Research Tools for Social Scientists

Thomas de Graaff

October 17, 2017

Contents

1 Introduction **2**
1.1 Preface . . . 2
1.2 Where to get R . . . 2
1.3 Structure of the book . . . 2

2 Why bother? **2**
2.1 Why do we need all this? . . . 3
2.2 Why use R and not other applications? . . . 3
2.3 R Markdown: a wonderful life in plain text . . . 4
2.4 Regression analysis: not again! . . . 4

3 Basic R Usage **5**
3.1 For absolute beginners . . . 5
3.2 How to use it . . . 5
3.3 Reading and writing data . . . 7
3.4 Ehmmm, dataframes . . . 7
3.5 Functions . . . 9
3.6 Regression modeling . . . 10
3.7 Making plots . . . 13
3.8 Recap . . . 16

4 Digression: Linear Regression and how to apply it **16**
4.1 Theoretical background . . . 17
4.2 Applications of linear regression . . . 20

5 Network analysis with R **26**
5.1 Introduction . . . 26
5.2 Creating networks . . . 26
5.3 Network characteristics . . . 34

6 `{r} # edge_density(g_florence$PADGB) #` **34**

7 **34**

8 Which give the proportion of the present edges from all possible edges in the network. **34**

9 **34**

10 The diameter of a network can be retrieved as: **34**

11 **34**

12 `{r} # diameter(g_florence$PADGB) #` **34**

13 `{r} # deg <- degree(g_power_law) # deg #` **34**

14 **34**

15 which is not that insightful, but a histogram is: **34**

16 **34**

17 `{r, message=FALSE} # hist(deg) #` **34**

18 `{r} # distances(g_florence$PADGB) #` **35**

19 In conclusion **35**

1 Introduction

1.1 Preface

I started this online book as background material for the course Network Analysis and from the need to teach social science students (which includes Business Administration and Economics students) the a basic understanding of `R`, network techniques and the intuition behing applying linear regression. I intend to work on this book on the fly meaning that during courses I try to see what the needs of students really are. Moreover, over the coming years I intend to add additional chapters, specifically about the use of stated preference modeling–and its corresponding logit estimation–and working with geographical data. Obviously, all in `R`. At the moment this book still has not yet passed its infant stage. Therefore, all input in the form of additions, comments, critique and remarks are *highly* appreciated.

1.2 Where to get `R`

First, you need to install `R` itself. You can do this by downloading this from CRAN (we choose here the server from the Netherlands). Choose your appropriate operating system, choose the `base` system, download `R` and install it. That's it!

The base distribution of `R` comes with a built-in editor, where you can write your script (more about scripts in subsection 3.2.1). This editor is however *very* basic. Therefore, it is very much advised that you download and install the free editor `RStudio` as well. Again, choose your operating system and just install the latest version. The very short video (1.5 minutes) on RStudio's website gives an overview of the *basic* features (it can do quite some more stuff).

1.3 Structure of the book

In its present state, the book contains four chapters dealing with contents. The first chapter gives a justification for all this. The second chapter concerns the basic usage of `R`. The third deals with the fundamentals behind regression analysis and how to apply it in `R`. The fourth chapter lays out how to do network analysis in `R`.

2 Why bother?

This chapter deals with the justification for this book and the rationale behind using the `R` software environment versus other software applications. The first section deals with understanding the various criteria I (and others) maintain for software for proper research workflows. Thereafter, I deal with `R` and subsequently with using RMarkdown files (files with the `.rmd` extension). Finally, I say something about the need to revisit regression analysis again.

