Quantitative Politics with R

Erik Gahner Larsen and Zoltán Fazekas

November 29, 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 also social media data, texts, images, geographic data (*GIS*), and so forth. For that and other reasons listed below, `R` is a good thing to learn.

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

Compared to other statistical packages, such as Excel, SPSS, Stata and SAS, you will experience that `R` is somewhat 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 or something just as embarrasing.

In this chapter you will find an introduction to `R`. The introduction takes place in three steps. First, we ask the obvious and important question, why R? Second, we help you install what you need. Third, we introduce 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 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 do not 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. Specifically, they use the package `ggplot2`, a very popular package used to create figures. We will work with this package later.

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 `R`. 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. In other words, it is easy to document what you are doing in `R` with commands (in a script). 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 `R`, you will have to install 1) the `R` language and 2) the graphical user interface RStudio. 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 next.

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.

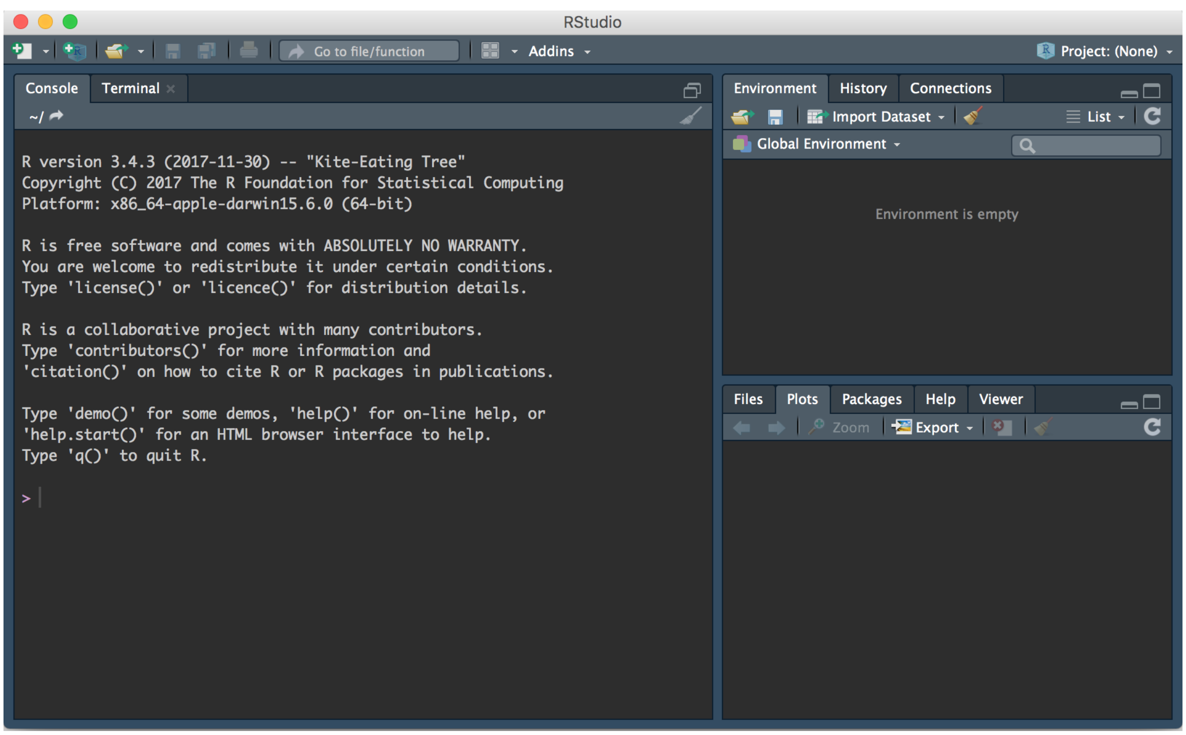


Figure 1.1: Graphical interface in RStudio

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.

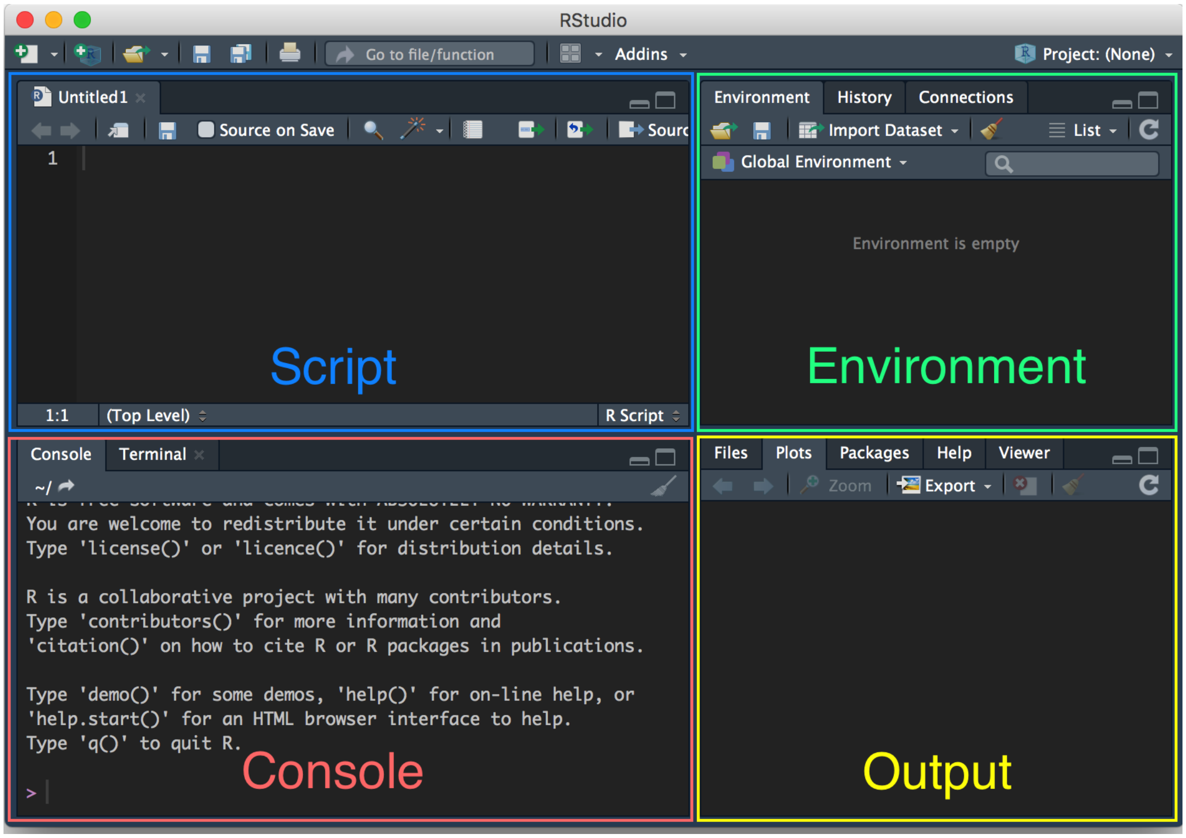


Figure 1.2: Graphical interface in RStudio, explained

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 can add code and make changes to your script. 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 the figures you create as well as documents. The *console* is where you can see your output and run commands.

Importantly, everything you do in `R` can be written as commands. This ensures that you will always be able to document your work (in the `script` window). 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 the following:

```
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 you have marked. Insert the code below in your script, mark it, run it and see how the output shows up 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` works via 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 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 to do what you want. The sky is the limit and the only thing you need to learn now 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:

```r
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 if you don't need them. For this reason, most scripts begin with loading the packages that you need. 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.

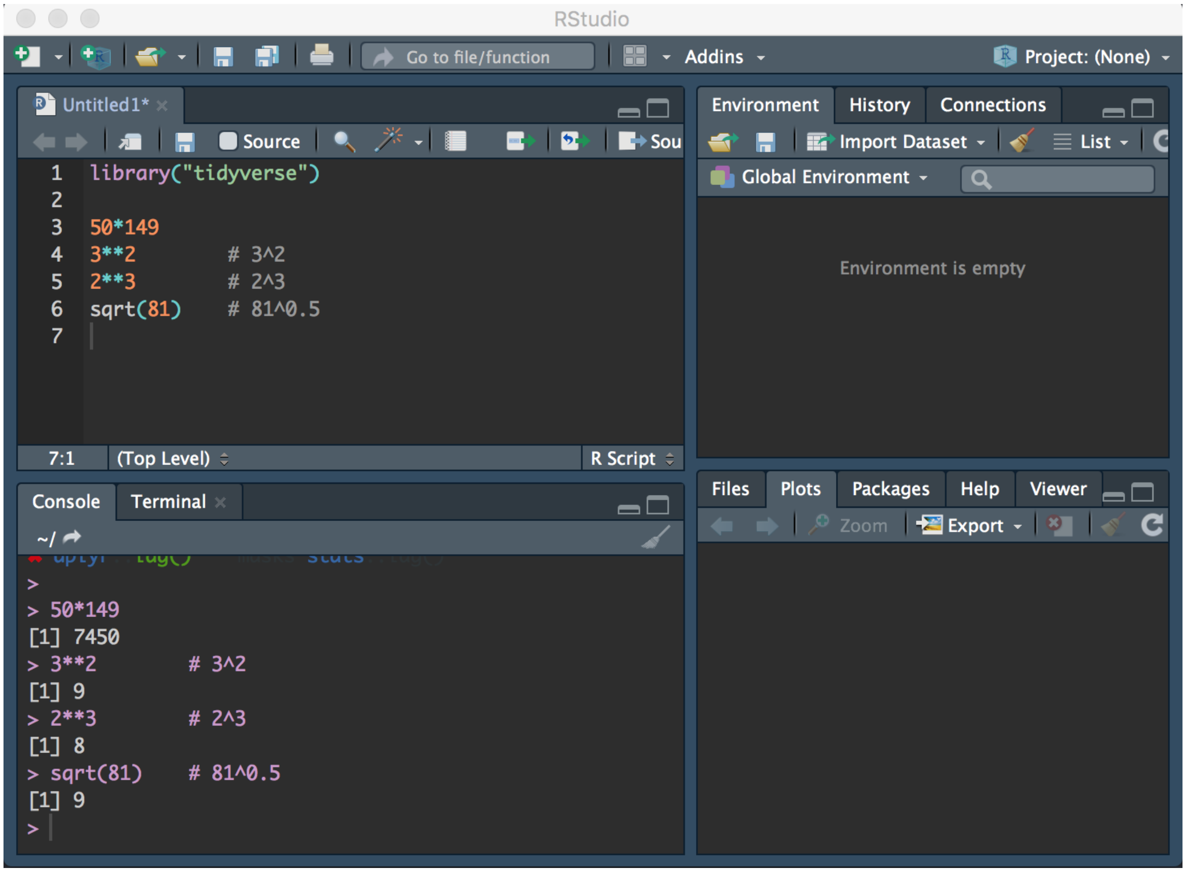


Figure 1.3: A script in RStudio

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 the last chapter from your script window: write 2+2 and run the code. This should look like the output below.

```
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 example, want to connect two different surveys, you can have them both loaded in the memory at the same time and work with them. This is not possible in SPSS or Stata.

To save something in an object, 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 below 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 output from 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 making mistakes.

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.

```
y
```

```
[1] 9
```

Alternatively, when we create the object, we can include it all in a parenthesis as we do below.

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

```
[1] 9
```

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

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.

```
1:10
```

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

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

```
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 happening in `R` happens with functions. A vector can look like this:

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

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

This is a *numerical* vector. A vector is a collection of values of the same type. We can save any vector in an object. In the code below we save four numbers (14, 6, 23, 2) in the object `x`.

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

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

[2] In the example with `1:10`, this is similar to writing `c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` and `c(1:10)`. 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.

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.

```
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 get the third number in the object.

```
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 the example below we will get all values except for 2. Also, note that since we are not assigning anything to an object (with `<-`), we are not making any changes to `x`.

```
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 median, the mean, the standard deviation etc.

```
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 of 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 get 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 specify 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 on 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 than or equal to
x >= 2.01      # greater than or equal to
```

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

2.4 Text as data

In addition to numbers we can and will 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.

```
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"*).

```
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 are: `character` (text), `numeric` (numbers), `integer` (whole numbers), `factor` (categories) and `logical` (logical).

```
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.

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

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

To get the number of characters for each element in our object, we can use the function `nchar()`:

```r
nchar(p)
```

```
[1] 18 12 23 17 25  9
```

We can also convert the characters in different ways. First, we can convert all characters to uppercase with `toupper()`. Second, we can concert all characters to lowercase with `tolower()`.

```r
toupper(p)
```

```
[1] "CONSERVATIVE PARTY"         "LABOUR PARTY"
[3] "SCOTTISH NATIONAL PARTY"    "LIBERAL DEMOCRATS"
[5] "DEMOCRATIC UNIONIST PARTY"  "SINN FÉIN"
```

```r
tolower(p)
```

```
[1] "conservative party"         "labour party"
[3] "scottish national party"    "liberal democrats"
[5] "democratic unionist party"  "sinn féin"
```

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.

```r
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 stored in the object. 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 specify, 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]

As a first step we can create new objects with more information: `leader` (information 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

[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

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()` again to show that `uk2017` is a data frame.

```
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`.

```
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 of the data frame.

```
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
```

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()`.

```
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()`.

```
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 show the full data frame. One way to see 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()` to 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).

```r
View(uk2017)
```

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. However, now 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 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 of 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 illustrate how we can 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 the logical operator `==` 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.

```
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 to what 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 (more on this specific function in `R` later). First, we save the model in the object `uk2017_lm` and then use `summary()` to get the results.

```
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
```

The coefficient for `votes` is positive and statistically significant ($p < 0.05$). In other words, as the vote share increases, so does the number of seats.

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 can 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. In other words, we need to tell `R` where it should be saving the file and - when we want to import a data frame - where to look for a file. To see where `R` is currently working from (the *working directory*) you can type `getwd()`. This will return the place where `R` is currently going to save the file if we do not change it.

```
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. In the code below I change the working directory to the folder `book` in the folder `qpolr` in the Dropbox folder. Do also note that we are using forward slash (`/`) and not backslash (`\`).

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

An easy way to control the working directory is to open an `R`-script directly from the folder you want to have as your working directory. Specifically, instead of opening RStudio and finding the script, find the script in your folder and open 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`).

```
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`.

```
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) and `readr` (H. Wickham & Francois, 2015). If you install and load the package `rio`, you can use the functions `import()` and `export()`.

```
# 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. To see 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()
```

(PART) Working with data

Chapter 3

Data management

There are multiple ways to manage data in `R` and in particular different 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 and create new variables. Noteworthy, there are multiple packages we can use to manipulate data frames, but the best is without a doubt `dplyr` (Hadley Wickham & Francois, 2016). This is part of the `tidyverse` package so you do not need to install any new packages if you have already installed `tidyverse`.

The package provides some basic functions making it easy to work with data frames. These functions include `select()`, `filter()`, `arrange()`, `rename()`, `mutate()` and `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 that you will need to save in a new object or overwrite the existing object with your data frame.

As the `dplyr` package is part of the `tidyverse`, the first thing we do is to call the `tidyverse`.

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

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

We will use the dataset we created in the previous chapter. If you do not have it, you can download it here (make sure to have the data file saved in your working directory): http://qpolr.com/data/uk2017.csv

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

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

```
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 task 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:

```
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
```

Again, this is not saved in a new data frame. If we want to save this in a new data frame, say uk2017_pv, we need to assign the output from select() to our object.

```
uk2017_pv <- select(uk2017, party, votes)
```

There are multiple different functions that can help us find 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 the text seat.

```
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 similar to contains() that can be of help 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.

```
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
```

Last, 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).

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

```
Warning: package 'bindrcpp' was built under R version 3.4.4
```

```
                      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`. Again, 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.

```r
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 greatest 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 that 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`.

```r
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 in the election.

```r
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
```

We can also use the sum() function to find the proportion of seats a party got in a variable, seats_prop.

```r
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.

```r
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 as the number of functions we use increases. Furthermore, the likelihood of making a stupid mistake, e.g. by including an extra (or) increases substantially. 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 done and do not want to do more to our data frame.

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.

```
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` or `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 get the same information 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 FALSE                  2       3        42.4       22.7
2 TRUE                   4       0.7      40         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.

Chapter 4

Get existing data

There are multiple ways you can get data into `R`. In this chapter we introduce different strategies for getting data into `R` from a variety of political data sources. First, we look at data included in packages. Second, we show how you can find datasets online and introduce a resource with a lot of links to political datasets. Third, we introduce a series of different packages that makes it easy to get data into `R`.

Throughout the chapter we will use the `tidyverse` package so make sure to load this.

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

4.1 Using data from data packages

A lot of the packages we are working with, including packages in the `tidyverse`, include datasets. To illustrate this, we will be using the package `poliscidata`.[1] The first thing we will need to do is to install the package.

```
install.packages("poliscidata")
```

Next, we will need to load the package with `library()`.

