each of the coefficients in the model. Recall that according to the null hypotheses, the value of the coefficient of interest is zero. The *p-value* tells us whether can can reject the null hypotheses or not.

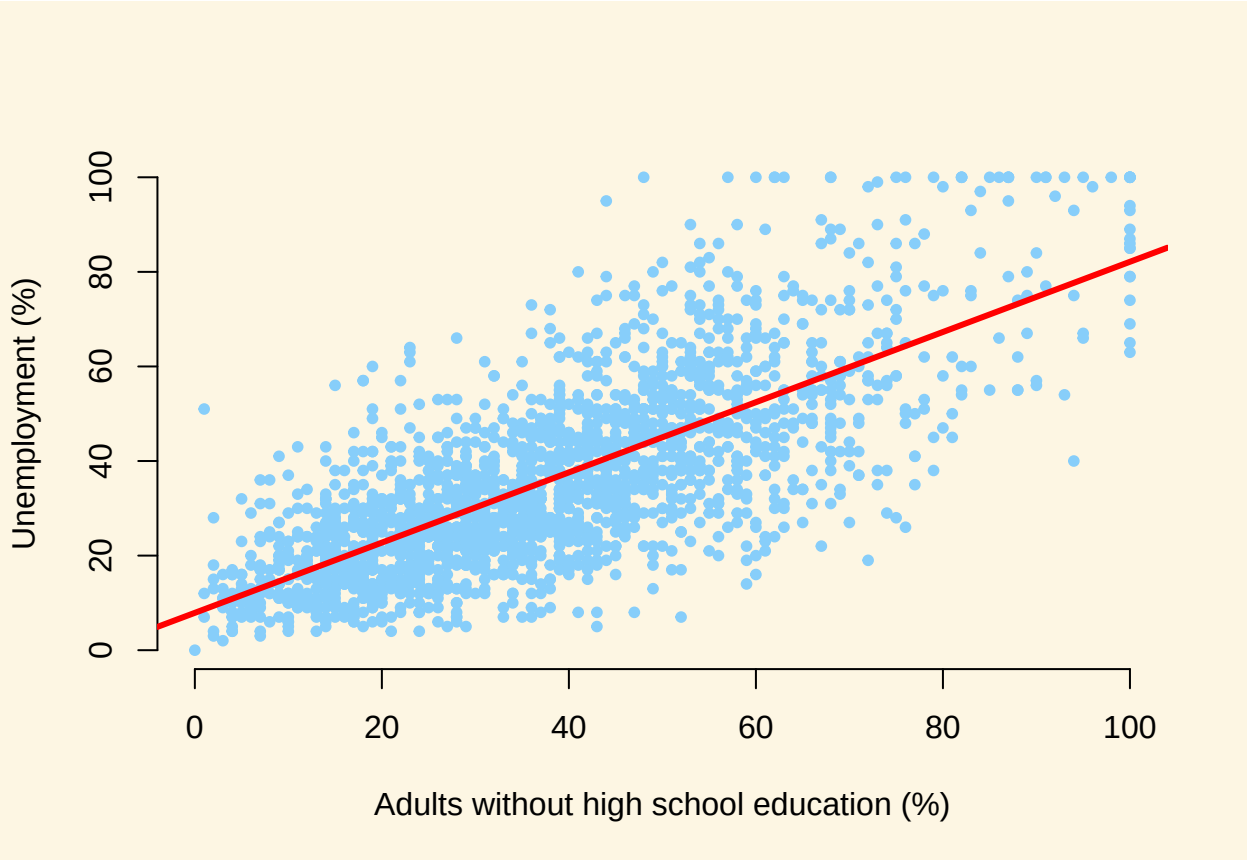
#	Item	Description
7	R^2 and *adj-*R^2	tell us how much of the variance in our model is accounted for by the *independent* variable. The *adjusted* R^2 is always smaller than R^2 as it takes into account the number of *independent* variables and degrees of freedom.