2.1 Why do we need all this?

The first question that arises, or at least should arise, is why bother? Why learn students in the social sciences relatively new and specific software tools for education and research. 'Old' tools[1] such as `Word`, `Excel` and `STATA` do just as fine right? I would say that it depends, but that these tools are severely limited. If you are only interested in straightforward regression and anova techniques and collect your sample by survey research, then `Stata` and to a much lesser extent `Excel`(or even `SPSS`, the horror…) would definitely suffice. But if you would like to do more fancy and cool stuff, including creating beautiful diagrams, nice maps, simulation analysis, network analysis, and even whole books in `html` or `pdf` then you need more elaborate tools.[2]

And note that, even though most of the best tools out there[3] are open source (so they are **free**, as in **free beer**), they will cost you something dearly: namely, your time. The learning curve of these tools are usually quite steep (which also means they will pay-off quickly). So, choose your battles carefully. The best tools, I would argue, have the following set of characteristics:

- They are open source. The most important argument to use an open source package is reproducibility. Your work is simply less accessible and thus reproducible if the code can only be run with applications that costs over 1,000 Euros.
- The learning curve is reasonable. First, and foremost, students should be able to use the package for straightforward research. If that is not possible after one six-week's course, the software package is not particularly suitable for social science students
- They are scriptable. A software package should be scriptable, both internally as externally. Internally scriptable indicates that within the package scripts or programs can be written so that every step within the workflow can be reproduced. With externally scriptable I mean that the software package should also be used in combination with other software packages or languages, such as LaTeX, markdown, make, html, sql, C++, etc.
- There is a large community that uses these tools. Nobody wants to be locked in with obsolete technology. A large userbase ensures a high probability that the software package will be used and maintained in the future as well. Moreover, all sorts of indirect effects, such as user written routines, packages and documentation, come along for free with a large community.
- Flexibility: Ideally, a software approach should be both extendable and scalable. The former ensures that slight deviations from standard approaches can relatively easy be implemented. The latter is important when the size of the database increases, as typically is the case with recent improvements in remote sensing techniques.

One of the tools that meets all these requirements, and more, is `R` and the next section lays out why "even" social scientists should learn `R`.

2.2 Why use `R` and not other applications?

Ask any data scientist at the moment for the software tools most used and they will most likely answer `R` or `Python`. Of course, that should not be a valid answer (many people use `Word` as well and nobody would argue that `Word` is brilliantly programmed or designed), but it indicates the popularity (and the community) that uses `R`.

Where 10 years ago most social scientists still used `SPSS` (and the economists `Stata`), that has now changed completely (well, the economists still use `Stata`, but the rest of the world moved on). And for good reasons, namely:

1. It is open source and thus free;
2. `R` is flexible and thus multi-purpose;

[1] Some of the tools I use are actually quite old, even so far as from the 1970s and they are still very good.
[2] Actually, this "book" is actually written by a combination of `R` and `RStudio`.
[3] Including `Markdown`, `LaTeX`, `Python`, `Git` and `GitHub`.

3. there is now a **very** large userbase; everything you can dream of (that is in the context of data science/management), somebody else most likely already programmed;[4]
4. it generates beautiful pictures, diagram, maps, and histograms (even 3D pie diagrams for the masochists amongst us);
5. relative to `Stata` or `Excel` it is fast, which is great for larger (spatial) databases.

In general, you can use `R` for statistical analysis, simulation analysis, data management, visual display of data, creating documents (and presentations), and even GIS applications. In that respect it is far more flexible than `Stata`. Last but perhaps not least, `R` is more and more used outside academia as well. Twitter, Facebook, Booking and Google use `R`, just as companies as the New York Times for creating interactive website diagrams.

Social science students might find working with `R` initially strange, cumbersome or even frustrating. All the lovely drop-menus that are still provided by `Excel` and `Stata` have disappeared, and the whole thing is completely script-driven. In fact, `R` is a full-blown program language. I am aware that this needs some adaptation. However, hopefully, learning `R` in combination with `RStudio` will pay-off; if not in becoming more efficient and reproducable, then at least in the fact that you start to understand a *different* way of doing things and that the office suite (Word, Excel and Powerpoint) is not the only option out there.

2.3 R Markdown: a wonderful life in plain text

`RStudio` does not only provide an editor for `R` but comes with all kinds of various other goodies installed. One of them is the ability to use R Markdown. R Markdown is a specific variant on the more generic Markdown markup language. Markup languages is not something that social science students are familiar with, but actually are very often used. Essentially, a markup language very specifically determines how pieces of text should look like, e.g., bold, italic, or being a header. One of the most famous markup languages is `html` and another well-known is LaTeX.