```
library("poliscidata")
```

[1] For more information on the package and the included packages, see: https://cran.r-project.org/web/packages/poliscidata/poliscidata.pdf

There are multiple datasets in the `poliscidata` package. We will focus on the dataset `states`, a dataset with variables about the 50 states in the United States. We use the function `names()` to get a list of all variables in the data frame states (it takes up a lot of space but gives an indication of the variety of variables in the data frame).

```
names(states)
```

```
 [1] "abort_rank3"       "abortion_rank12"   "adv_or_more"
 [4] "ba_or_more"        "cig_tax12"         "cig_tax12_3"
 [7] "conserv_advantage" "conserv_public"    "dem_advantage"
[10] "govt_worker"       "gun_rank3"         "gun_rank11"
[13] "gun_scale11"       "hr_cons_rank11"    "hr_conserv11"
[16] "hr_lib_rank11"     "hr_liberal11"      "hs_or_more"
[19] "obama2012"         "obama_win12"       "pop2000"
[22] "pop2010"           "pop2010_hun_thou"  "popchng0010"
[25] "popchngpct"        "pot_policy"        "prochoice"
[28] "prolife"           "relig_cath"        "relig_prot"
[31] "relig_high"        "relig_low"         "religiosity3"
[34] "romney2012"        "smokers12"         "stateid"
[37] "to_0812"           "uninsured_pct"     "abort_rate05"
[40] "abort_rate08"      "abortlaw3"         "abortlaw10"
[43] "alcohol"           "attend_pct"        "battle04"
[46] "blkleg"            "blkpct04"          "blkpct08"
[49] "blkpct10"          "bush00"            "bush04"
[52] "carfatal"          "carfatal07"        "cig_tax"
[55] "cig_tax_3"         "cigarettes"        "college"
[58] "conpct_m"          "cons_hr06"         "cons_hr09"
[61] "cook_index"        "cook_index3"       "defexpen"
[64] "demhr11"           "dem_hr09"          "demnat06"
[67] "dempct_m"          "demstate06"        "demstate09"
[70] "demstate13"        "density"           "division"
[73] "earmarks_pcap"     "evm"               "evo"
[76] "evo2012"           "evr2012"           "gay_policy"
[79] "gay_policy2"       "gay_policy_con"    "gay_support"
[82] "gay_support3"      "gb_win00"          "gb_win04"
```

```
 [85] "gore00"           "gun_check"         "gun_dealer"
 [88] "gun_murder10"     "gun_rank_rev"      "gunlaw_rank"
 [91] "gunlaw_rank3_rev" "gunlaw_scale"      "hispanic04"
 [94] "hispanic08"       "hispanic10"        "indpct_m"
 [97] "kerry04"          "libpct_m"          "mccain08"
[100] "modpct_m"         "nader00"           "obama08"
[103] "obama_win08"      "over64"            "permit"
[106] "pop_18_24"        "pop_18_24_10"      "prcapinc"
[109] "region"           "relig_import"      "religiosity"
[112] "reppct_m"         "rtw"               "secularism"
[115] "secularism3"      "seniority_sen2"    "south"
[118] "state"            "to_0004"           "to_0408"
[121] "trnout00"         "trnout04"          "unemploy"
[124] "union04"          "union07"           "union10"
[127] "urban"            "vep00_turnout"     "vep04_turnout"
[130] "vep08_turnout"    "vep12_turnout"     "womleg_2007"
[133] "womleg_2010"      "womleg_2011"       "womleg_2015"
```

While the data is available, it is not possible to see in the *Environment* window. To see the data frame, we can save `states` in an object of the same name.

```r
states <- states
```

Now we can see in the `Environment` window that we have 50 observations of 135 variables. We will be using this data later, but for now we will see that we have actual data. Using the `table()` function we can show the distribution of observations in the `gay_policy` variable, showing data on the Billman's policy scale (4 ordinal categories).

```r
table(states$gay_policy)
```

```

     Most liberal           Liberal      Conservative Most conservative 
                6                14                10                20 
```

Here we see that 6 states have a most liberal score, 14 have a liberal score, 10 have a conservative score, and 6 have a most conservative score.

4.2 Download data from webpages

A lot of the political datasets you will find are available online and can be downloaded for free. A free resource with an overview of political datasets can be found here: https://github.com/erikgahner/PolData

In this dataset with political datasets, you can find datasets from different topics (international relations, political institutions, democracy etc.). For each dataset you will also be able to see whether it is possible to download the data for free, and if so, what the link to the dataset is.

To illustrate this, we can find the link to download the Global Media Freedom dataset. The dataset is available as a `.csv` file and get into `R` with the `read.csv()` function.

```
gmd <- read.csv(
  "http://faculty.uml.edu/Jenifer_whittenwoodring/GMFD_V2.csv"
  )
```

The dataset consists of the following four variables: `id`, `year`, `country`, `mediascore`.

In the next sections, we will introduce different packages, that can make it easier to work with different datasets.

4.3 Data: European Social Survey (`essurvey`)

To get data from European Social Survey (ESS), we will be using the `essurvey` package. If you do not have a free user, the first step is to go online and create a user: http://www.europeansocialsurvey.org/user/new

The next thing you need to do is to install the package.

```
install.packages("essurvey")
```

And then load the package.

```
library("essurvey")
```

Now you need to set the email you used to register an account. If you don't do this, ESS will not be able to confirm that you have an account, and you will not be able to get access to the data.

```r
set_email("your@mail.com")
```

There are multiple functions to use in order to get data, and for an overview of some of them, check out https://ropensci.github.io/essurvey/.

Here, we will provide an example on how to reproduce the main result in Larsen (2018). Here we use the `import_country()` function to import data from Denmark in Round 6 of the ESS.

```r
ess <- import_country("Denmark", 6)
```

All the recodings are made with the `mutate()` function.

```r
ess <- ess %>%
  mutate(
    stfgov = ifelse(stfgov > 10, NA, stfgov),
    reform = case_when(inwmme < 2 ~ 0,
                       inwmme == 2 & inwdde < 19 ~ 0,
                       inwmme == 2 & inwdde > 19 ~ 1,
                       inwmme > 2 ~ 1,
                       TRUE ~ NA_real_)
  )
```

And the regression model can be achieved with the `lm()` function.

```r
lm(stfgov ~ reform, data=ess)
```

4.4 Data: Manifesto Project Dataset (manifestoR)

To use data from the Manifesto Project Dataset, you need to create an account as well. This can be done at: https://manifesto-project.wzb.eu/signup

Next, install and load the package.

```r
# install the package
install.packages("manifestoR")

# load the package
library("manifestoR")
```

You now need to go to your profile page at https://manifesto-project.wzb.eu/. You will need to click on the button to get an API key. You can now click 'download API Key file (txt)' and place this file in your working directory - or copy your key and use the code below.

```
mp_setapikey(key = "yourKeyHere")
```

You are now able to download text data from the Manifesto Project into R. We use the mp_corpus() function to download election programmes texts and codings, in this case from Denmark.

```
manifesto_dk <- mp_corpus(countryname == "Denmark")
```

To see some of the content from the manifesto data, you can try the code below.

```
head(content(manifesto_dk[[1]]))
```

If you want to find a more detailed description of how to look at the data, please see https://cran.r-project.org/web/packages/manifestoR/vignettes/manifestoRworkflow.pdf.

4.5 Data: Varieties of Democracy (vdem)

To get data from Varieties of Democracy into R, we are going to use the vdem package. This package is not on CRAN, and accordingly, we cannot use install.packages() to install it. Instead, we will have to use the function install_github() as it is on GitHub. In order to do this, you need to have the package devtools. To install this package, you can uncomment the first line below. The second line says that we are using the install_github() function from the devtools package (with ::).

```
#install.packages("devtools")
devtools::install_github("xmarquez/vdem")
```

When the package is installed, use library() to load it.

```
library("vdem")
```

To get the main democracy indices from the data, we can use the `extract_vdem()` function.

```
vdem_data <- extract_vdem(section_number = 1)
```

This gives us a dataset with 17,604 observations of 55 variables. To see the first observations, use `head()` (output not shown).

```
head(vdem_data)
```

Chapter 5

Create data

In this chapter we will introduce different ways to create your own data. Specifically, we will show how to create data from existing files, how to scrape tables from webpages and how to get data from Twitter.

5.1 Create data from files

You will often encounter that the data of interest is not available in a format or structure that you will need for your analysis. Accordingly, as a first step, you will need to collect multiple files and turn them into a single dataset.

Here, we will use the example of election results from the Electoral Calculus. The example is from Matt Riggott (see the script here) and shows how we can download multiple files and connect them into a single dataset. Each file we will work with contains the election results from general elections in the UK.

As always, the first thing we will do is to load the `tidyverse` package.

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

Next, to get a sense of the data we will be looking at, go to your browser (e.g. Google Chrome or Safari) and open this link: https://www.electoralcalculus.co.uk/electdata_1955.txt

In this file, you will see multiple lines. These are the election results from 1955 (as indicated by the filename, `electdata_1955.txt`. The first line in the file is: `Name;MP;Area;County;Electorate;CON;LAB;LIB;NAT;MIN;OTH`. These are the variable names and are separated by `;`. By using the function `read_delim()` from the tidyverse package, we can load this file into R as a data frame. Notice that we specify that ; is used to separate fields. We save the file in the object el_1955.

```
el_1955 <- read_delim(
  "https://www.electoralcalculus.co.uk/electdata_1955.txt",
  delim = ";"
  )
```

To inspect the data, we can use the function head() (output not shown).

```
head(el_1955)
```

The above output shows that the data is loaded succesfully and saved in a data frame. We could do this for all elections manually, but that would take a lot of time and increase the odds of making mistakes. Instead, we will create a function that downloads all files. First, we specify the elections we are interested in (from 1955 to 2017) and save this information in the object election_years.

```
election_years <- c("1955", "1959", "1964", "1966", "1970", "1974feb",
"1974oct", "1979", "1983", "1987", "1992ob", "1997",
"2001ob", "2005ob", "2010", "2015", "2017")
```

Second, we use read_delim() again, but as part as a function where we use the read_delim() on the year we specify. We call this function read_election_data().

```
read_election_data <- function(election) {
  url <- paste0("http://www.electoralcalculus.co.uk/electdata_",
                election, ".txt")
  read_delim(url, delim = ";") %>%
  mutate(year = election)
}
```

With this function, we can specify any election year and get the data, e.g. read_election_data(2017) to get the data from 2017. Here, we use lapply() to run the function on all the election years mentioned in the object election_years above. To connect all elections, we use the function bind_rows(). We save the output in the object elections.

```
elections <- bind_rows(lapply(election_years, read_election_data))
```

To see whether it has worked, use `head()` on the object (output not shown).

```
head(elections)
```

5.2 Scrape data from tables

To scrape data from tables online, we use the `rvest` package. Remember to install it if you haven't already done so.

```
library("rvest")
```

In the example below, we will show how to easily scrape a table from a Wikipedia page. The first thing we do is to specify the link to the Wikipedia page and save it in the object `url`. In the example we will be looking at the election results from the 2014 European Parliament election in the United Kingdom.

```
url <- c(
  "https://en.wikipedia.org/wiki/European_Parliament_election,_2014_(United_Kingdom)"
  )
```

Next, we use the `read_html()` function to save the content on the Wikipedia page. We save the data in the object `wikipage`

```
wikipage <- read_html(url)
```

We can use the function `class()` to see what type of content we have in the object.

```
class(wikipage)
```

Here, we see that we have an xml_document and xml_node in our object. We want to save the tables in our data. To do this, we use the function `html_nodes()`.

```
data_table <- html_nodes(wikipage, "table")
```

If you type `data_table`, you can see an overview of all the tables we have saved. Here, we would like to use the data on the number of votes the different parties got in the 2014 European Parliament election in the United Kingdom. We can see that this table is number 14 in our object. We save this table in the object `data_results`.

```
data_results <- data_table[[14]]
```

We can now use the function `html_table()` to save the table as a data frame. We use the option `fill=TRUE` as there are empty cells in the table. We save the table in the object `ep14_raw`.

```
ep14_raw <- html_table(data_results, fill=TRUE)
```

To ensure that it is a data frame we are working with, we can use the function `class()` again.

```
class(ep14_raw)
```

We call the object `ep14_raw` as it is a raw table that needs further changes before we are satisfied. To get a sense of one of the issues with the data frame, we look at the last observations in the data frame with `tail()` (output not shown).

```
tail(ep14_raw)
```

Here we see that the last three rows are aggregated numbers unrelated to the votes for the individual parties. Accordingly, we would like to remove these observations. To remove the specific rows, we save the object without observations 32, 33 and 34.

```
ep14_raw <- ep14_raw[-c(32:34), ]
```

Next, we use `head()` to see what our data frame looks like for the first observations (output not shown).

```
head(ep14_raw)
```

We see two main issues. First, that the variable names are not unique and will need to be changed. We are interested in four of the variables, namely the name of the party, the number of votes, the vote share and the number of seats. We give the relevant variables names and give the other variables unimportant names (as we are going to ignore those).

```
names(ep14_raw) <- c("V1", "party", "votes", "share", "V5",
                     "seats", "V7", "V8", "V9", "V10")
```

Next, we can see that the first row is not an observation but variable names as well. Accordingly, we need to remove this observation as well.

```
ep14_raw <- ep14_raw[-c(1), ]
```

To remove the irrelevant variables in our data frame, we use the `select()` function to select the relevant variables.

```
ep14_raw <- ep14_raw %>%
  select(party, votes, share, seats)
```

The last thing to do is to tell `R` that three variables, `votes`, `share` and `seats` are numeric. Notice how we use the function `gsub()` to get rid of commas. We save this data frame in the object `ep14`.

```
ep14 <- ep14_raw %>%
  mutate(
    votes = as.numeric(gsub(",","", votes)),
    share = as.numeric(share),
    seats = as.numeric(seats)
  )
```

Inspect the final data frame. In this case, we do not have a lot of observations and we simply show them all.

```
ep14
```

Last, we create a figure showing the vote share and seats for the parties (notice that you will also need the package `ggrepel` to create the figure). (Output not shown)

```r
ggplot(ep14, aes(x = share, y = seats)) +
  geom_point() +
  theme_minimal() +
  ggrepel::geom_text_repel(
    aes(label = ifelse(share > 15, party, NA)),
      size = 4.5,
      point.padding = .2,
      box.padding = .4
    ) +
  labs(
    y = "Number of seats",
    x = "Vote share",
    title = "2014 European Parliament election, United Kingdom"
  )
```

5.3 Scrape political speeches

A lot of the text we can scrape online is not in the form of spreadsheets but in the form of nothing but text. To show how to scrape such text, we will focus on British political speeches from http://www.britishpoliticalspeech.org/speech-archive.htm.

Specifically, we will select the speech the Leader's speech by Theresa May in Manchester from 2017. First, as in the previous example, we specify the url of the page we would like to scrape. In this speech, Theresa May is talking extensively about the British Dream.

```r
url <- c(
  "http://www.britishpoliticalspeech.org/speech-archive.htm?speech=367"
  )
```

To get the content of the page with the speech, we save the content of the page in the object `speechpage`.

```r
speechpage <- read_html(url)
```

Next, to select the actual part of the page containing the speech, we select the content within the `<p></p>` tags.

```
data_speech <- html_nodes(speechpage, "p")
```

To get the actual text, we use the function `html_text()`.

```
data_speech_text <- html_text(data_speech)
```

Now we have all the text we need to use. However, to create a dataset with the words in the speech, we will use some functions from the package `tidytext` (as always, remember to install the package if you do not already have it installed) (Silge & Robinson, 2016).

```
library("tidytext")
```

The first function we are going to use is not part of the package but will be used to convert our speech into a data frame using the `data_frame()` function.

```
data_speech_df <- data_frame(text = data_speech_text)
```

While in a data frame, it is still just a lot of sentences on different rows. To unnest all the sentences in our text column into a word column, we use the function `unnest_tokens()`.

```
words <- data_speech_df %>% unnest_tokens(word, text)
```

This gives os an object, `words`, with 7,116 observations. However, a lot of these words are irrelevant stop words (most common words that we are not interested in such as *the*, *is*, *at*, *which*) that we would like to remove. We use the `anti_join()` function to remove all stop words.

```
words <- words %>% anti_join(stop_words, by = "word")
```

Last, we can count the words in the speech and calculate the number of occurences.

```
words %>% count(word, sort = TRUE)
```

We see that *people* is mentioned 49 times, and *britain* is mentioned 36 times. *dream* and *british* are mentioned 33 and 29 times, respectively.

5.4 Get data from Twitter

To get data from Twitter, we are going to use the `rtweet` package (Kearney, 2018). The first thing we do is to load the package (remember to install if if you have not already done so). You can find more information about the package here: https://rtweet.info/

```
library("rtweet")
```

Next, to make sure you can collect data, you need to have a Twitter user. You can register for free at https://twitter.com/. You will need this in order to use the `rstats2twitter` app. Last, make sure to install the `httpuv` package as well.

