Introduction to Statistics

Contents

About this course **1**

1 Introduction to R and RStudio **1**
1.1 Learning objectives 1

2 Data Stuctures and Data Types **4**
2.1 Seminar 4

3 Creating and Exploring Data Frames **8**
3.1 Seminar 8

4 Data Import (from csv, txt, and excel) and Saving Data Frames **11**
4.1 Seminar 11

5 Descriptive Statistics **14**
5.1 Seminar 14

About this course

This course is an introduction to statistics, R and RStudio. Our primary aims are to introduce you to and help you become familiar with RStudio and quantitative methodologies critical to your development as an analyst.

By the end of the course, you should be able to understand fundamental research methods, apply them to real world problems and acquire competency in performing statistical functions using R.

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.

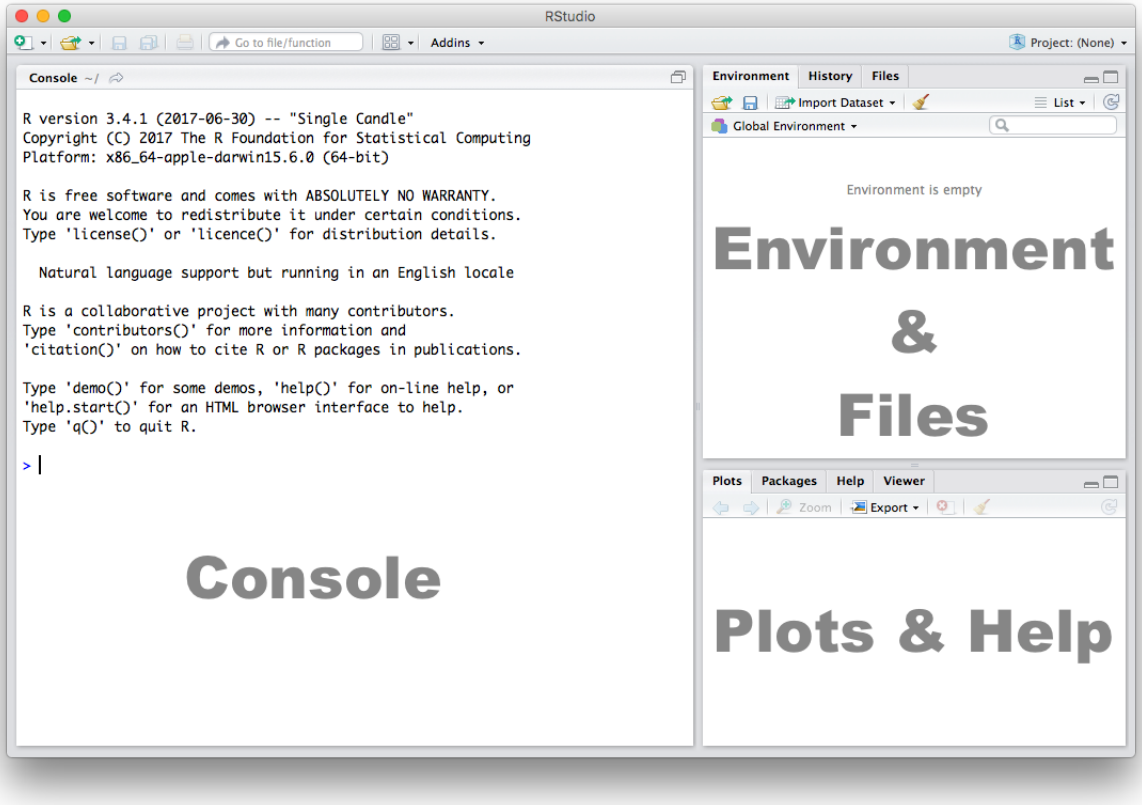
To install R and RStudio on your computer, download both from the following sources:

- 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
```

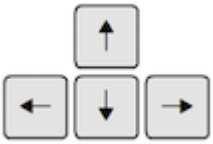
```
[1] 6
```