One of the problems with these markup languages is that they are complex, difficult, or just *annoying* to read and write in (just try for yourself with writing a page in `html`). That's why Markdown is invented as a lightweight markup language in 2004 by John Gruber. And with lightweight it is meant that there are only a few syntax operations (more or less only headers, bold, italics, lists, and hyperlinks). Very quickly it became a huge success (especially for web based applications). Now various well known applications (such as Wordpress, GitHub and others) make use of this format.

R Markdown is a variant on Markdown in combination with other tools (you do not have to know them, they are all under the hood of RStudio). And with R Markdown you can enable people > "to write using an easy-to-read, easy-to-write plain text format, and optionally convert it to structurally valid XHTML (or HTML)"

That means that with R Markdown you can easily write a piece of text and then convert it to html. However, you are not restricted to html! You can convert the same text to a pdf or open office format! This makes it very flexible. In fact, this book has been written in R Markdown as well. And you are not restricted to texts. You can create slides as well if you want. Even more bonkers: you can put R code in your text with the results! That means that in one document you can have your script as your report file, which is truly wonderful for reproducable research.

2.4 Regression analysis: not again!

I will also spend some time on the use of regression analysis and try to explain in my own words how to use it. In general, my experience is that many students have little or no experience in regression analysis. Moreover, and perhaps even worse, they have little or no intuition for regression analysis. This sort of sucks,

[4] CRAN packages give a great overview of all the official packages out there and the wide range of applications, and again they are all free!

the more because regression is the most often statistical technique used in the social sciences (heck, in all sciences).

In my perception, students are taught the principles of statistics (typically this involves lots of things as ANOVA), where in the last course, one or two hours is spent on regression analysis. This is fine, as long as it comes back in a course as *applied* statistics or econometrics (how and when to do this stuff?) or in other applied courses. But usually this is not the case. Most courses don't care about regression analysis and the first moment you have to use it again is when writing your thesis and at that time it is a bit too late to teach the applied stuff again.

So, yeah, therefore a bit of regression analysis from my perspective. But again, you only learn this by doing and under the guidance of various teachers (and in various cirumstances).

3 Basic R Usage

```
library("swirl")
library("rio")
library("ggplot2")
library("dplyr")
```

The R programming language and software environment for statistical computing is an implementation of the proprietary S programming language by Ross Ithaka and Robert Gentleman in 1992. It quickly gained in popularity (see, e.g, this Nature article from 2014 Programming tools: Adventures with R) and now has more than **official** 10,000 user contributed packages (see as well the blog piece On the growth of CRAN packages).

3.1 For absolute beginners

At the moment there are *many* tutorials, blogs, youtube clips, and background materials about using R on the internet. I therefore do not intend to write a complete handbook, but focus instead on what I need for my courses. Moreover, I do not intend to teach the very basic stuff. There is very good tutorial package out there called Swirl (see as well the website: http://swirlstats.com/students.html). I very much recommend using this package for absolute beginners. The way to do this is rather simple. First install the package by typing:

```
install.packages("swirl")
```

then start Swirl by first loading the package:

```
library("swirl")
```

And then call the function by typing:

```
swirl()
```

In the first menu choose `R Programming`. Now, there are 15 lessons. I find the first four the most useful (Basic Building Block, Workspace and Files, Sequences of Numbers, Vectors), but others are very useful as well to go through. The command `main()` by the way brings you always back to the main menu, and do not go for the credits on coursera (that is now a paid online course).

3.2 How to use it

R is truly a programming language in the sense that there is no graphical user interface (GUI) involved. You need to type your own commands. And for beginners this sort of sucks. It seems slower, you have no idea

which command to type in, and you frequently make many mistakes. However, when you start to use it more, the speed of getting things done goes up (sometimes exponentially), you have a better grasp on the basic commands, and the number of mistakes go down. And perhaps you even reach that wonderful sensation: you feel in control of things.