```
library("httpuv")
```

Noteworthy, we cannot just collect data without any limits. In most cases, we have a liit of 18,000 observations per 15 minutes.

5.4.1 Data on Twitter user

To get data on a Twitter user, we can use different functions. There is a distinction betweeen friends and followers. The accounts a user follows are called friends, whereas followers are the accounts that follow a user. Here, we will use the `get_friends()` function to get information on the people Donald Trump is following.

```
trump_following <- get_friends("realDonaldTrump")
```

When we do that, all we get is a series of user IDs for the people Donald Trump is following. We can use the `lookup_users()` function toget more information about the individual accounts.

```
trump_following <- lookup_users(trump_following$user_id)
```

This gives us a lot more information on the individual users, including their Twitter handle, name and description. To see all the information saved, you can use the `names()` function.

```
names(trump_following)
```

To save information on the user ID, the handle, name and the description, we create a new object called `trump_data` just with these variables.

```
trump_data <- trump_following %>%
  select(user_id, screen_name, name, description)
```

You can use `head(trump_data)` to see what the data looks like. To get information on the followers of Donald Trump, you can use the `get_followers()` function. However, this will take a lot of time to get (we are talking days!).

To get the most recent tweets from, Donald Trump, we can use the `get_timeline()` function.

```
trump_tweets <- get_timeline("realDonaldTrump", n = 100)
```

To search for tweets from specific users, we can use the `search_users()` function. Below, we search for tweets from users with `politics` (via Twitters search query).

```
politics_users <- search_users("politics", n = 50)
```

Next, we can use the `get_favorites()` function to get data on the tweets a user has favorited. Here, we save the favorites from Boris Johnson and save it in the object `tweets_bj`.

```
tweets_bj <- get_favorites("BorisJohnson")
```

To get a sense of what this data looks like, you can use the `head()` function.

```
head(tweets_bj)
```

5.4.2 Data on trends

To get data on what is trending in a certain part of the world, we can use the `get_trends()` function. Below, we get the 50 most trending topics in the United Kingdom. On October 29, 2018, `#NationalCatDay` and `Angela Merkel` are both trending (not for the same reason though).

```
trends_uk <- get_trends("united kingdom")
```

5.4.3 Data on tweets

Last, to get data on specific tweets, we first use the `search_tweets()` function. Below, we get the most recent 100 tweets mentioning `brexit`. We also specify that we are not interested in retweets.

```
brexit <- search_tweets(
  "brexit", n = 100, include_rts = FALSE
)
```

This gives us a data frame with 100 observations and 88 variables. You can use the `names()` function to get a list of all variables in the data frame.

You can also use the search operators provided by Twitter, e.g. by filtering only tweets linking to news articles.

```
news <- search_tweets("filter:news", n = 100)
```

We acn combine the two searches above and only search for tweets with Brexit related news.

```
brexit_news <- search_tweets("brexit filter:news", n = 100, include_rts = FALSE)
```

If we want to only include tweets with a video, we can use `"filter:video"`:

```
videos <- search_tweets("filter:video", n = 100, include_rts = FALSE)
```

To look up data on a specific tweet, use the function `lookup_tweets()`. You can find the id on a tweet by looking in the url for a tweet (or in the variable `status_id`).

```
lookup_tweets("1065623990746710022")
```

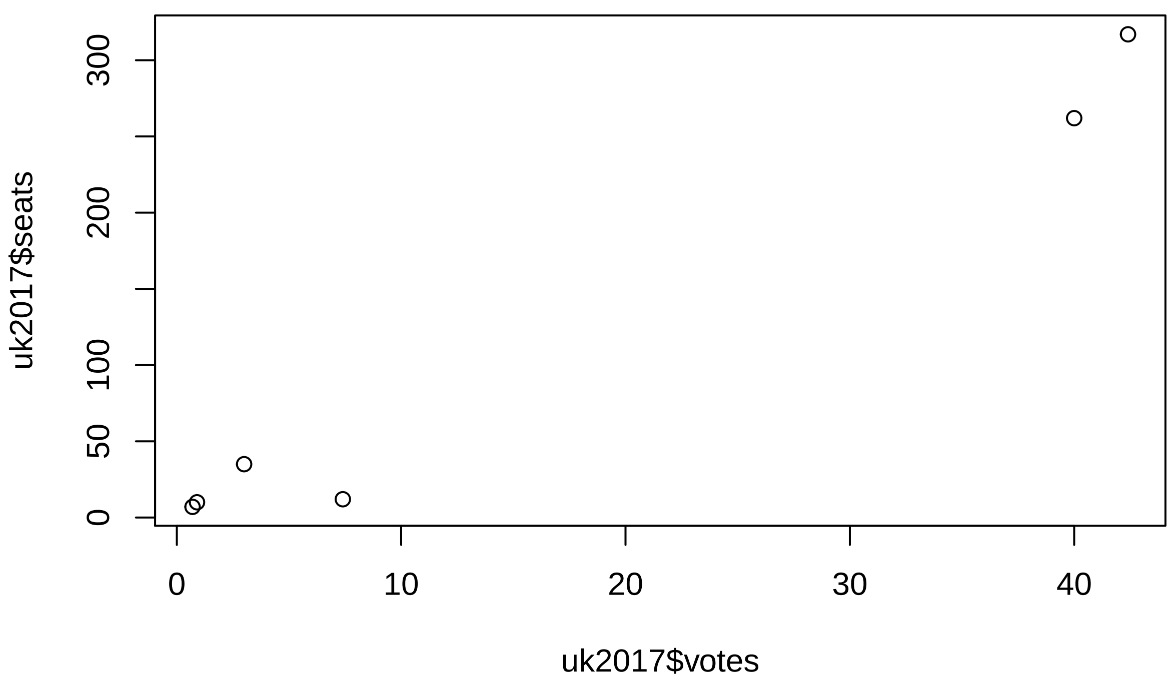
(PART) Presenting data

Chapter 6

Data visualisation

Visualising data is important (Healy & Moody, 2014). As with everything in R, there are a lot of different ways to visualise data. One simple way to visualise data is to use *base* functions in R (i.e. functions that come when you install the R language). Below you will see an example on this.

```
plot(x=uk2017$votes, y=uk2017$seats)
```



There is nothing inherently wrong with using a function like this, but the moment we want to tweak the figure, it gets complicated. Accordingly, we will not use the standard functions in R but the package `ggplot2` (H. Wickham, 2009). This package makes it easy to create beautiful figures in R.

`ggplot2` creates more beautiful figures with better defaults, it is very customizable, and it works within the tidyverse (together with `dplyr`). For those reasons it is becoming incredibly popular among practitioners and academics alike. That being said, there is an element of personal preference when it comes to data visualisations and `ggplot2` is not perfect. While the defaults are good, they could be better. Furthermore, there are functions in the package you should *never* use (such as `qplot()`, short for *quick plot*).

6.1 The basics of ggplot2

You can load `ggplot2` by loading the `tidyverse` (alternatively you can just load the `ggplot2` package).

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

The two g's (`gg`) in `ggplot2` are short for *grammar of graphics*. The philosophy is that we are working with building blocks in the form of a sentence structure where we can add more components to our visualisation, e.g. change colours and add text. This makes it easy to first create a figure and then tweak it till we are satisfied.

These building blocks are:

1. Data (the data frame we will be using)
2. Aesthetics (the variables we will be working with)
3. Geometric objects (the type of visualisation)
4. Theme adjustments (size, text, colours etc.)

6.1.1 Data

The function we will be using is `ggplot()`. Here, we will be using the `states` data from the `poliscidata` package introduced in Chapter 4.

```
library("poliscidata")
states <- states
```

The first thing we always have to specify in our function is the data frame. In other words, you will *always* have to use a data frame.

```r
ggplot(states)
```

Do note that if you run the code above - and have the `states` in your working memory, we will not get anything but an empty plot. The only thing we have done so far is telling `R` that we would like to create a coordinate system and data from `uk2017` should play some role, but this is of course not enough.

6.1.2 Aesthetics

The next thing we have to specify is what variables in the data frame we will be using and what role they play. To do this we will use the function `aes()` *within* the `ggplot()` function after the data frame (remember the comma after the data frame).

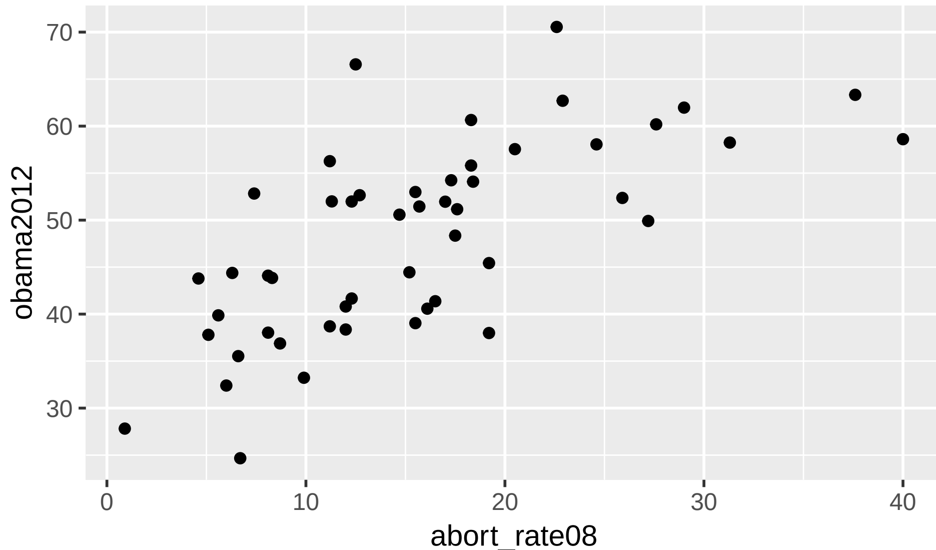
```r
ggplot(states, aes(x = abort_rate08, y = obama2012))
```

In the example above we specify that we are working with *two* variables, x (Number of abortions per 1,000 women aged 15-44 in 2008) and y (Obama vote share in 2012). If you only will be working with one variable (e.g. a histogram), you should of course only specificy one variable, x. However, now we have only told `R` what variables we would like to work with, but it is still not enough to actually create a figure.

6.1.3 Geometric objects

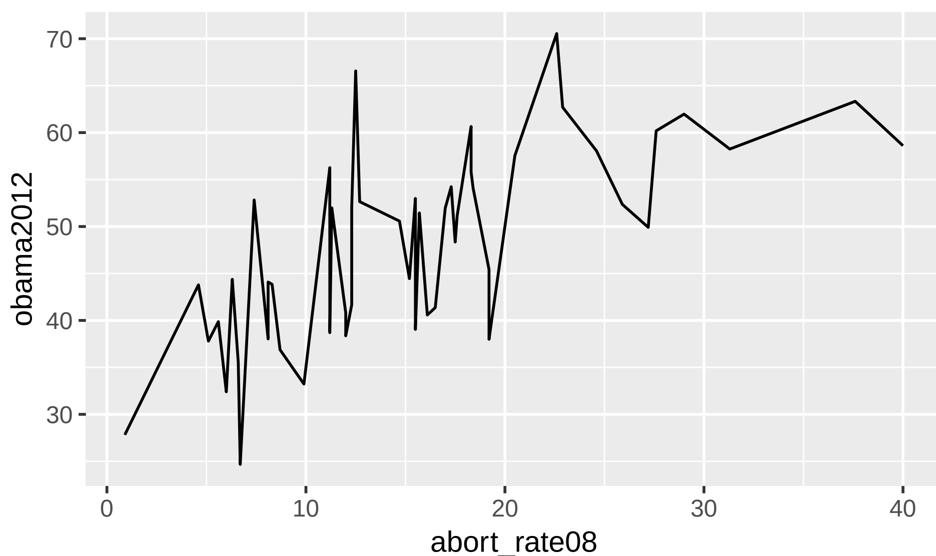
Now we will need to add the geometric object, we would like to visualise. We need to go to a new line and tell `R` to follow along. To do this, we add a plus (`+`) at the end of the line. On the new line we add the type of geometric object (`geom_`), we want add. To replicate the plot above we use `geom_point()`.

```r
ggplot(states, aes(x = abort_rate08, y = obama2012)) +
  geom_point()
```



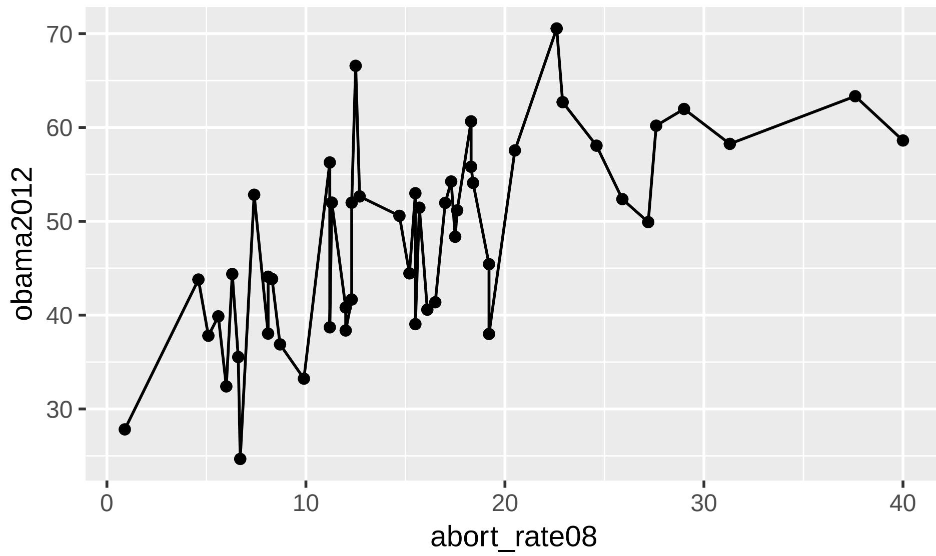
This is a standard `ggplot2` plot with all its defaults. If we instead a scatter plot wanted a line plot, we can change `geom_point()` to `geom_line()`.

```
ggplot(states, aes(x = abort_rate08, y = obama2012)) +
  geom_line()
```



The above figure is somewhat misleading so it is just to show the logic of the how geometric objects work. Interestingly, we can add multiple geometric objects to the same plot. Below, we add both geometric objects used above.

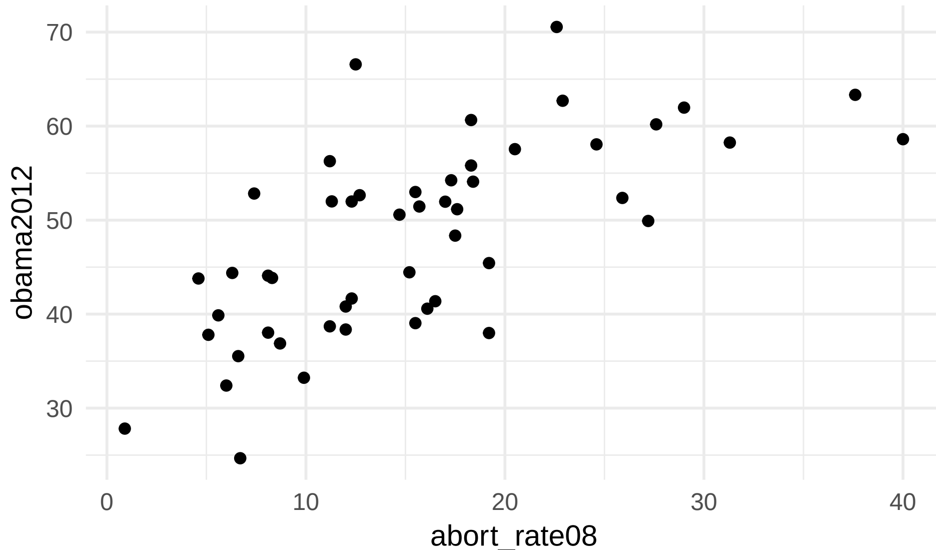
```
ggplot(states, aes(x = abort_rate08, y = obama2012)) +
  geom_line() +
  geom_point()
```



6.1.4 Theme adjustments

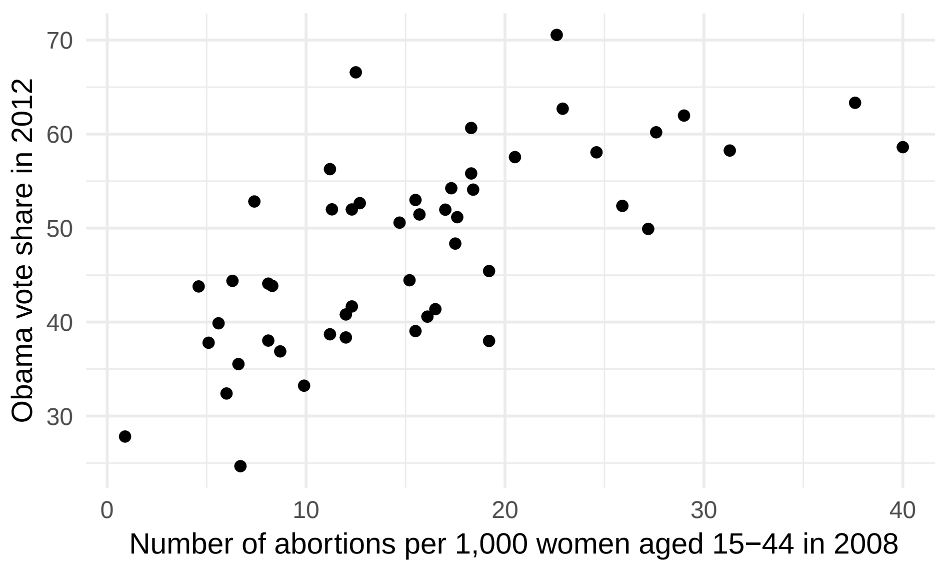
What you will see in a typical plot is that it is not done. The axes simply have the variable names, the colours are not great etc. Accordingly, we often need to add and change elements of our plot. Here we add the theme of the plot (described in detail below).

