Quantitative Politics with R

Erik Gahner Larsen and Zoltán Fazekas

February 25, 2018

Table of Contents

Chapter 1

Introduction

If you want to conduct quantitative analyses of political phenomena, `R` is by far the best software you can use. Importantly, data analysis is no longer restricted to analyzing survey data, but does now include social media data, texts, images, geographic data (*GIS*), and so forth.

In this book, we aim to provide an easily accessible introduction to `R` for the study of different types of data. The book will teach you how to get different types of data into `R` and manipulate, analyze and visualize the data.

Compared to other statistical softwares, such as Excel, SPSS, Stata and SAS, you will experience that `R` is completely different. First in a bad way: things are not as easy as they used to be. Then in a good way: once you learn how to do different tasks in `R`, you will be ashamed when you look back at the old you doing analyses in SPSS.

In this chapter you will find an introduction to `R`. First, we ask the obvious and important question, why R? Second, we help you install what you need. Third, we introduce you to the basic logic of `R` so you are ready for the chapters to come.

1.1 Why `R`?

First, `R` is an *open source* statistical programming language. `R` is free, and while you might not pay for Stata or SPSS because you are a student, you will not have free access to this forever. This is not the case with `R`. On the contrary, you will *never* have to pay for `R`.

Second, `R` provides a series of opportunities you don't have in SPSS and Stata. `R` has an impressive package ecosystem on CRAN (the **c**omprehensive **R** **a**rchive **n**etwork) with more than 12,000 packages created by other users of `R`.

Third, some of the most beautiful figures you will find today are created in `R`. Big media outlets such as The New York Times and FiveThirtyEight use `R` to create figures. In particular the package `ggplot2` is popular to create figures and we will work with this package below.

Fourth, there is a great community of `R` users that are able to help you when you encounter a problem (which you undoubtly will). `R` is a very popular software and in great demand meaning that you will not be the first (nor the last) to experience specific issues in your data analysis. Accordingly, you will find a lot of help on Google and other places to a much greater extent than for other types of software.

Fifth, while you can't do as much point-and-click as in SPSS and Stata, this approach facilitates that you can reproduce your work. When you are doing something i `R` with commands (in a script) is it easy to document. So, while you do not see a pedagogical graphical user interface in `R` with a limited set of buttons to click, this is more of an advantage than a limitation.

1.2 Installing `R`

To install the `R`, you will have to install 1) the `R` language and 2) RStudio, the graphical user interface. To install the `R` language, follow this procedure:

1. Go to https://cloud.r-project.org.
2. Click *Download R for Windows* if you use Windows or *Download R for (Mac) OS X* if you use Mac.

If you use Windows:

3. Click on *base*.
4. Click the top link where you can download `R` for Windows.
5. Follow the installation guide.

If you use Mac:

3. Select the most recent `.pkg` file under *Files:* that fits your OS X.
4. Follow the installation guide.

If you encounter problems with the installation guide, make sure that you did download the correct file *and* that your computer meets the requirements. If you did this and still encounter problems, you should get an error message you can type into Google and find relevant information on what to do.

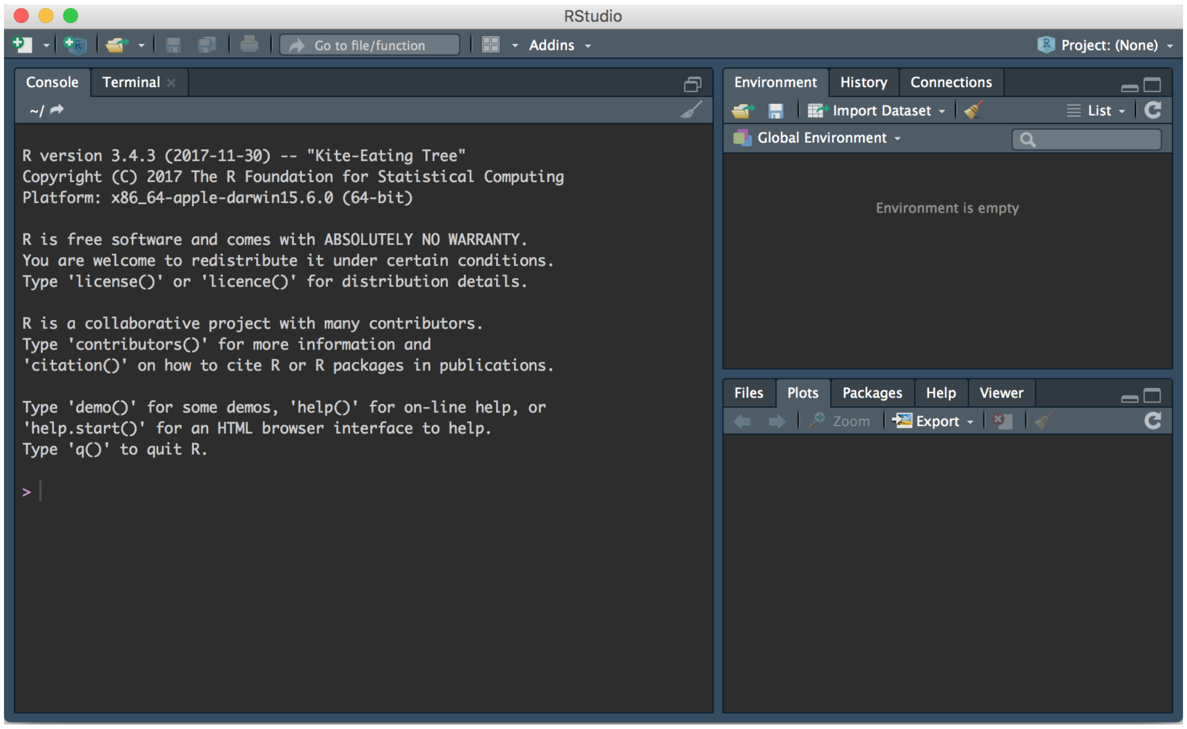
You should now have the `R` language installed on your computer.

1.3 Installing RStudio

RStudio is an integrated development environment (IDE) and makes it much easier to work in `R` compared to the standard ("base") R. This is also available for free. To install RStudio, follow these steps:

1. Go to: https://www.rstudio.com/products/rstudio/download/#download.
2. Click on the installer file for your platform, e.g. Windows or Mac OS X.
3. Follow the installation guide.