In contrast with programs such as `Excel` or `SPSS`, there are two big differences: (*i*) you use scripts and (*ii*) you make frequently use of packages which are essentially written by other `R` users.

3.2.1 Scripts

The use of scripts or program files is somewhat alien to most. Although programs such as `Stata` also makes use of scripts in the form of so-called `do` files. You start a new script by clicking on File > New Script (`R` editor) or File > New File > New Script (`RStudio`). You now have a new *empty* file (which you have to save from time to time). If you fill in this file with commands, you are actually programming. The **huge** benefit of this procedure is that you record what you have done and that you can easily change something without **retyping** the whole thing over again.

As an example, assume that somebody gives you a dataset with 2 variables and ask you to analyse this dataset. With `SPSS` you read this dataset in and then click on various butttons so you get some output. Now, assume further that this person actually has forgotten a variable (this happens more often than you think) and gives you a new dataset with 3 variables. Then you have to do all the clicking again (and hopefully you remembered on which buttons you actually clicked).

When you have a script file you only have to change the code in 1 or 2 places and run it all again. No sweat! So, writing up all the commmands and saving it for later, might cost you some time in the **beginning**, but there are **huge** time savers later on! To run a script you simply need to press the button Edit > Run all (`R` editor) or Code > Run Region > Run All (`RStudio`). Nobody does that however, because there are numerous Keyboard Shortcuts (I advise you to learn them, because they make you considerably faster. Actually, most Keyboard Shortcuts work in a wide variety of editors–even in `Word`.)

3.2.2 Help!

Sometimes you have found a command you would think you could use, but you do not know how, then you need to use the `help()` command, where the thing you want help on should go between parentheses. A shortcut for this would be to use the `?` operator in front of the command. For instance, you know that the command `sqrt` could be useful to find square roots, but how? Then type:

```
help(sqrt)
?sqrt
```

And the appropiate documentation will pop up. Not that the documentation will make sense immediately, but a very efficient way to understand what is going on is to look for the examples at the bottom of the documentation and copy and play with them. In fact, you can immediately run an example by using the `example` command, as follows

```
example(sqrt)
```

3.2.3 Packages

`R` (as many other software programs/languages nowadays) depends heavily on packages written by other parties, usually users of `R`. There are now many packages out there. You can find the 'official' ones on the CRAN website, but there are many more. Packages have to be installed (both `R` editor and `RStudio` have a separate package manager) and afterwards loaded. Say, for instance we want to use the awesome `ggplot2`

package (a package to make plots look nice, actually to make more elaborate plots but anyway), then to install and load the package we give the commands:

```
install.packages("ggplot2")
library("ggplot2")
```

Now, we can use the commands from this package as if they were built-in.

Gradually, we come across some useful packages. Those we will use in a chapter will always be listed at the start of the chapter.

3.2.4 Using comments

A final word in this section about the use of comments. **Do it!**. Really, it will make your future life much easier if you have documented what you have done. You can insert comments by using the `#` operator (everything after the hashtag is a comment), so, e.g.,

```
2+2 # always wanted to know about the outcome, but were afraid to ask
```

3.3 Reading and writing data

To properly do statistics one needs data (duh!). Luckily, there are numerous ways to get data in `R`.

When you just have a `csv` text file (comma separated file), it is easy, you just type:

```
df <- read.csv(file="my_data.csv", header = TRUE)
```

and you read in the `data.csv` in a data frame variable called `df`. Note that the original header variables are preserved. If you would like to store your data you can do the reverse, namely:

```
write.csv(df, file = "my_data.csv")
```

Now sometimes you do not have nicely formatted `.csv` or `.txt`files, but nasty `.dta` files from Stata or `.xlsx` files from `Excel`. Here the package Rio comes very handed, being the swiss-army knife of data converters in `R`. Assume you have the `mtcars` dataset in various formats, then you can do

```
library("rio")

x <- import("mtcars.sav")  # SPSS data file
y <- import("mtcars.xlsx") # Excel data file
z <- import("mtcars.dta")  # Stata data file
```

and all dataframes `x`, `y` `z` should be identical.