```
ggplot(states, aes(x = abort_rate08, y = obama2012)) +
  geom_point() +
  theme_minimal()
```



We can also easily change the labels by using `xlab()` and `ylab()`.

```
ggplot(states, aes(x = abort_rate08, y = obama2012)) +
  geom_point() +
  theme_minimal() +
  ylab("Obama vote share in 2012") +
  xlab("Number of abortions per 1,000 women aged 15-44 in 2008")
```



This is the basic logic of `ggplot2`.

6.2 Plotting one variable: distributions

Table 6.1 shows the geometric objects we will be working with below. In addition to the name of the object, you will also find a link where you can find more illustrations and examples on how they work.

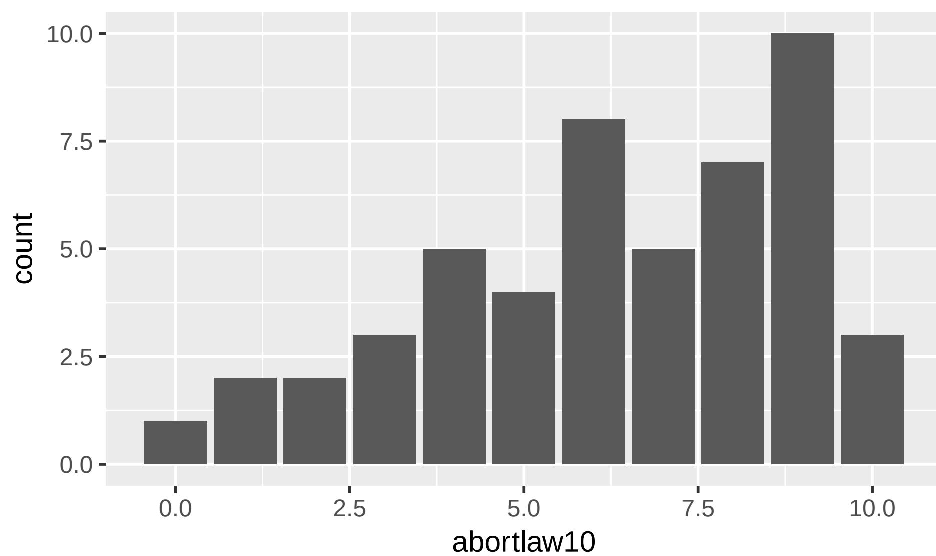
Table 6.1: Selected geometric objects with `ggplot2`

Name	Function	Cookbook for R
Bar plot	`geom_bar()`	Bar and line graphs
Histogram	`geom_histogram()`	Plotting distributions
Density plot	`geom_density()`	Plotting distributions

6.2.1 Bar plot

The first plot we will do is a bar plot. To do this we use a variable on the number of restrictions on abortion (`abortlaw10`) and `geom_bar()`.

```
ggplot(states, aes(x=abortlaw10)) +
  geom_bar()
```

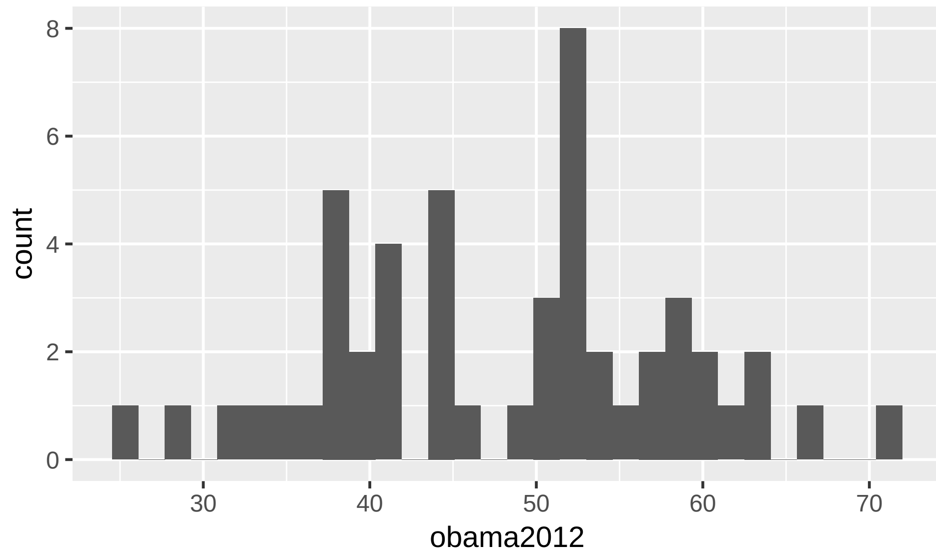


6.2.2 Histograms

The next figure we will work with is the histogram. Here we will plot the distribution of Obama's vote share in 2012 (the `obama2012` variable) and use `geom_histogram()`.

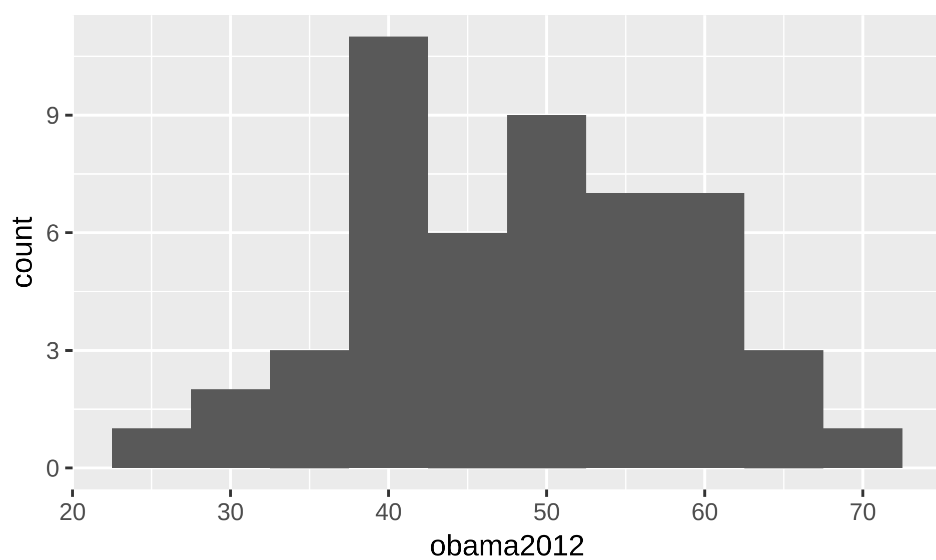
```
ggplot(states, aes(x=obama2012)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



As you can see, we get a message about the use of a default binwidth. This is to emphasize the importance of specifying the binwidth yourself. We can change the bin width by adding `binwidth` to `geom_histogram()`.

```
ggplot(states, aes(x=obama2012)) +
  geom_histogram(binwidth = 5)
```

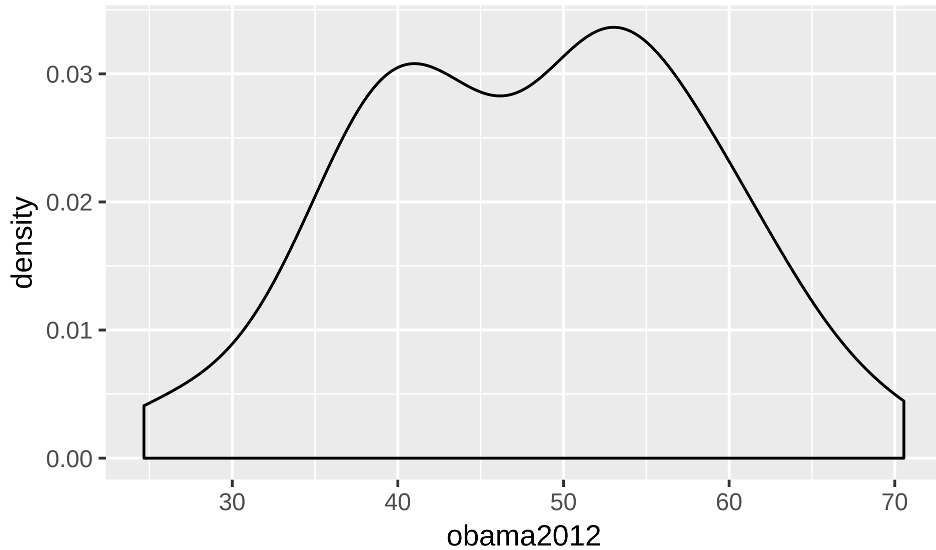


Play around with different binwidths to see how it affects the distribution in the figure.

6.2.3 Density plots

The histogram is not the only way to show the distribution of a variable. To make a density plot, you can use `geom_density()`. We use the `obama2012` variable again.

```
ggplot(states, aes(x=obama2012)) +
  geom_density()
```



Do compare the density plot to the histograms above.

6.3 Plotting two variables: relationships

To show how different variables are related, Table 6.1 shows the geometric objects we will be working with below as well as link where you can find more information.

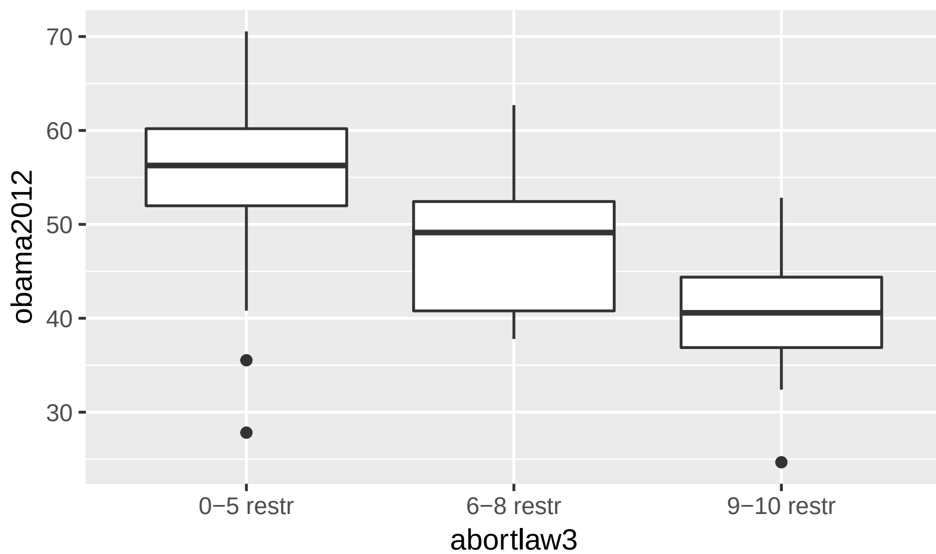
Table 6.2: Selected geometric objects for relations in `ggplot2`

Name	Function	Cookbook for R
Box plot	`geom_boxplot()`	Plotting distributions
Scatter plot	`geom_point()`	Scatterplots

6.3.1 Box plot

For the box plot, we will be using `geom_boxplot()` to show how the vote share for Obama is related to abortion laws (here with the `abortlaw3` variable, i.e. abortion restrictions with three tiers of number of restrictions).

```
ggplot(states, aes(x=abortlaw3, group=abortlaw3, y=obama2012)) +
  geom_boxplot()
```

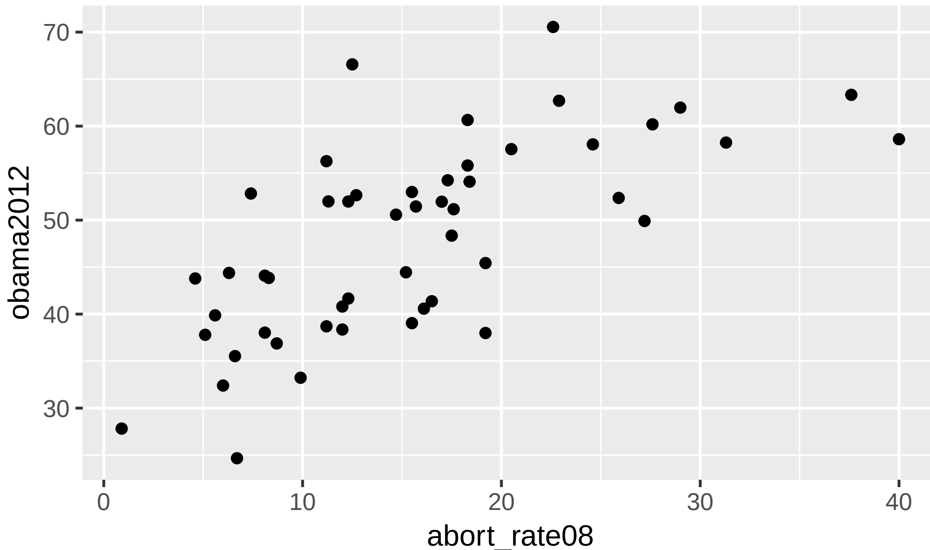


Here we can see that Obama got a greater vote share in states with less restrictions on abortion.

6.3.2 Scatter plots

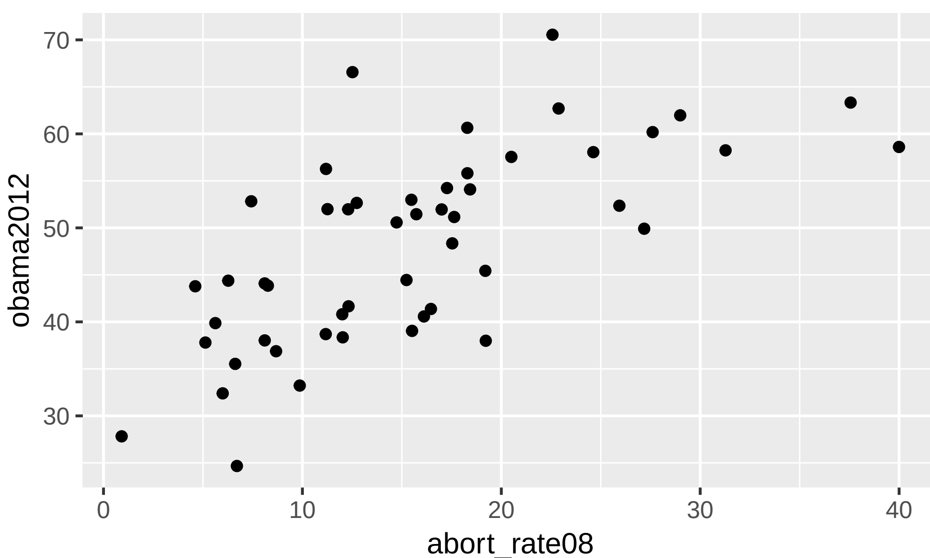
To illustrate the relation between number of abortions and Obama's vote share, measured with the variables `abort_rate08` and `obama2012`, we will create a scatter plot with `geom_point()`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point()
```



If we are working with a lot of observations, there will be an overlap in the points. To show all of the observations, we can add some small, random noise to the observations, so we can see more of them. To do this, we can use `geom_jitter()` instead of `geom_point()`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_jitter()
```

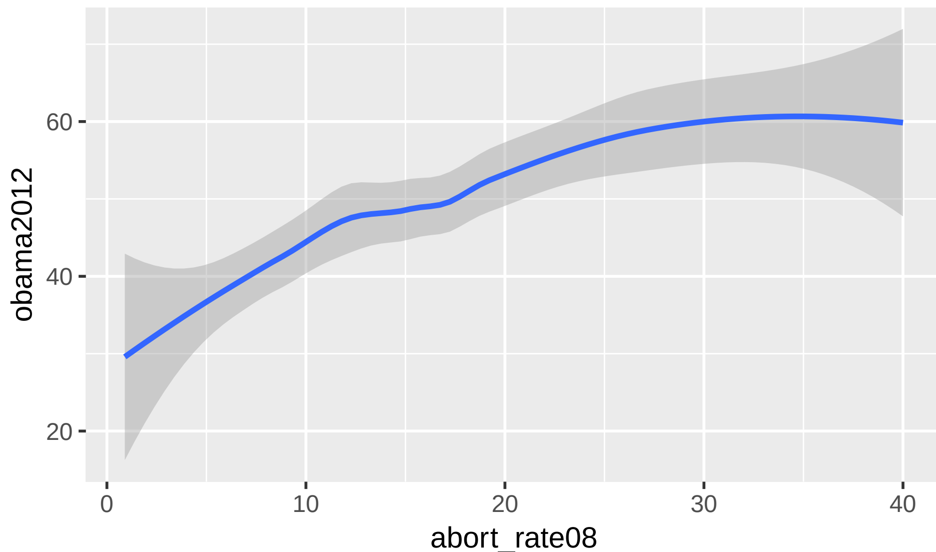


We can also use `geom_point(position = "jitter")` instead of Instead of `geom_jitter()`. However, in this particular case, as we only have 50 observations, it is not a major concern.