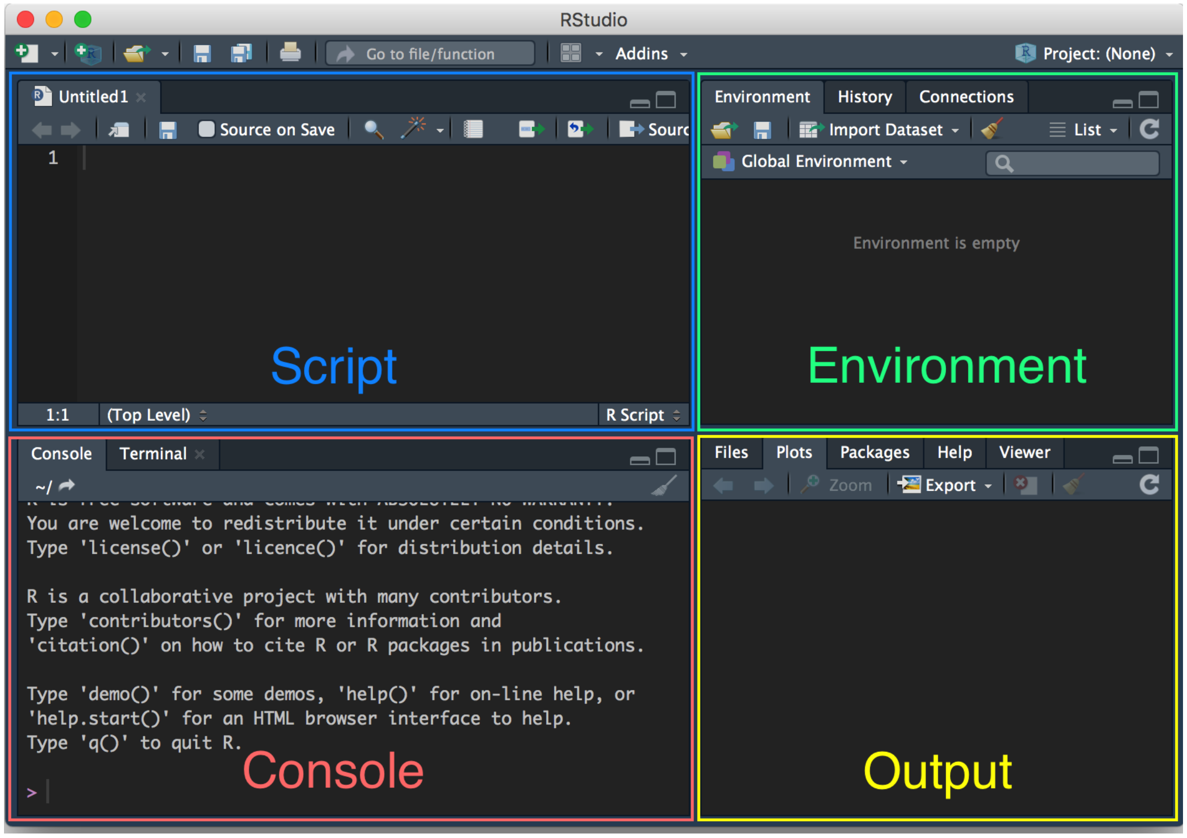
You should now have RStudio installed on your computer. When you open `R` you will see a graphical interface as in Figure 1.1.



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

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

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

>
```

Figure 1.1: Graphical interface in RStudio

There are three different windows. However, one is missing, and that is the window where you will write most of your scripts. You can get this window by going to the top menu and select `File` → `New File` → `R Script`. This should give you four windows as shown in Figure 1.2.



```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

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

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

>
```

Figure 1.2: Graphical interface in RStudio, explained

In the figure, we have emphasized the four windows: script, environment, output, and console. The *script* is where you will have your `R` code and make changes. The *environment* is where you can see what datasets, variables and other parts you have loaded into R. The *output* is where you can see figures you create. The *console* is where you can see some output and run commands.

Everything you do in `R` can be written as commands. This ensures that you will always be able to document your work (in your script). In the console, you can see a prompt (`>`). Here, you can write what what you want `R` to do. Try to write `2+2` and hit `Enter`. This should look like this:

```
2+2
```

```
[1] 4
```

The code you have entered in the console cannot be traced later. Accordingly, you will have to save the commands you want to keep in the script. Even better, you should write your commands in the script and "run" them from there. If you write `2+2` in the script, you can mark it and press `CTRL+R` (Windows) or `CMD+ENTER` (Mac). Then it will run the part of the script that is marked in the console. Insert the code below in your script and run it in the console:

```
50*149
3**2         # 3^2
2**3         # 2^3
sqrt(81)     # 81^0.5
```

As you can see, we have used `#` as well. The `#` sign tells `R` that everything after that sign on that line shouldn't be read as code but as a comment. In other words, you can write comments in your script that will help you remember what you are doing - and help others understand the meaning of your script. For now, remember to document everything you do in your script.

Notice also that we use a function in the bottom, namely `sqrt()`. A lot of what we will be doing in `R` is with functions. For example, to calculate a mean later we will use the `mean()` function. In the next section we will use functions to install and load packages.

1.4 Installing R packages

We highlighted above that one of the key advantages of using `R` is the package system. In `R`, a package is a collection of data and functions that makes it easier for you do to what you want. The sky is the limit and the only thing you need to learn know is how to install and load packages.

To install packages, you will have to use a function called `install.packages()`. We will install a package that installs a lot of the functions we will be using to manipulate and visualise data. More specifically, we will work within the tidyverse (Hadley Wickham, 2017). You can read more at tidyverse.org. To intall this package type:

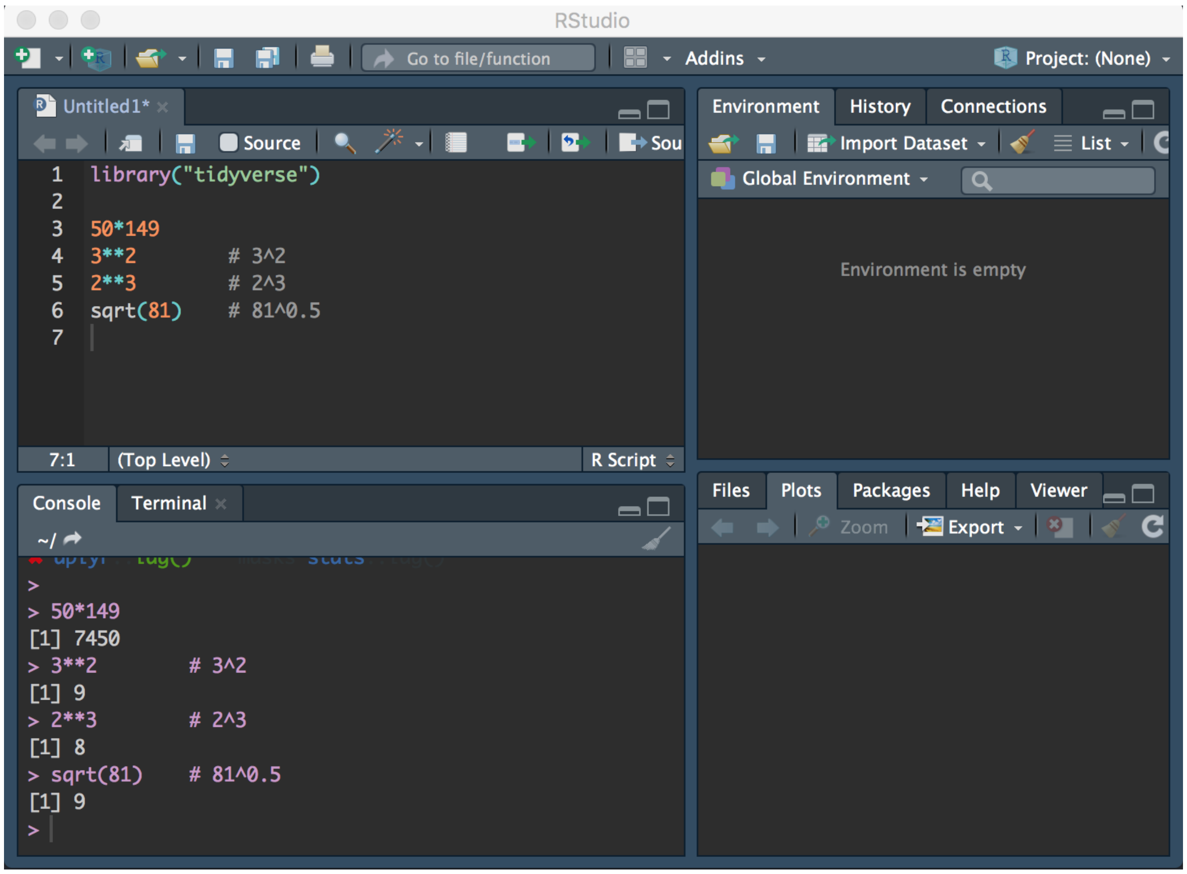
```
install.packages("tidyverse")
```

You only need to install the package once. In other words, when you have used `install.packages()` to install a packagae, you will not need to install that specific package again. Note that we put `tidyverse` in quotation marks. This is important when you install a package. If you forget this, you will get an error.

While you only need to install a package once, you need to load the package every time you open `R`. This is a good thing as you don't want to have all your installed `R` packages working at the same time. For this reason, most scripts begin with loading the packages that is needed. To load a package, we use the function `library()`:

```r
library("tidyverse")
```

To recap, it is always a good idea to begin your script with the package(s) you will be working with. If we want to have a script where we load the `tidyverse` package and have some of the commands we ran above, the script could look like the script presented in Figure 1.3.



```r
library("tidyverse")

50*149
3**2         # 3^2
2**3         # 2^3
sqrt(81)     # 81^0.5
```

```
>
> 50*149
[1] 7450
> 3**2         # 3^2
[1] 9
> 2**3         # 2^3
[1] 8
> sqrt(81)     # 81^0.5
[1] 9
>
```

Figure 1.3: A script in RStudio

If you want to save your script you can select `File` → `Save`, where you can pick a destination for your script.

1.5 Errors and help

As noted above, you will encounter problems and issues when you do stuff in `R`. Sadly, there are many potential reasons to why your script might not be working. Your version of `R` or/and RStudio might be too old or too new, you might be using a function that has a mistake, you might not have the data in the right format etc.

Consequently, we cannot provide a comprehensive list of errors you might get. The best thing to do is to learn how to find help online. Here, the best advice is to use Google and, when you search for help, always remember to mention R in your search string, and, if you are having problems with a specific package, also the name of the package.

Chapter 2

Basics

Remember that everything you do in `R` can be written as commands. Repeat what you did in last chapter from your script window: write `2+2` and run the code. This should look like this:

```
2+2
```

```
[1] 4
```

You are now able to conduct simple arithmetics. This shows that `R` can be used as a calculatur and you can now call yourself an `R` user. In other words, knowing how to use `R` is not a binary category where you either can use `R` or not, but a continuum where you will always be able to learn more. That's great news!

2.1 Numbers as data

Next, we will have to learn about variable assignment and in particular how we can work with *objects*. Everything you will use in `R` is saved in objects. This can be everything from a number or a word to complex datasets. A key advantage of this compared to other statistical programmes is that you can have multiple datasets open at the same time. If you, for exampel, want to connect two different surveys, you can have them both loaded at the same time. This is not possible in SPSS and Stata.

To save something in an objet, we need to use the *assignment operator*, `<-`, which basically tells `R` that anything on the right side of the operator should be assigned to the object on the left side. Let us try to save the number 2 in the object `x`

```
x <- 2
```

Now `x` will return the number 2 whenever we use `x`. Let us try to use our object in different simple operations. Write the operations in your R-script and run them individually and see what happens.

```
x
x * 2
x * x
x + x
```

If it is working, `R` should return the values `2`, `4`, `4` and `4`. If you change the object `x` to have the number 3 instead of 2 and run the script again, you should get a new output.[1] This is great as you only need to change a single number to change the whole procedure. Accordingly, when you are working with scripts, try to save as much you can in objects, so you only need to change numbers once, if you want to make changes. This also reduces the likelihood of you making a mistake.

We can also use our object to create other objects. In the example below we will create a new object `y`. This object returns the sum of `x` and 7.

```
y <- x + 7
```

One thing to keep in mind is that we do not get the output in `y` right away. To get the output, we can just write `y`, or we can, when we create the object, include it all in a parenthesis as we do below.

```
(y <- x + 7)
```

```
[1] 9
```

Luckily, we are not limited to save only one number in an object. On the contrary, in most objects we will be working with, we will have multiple numbers. The code below will return a row of numbers from 1 to 10.

[1]More specifically, `3`, `6`, `9` and `6`.

```r
1:10
```

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

We can save this row of numbers in an object (using `<-`), but we can also use them directly, e.g. by taking every number in the row and add 2 to all of them.

```r
1:10 + 2
```

```
 [1]  3  4  5  6  7  8  9 10 11 12
```

When you will be working with more numbers, you have to tell `R`, that you are working with multiple numbers. To do this, we use the function `c()`. This tells `R` that we are working with a vector.[2] The function `c()` is short for *concatenate* or *combine*.[3] Remember that everything that happens in `R` happens with functions. A vector looks like this:

```r
c(2, 2, 2)
```

```
[1] 2 2 2
```

This is a *numerical* vector. Again, a vector is a collection of values of the same type. We can save any vector in an object without any problems. In the code below we save four numbers (14, 6, 23, 2) in the object `x`.

```r
x <- c(14, 6, 23, 2)
x
```

```
[1] 14  6 23  2
```

We can then use this vector to calculate new numbers (just as we did above with `1:10`), for example by multiplying all the numbers in the vector with 2.

[2]In the example with `1:10`, this is similar to writing `c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`. In other words, we have a hidden `c()` when we type `1:10`.

[3]`c()` creates a vector with *all* elements in the parenthesis. Since a vector can only have one type of data, and not both numbers and text (cf. next section), `c()` will ensure that all values are reduced to the level all values can work with. Consequently, if just one value is a letter and not a number, all values in the vector will be considered text.