```
5 * 3
```

```
[1] 15
```

```
7 / 2
```

```
[1] 3.5
```



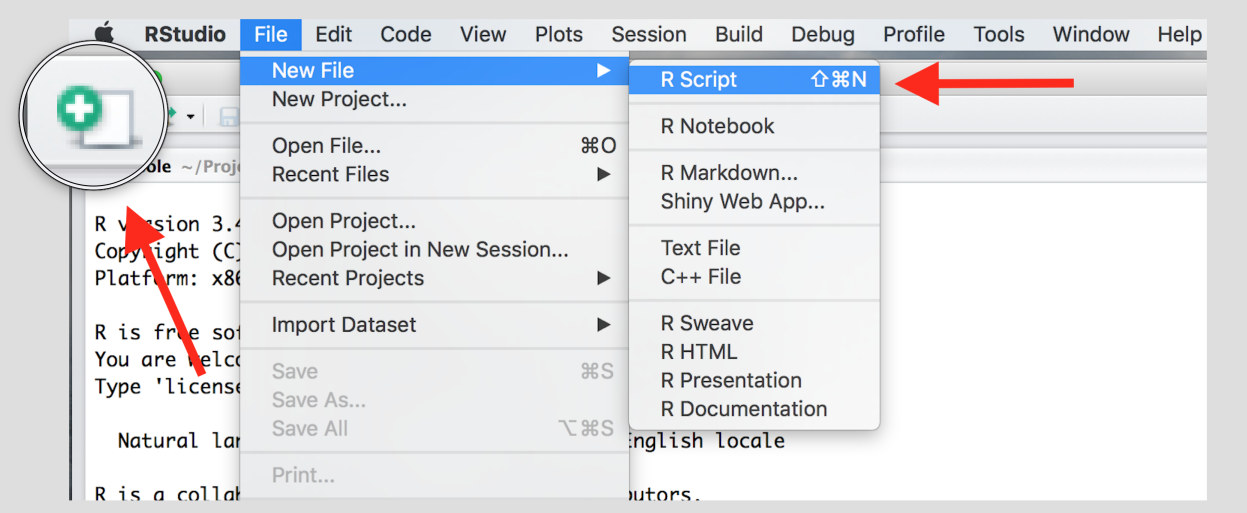
You can use the cursor or arrow keys on your keyboard to edit your code at the console:- Use the UP and DOWN keys to re-run something without typing it again- Use the LEFT and RIGHT keys to edit

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!

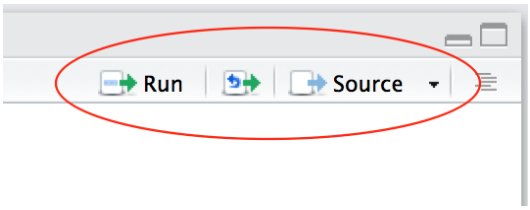
1.1.4 Scripts

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 Data Stuctures and Data 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.

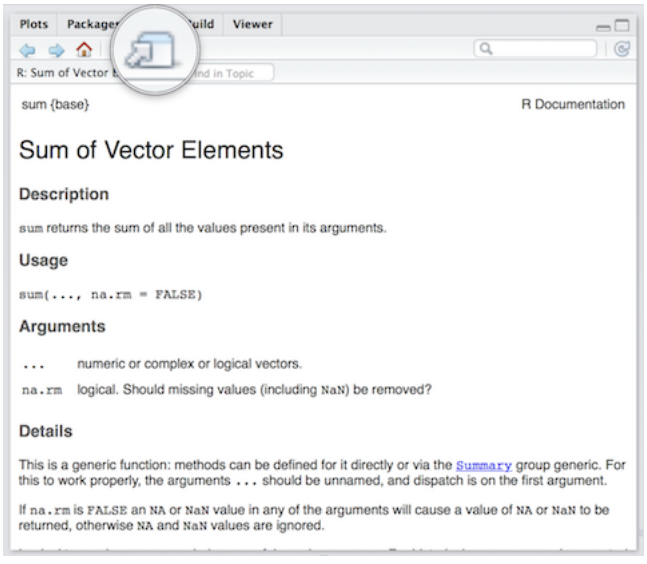
2.1.2 Getting Help

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:

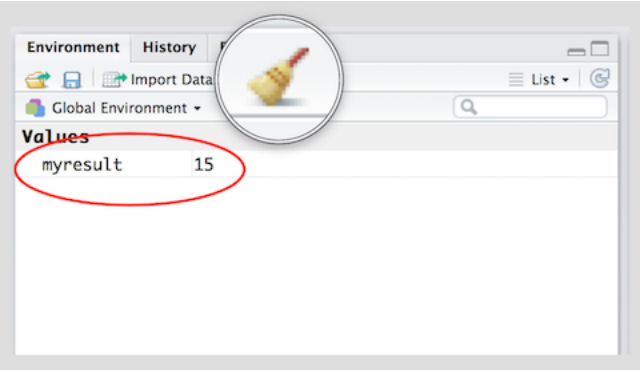
```
myresult <- sum(1, 4, 10)
```

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 Creating and Exploring Data Frames

3.1 Seminar

In this session we introduce data frames using R-syntax.

3.1.1 Clearing your workspace from previous work

To remove a specific object or in this case a matrix, we can use the following command.