Now let's add a regression line to the scatter plot using the `abline()` function.

First we run the same `plot()` function as before, then we overlay a line with `abline()`:

```
plot(
  x = communities$PctNotHSGrad,
  y = communities$PctUnemployed,
  xlab = "Adults without high school education (%)",
  ylab = "Unemployment (%)",
  frame.plot = FALSE,
  pch = 20,
  col = "LightSkyBlue"
)

abline(model1, lwd = 3, col = "red")
```



We can see by looking at the regression line that it matches the coefficients we estimated above. For example, when `PctNotHSGrad` is equal to zero (i.e. where the line intersects the Y-axis), the predicted value for `PctUnemployed` seems to be above 0 but below 10. This is good, as the *intercept* coefficient we estimated in the regression was 7.8952023.

Similarly, the coefficient for the variable `PctNotHSGrad` was estimated to be 0.7423853, which implies that a one point increase in the percentage of citizens with no high-school education is associated with about 0.7423853 of a point increase in the percentage of citizens who are unemployed. The line in the plot seems to reflect this: it is upward sloping, so that higher levels of the no high-school variable are associated with higher levels of unemployment, but the relationship is not quite 1-to-1. That is, for each additional percentage point of citzens without high school education, the percentage of citizens who are unemployed increases by a little less than one point.

While the `summary()` function provides a slew of information about a fitted regression model, we often need to present our findings in easy to read tables similar to what you see in journal publications. The `texreg` package we loaded earlier allows us to do just that.

Let's take a look at how to display the output of a regression model on the screen using the `screenreg()` function from `texreg`.

```
screenreg(model1)
```

```
==========================
              Model 1
--------------------------
(Intercept)      7.90 ***
                (0.65)
PctNotHSGrad     0.74 ***
                (0.01)
--------------------------
R^2              0.55
Adj. R^2         0.55
Num. obs.     1994
RMSE            13.52
==========================
*** p < 0.001, ** p < 0.01, * p < 0.05
```

Here, the output includes some of the most salient details we need for interpretation. We can see the coefficient for the `PctNotHSGrad` variable, and the estimated coefficient for the intercept. Below these numbers, in brackets, we can see the standard errors. The table also reports the R^2, the adjusted R^2, the number of observations (n) and the root-mean-squared-error ($RMSE$).

One thing to note is that the table does not include either t-statistics or p-values for the estimated coefficents. Instead, the table employs a common device of using stars to denote whether a variable is statistically significant at a given alpha level.

- `***` indicates that the coefficient is significant at the 99.9% confidence level (alpha = 0.001)
- `**` indicates that the coefficient is significant at the 99% confidence level (alpha = 0.01)
- `*` indicates that the coefficient is significant at the 95% confidence level (alpha = 0.05)

Returning to our example, are there other variables that might affect the unemployment rate in our dataset? For example, is the unemployment rate higher in rural areas? To answer this question, we can swap `PctNotHSGrad` for a different independent variable. Let's use the variable `population`, which measures the proportion of adults who live in cities (rather than rural areas). Again, we can transform this proportion to a percentage with the following code:

```
communities$population <- communities$population * 100
```

Let's fit a linear model using `population` as the independent variable:

```
model2 <- lm(PctUnemployed ~ population, data = communities)
summary(model2)
```

```
Call:
lm(formula = PctUnemployed ~ population, data = communities)

Residuals:
    Min      1Q  Median      3Q     Max
-35.252 -14.715  -3.946  11.054  64.980

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 35.02042    0.49206  71.171  < 2e-16 ***
population   0.23139    0.03532   6.552  7.2e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 20.01 on 1992 degrees of freedom
Multiple R-squared:  0.0211,    Adjusted R-squared:  0.02061
F-statistic: 42.93 on 1 and 1992 DF,  p-value: 7.201e-11
```

We can show regression line from the `model2` just like we did with our first model.

```r
plot(
  x = communities$population,
  y = communities$PctUnemployed,
  xlab = "Adults living in cities (%)",
  ylab = "Unemployment (%)",
  frame.plot = FALSE,
  pch = 20,
  col = "LightSkyBlue"
)

abline(model2, lwd = 2, col = "red")
```

So we now have two models! Often, we will want to compare two estimated models side-by-side. We might want to say how the coefficients for the independent variables we included differ in `model1` and `model2`, for example. Or we may want to ask: Does `model2` offer a better fit than `model1`?