6.3.3 Line plots

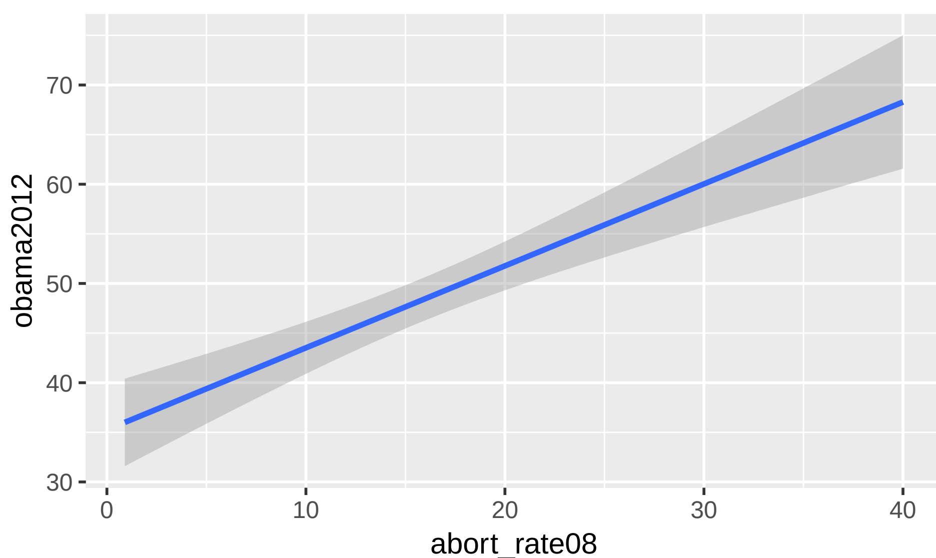
To create a regression line we can use the `geom_smooth()` function. Here we will again look at the relation between `abort_rate08` and `obama2012`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_smooth()
```



Here we can see that as the abortion rate increases, so does the vote share for Obama. As we can also see, this is a smoothing function. To have a linear line instead we can specify that we will be using `method="lm"` as an option.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_smooth(method="lm")
```



6.4 Manipulating plots

6.4.1 Themes

As you could see in the plots above, we have used a default theme in `ggplot2`. Table 6.3 shows a series of themes to be found in `ggplot2`and the package `ggthemes`. These are just a selection of some of the themes.

Table 6.3: Selected themes for `ggplot2`

Function	Package	Description
theme_bw()	`ggplot2`	Black elements on white background
theme_minimal()	`ggplot2`	Minimalistic
theme_classic()	`ggplot2`	Theme without grid lines
theme_base()	`ggthemes`	Copy of the base theme in `R`
theme_economist()	`ggthemes`	The Economist theme
theme_fivethirtyeight()	`ggthemes`	FiveThirtyEight theme
theme_tufte()	`ggthemes`	Tufte (1983) theme

Figure 6.1 shows the look of the different themes. The order is: Standard, `theme_bw()`, `theme_minimal()`, `theme_classic()`, `theme_base()`, `theme_economist()`, `theme_fivethirtyeight()`, `theme_tufte()`.

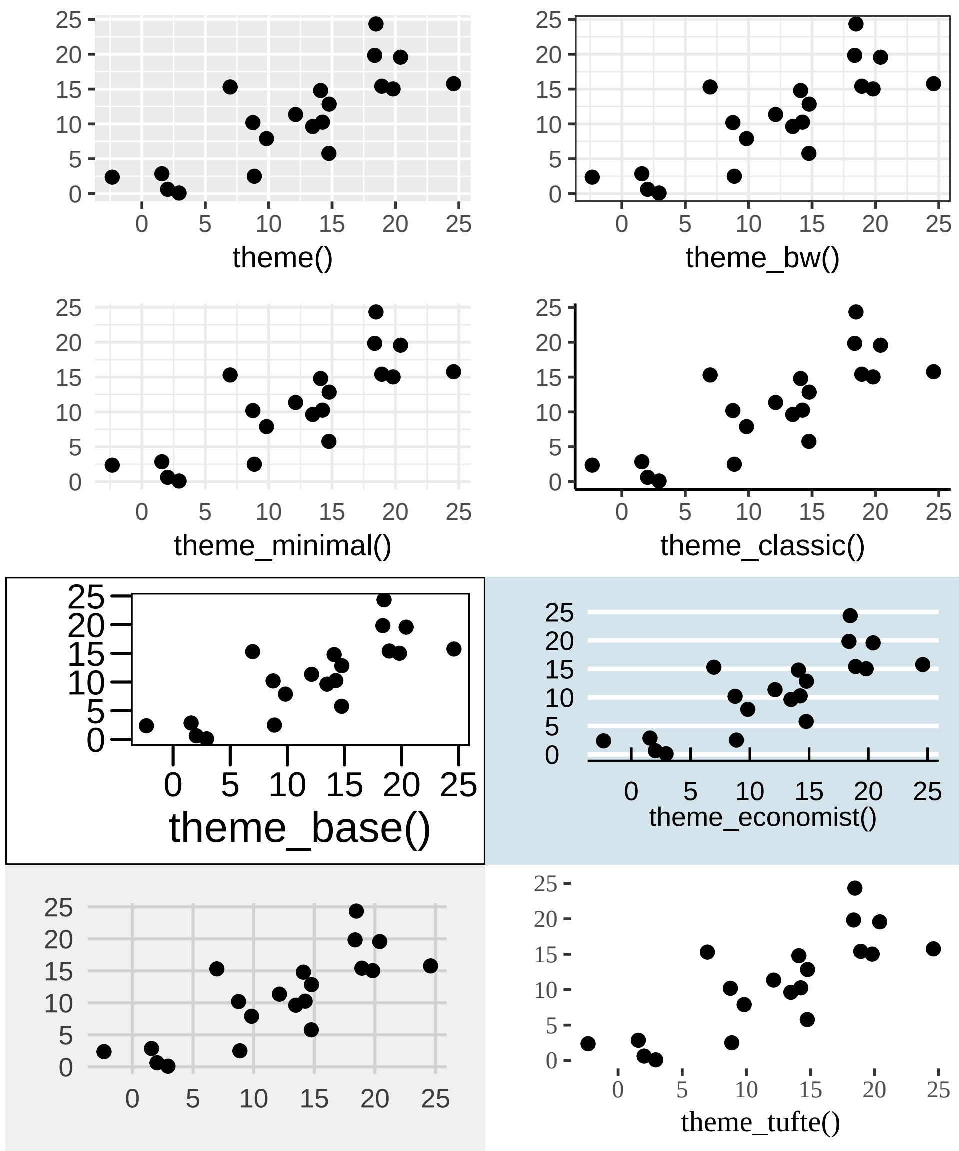
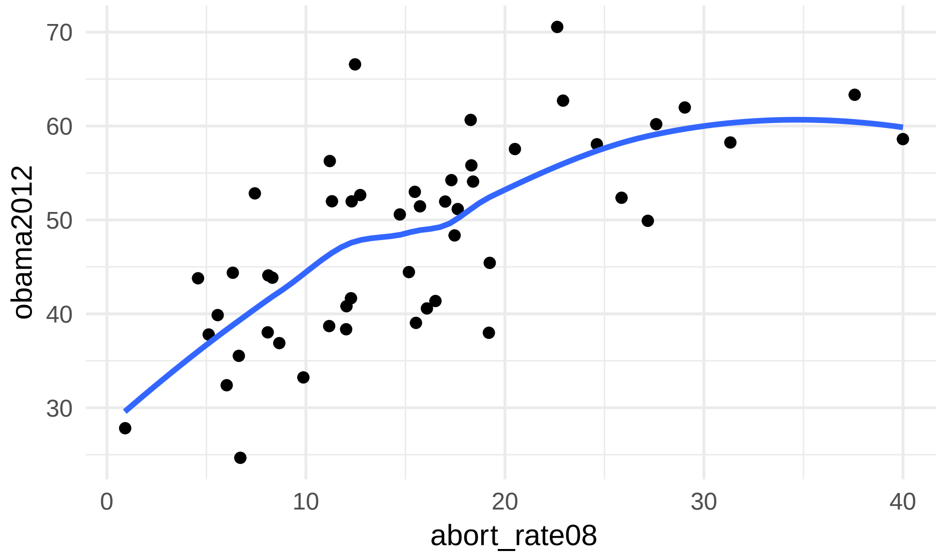


Figure 6.1: Eight themes

You can find a lot more resources online related to `ggplot2`. In addition to the links above, do consult ggthemr and ggplot2 extensions.

Below, we will be using `theme_minimal()` as the theme when we work with out plots.

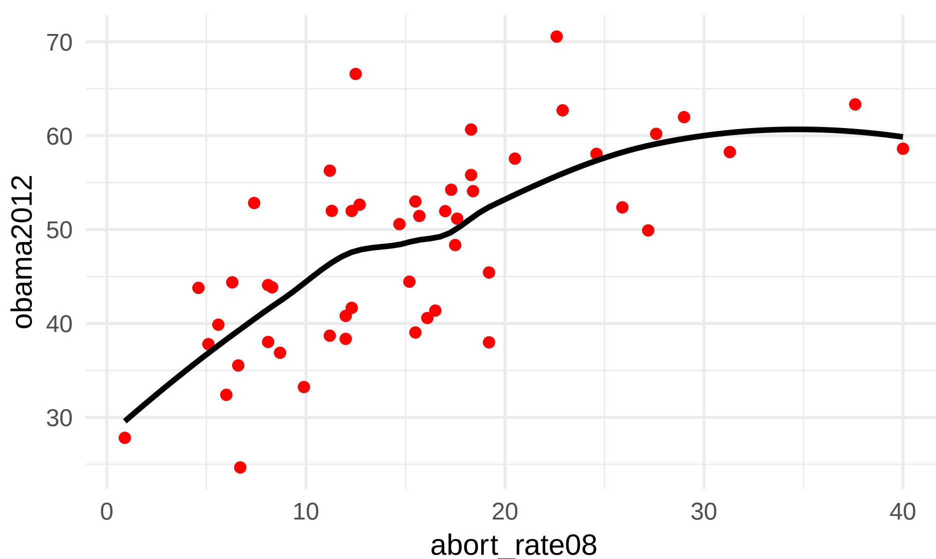
```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(position = "jitter") +
  geom_smooth(se=FALSE) +
  theme_minimal()
```



6.4.2 Colours

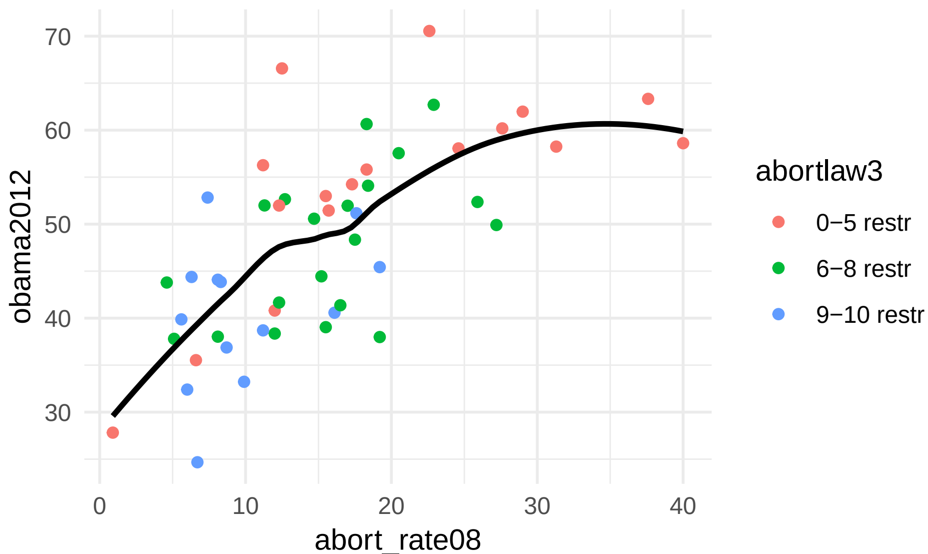
If we want to change the colours of the points in our plot, we can add the `colour=""` option to our geometric objects. In the example below we change the colour of our points from black to red and the colour of the line to black.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(colour="red") +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal()
```



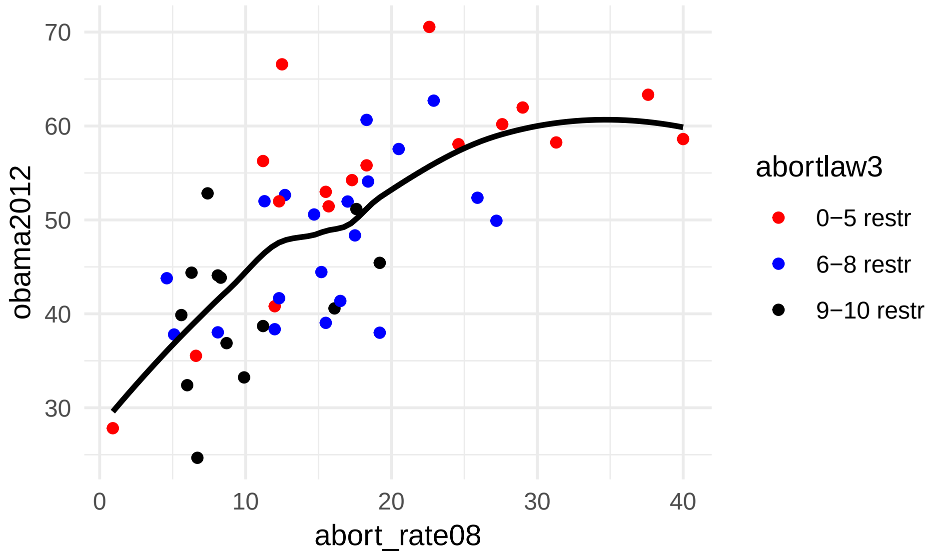
If we want to give points a value based on the value of a specific variable, we need to specificy this within `aes()`. When we add `colour=abortlaw3` to our `aes()`, we will see different colours for states with different restrictions on abortion.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3)) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal()
```