```
ls()
```

```
[1] "chapter_header"
```

```
rm(mat1)
```

```
Warning in rm(mat1): object 'mat1' not found
```

Otherwise we can remove everything, which is good practice when starting new work. The same can also be achieved by clicking on the yellow broomstick in the Enviornemnt and Files Panel in R Studio (top right hand panel)

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

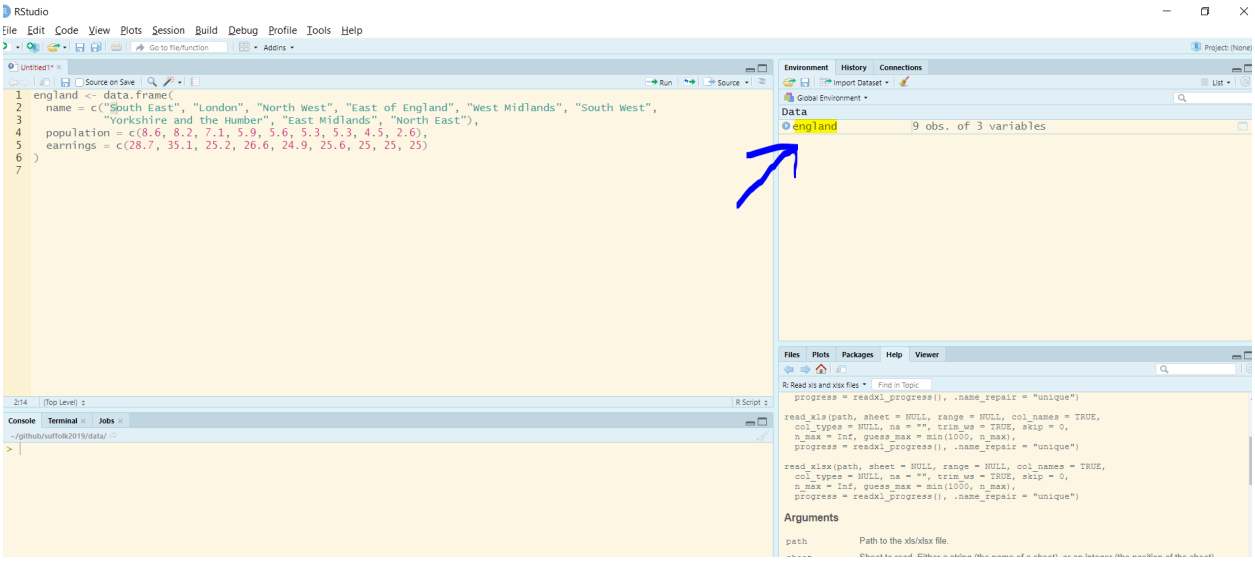
3.1.2 Creating data frames

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)
)
```

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



3.1.3 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$population
```

```
[1] 8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
```

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

```
england[, "population" ]
```

```
[1] 8.6 8.2 7.1 5.9 5.6 5.3 5.3 4.5 2.6
```

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

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

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

You can also explore the variable further. For instance, calculate the average population in England and then select regions that have a population higher than this average. Again the dataset object is followed by the `$`and then the variable name like so