```r
x * 2
```

```
[1] 28 12 46  4
```

If we are only interested in a single value from the vector, we can get this value by using brackets, i.e. `[ ]`, which you place just after the object (so no space between the name of the object and the brackets!). By placing the number 3 in the brackets we can get the third number in the object.

```r
x[3]
```

```
[1] 23
```

As you can see, we get the third element, 23. We can use the same procedure to get all values with the exception of one value by including a negative sign in the brackets. In this example we will get all values except for 2. Also, note that since we are not assigned anything to an object (with `<-`), we are not making any changes to `x`.

```r
x[-2]
```

```
[1] 14 23  2
```

Now we can try to use a series of functions on our object. The functions below will return different types of information such as the number of values, the median, the mean, the standard deviation etc.

```r
length(x)     # length of vector, number of values
min(x)        # minima value
max(x)        # maxima value
median(x)     # the median
sum(x)        # the sum
mean(x)       # the mean
var(x)        # the variance
sd(x)         # the standard deviation
```

The functions should return the values 4, 2, 23, 10, 45, 11.25, 86.25 and 9.287088.

If we for some reason wants to add an extra number to our vector `x`, we can either create a new vector with all the numbers or just overwrite the existing vector with the addition of an extra number:

```
x <- c(x, 5)
x
```

```
[1] 14  6 23  2  5
```

We now have five values in our vector instead of four. The value 5 has the last place in the vector but if we had added 5 before `x` in the code above, 5 would have been in the beginning of the vector.

Try to use the `mean()` function on the new object `x`

```
mean(x)
```

```
[1] 10
```

Now the mean is 10 (before we added the value 5 to the object the mean was 11.25).

2.2 Missing values (NA)

Up until now we have been lucky that all our "data" has been easy to work with. However, in the real world - and thereby for most of the data we will work with - we will encounter missing values. In Stata you will see that missing values gets a dot ('.'). In `R`, all missing values are denoted `NA`. Let us try to add a missing value to our object `x` and take the mean.

```
x <- c(x, NA)


mean(x)
```

```
[1] NA
```

We do not get a mean now but just `NA`. The reason for this is that `R` is unable to calculate the mean of a vector with a missing value included. In order for `R` to calculate the mean now, we need to specifcy that it should remove the missing values before calculating the mean. To do this, we add `na.rm=TRUE` as an *option* to the function. Most functions have a series of options (more about this later), and the default option for the `mean()` function is not to ignore the missing values.

```
mean(x, na.rm=TRUE)
```

```
[1] 10
```

Now we get the same mean as before we added NA to the object.

2.3 Logical operators

In R a lot of what we will be doing is using logical operators, e.g. testing whether something is equal or similar to something else. This is in particular relevant when we have to recode objects and only use specific values. If something is true, we get the value TRUE, and if something is false, we get FALSE. Try to run the code below and see what information you get (and whether it makes sense).

```
x <- 2

x == 2         # equal to
x == 3
x != 2         # not equal to
x < 1          # less than
x > 1          # greater than
x <= 2         # less or equal to
x >= 2.01      # greater or equal to
```

The script will return TRUE, FALSE, FALSE, FALSE, TRUE, TRUE and FALSE. If you change x to 3, the script will return other values.

2.4 Text as data

In addition to numbers we can also work with text. The difference between text and numbers in R is that we use quotation marks to indicate that something is text (and not an object).[4] As an example, we will create an object called p with the political parties from the United Kingdom general election in 2017.

[4]Alternatively, you can use ' instead of ". If you want more information on when you should use ' instead of", see http://style.tidyverse.org/syntax.html#quotes.

```r
p <- c("Conservative Party", "Labour Party", "Scottish National Party",
       "Liberal Democrats", "Democratic Unionist Party", "Sinn Féin")


p
```

```
[1] "Conservative Party"        "Labour Party"
[3] "Scottish National Party"   "Liberal Democrats"
[5] "Democratic Unionist Party" "Sinn Féin"
```

To see what type of data we have in our object, `p`, we can use the function `class()`. This function returns information on the type of data we are having in the object. If we use the function on `p`, we can see that the object consists of characters (i.e. *"character"*).

```r
class(p)
```

```
[1] "character"
```

To compare, we can do the same thing with our object `x`, which includes numerical values. Here we see that the function `class()` for `x` returns `"numeric"`. The different classes a vector can have is: `character` (text), `numeric` (numbers), `integer` (whole numbers), `factor` (categories) and `logical` (logical).

```r
class(x)
```

```
[1] "numeric"
```

To test whether our object is numerical or not, we can use the function `is.numeric()`. If the object is numeric, we will get a `TRUE`. If not, we will get a `FALSE`. This logical structure can be used in a lot of different scenarios as we will see later. Similar to `is.numeric()`, we have a function called `is.character()` that will show us whether the object is a charater or not.

```r
is.numeric(x)
is.character(x)
```

Try to use `is.numeric()` and `is.character()` on the object `p`.

In the same way we could get specific values from the object when it was numeric, we can get specific values when it is a character object as well.

```
p[3]
```

```
[1] "Scottish National Party"
```

```
p[-3]
```

```
[1] "Conservative Party"        "Labour Party"
[3] "Liberal Democrats"         "Democratic Unionist Party"
[5] "Sinn Féin"
```

While `p` is a short name for an object and easy to write, it is not telling for what we actually have in the object. Accordingly, let us create a new object called `party` with the same information as in `p`. When you name objects remember that they are case sensitive so `party` will be a different object than `Party`.[5]

```
party <- p

party
```

```
[1] "Conservative Party"        "Labour Party"
[3] "Scottish National Party"   "Liberal Democrats"
[5] "Democratic Unionist Party" "Sinn Féin"
```

2.5 Data frames

In most cases, we will not be working with one variable (e.g. information on party names), but multiple variables. To do this in an easy way, we can create *data frames* which is similar to a dataset in SPSS and Stata. The good thing about `R`, however, is that we can have multiple data frames open at the same time. The cost of this is that we have to specifcy, when we do something in `R`, exactly what data frame we are using.

Here we will create a data frame with more information about the parties from the United Kingdom general election, 2017.[6]

[5]If you want more information on how to name objects, see http://style.tidyverse.org/syntax.html#object-names.

[6]The information is taken from https://en.wikipedia.org/wiki/United_Kingdom_general_election,_2017

As a first step we can create new objects with more information: `leader` (ifnormation on the party leader), `votes` (the vote share in percent), `seats` (the number of seats) and `seats_change` (change in seats from the previous election). Do note that the order is important as we are going to link these objects together in a minute, where the first value in each object is for the Conservative Party, the second for the Labour Party and so on.