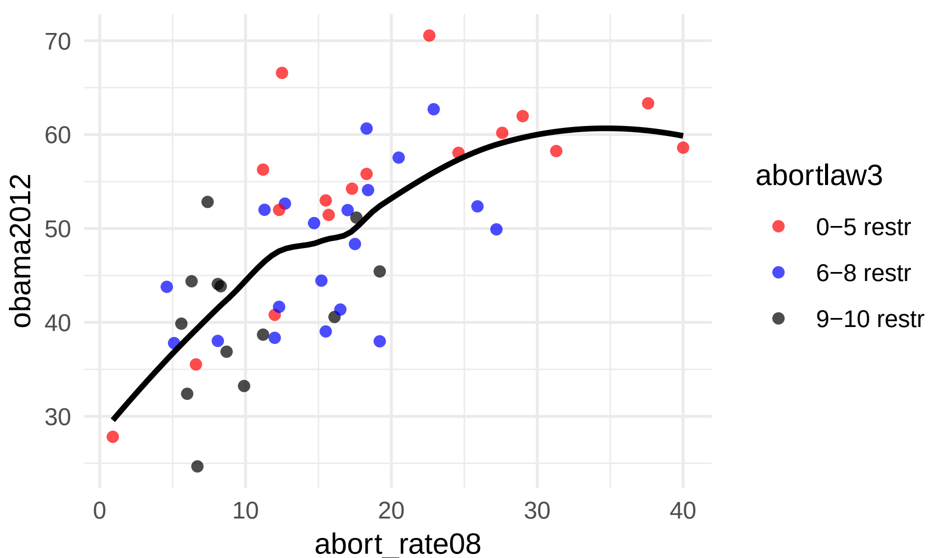
If we want to change these colours, we can use `scale_colour_manual()`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3)) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black"))
```



The colours are very bright. If we want to make them less so we can add `alpha` to `geom_point()` to add transparency to the points. Below we use an alpha of 0.7 (if we want more transparency we can use a lower alpha level).

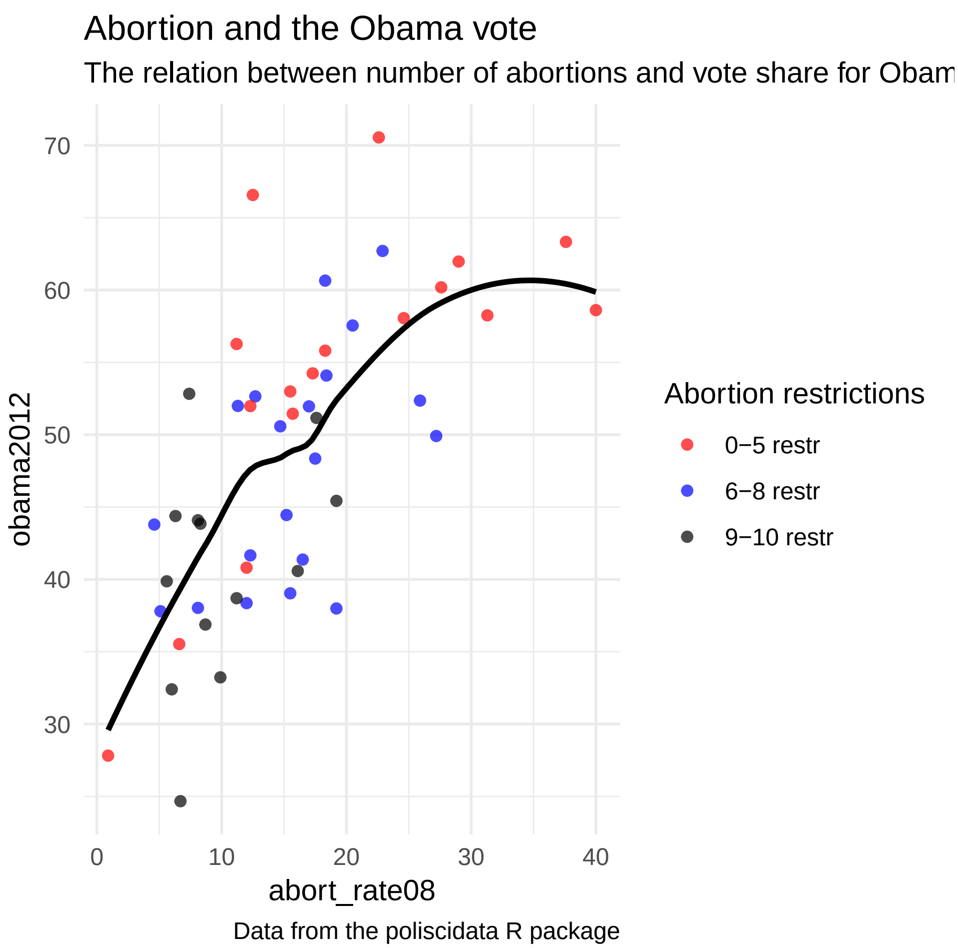
```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black"))
```



6.4.3 Labels

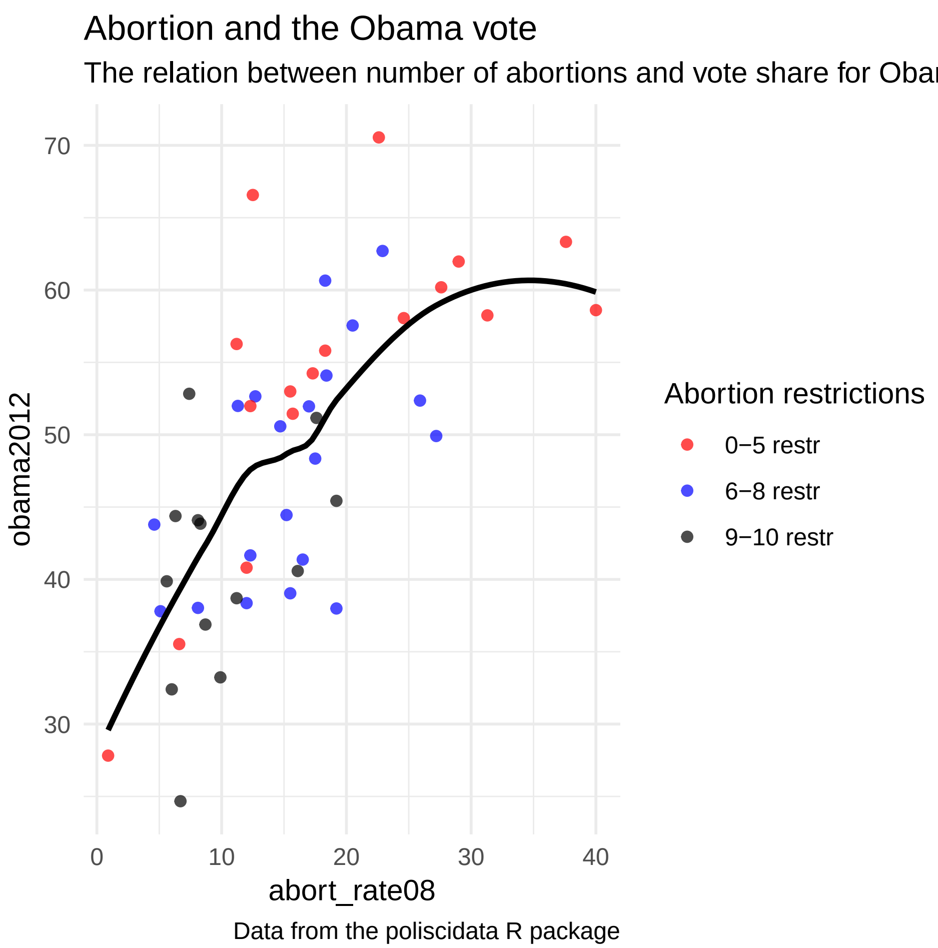
Make sure that your figure have labels that helps the reader understand what is going on. To do this, you can add `labs()` to your figure. Here we will add a title, subtitle and caption.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black")) +
  labs(
    title = "Abortion and the Obama vote",
    subtitle = "The relation between number of abortions and vote share for Obama",
    caption = "Data from the poliscidata R package",
    colour = "Abortion restrictions"
  )
```



Last, we can see that the legend title is `abortlaw3`. We can change this by adding `colour` to `labs()` as well.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black")) +
  labs(
    title = "Abortion and the Obama vote",
    subtitle = "The relation between number of abortions and vote share for Obama",
    caption = "Data from the poliscidata R package",
    colour = "Abortion restrictions"
  )
```

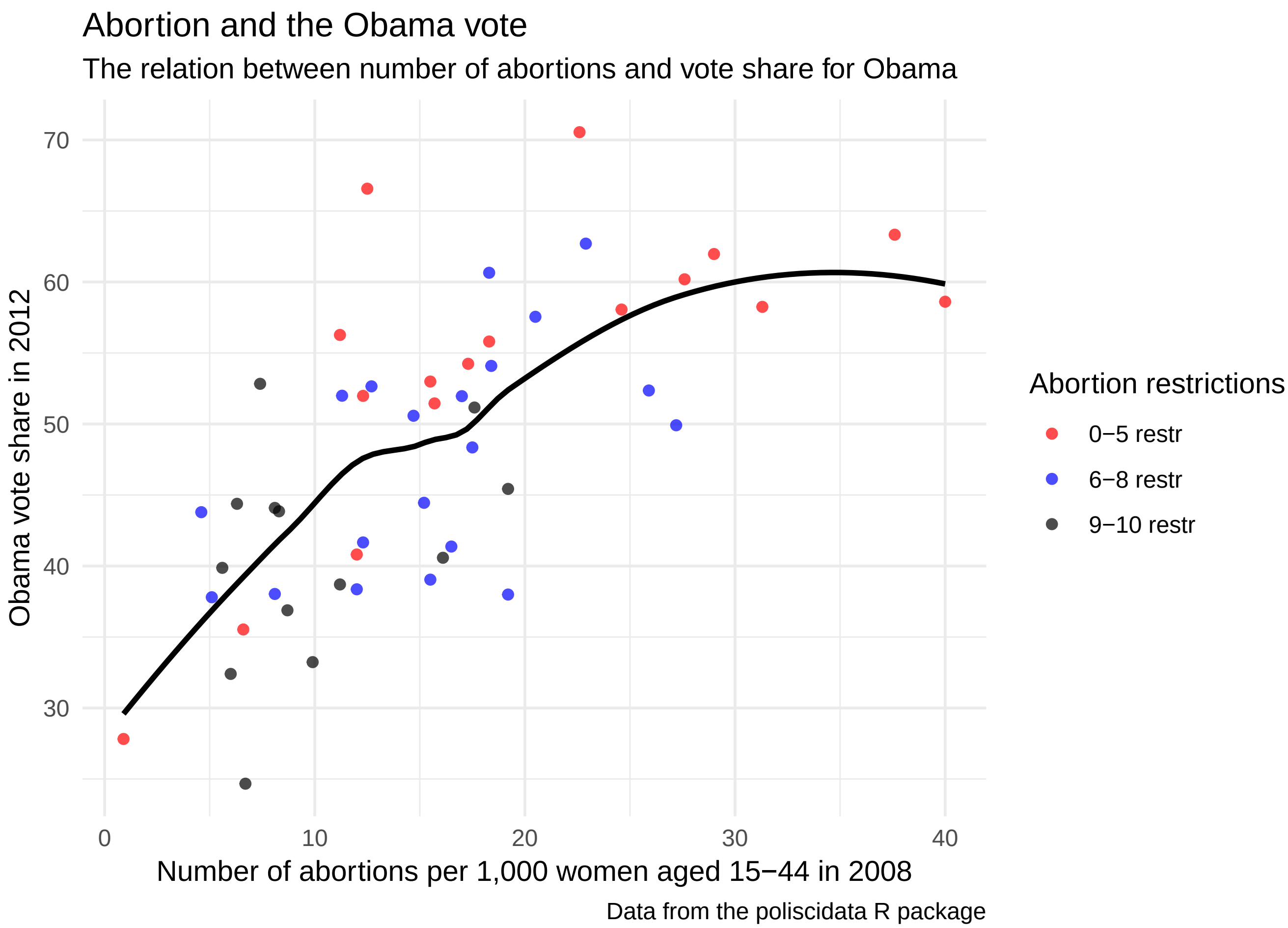


6.4.4 Axes

Related to labels are the axes. Always label the axes so they have meaningful names. The variable name is not a meaningful name. We add `x` and `y` to the `labs()` addition in our plot.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(se=FALSE, colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black")) +
  labs(
    title = "Abortion and the Obama vote",
    subtitle = "The relation between number of abortions and vote share for Obama",
    caption = "Data from the poliscidata R package",
```

```
    colour = "Abortion restrictions",
    y = "Obama vote share in 2012",
    x = "Number of abortions per 1,000 women aged 15-44 in 2008"
  )
```

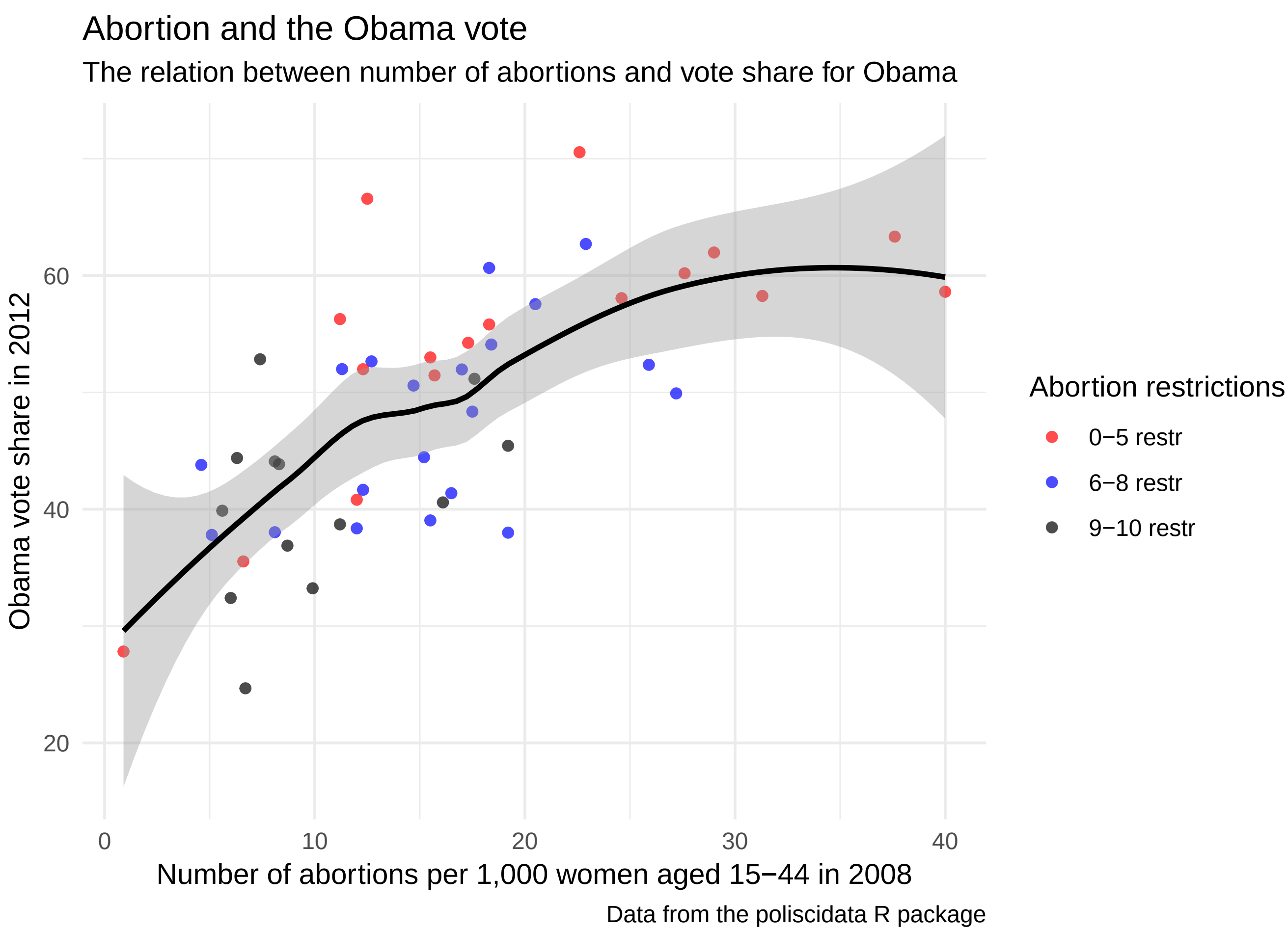


6.4.5 Confidence intervals

We can have confidence intervals in our figure by not having `se` (standard errors) set to `FALSE`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black")) +
```

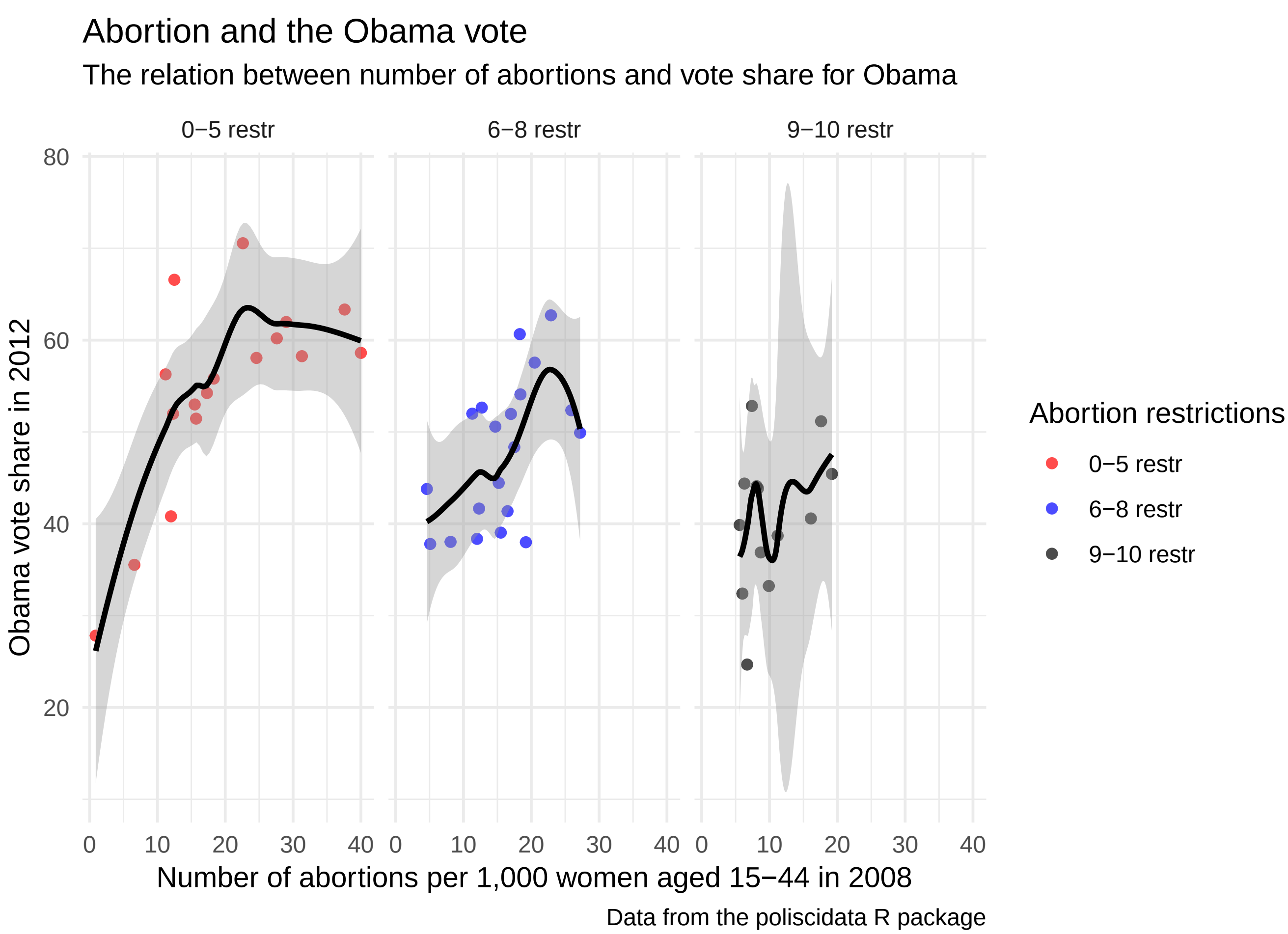
```
  labs(
    title = "Abortion and the Obama vote",
    subtitle = "The relation between number of abortions and vote share for Obama",
    caption = "Data from the poliscidata R package",
    colour = "Abortion restrictions",
    y = "Obama vote share in 2012",
    x = "Number of abortions per 1,000 women aged 15-44 in 2008"
  )
```



6.4.6 Making multiple plots in one

If we would prefer to have the plots for different observations, we can specify that with `facet_grid()`.