```
mean(england$population)
```

```
[1] 5.9
```

```
avg.pop = mean(england$population)
england [england$population > avg.pop,
         c("name", "population")]
```

```
             name population
1      South East        8.6
```

```
2          London        8.2
3      North West        7.1
4 East of England        5.9
```

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.

4 Data Import (from csv, txt, and excel) and Saving Data Frames

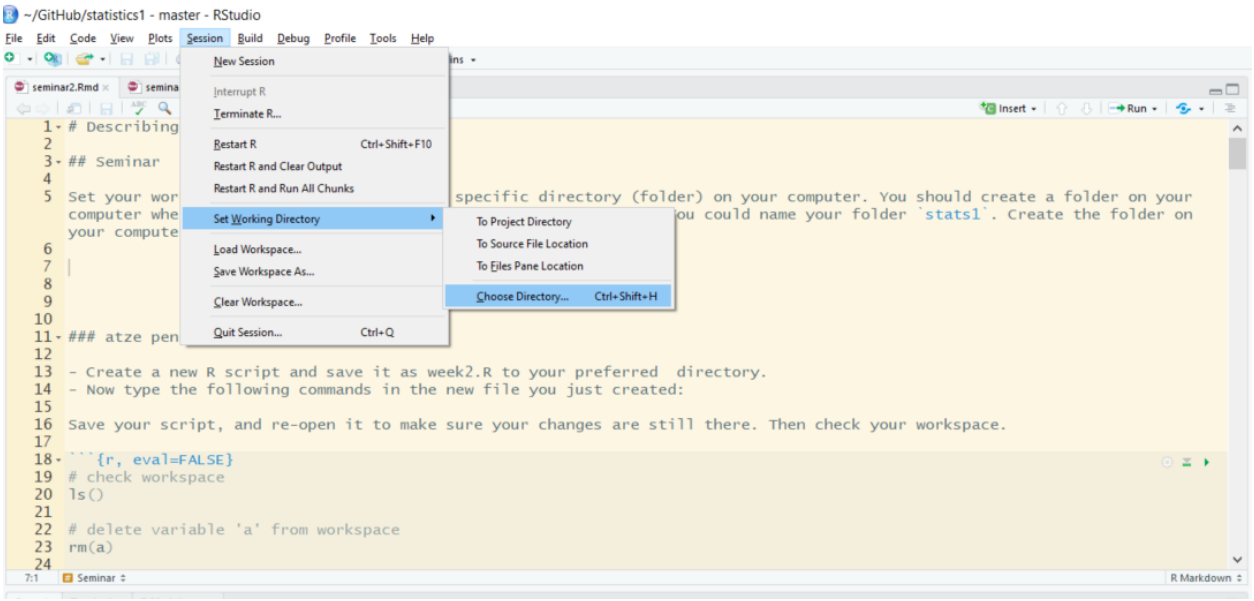
4.1 Seminar

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

4.1.1 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 `Stats101`. 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 `Stats101` 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.

Again, as we are starting a new script we should check the contents of our workspace like so:

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

Again 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:

```
# 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:

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

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

4.1.2 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".

4.1.3 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.

Essentially, its best to work with this data format as it can be easily loaded into R, but also easy opening outside of R.

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

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

4.1.4 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")`.

```
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.

```
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

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

Now, let's load the file:

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

4.1.5 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.

4.1.6 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)
```

4.1.7 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)

Lets save the data we have open on pereceptions of non-western immigration. It is important to select a

new file name, i.e. "newdata.csv", otherwise R overwrites the original dataframe and data may be lost. If updating a dataframe, it is good practice to save a file as V1.0, then V1.1 and so on. Let's try this below:

```
write.table(dat3,
            file = "non_western_immigrants_V1.1.csv",
            sep = ",",
            row.names = FALSE,
            col.names = TRUE
)
```

5 Descriptive Statistics

5.1 Seminar

Descriptive statistics are a good way to get a "feel" for the data and are an important first step for data analysis. Here we will explore two types: central tendency and dispersion.

Again, as good practice, let's first clear our workspace:

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

5.1.1 Central tendency

Central tendency explores the value of the observation at the center of a variable's distribution. This is the average or "typical" observation. What measure of central tendency you use depends on type of variable. In general, you use the following measures:

- Categorical variables - mode
- Ordinal variables - median
- Continuous variables - mean

As a recap:

- Categorical variables are unranked categories, such as political parties.
- Ordinal variables have categories that are ranked on a scale, i.e. council tax bands.
- Continuous variables have integer values and are simply not countable, i.e.income.
- Count variables represent countable data such as crime incidents
- Binary variables have only two categories i.e. employed or unemployed.

Let's first open some real data again:

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

```
[1] 1049   13
```

We can explore this in R by summarising our data: .

```
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
```

However, this can be less useful when you have many variables in the data frame. So instead you can summzarise the variables of interest by subsetting:

```
summary (dat1[ , c("IMMBRIT", "over.estimate")])
```

```
    IMMBRIT       over.estimate
 Min.   :  0.00   Min.   :0.0000
 1st Qu.: 10.00   1st Qu.:0.0000
 Median : 25.00   Median :1.0000
 Mean   : 29.03   Mean   :0.7235
 3rd Qu.: 40.00   3rd Qu.:1.0000
 Max.   :100.00   Max.   :1.0000
```

Here the mean and median is reported. Again the median is relevant for ordinal values, while the mean is best for continuous, count and binary variables.

Large differences between the median and mean could indicate that a variable is skewed; when a variable has extreme values or many 0s which drags the mean either side of the median. For example, we often refer to "median incomes," even though income is a continous variable. Here the median is used because the mean is dragged to the right by extremely wealthy outliers, while most incomes are in fact clustered as the lower end. The median income in the UK for 2014/15, before housing costs, was £473 per week. But for the mean, weekly income is placed at £581.

You may have noticed that the mode is not reported. To gain the mode you can create a frequency table and then use the order function to display the most common categories first. This is done by using the table and order functions:

```
a <- table(dat1$party_self)
a[order(a, decreasing = TRUE)]
```

```
  7   1   2   6   5   4   3
383 284 280  32  31  23  16
```

As you can see, category 7 is the most common category, with the number of observations reported below the category.

5.1.2 Dispersion

Dispersion measures the spread of values within a variable. Again, the precise measure of dispersion depends on the level of measurement:

- Categorical/binary variables - proportion of each category
- Ordinal variables - range or interquartile range
- Continuous/count variables - variance or the standard deviation

Categorical variables and proportional percentages

Let's first start with categorical variables (not this is also applicable for binary variables). Here we should again look at the a frequency table to understand the proportion of each category, in this case the proportion of observations for each political party:

```
#absolute frequency
table1 <- table (dat1$party_self)
```

```
#and the percentages, rounded by two decimal spaces and then multiplied by 100
round (table1 /sum(table1), 2)*100
```