```
leader <- c("Theresa May", "Jeremy Corbyn", "Nicola Sturgeon",
            "Tim Farron", "Arlene Foster", "Gerry Adams")
votes <- c(42.4, 40.0, 3.0, 7.4, 0.9, 0.7)
seats <- c(317, 262, 35, 12, 10, 7)
seats_change <- c(-13, 30, -21, 4, 2, 3)
```

The next thing we have to do is to connect the objects into a single object, i.e. our data frame. A data frame is a collection of different vectors of the same length. In other words, for the objects we have above, as they have the same number of information, they can be connected in a data frame. `R` will return an error message if the vectors do not have the same length.

We can have different types of variables in a data frame, i.e. both numbers and text variables. To create our data frame, we will use the function `data.frame()` and save the data frame in the object `uk2017`.

```
uk2017 <- data.frame(party, leader, votes, seats, seats_change)

uk2017 # show the content of the data frame
```

```
                      party          leader votes seats seats_change
1        Conservative Party     Theresa May  42.4   317          -13
2              Labour Party   Jeremy Corbyn  40.0   262           30
3  Scottish National Party Nicola Sturgeon   3.0    35          -21
4         Liberal Democrats      Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

To see what type of object we are working with, we can use the function `class()` to show that `uk2017` is a data frame.

```r
class(uk2017)
```

```
[1] "data.frame"
```

If we would like to know what class the individual variables in our data frame are, we can use the function `sapply()`. This function allows us to apply a function to a list or a vector. Below we apply `class()` on the individual variables in `uk2017`.

```r
sapply(uk2017, class)
```

```
       party       leader        votes        seats seats_change 
    "factor"     "factor"    "numeric"    "numeric"    "numeric" 
```

Here we can see that we have data as a `factor` as well as numerical variables. We can get similar information about our data by using the function `str()`. This function returns information on the structure in the data frame.

```r
str(uk2017)
```

```
'data.frame':	6 obs. of  5 variables:
 $ party       : Factor w/ 6 levels "Conservative Party",..: 1 3 5 4 2 6
 $ leader      : Factor w/ 6 levels "Arlene Foster",..: 5 3 4 6 1 2
 $ votes       : num  42.4 40 3 7.4 0.9 0.7
 $ seats       : num  317 262 35 12 10 7
 $ seats_change: num  -13 30 -21 4 2 3
```

Here we can see that it is a data frame with 6 observations of 5 variables. If the rows (i.e. observations) have names, we can get these by using `rownames()`. We can get the names of the columns, i.e. the variables in our data frame, by using `colnames()`.

```r
colnames(uk2017)
```

```
[1] "party"        "leader"       "votes"        "seats"       
[5] "seats_change"
```

If we want to see the number of columns and rows in our data frame, we can use `ncol()` and `nrow()`.

```r
ncol(uk2017)
```

```
[1] 5
```

```r
nrow(uk2017)
```

```
[1] 6
```

If we are working with bigger data frames, e.g. a survey with thousands of respondents, it might not be useful to just show the full data frame. One way to see just a few of the observations is by using `head()`. If not specified further, this function will show the first six observations in the data frame. In the example below, we will tell `R` to show the first three observations

```r
head(uk2017, 3)  # show the first three rows
```

```
                     party          leader votes seats seats_change
1       Conservative Party     Theresa May  42.4   317          -13
2             Labour Party   Jeremy Corbyn  40.0   262           30
3 Scottish National Party Nicola Sturgeon   3.0    35          -21
```

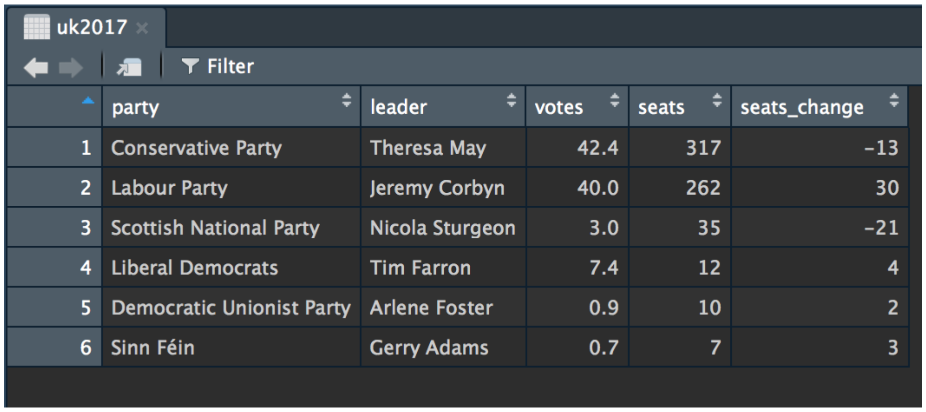
In the same way, we can use `tail()` show the last observations in a data frame. Here we see the last four observations in our data frame.

```r
tail(uk2017, 4)  # show the last four rows
```

```
                      party          leader votes seats seats_change
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4         Liberal Democrats      Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

If you want to see your data frame in a new window, you can use the function `View()` (do note the capital letter V - not v).

```
View(uk2017)
```



	party	leader	votes	seats	seats_change
1	Conservative Party	Theresa May	42.4	317	-13
2	Labour Party	Jeremy Corbyn	40.0	262	30
3	Scottish National Party	Nicola Sturgeon	3.0	35	-21
4	Liberal Democrats	Tim Farron	7.4	12	4
5	Democratic Unionist Party	Arlene Foster	0.9	10	2
6	Sinn Féin	Gerry Adams	0.7	7	3

Figure 2.1: Data frame with View(), RStudio

When you are working with variables in a data frame, you can use $ as a *component selector* to select a variable in a data frame. This is the base R way, i.e. brackets and dollar signs. In the next chapter we will work with other functions that makes it easier to work with data frames.

If we, for example, want to have all the vote shares in our data frame `uk2017`, we can write `uk2017$votes`.

```
uk2017$votes
```

```
[1] 42.4 40.0  3.0  7.4  0.9  0.7
```

Contrary to working with a vector in a single dimension, we have two dimensions in a data frame (rows horisontally and columns vertically). Just as for a single vector, we need to work with the brackets, `[ ]`, in addition to our object, but we need to specify the rows and columns we are interested in. If we want to work with the first row, we need to specify `[1, ]` after the object. The comma is seperating the information on the rows and columns we want to work with. When we are not specifying anything after the comma, that means we want to have the information for *all* columns.

```
uk2017[1,] # first row
```

```
               party      leader votes seats seats_change
1 Conservative Party Theresa May  42.4   317          -13
```

Had we also added a number after the comma, we would get the information for that specific column. in the example below we want to have the information on the first row in the first column (i.e. the name of the party on the first row).

```
uk2017[1, 1] # first row, first column
```

```
[1] Conservative Party
6 Levels: Conservative Party Democratic Unionist Party ... Sinn Féin
```

If we want to have the names of all parties, i.e. the information in the first column, we can specify that we want all rows but only for the first column.