It is often useful to print the salient details from the estimated models side-by-side. We can do this by using the `screenreg()` function.

```
screenreg(list(model1, model2))
```

```
========================================
               Model 1       Model 2
----------------------------------------
(Intercept)      7.90 ***     35.02 ***
                (0.65)       (0.49)
PctNotHSGrad     0.74 ***
                (0.01)
population                    0.23 ***
                             (0.04)
----------------------------------------
R^2              0.55         0.02
Adj. R^2         0.55         0.02
Num. obs.     1994         1994
RMSE            13.52        20.01
========================================
*** p < 0.001, ** p < 0.01, * p < 0.05
```

What does this table tell us?

- The first column replicates the results from our first model. We can see that a one point increase in the percentage of citizens without high-school education is associated with an increase of 0.7423853 percentage points of unemployment, on average.
- The second column gives us the results from the second model. Here, a one point increase in the percentage of citizens who live in cities is associated with an increase of 0.2313906 percentage points of unemployment, on average
- We can also compare the R^2 values from the two models. The R^2 for `model1` is 0.5529732 and for `model2` is 0.0210968. This suggests that the model with `PctNotHSGrad` as the explanatory variable explains about 55.2973193% of the variation in unemployment. The model with `population` as the explanatory variable, on the other hand, explains just 2.1096753% of the variation in unemployment.

Finally, and this is something that might help with your coursework, let's save the same output as a Microsoft Word document using `htmlreg()`.

```
htmlreg(list(model1, model2), file = "regression_model.doc")
```

If you're using a Mac, you might want to save the file as `.html` if the Word document isn't formatted correctly.

```
htmlreg(list(model1, model2), file = "regression_model.html")
```

9.1.2 Fitted values

Once we have estimated a regression model, we can use that model to produce fitted values. Fitted values represent our "best guess" for the value of our dependent variable for a specific value of our independent variable.

Let's calculate the fitted values manually and then we'll show you how to do it in R. The fitted value formula is:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 * X_i$$

Let's say that, on the basis of `model1` we would like to know what the unemployment rate is likely to be for a community where the percentage of adults without a high-school education is equal to 10%. We can substitute in the relevant coefficients from `model1` and the value for our X variable (`10` in this case), and we get:

$$\hat{Y}_i = 7.9 + 0.74 * 10 = 15.3$$

To calculate fitted values in R, we use the `predict()` function.

The predict function takes two main arguments.

Argument	Description
`object`	The `object` is the model object that we would like to use to produce fitted values. Here, we would like to base the analysis on `model1` and so specify `object = model1` here.
`newdata`	This is an optional argument which we use to specify the values of our independent variable(s) that we would like fitted values for. If we leave this argument empty, R will automatically calculate fitted values for all of the observations in the data that we used to estimate the original model. If we include this argument, we need to provide a `data.frame` which has a variable with the same name as the independent variable in our model. Here, we specify `newdata = data.frame(PctNotHSGrad = 10)`, as we would like the fitted value for a community where 10% of adults did not complete high-school.

```r
predict(model1, newdata = data.frame(PctNotHSGrad = 10))
```

```
       1
15.31906
```

This is the same as the result we obtained when we calculated the fitted value manually. The good thing about the `predict()` function, however, is that we will be able to use it for *all* the models we study on this course, and it can be useful for calculating many different fitted values.

9.1.3 Additional Resources

- Linear Regression - Interactive App

10 Prediction and assessing prediction accuracy

10.1 Seminar

Will be uploaded tomorrow