```
ggplot(states, aes(x=abort_rate08, y=obama2012)) +
  geom_point(aes(colour=abortlaw3), alpha=0.7) +
  geom_smooth(colour="black") +
  theme_minimal() +
  scale_colour_manual(values = c("red", "blue", "black")) +
  labs(
    title = "Abortion and the Obama vote",
    subtitle = "The relation between number of abortions and vote share for Obama",
    caption = "Data from the poliscidata R package",
    colour = "Abortion restrictions",
    y = "Obama vote share in 2012",
    x = "Number of abortions per 1,000 women aged 15-44 in 2008"
  ) +
  facet_grid(~ abortlaw3)
```



6.5 Saving plots

When you have a plot you would like to save, you can use `ggsave()`. Do keep in mind that it will only save the last plot you have created.

```
ggsave("fig1-abortion.png")
```

The figure will be saved in your working directory. The file type `.png` can be replaced to whatever format you would prefer your figure to be in. If you have saved your figure in an object, you can save it by specifying this before the file name.

```
ggsave(fig1, "fig1-abortion.png")
```

Often you will see that you are not totally satisfied with the size of your figure. To change this, you can use `width` and `height`.

```
ggsave(fig1, "fig1-abortion.png", width = 4, height = 4)
```

(PART) Regression

Chapter 7

OLS regression

To provide a simple example of how to conduct an OLS regression, we will use the same data as in the visualisation chapter, i.e. the `states` data frame from the package `poliscidata`.

```
library("poliscidata")

states <- states
```

7.1 Bivariate linear regression

To conduct a bivariate linear regression, we use the `lm()` function (short for linear models). We need to specify the dependent variable, independent variable and the data frame. Below we specify `obama2012` as the dependent variable and `abort_rate08` as the independent variable. Notice that we use the ~ symbol to separate the dependent variable from the independent variable. We save the output in the object `reg_obama`.

```
reg_obama <- lm(obama2012 ~ abort_rate08, data = states)
```

If we type `reg_obama`, we can see the intercept and coefficient in the model.

```
reg_obama
```

```

Call:
lm(formula = obama2012 ~ abort_rate08, data = states)
```

```
Coefficients:
 (Intercept)  abort_rate08  
     35.2589        0.8257  
```

Here we see that the intercept is 35.26, which is the predicted vote share for Obama in 2012 when we extrapolate to a state with an abortion rate of 0. The coefficient is 0.83, which is the increase in the vote share for Obama when there is an one-unit increase in the abortion rate.

However, this is not enough information. We need, for example, also information on the standard errors as well as model statistics. To get this, we use the function `summary()` on our object.

```r
summary(reg_obama)
```

```

Call:
lm(formula = obama2012 ~ abort_rate08, data = states)

Residuals:
     Min       1Q   Median       3Q      Max 
-16.1208  -5.6516   0.6785   4.7242  20.9904 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)   35.2589     2.2970  15.350  < 2e-16 ***
abort_rate08   0.8257     0.1297   6.366 6.91e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.654 on 48 degrees of freedom
Multiple R-squared:  0.4578,	Adjusted R-squared:  0.4465 
F-statistic: 40.52 on 1 and 48 DF,  p-value: 6.912e-08
```

Here we can see that the estimate for `abort_rate08` is statistically significant. We can further see that the R-squared is 0.46 which indicates that 46% of the variation in the vote share is explained by our independent variable.

To convert the results from our analysis into a data frame, we can use the package `broom` (Robinson, 2018).

```
library("broom")
```

As a first example, we can save the estimates and test statistics in a data frame by using the function `tidy()`. We save the output in a new object `reg_obama_tidy` and show this output as well.

```
reg_obama_tidy <- tidy(reg_obama)

reg_obama_tidy
```

```
          term  estimate std.error statistic      p.value
1  (Intercept) 35.258869 2.2970329 15.349745 3.818291e-20
2 abort_rate08  0.825655 0.1297048  6.365649 6.911913e-08
```

If we would also like to have the confidence intervals, we can add the `conf.int = TRUE`.

```
reg_obama_tidy <- tidy(reg_obama, conf.int = TRUE)

reg_obama_tidy
```

```
          term  estimate std.error statistic      p.value   conf.low
1  (Intercept) 35.258869 2.2970329 15.349745 3.818291e-20 30.6403753
2 abort_rate08  0.825655 0.1297048  6.365649 6.911913e-08  0.5648661
  conf.high
1 39.877364
2  1.086444
```

This is useful if you would like to visualise the results. However, often we also want to save predictions and residuals based on our model. To do this, we can use the function `augment()`. Below we save the output in the object `reg_obama_aug`.

```
reg_obama_aug <- augment(reg_obama)
```

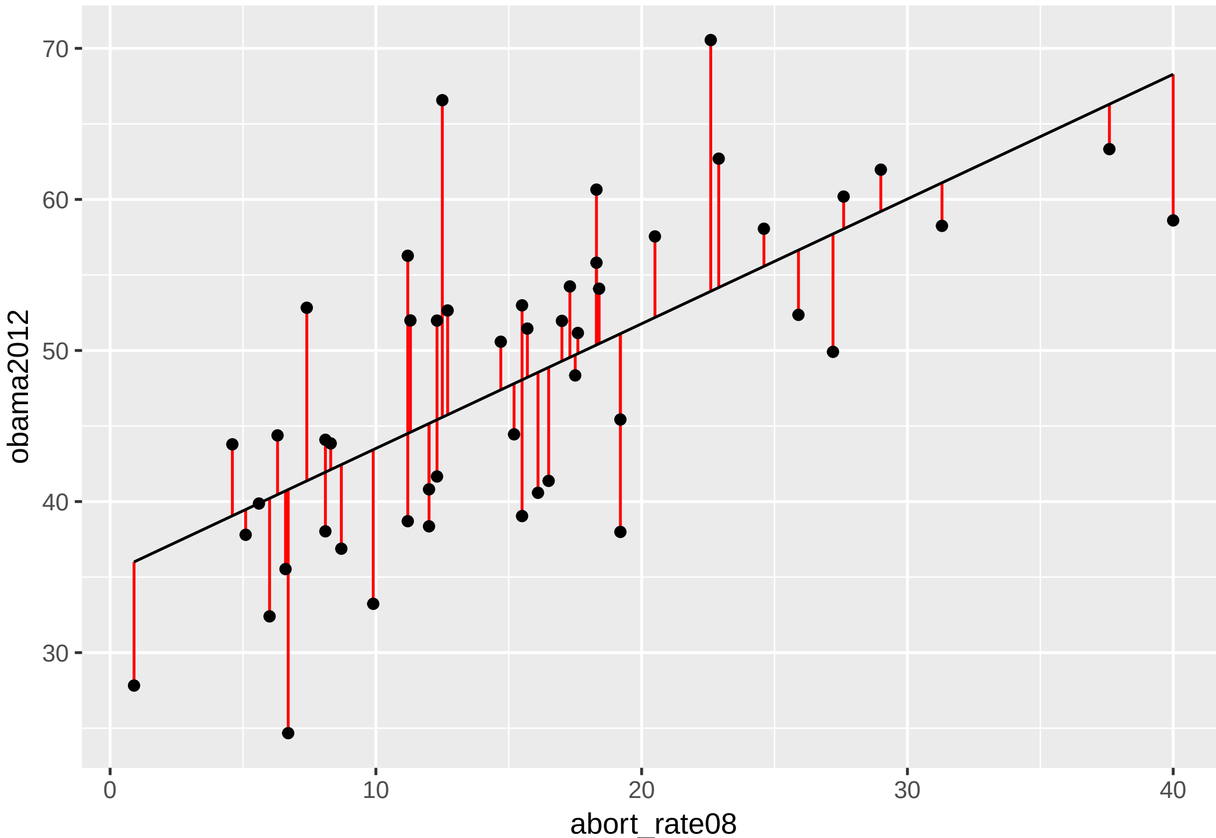
To see the data in the new object, use `head()`. Here you see that there is a variable called `.fitted`. This variable is the predicted value for each observation.

```r
head(reg_obama_aug)
```

```
  obama2012 abort_rate08  .fitted  .se.fit    .resid        .hat   .sigma
1     40.81         12.0 45.16673 1.179911 -4.356729 0.02376297 7.708398
2     38.36         12.0 45.16673 1.179911 -6.806729 0.02376297 7.669635
3     36.88          8.7 42.44207 1.406179 -5.562068 0.03375074 7.691025
4     44.45         15.2 47.80882 1.083835 -3.358825 0.02005065 7.719335
5     60.19         27.6 58.04695 1.893732  2.143053 0.06121236 7.728453
6     51.45         15.7 48.22165 1.082515  3.228348 0.02000184 7.720544
      .cooksd .std.resid
1 0.004039088 -0.5760815
2 0.009859143 -0.9000401
3 0.009544419 -0.7392523
4 0.002010333 -0.4432886
5 0.002722341  0.2889684
6 0.001852473  0.4260579
```

We can use this data frame to visualise the residuals (with the colour red below).

```r
ggplot(reg_obama_aug, aes(x=abort_rate08, y=obama2012)) +
  geom_segment(aes(xend=abort_rate08, y=obama2012, yend=.fitted),
        colour="red") +
  geom_point() +
  geom_line(aes(x=abort_rate08, y=.fitted))
```



7.2 Multiple linear regression

To conduct a multiple linear regression, we simply need to add an extra variable to our model. Accordingly, the only difference between the example above and the example here is the addition of a new variable. Here, we want to examine whether the effect of `abort_rate08` holds when we control for population density (`density`). Notice that we add a `+` before adding the variable to the list of variables.

```
reg_obama_full <- lm(obama2012 ~ abort_rate08 + density, data = states)
```

We use the `summary()` function to get the output of the model.

```
summary(reg_obama_full)
```

```
Call:
lm(formula = obama2012 ~ abort_rate08 + density, data = states)
```

```
Residuals:
     Min       1Q   Median       3Q      Max 
-16.1719  -5.5567  -0.2101   4.3195  21.5132 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept)  36.019160   2.328169  15.471  < 2e-16 ***
abort_rate08  0.681420   0.161482   4.220 0.000111 ***
density       0.007656   0.005214   1.468 0.148669    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.564 on 47 degrees of freedom
Multiple R-squared:  0.4815,	Adjusted R-squared:  0.4595 
F-statistic: 21.83 on 2 and 47 DF,  p-value: 1.976e-07
```

In the output we see that the coefficient for `abort_rate08` is slightly smaller compared to the bivariate model but still statistically significant. Again we can use the `tidy()` function to get a data frame with the results.

```
reg_obama_full_tidy <- tidy(reg_obama_full)
```

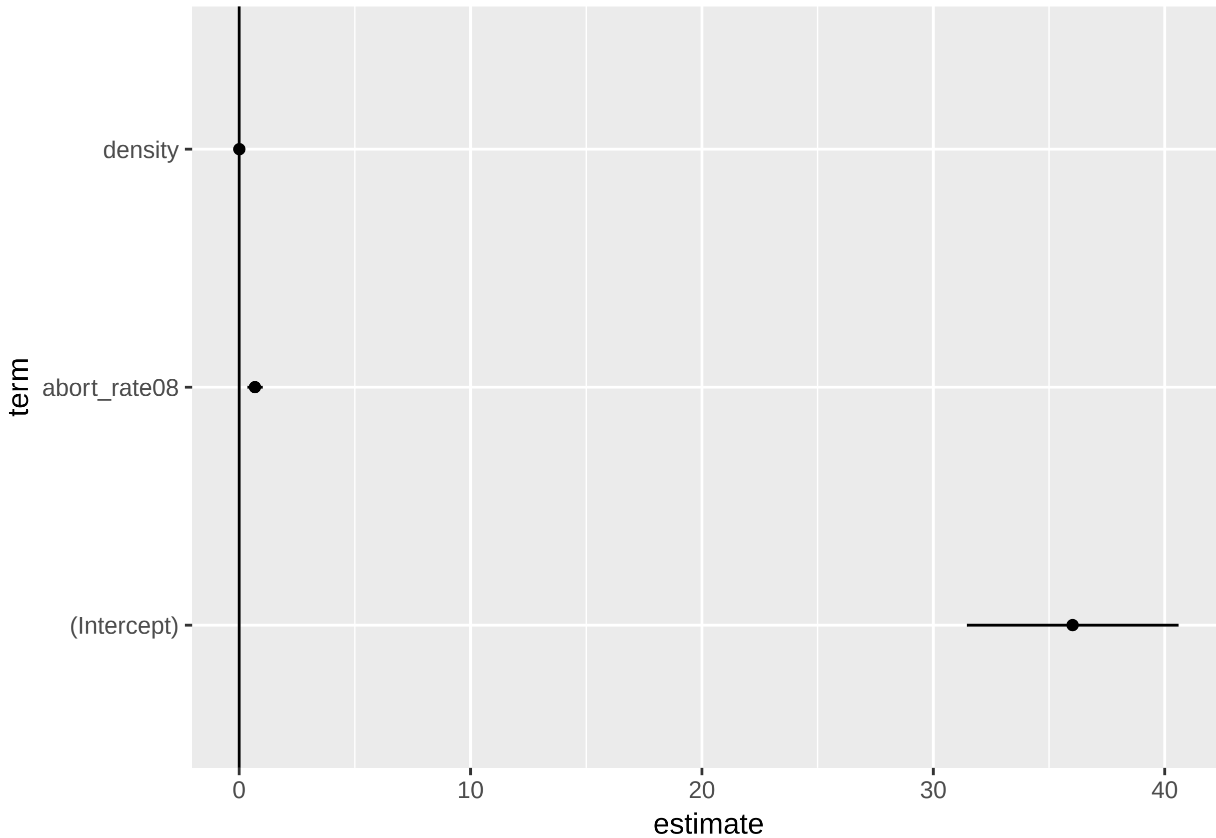
We further calculate the 95% confidence intervals for the estimates.