```
uk2017[, 1] # first column
```

```
[1] Conservative Party         Labour Party
[3] Scottish National Party    Liberal Democrats
[5] Democratic Unionist Party Sinn Féin
6 Levels: Conservative Party Democratic Unionist Party ... Sinn Féin
```

Interestingly, the functions we have talked about so far can all be applied to data frames. The `summary()` function is very useful if you want to get an overview of all your variables in your data frame. For the numerical variables in the data frame, the function will return information such as the mean and the median.

```
summary(uk2017)
```

```
                       party                 leader       votes
 Conservative Party       :1   Arlene Foster  :1   Min.   : 0.700
 Democratic Unionist Party:1   Gerry Adams    :1   1st Qu.: 1.425
 Labour Party             :1   Jeremy Corbyn  :1   Median : 5.200
 Liberal Democrats        :1   Nicola Sturgeon:1   Mean   :15.733
 Scottish National Party  :1   Theresa May    :1   3rd Qu.:31.850
 Sinn Féin                :1   Tim Farron     :1   Max.   :42.400
     seats        seats_change
 Min.   :  7.0   Min.   :-21.0000
 1st Qu.: 10.5   1st Qu.: -9.2500
 Median : 23.5   Median :  2.5000  
 Mean   :107.2   Mean   :  0.8333  
 3rd Qu.:205.2   3rd Qu.:  3.7500  
 Max.   :317.0   Max.   : 30.0000  
```

We can also use the functions on our variables as we did above, e.g. to get the maximum number of votes a party got with the function `max()`.

```
max(uk2017$votes)
```

```
[1] 42.4
```

If we want to have the value on a specific variable in our data frame, we can use both `$` and `[ ]`. Below we get the second value in the variable `party`.

```
uk2017$party[2]
```

```
[1] Labour Party
6 Levels: Conservative Party Democratic Unionist Party ... Sinn Féin
```

To combine a lot of what we have used above, we can get informatin on the name of the party that got the most votes. In order to do this, we specify that we would like to have the name of the party for the party where the number of votes equals the maximum number of votes. In other words, when `uk2017$votes` is equal to `max(uk2017$votes)`, we want to get the information on `uk2017$party`. We use `==` to test whether something is equal to.

```
uk2017$party[uk2017$votes == max(uk2017$votes)]
```

```
[1] Conservative Party
6 Levels: Conservative Party Democratic Unionist Party ... Sinn Féin
```

As we can see, the Conservative Party got the most votes in the 2017 election. We can use the same procedure if we want to get information on the party that got the minimum number of votes. To do this we use `min()`. Here we can see that this is Sinn Féin in our data frame.

```r
uk2017$party[uk2017$votes == min(uk2017$votes)]
```

```
[1] Sinn Féin
6 Levels: Conservative Party Democratic Unionist Party ... Sinn Féin
```

The sky is the limit when it comes t owhat we can do with data frames, including various types of statistical analyses. To give one example, we can use the `lm()` function to conduct an OLS regression with `votes` as the independent variable and `seats` as the dependent variable. First, we save the model in the object `uk2017_lm` and then use `summary()` to get the results.

```r
uk2017_lm <- lm(seats ~ votes, data = uk2017)

summary(uk2017_lm)
```

```

Call:
lm(formula = seats ~ votes, data = uk2017)

Residuals:
      1       2       3       4       5       6 
 20.890 -17.105  18.054 -36.122   7.933   6.350 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   -4.310     13.405  -0.321 0.763932    
votes          7.085      0.558  12.698 0.000222 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 24.81 on 4 degrees of freedom
Multiple R-squared:  0.9758,	Adjusted R-squared:  0.9697 
F-statistic: 161.2 on 1 and 4 DF,  p-value: 0.0002216
```

2.6 Import and export data frames

Most of the data frames we will be working with in `R` are not data frames we will build from scratch but on the contrary data frames we will import from other files such as files made for Stata, SPSS or Excel. The most useful filetype to use when you work with data in files is `.csv`, which stands for *comma-separated values*. This is an open file format and be opened in any software. To export and import data frames to `.csv` files, we can use `write.csv()` and `read.csv()`.

First of all we need to know where `R` is working from, i.e. what our *working directory* is. To get this you can type `getwd()` and see where your data will be saved.

```r
getwd()
```

If you would like to change this, you can use the function `setwd()`. This function allows you to change the working directory to whatever folder on your computer you would like to use.

```r
setwd("/Dropbox/qpolr/book")
```

An easy way to control the working directory is to open an R-script directly from the folder so it also opens RStudio that way. This will automatically set the working directory to the folder with the R-script.

Once we know where we will save our data, we can use `write.csv()` to save the data. In the code below we first specify that we want to save the data frame `uk2017` and next the filename of the file (`uk2017.csv`).

```r
write.csv(uk2017, "uk2017.csv")
```

Do note that we need to put the file in quotation marks. Next, we can import the file into `R` the next time we open `R` with the function `read.csv()` and save the data frame in the object `uk2017`.

```r
uk2017 <- read.csv("uk2017.csv")
```

As with most stuff in `R`, there are multiple ways of doing things. To import and export data, we have packages like `foreign` (R Core Team, 2015), `rio` (C. Chan, Chan, & Leeper, 2016) og `readr` (H. Wickham & Francois, 2015). If you install and load the package `rio`, you can use the functions `import()` and `export()`.

```r
# export data with the rio package
export(uk2017, "uk2017.csv")

# import data with the rio package
uk2017 <- import("uk2017.csv")
```

2.7 Environment

We have worked with a series of different objects. T osee what objects we have in our memory, we can look in the *Environment* window, but we can also use the function `ls()`(*ls* is short for *list objects*).

```r
ls()
```

```
 [1] "leader"       "p"            "party"        "seats"
 [5] "seats_change" "uk2017"       "uk2017_lm"    "votes"
 [9] "x"            "y"
```

If we would like to remove an object from the memory, we can use the function `rm()` (*rm* is short for *remove*). Below we use `rm()` to remove the object `x` and then `ls()` to check whether `x` is gone.

```r
rm(x)

ls()
```

```
[1] "leader"       "p"            "party"        "seats"
[5] "seats_change" "uk2017"       "uk2017_lm"    "votes"
[9] "y"
```

If you would like to remove *everything* in the memory, you can use `ls()` in combination with `rm()`.

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

ls()
```

Chapter 3

Data management

There are multiple ways to manage data in `R` and in particular ways to create and change variables in a data frame. In this chapter, we show different ways of working with data frames with a focus on how to change variables. Noteworthy, there are multiple packages we can use to manipulate data farmes, but the best is without a doubt `dplyr` (Hadley Wickham & Francois, 2016).