```
 1  2  3  4  5  6  7
27 27  2  2  3  3 37
```

This variable does not have labels for the categories, but by looking at the codebook of this data set we know the categories are as followed: 1 - Conservatives 2 - Labour 3 - SNP 4 - Green 5 - UKIP 6 - BNP 7 - Other

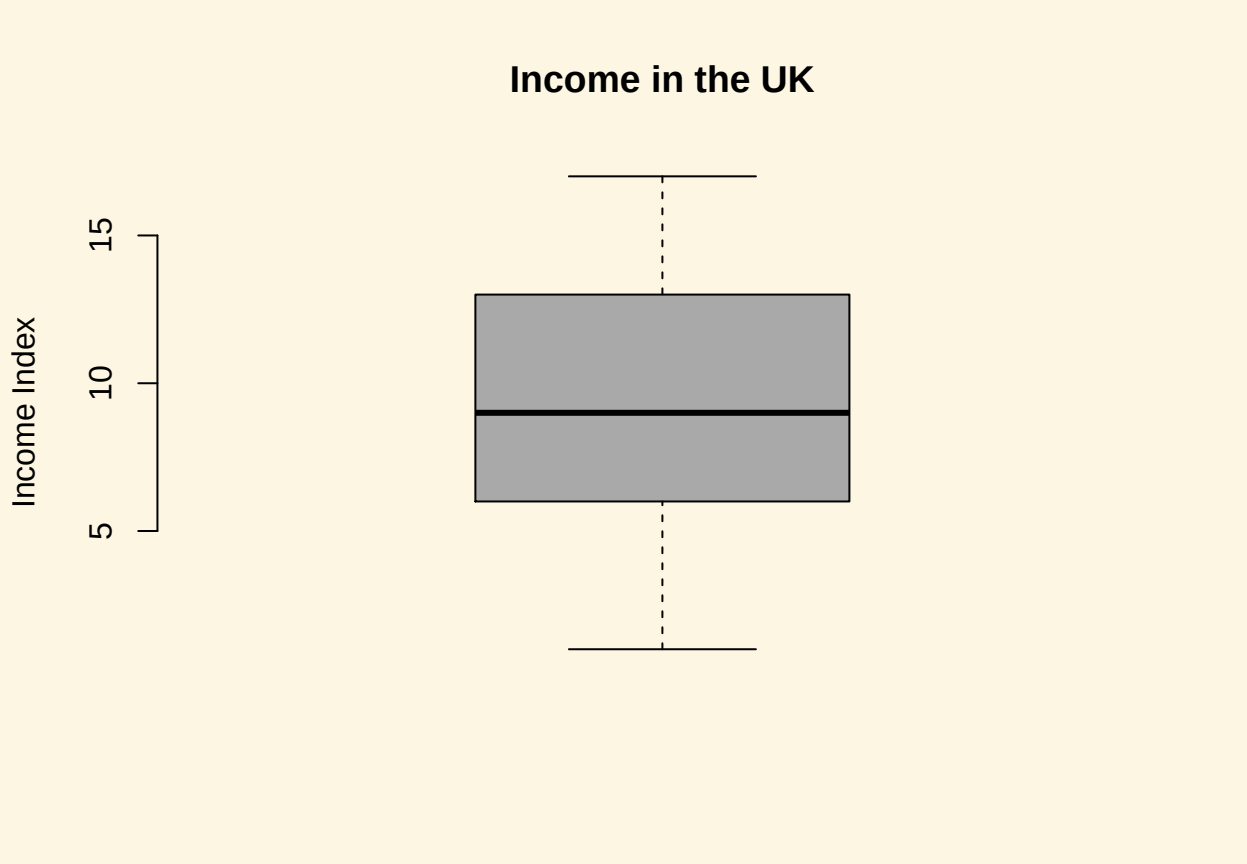
Perhaps this is not the best sample of our population, since there is no category for the Liberal Democrats!

Ordinal variables and the interquartile range

For ordinal variables, the most useful way to explore dispersion is by looking at the interquartile range. This can be visualised by plotting a simple box plot.

Note that with plots in R we have to provide arguments to design the plot, which are seperated by commas. Below we select: i) the variable of interest within the data frame (using the $ symbol), ii) add labels for the graph and y-axis, iii) remove the frame around the plot using the frame.plot function iv) choose the colour.

```
boxplot(
  dat1$HHInc,
  main = "Income in the UK",
  ylab = "Income Index",
  frame.plot = FALSE,
  col = "darkgray"
)
```



The box that is displayed within the plot represents the location of 50% of the data, between the 25% and 75% quartiles. The line within the box tells us the median. The two lines outside of the box shows us the data that falls within 1.5 times the quartiles.

Continuous variables and the standard deviation

For continuous (and count) variables we can look at the standard deviation. This tells us where 68% of the data can be found, since 68% of the data falls within one standard deviation of the mean. This can be done in R with a simple function. Here we are looking at British perceptions of non-Western immigration:

```
sd(dat1$IMMBRIT)
```

```
[1] 21.06331
```

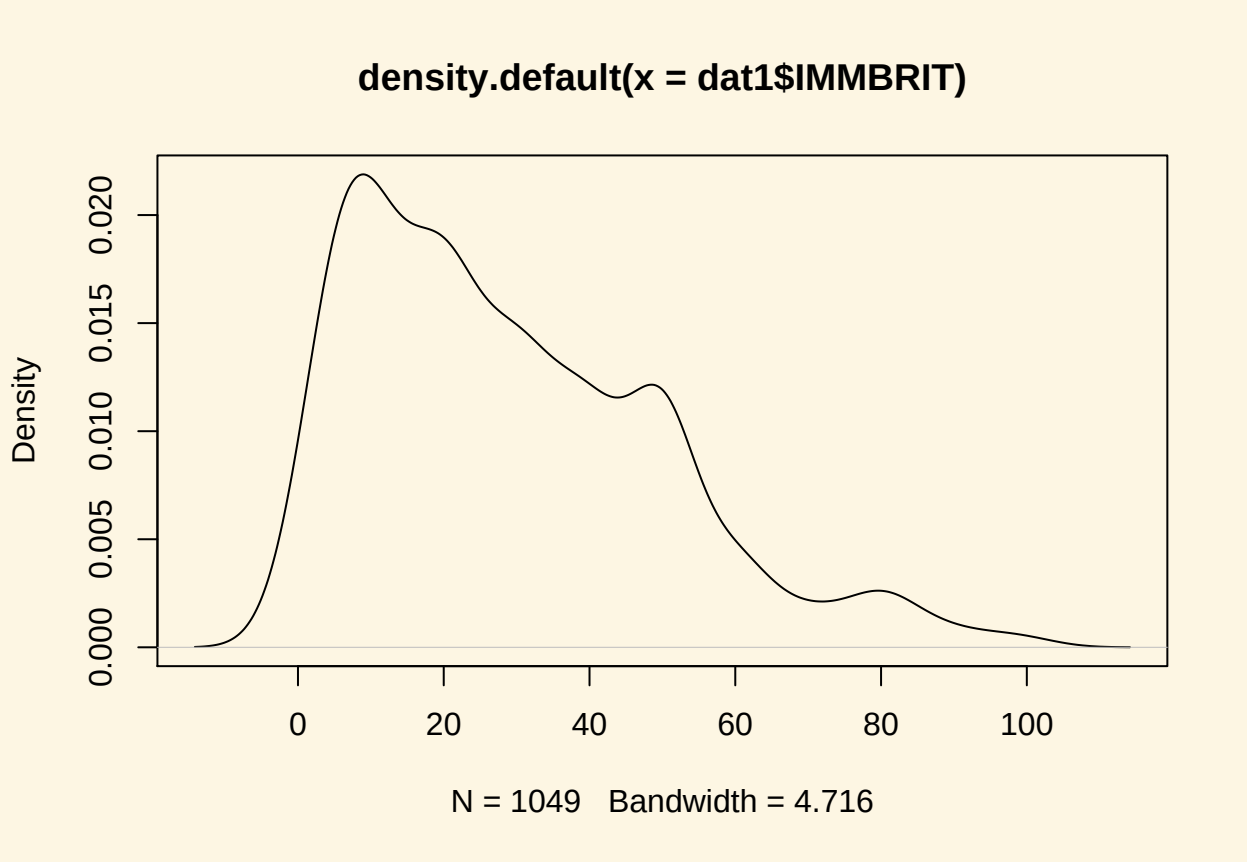
We can then compare this to the mean, which is another way to see if a continous variable is skewed:

```
mean(dat1$IMMBRIT)
```

```
[1] 29.03051
```

With the standard deviation at 21 and the mean at 29, we can see there is a large gap which indicates skewedness. But we can also simply visualise this by plotting the density of the variable:

```
plot(density(dat1$IMMBRIT))
```



As we can see most of the data is in the left hand side of the graph with fewer larger values on the far right of the graph. This drags the mean to the right which explains the large difference with the standard deviation. In this case it would be more appropriate to use the median (as is the case with income in the example above). When we run the median below, it is much closer to the centre of the data:

```
median(dat1$IMMBRIT)
```

```
[1] 25
```

To clear the graphs from our workspace, we can use the dev.off function with brackets:

```
dev.off()
```

```
null device
          1
```