```
reg_obama_full_tidy <- reg_obama_full_tidy %>%
  mutate(
    ci_low = estimate - 1.96 * std.error,
    ci_high = estimate + 1.96 * std.error
  )
```

We can then visualise the results.

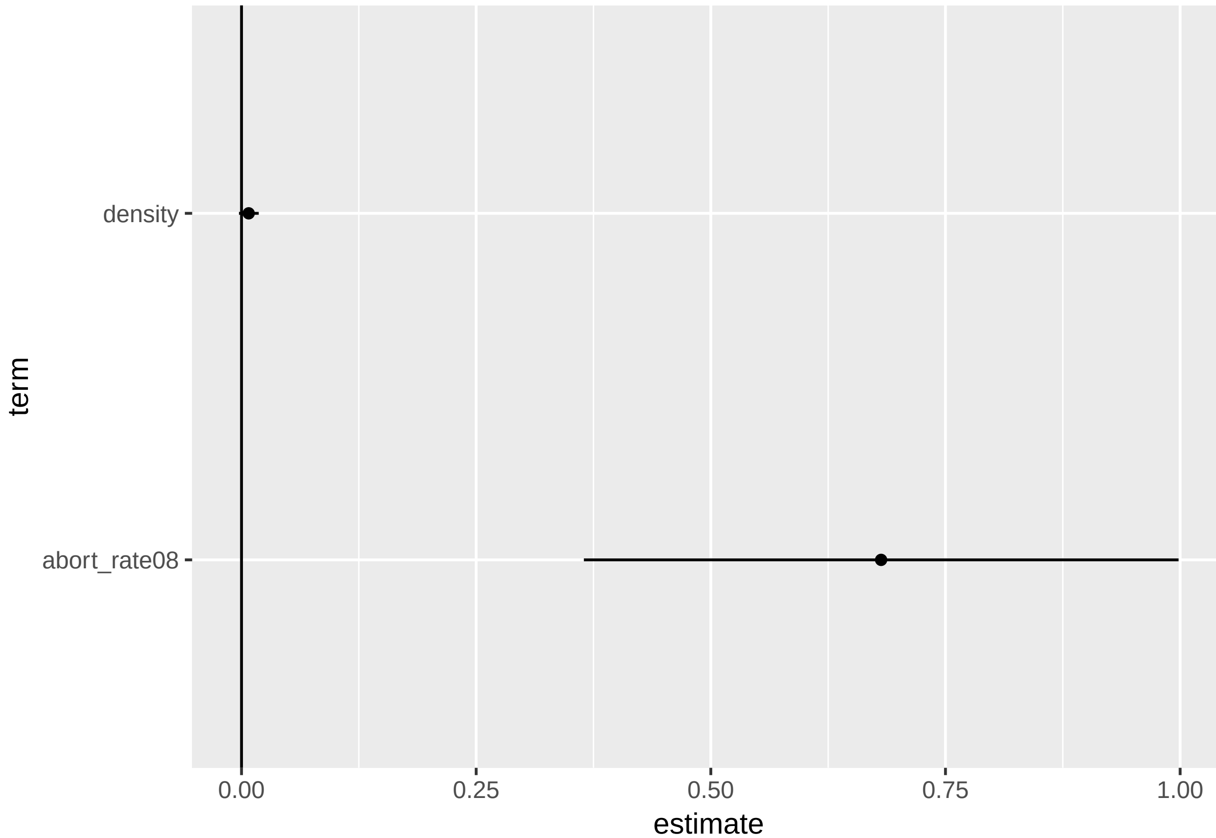
```
ggplot(reg_obama_full_tidy, aes(estimate, term, xmin = ci_low,
                                xmax = ci_high, height = 0)) +
     geom_point() +
```

```
    geom_vline(xintercept = 0) +
    geom_errorbarh()
```



In some cases the intercept is not relevant. In the code below, we use the `filter()` function to visualise all effects except for the intercept.

```
reg_obama_full_tidy %>%
  filter(term != "(Intercept)") %>%
  ggplot(aes(estimate, term, xmin = ci_low,
             xmax = ci_high, height = 0)) +
    geom_point() +
    geom_vline(xintercept = 0) +
    geom_errorbarh()
```



7.3 Diagnostic tests

To get diagnostic plots, we will use the `fortify()` function from `ggplot2`. This allows us to get the following variables realted to model fit statistics:

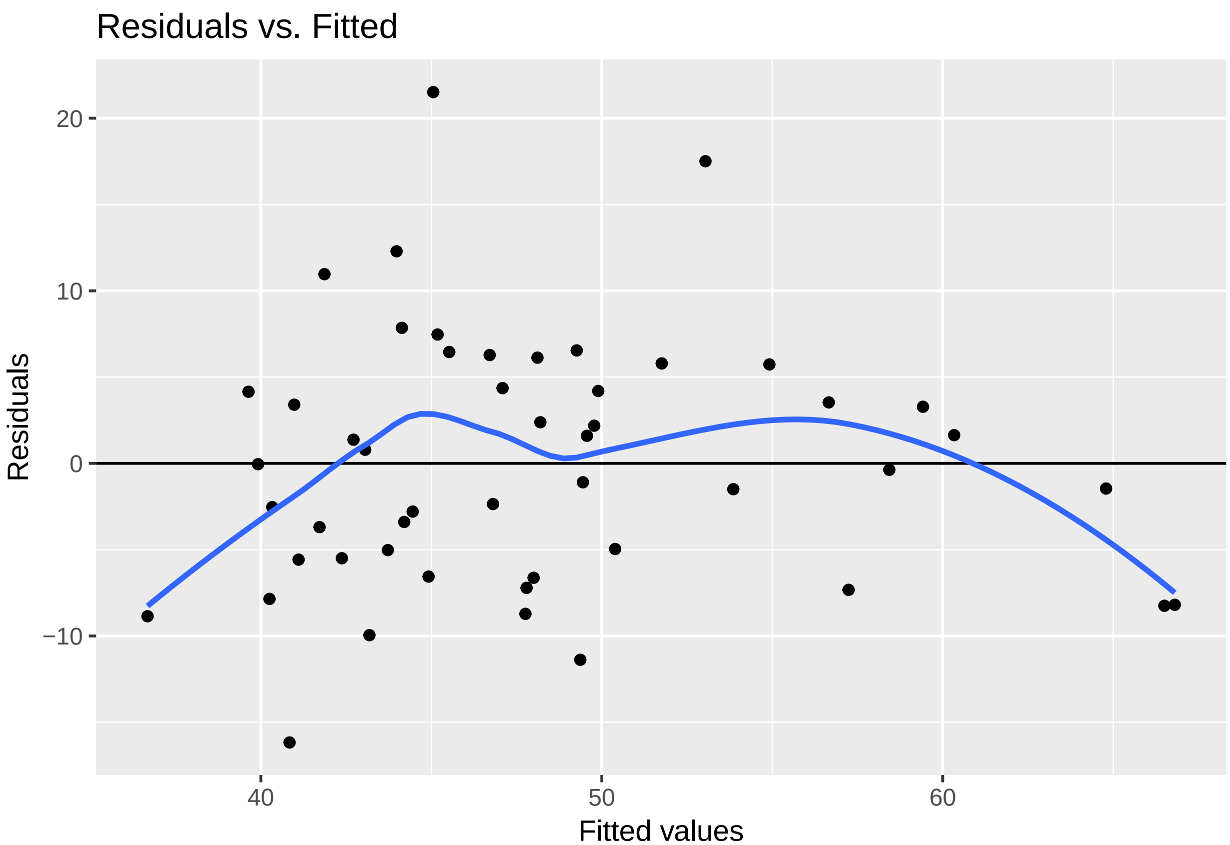
1. `.hat`: Diagonal of the hat matrix
2. `.sigma`: Estimate of residual standard deviation when corresponding observation is dropped from model
3. `.cooksd`: Cooks distance, using `cooks.distance()`
4. `.fitted`: Fitted values of model
5. `.resid`: Residuals
6. `.stdresid`: Standardised residuals

First, we use `fortify()` on our linear model:

```
reg_fortify <- fortify(reg_obama_full)
```

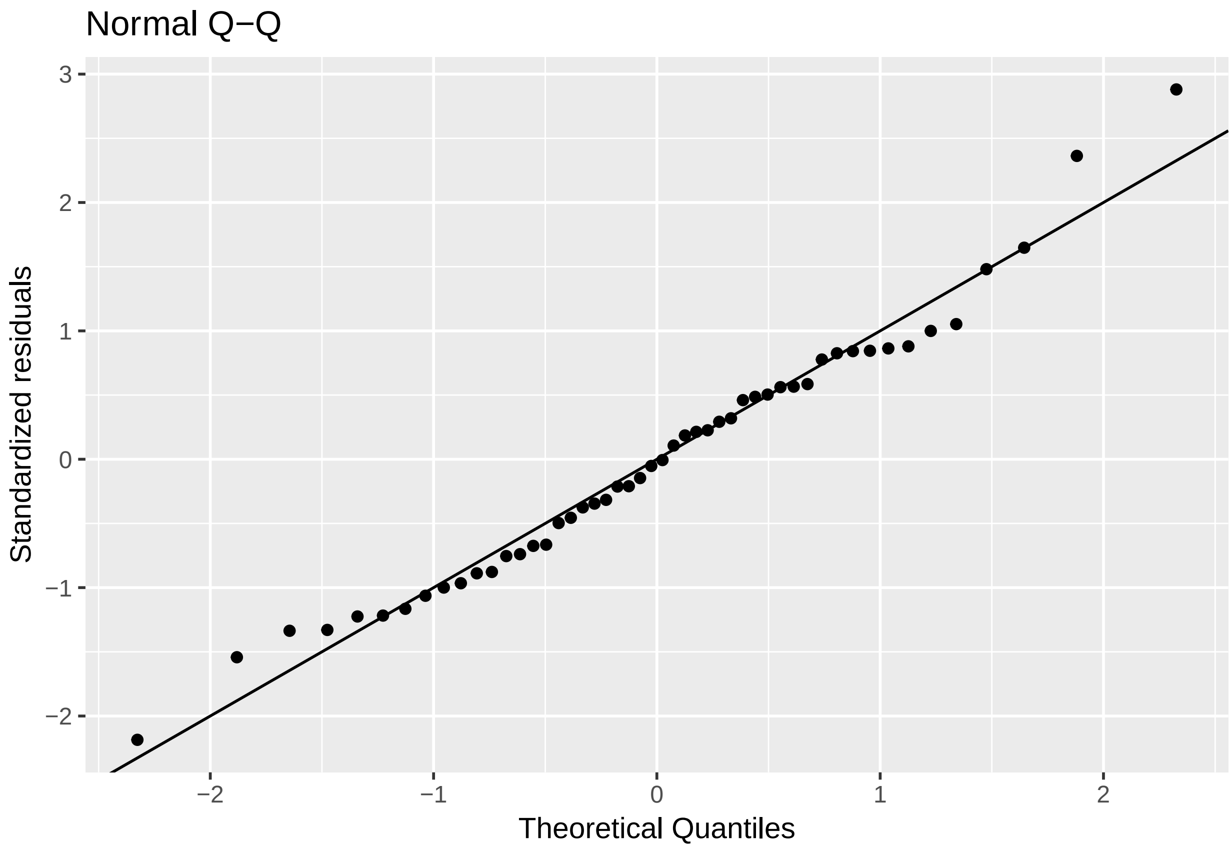
To see how our residuals are in relation to our fitted values, we can plot `.fitted` and `.resid`.

```
ggplot(reg_fortify, aes(x = .fitted, y = .resid)) +
  geom_point() +
  geom_hline(yintercept = 0) +
  geom_smooth(se = FALSE) +
  labs(title = "Residuals vs. Fitted",
       y = "Residuals",
       x = "Fitted values")
```



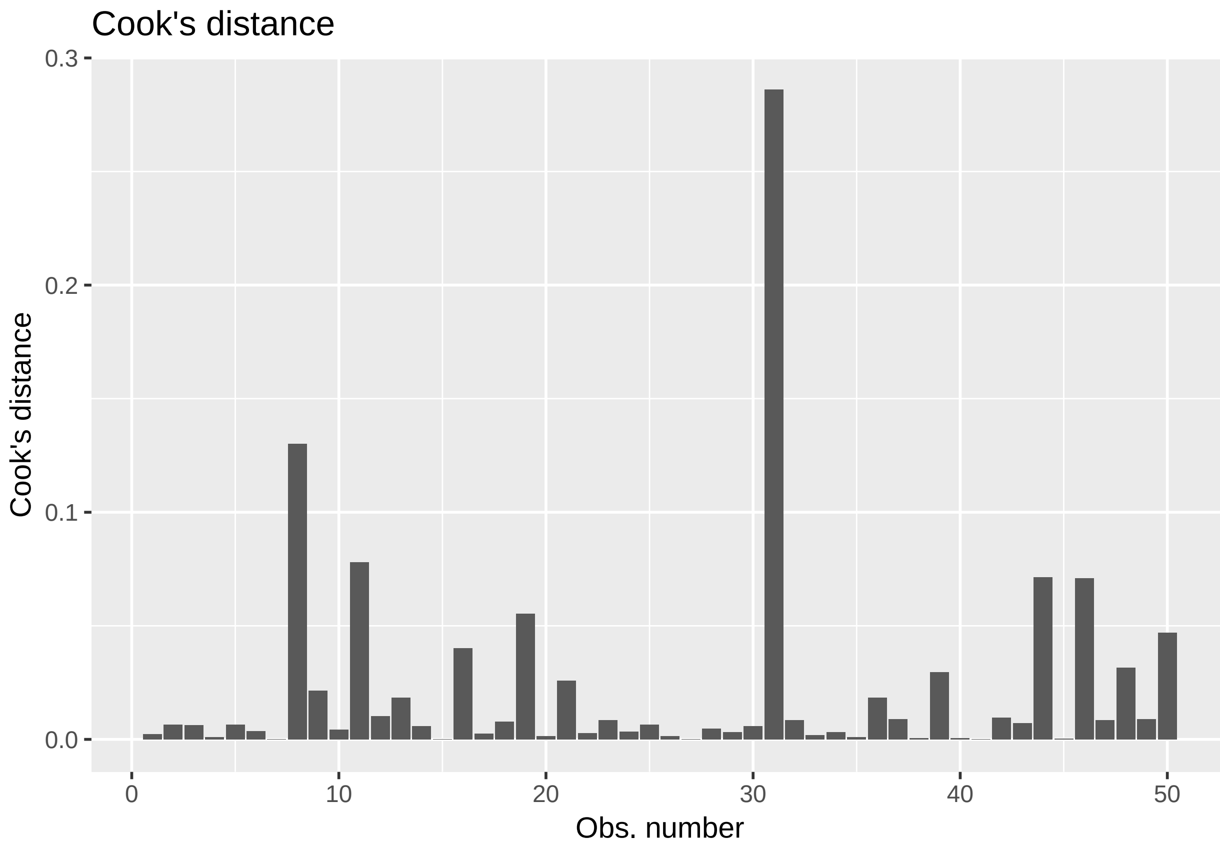
To see whether our residuals are normally distributed, we create a normal Q-Q plot with the standardized residuals.

```
ggplot(reg_fortify) +
  stat_qq(aes(sample = .stdresid)) +
  geom_abline() +
  labs(title = "Normal Q-Q",
       y = "Standardized residuals",
       x = "Theoretical Quantiles")
```



To estimate the influence of individual observations, we plot the Cook's distance for each state.

```
ggplot(reg_fortify, aes(x = seq_along(.cooksd), y = .cooksd)) +
  geom_col()  +
  labs(title = "Cook's distance",
       y = "Cook's distance",
       x = "Obs. number")
```



7.4 Setting up regression tables

To export regression tables from `R`, we are going to use the package `stargazer` (remember to install the package if you haven't already done so).

```
library("stargazer")
```

First, we use the `stargazer()` function to show the output from the object `reg_obama`. Notice that we also add the option `type = "text"`. If we do not do that, we will get the output as LaTeX code.

```
stargazer(reg_obama, type = "text")
```

```
=================================================
                         Dependent variable:
                    -----------------------------
```

```
                             obama2012
-----------------------------------------------
abort_rate08                 0.826***
                              (0.130)


Constant                     35.259***
                              (2.297)


-----------------------------------------------
Observations                    50
R2                             0.458
Adjusted R2                    0.446
Residual Std. Error       7.654 (df = 48)
F Statistic           40.521*** (df = 1; 48)
===============================================
Note:               *p<0.1; **p<0.05; ***p<0.01
```

This shows the output from one regression model. To add more regression models to the table, simply add a comma and the name of the object with the model. Below we use the same code as above and add the model with control variables included, `reg_obama_full`.

```r
stargazer(reg_obama, reg_obama_full, type = "text")
```

```
=================================================================
                                 Dependent variable:
                    ---------------------------------------------
                                      obama2012
                           (1)                     (2)
-----------------------------------------------------------------
abort_rate08             0.826***                0.681***
                         (0.130)                 (0.161)


density                                           0.008
                                                 (0.005)
```

```
Constant                   35.259***              36.019***
                            (2.297)                (2.328)

-----------------------------------------------------------------
Observations                  50                     50
R2                           0.458                  0.482
Adjusted R2                  0.446                  0.459
Residual Std. Error     7.654 (df = 48)        7.564 (df = 47)
F Statistic         40.521*** (df = 1; 48) 21.827*** (df = 2; 47)
=================================================================
Note:                                  *p<0.1; **p<0.05; ***p<0.01
```

7.4.1 Exporting the regression table

To export the regression table, we use the option `out` to specify, where we want to save our regresion table. Below we save the table in the file `tab-regression.htm`.

```
stargazer(reg_obama, reg_obama_full, type = "text",
          out="tab-regression.htm")
```

An `.htm` file is a HTML file you can open in your browser (e.g. Google Chrome). To get it into Word, simply open the file via Word. You might have to do some extra changes before it is ready for a broader audience. Always try to make your tables look like tables in published articles and books.

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

Healy, K., & Moody, J. (2014). Data visualization in sociology. *Annual Review of Sociology*, *40*, 105–128.

Kearney, M. W. (2018). *Rtweet: Collecting twitter data*. Retrieved from https://cran.r-project.org/package=rtweet

Larsen, E. G. (2018). Welfare retrenchments and government support: Evidence from a natural experiment. *European Sociological Review*, *34*(1), 40–51.

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

Robinson, D. (2018). *Broom: Convert statistical analysis objects into tidy data frames*. Retrieved from https://CRAN.R-project.org/package=broom

Silge, J., & Robinson, D. (2016). Tidytext: Text mining and analysis using tidy data principles in r. *JOSS*, *1*(3). http://doi.org/10.21105/joss.00037

Tufte, E. R. (1983). *The visual display of quantitative information*. Graphics Press.

Wickham, H. (2009). *Ggplot2: Elegant graphics for data analysis*. Springer-Verlag New York. Retrieved from http://ggplot2.org

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