The package provides some basic functions making it easy to work with data frames. These functions include `select()`, `filter()`, `arrange()`, `rename()`, `mutate()` og `summarize()`.[1] `select()` allows you to pick variables by their names. `filter()` allows you to pick observations by their values. `arrange()` allows you to reorder the rows. `rename()` allows you to rename columns. `mutate()` allows you to create new variables based on the values of old variables. `summarize()` allows you to collapse many values to a single summary.

All these functions rely on data frames. In other words, you can not use these functions on other types of data in `R`. Furthermore, they all return a new data frame.

The `dplyr` package is a part of the `tidyverse`. First, load the `tidyverse`.

```r
library("tidyverse")
```

We will use the dataset we created in the previous chapter. If you do not have it, you can download it here: http://qpolr.com/data/uk2017.csv

[1]For another good introduction to `dplyr`, see: Managing Data Frames with the dplyr package.

```r
uk2017 <- import("uk2017.csv")
```

To see the information in the dataset, use `head()`.

```r
head(uk2017)
```

```
                      party          leader votes seats seats_change
1        Conservative Party     Theresa May  42.4   317          -13
2              Labour Party   Jeremy Corbyn  40.0   262           30
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4         Liberal Democrats      Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

3.1 Selecting variables: `select()`

When we work with large datasets, we often want to select the few variables that are of key interest to our project. For this, the `select()` function is perfect. If we only want to have information on the party name and the votes in the `uk2017` data frame, we can write:

```r
select(uk2017, party, votes)
```

```
                      party votes
1        Conservative Party  42.4
2              Labour Party  40.0
3   Scottish National Party   3.0
4         Liberal Democrats   7.4
5 Democratic Unionist Party   0.9
6                 Sinn Féin   0.7
```

There are multiple different functions that can help us finding specific variables in the data frame. We can use `contains()`, if we want to include variables that contain a specific word in the variable name. In the example below we look for variables that contain `seat`.

```r
select(uk2017, contains("seat"))
```

```
  seats seats_change
1   317          -13
2   262           30
3    35          -21
4    12            4
5    10            2
6     7            3
```

Other noteworthy functions that can be of help similar to `contains()` are functions such as `starts_with()`, `ends_with()`, `matches()`, `num_range()`, `one_of()` and `everything()`. The last function, `everything()` is helpful if we want to move a variable to the beginning of our data frame.

```r
select(uk2017, votes, everything())
```

```
  votes                      party          leader seats seats_change
1  42.4         Conservative Party     Theresa May   317          -13
2  40.0               Labour Party   Jeremy Corbyn   262           30
3   3.0   Scottish National Party Nicola Sturgeon    35          -21
4   7.4          Liberal Democrats      Tim Farron    12            4
5   0.9 Democratic Unionist Party   Arlene Foster    10            2
6   0.7                  Sinn Féin     Gerry Adams     7            3
```

We can use the negative sign if we want to remove a variable from the data frame.

```r
select(uk2017, -leader)
```

```
                      party votes seats seats_change
1        Conservative Party  42.4   317          -13
2              Labour Party  40.0   262           30
3   Scottish National Party   3.0    35          -21
4         Liberal Democrats   7.4    12            4
5 Democratic Unionist Party   0.9    10            2
6                 Sinn Féin   0.7     7            3
```

3.2 Selecting observations: `filter()`

To select only some of the observations in our data frame, but for all variables, we can use the function `filter()`. In the example below we select the observations in our data frame with a positive value on `seats_change` (i.e. greater than 0).

```
filter(uk2017, seats_change > 0)
```

```
                      party        leader votes seats seats_change
1              Labour Party Jeremy Corbyn  40.0   262           30
2         Liberal Democrats    Tim Farron   7.4    12            4
3 Democratic Unionist Party Arlene Foster   0.9    10            2
4                Sinn Féin    Gerry Adams   0.7     7            3
```

Importantly, we are *not* making any changes to the data frame `uk2017`. This will only hapen if we replace our existing data frame or create a new data frame. In the example below we create a new data frame, `uk2017_seatlosers`, with the observations losing seats from 2015 to 2017.

```
uk2017_seatlosers <- filter(uk2017, seats_change < 0)
uk2017_seatlosers
```

```
                    party          leader votes seats seats_change
1      Conservative Party     Theresa May  42.4   317          -13
2 Scottish National Party Nicola Sturgeon   3.0    35          -21
```

Last, if we want to drop observations that contain missing values on specific variables, we can use the function `drop_na()`.

3.3 Sorting observations: `arrange()`

We can use the function `arrange()` if we want to change the order of observations. In the example below we sort our data frame according to how many votes the party got, with the party getting the least votes in the top of our data frame.

```
arrange(uk2017, votes)
```

```
                      party           leader votes seats seats_change
1                 Sinn Féin     Gerry Adams   0.7     7            3
2 Democratic Unionist Party   Arlene Foster   0.9    10            2
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4         Liberal Democrats       Tim Farron   7.4    12            4
5              Labour Party    Jeremy Corbyn  40.0   262           30
6        Conservative Party      Theresa May  42.4   317          -13
```

If we prefer to have the parties with the maximum number of votes in the top, we can use the negative sign (`-`).

```
arrange(uk2017, -votes)
```

```
                      party           leader votes seats seats_change
1        Conservative Party      Theresa May  42.4   317          -13
2              Labour Party    Jeremy Corbyn  40.0   262           30
3         Liberal Democrats       Tim Farron   7.4    12            4
4   Scottish National Party Nicola Sturgeon   3.0    35          -21
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

3.4 Rename variables: `rename()`

In the case we have a variable we would prefer having another name, we can use the function `rename()`. In the example below we change the name of `party` to `party_name`.

```
rename(uk2017, party_name = party)
```

```
                 party_name           leader votes seats seats_change
1        Conservative Party      Theresa May  42.4   317          -13
2              Labour Party    Jeremy Corbyn  40.0   262           30
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4         Liberal Democrats       Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

3.5 Create variables: mutate()

The best way to create a new variable from existing variables in our data frame is to use the function `mutate()`. In the example below we create a new variable, `votes_m` with information on how many percentage points a party is from the average number of votes a party got.

```
mutate(uk2017, votes_m = votes - mean(votes))
```

```
                      party          leader votes seats seats_change
1        Conservative Party     Theresa May  42.4   317          -13
2              Labour Party   Jeremy Corbyn  40.0   262           30
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4          Liberal Democrats      Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
    votes_m
1  26.666667
2  24.266667
3 -12.733333
4  -8.333333
5 -14.833333
6 -15.033333
```

In another example we use the `sum()` function as well to find the proportion of seats a party got in a variable, `seats_prop`.

```
mutate(uk2017, seats_prop = seats / sum(seats))
```

```
                      party          leader votes seats seats_change
1        Conservative Party     Theresa May  42.4   317          -13
2              Labour Party   Jeremy Corbyn  40.0   262           30
3   Scottish National Party Nicola Sturgeon   3.0    35          -21
4          Liberal Democrats      Tim Farron   7.4    12            4
5 Democratic Unionist Party   Arlene Foster   0.9    10            2
6                 Sinn Féin     Gerry Adams   0.7     7            3
```

```
  seats_prop
1 0.49300156
2 0.40746501
3 0.05443235
4 0.01866252
5 0.01555210
6 0.01088647
```

3.6 The pipe operator: %>%

So far we have looked at a series of different functions. In most cases we want to combine these functions, e.g. when we both have to select specific variables and observations. Luckikly, there is nothing against using one function nested within another, as the example below shows.

```
filter(select(uk2017, party, votes), seats_change > 0)
```

```
                      party votes
1              Labour Party  40.0
2         Liberal Democrats   7.4
3 Democratic Unionist Party   0.9
4                 Sinn Féin   0.7
```

The problem is that it can be complicated to read, especially when as the number of functions we use increase. Furthermore, the likelihood of making a stupid mistake, e.g. by including an extra `(` or `)` increases substantially. Luckily, we can use the pipe operator, `%>%`, to make our code more readable.

The operator relies on a step-wise logic so we first specify the data frame and then a line for each function we want to run on the data frame.

In the example below we do the same as above but in a way that is easier to follow.

```
uk2017 %>%
  select(party, votes) %>%
  filter(seats_change > 0)
```

```
                     party votes
1             Labour Party  40.0
2          Liberal Democrats   7.4
3 Democratic Unionist Party   0.9
4                Sinn Féin   0.7
```

On the first line, we show that we are using the data frame `uk2017`. We end this line with `%>%`, telling `R` that we are not done yet but will have to put this into the function on the line below. The next line uses the input from the previous line and selects `party` and `votes` from the data frame. This line also ends with the pipe, `%>%`. The third line shows the observations in our data frame where `seats_change` is greater than 0. Note that we did not select `seats_change` as a variable with `select()`, so this is not crucial in order to use it (as long as it is in the `uk2017` data frame). Last, we do *not* end with a pipe as we are now done.

3.7 Running functions on variables: `apply()`

If we would like to run a function on some of our rows or columns we can use the function `apply()`. For example, we can get the average number of votes and seats for parties with a positive value on `seats_change` (i.e. parties with an increase in seats from 2015 to 2017.

The addition here is the function `apply()` on the data frame used above. The first thing we specify here is `MARGIN`, i.e. whether we want to run a function on our rows (1) or columns (2). The next thing we specify is the function together with any relevant options.

```r
uk2017 %>%
  filter(seats_change > 0) %>%
  select(votes, seats) %>%
  apply(MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
votes seats
12.25 72.75
```

In the case you want to apply a function to both rows and columns, you will have to specify `c(1, 2)`. It is not important to mention `MARGIN` eller `FUN` if you have the order right. In other words, we can simplify our example to the code below.

```
uk2017 %>%
  filter(seats_change > 0) %>%
  select(votes, seats) %>%
  apply(2, mean)
```

```
votes seats
12.25 72.75
```

3.8 Aggregating variables: summarize() and group_by()

If we want to create new variables with aggregated information, similar to the information we got in the previous section, we can use the function `summarize()`. In the example below we get a data frame with information on the number of observatins, given by `n()`, the minimum number of votes a party got (`votes_min`), the maximum number of votes a party got (`votes_max`) and the average number of votes a party got (`votes_mean`) (all in percentages).

```
uk2017 %>%
  summarize(party = n(),
            votes_min = min(votes),
            votes_max = max(votes),
            votes_mean = mean(votes))
```

```
  party votes_min votes_max votes_mean
1     6       0.7      42.4   15.73333
```

If we want this information for different groups, we can supply with `group_by()`. In the example below we will like to have the information both for parties with an increase in seats from 2015 to 2017 and not.

```
uk2017 %>%
  group_by(seats_change > 0) %>%
  summarize(party = n(),
            votes_min = min(votes),
            votes_max = max(votes),
            votes_mean = mean(votes))
```

```
# A tibble: 2 x 5
  `seats_change > 0` party votes_min votes_max votes_mean
  <lgl>              <int>     <dbl>     <dbl>      <dbl>
1 F                      2     3.00       42.4       22.7
2 T                      4     0.700      40.0       12.2
```

In the example, you can see the aggregated information. `T` is short for `TRUE` and is the aggregated information for the observations where `seats_change` is greater than 0.

3.9 Recoding variables: `recode()`

In a lot of cases we want to recode the information in a single variable. To do this, we can use `recode()`. Importantly, this function works for individual variables and not for a data frame. Let us use the `leader` variable in `uk2017` as an example.

```
uk2017$leader
```

```
[1] Theresa May      Jeremy Corbyn    Nicola Sturgeon Tim Farron
[5] Arlene Foster    Gerry Adams
6 Levels: Arlene Foster Gerry Adams Jeremy Corbyn ... Tim Farron
```

In the case that we want to replace Tim Farron in the variable with a new guy, we can do that with the code below.

```
recode(uk2017$leader, "Tim Farron" = "New guy")
```

```
[1] Theresa May      Jeremy Corbyn    Nicola Sturgeon New guy
[5] Arlene Foster    Gerry Adams
6 Levels: Arlene Foster Gerry Adams Jeremy Corbyn ... New guy
```

Noteworthy, we do not create any changes to the `leader` variable. If we want to save the changes, we can save the new variable to our data frame.

```
uk2017$leader_new <- recode(uk2017$leader, "Tim Farron" = "New guy")

uk2017$leader_new
```

```
[1] Theresa May     Jeremy Corbyn   Nicola Sturgeon New guy
[5] Arlene Foster   Gerry Adams
6 Levels: Arlene Foster Gerry Adams Jeremy Corbyn ... New guy
```

Last, `dplyr` in the `tidyverse` is not the only package with a `recode()` function. The package `car` (Fox & Weisberg, 2011) has a similar function worth exploring.

References

Chan, C., Chan, G. C. H., & Leeper, T. J. (2016). *Rio: A swiss-army knife for data file i/o*.

Fox, J., & Weisberg, S. (2011). *An R companion to applied regression* (Second). Thousand Oaks CA: Sage. Retrieved from http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

R Core Team. (2015). *Foreign: Read data stored by minitab, s, sas, spss, stata, systat, weka, dBase, ...* Retrieved from http://CRAN.R-project.org/package=foreign

Wickham, H. (2017). *Tidyverse: Easily install and load the 'tidyverse'*. Retrieved from https://CRAN.R-project.org/package=tidyverse

Wickham, H., & Francois, R. (2015). *Readr: Read tabular data*. Retrieved from http://CRAN.R-project.org/package=readr

Wickham, H., & Francois, R. (2016). *Dplyr: A grammar of data manipulation*. Retrieved from http://CRAN.R-project.org/package=dplyr