3.4 Ehmmm, dataframes

I already mentioned dataframes above, but have not yet explained what they are. Simply, it is your data. Lets construct a simple dataframe:

```
Names  <- c("Thomas", "Erik", "Mark", "Eveline")
Grades <- c(5, 8, 6.5, 7)
Female <- c(FALSE,FALSE,FALSE,TRUE)

df_grades <- data.frame(Names, Grades, Female)
```

First,note a couple of things. A dataframe can consists of various data_types, in this case strings (the names)—so variable names are always in strings—, numbers (the grades) and so-called Booleans (someone is female or not). Secondly, we have named the variables. So, we now have a dataframe called df_grades. Great, now what? Well, we can do a couple of things. By using the command `head()` we can show the first 6 rows of this dataframe.

```
head(df_grades)
```

```
##     Names Grades Female
## 1  Thomas    5.0  FALSE
## 2    Erik    8.0  FALSE
## 3    Mark    6.5  FALSE
## 4 Eveline    7.0   TRUE
```

Because we only have four observations this actually gives our whole dataframe (you can also just type `df_grades` to get this). Using square brackets `[]` allows you get specific information from this dataset, where the first index denotes the row and the second index denotes the column. Look at the following examples:

```
df_grades[1,2]
```

```
## [1] 5
```

```
df_grades[1,]
```

```
##    Names Grades Female
## 1 Thomas      5  FALSE
```

```
df_grades[3]
```

```
##   Female
## 1  FALSE
## 2  FALSE
## 3  FALSE
## 4   TRUE
```

```
df_grades["Names"]
```

```
##     Names
## 1  Thomas
## 2    Erik
## 3    Mark
## 4 Eveline
```

Most of the statistical stuff we will do invokes the use of dataframe and specific variables from that dataframe.

How to change variables or generate new variables in such a dataframe. Well, note that you can always access elements from an object by the `$`-sign. So, let's suppose we want to add a new variable to our dataframe where we add a whole point to the `Grades` variable. We can do this as follows:

```
df_grades$Grades_revised <- df_grades$Grades + 1
```

Now, the grades dataframe looks like:

```
head(df_grades)
```

```
##     Names Grades Female Grades_revised
## 1  Thomas    5.0  FALSE            6.0
## 2    Erik    8.0  FALSE            9.0
## 3    Mark    6.5  FALSE            7.5
```

```
## 4 Eveline    7.0   TRUE            8.0
```

We can even do this in a more simple way, by invoking the most brilliant `dplyr` package. Say, we want to create a log variable of the "Grades" variable and then a new variable where we add +1. No idea why, but we want it. Then

```
df_grades <- mutate(df_grades,
               log_grades = log(Grades),
               log_grades_revised = log_grades + 1)
```

This does oes the trick. Note that I only have to invoke `mutate` once. `dplyr` can be used for filtering, selecting, reshaping, sorting and a whole lot of other stuff. A very handy cheatsheet can be found here.

Now, the grades dataframe looks like:

```
head(df_grades)
```

```
##     Names Grades Female Grades_revised log_grades log_grades_revised
## 1  Thomas    5.0  FALSE            6.0   1.609438           2.609438
## 2    Erik    8.0  FALSE            9.0   2.079442           3.079442
## 3    Mark    6.5  FALSE            7.5   1.871802           2.871802
## 4 Eveline    7.0   TRUE            8.0   1.945910           2.945910
```

3.5 Functions

I already referred to the concepts of functions above. Everything in `R` that has the symbols `()` just behind a word denotes a function. And most things in `R` are really a function. Examples you have already seen above are `sqrt()`, `head()`, and `read.csv()`. As a user, you can define your own functions. (But always do this before you 'call' this function.) Suppose, we want to create a function that calculates the function $z = x^2 + y^2 + 10 * x + 12 * y - 10$ for every x and y we would like to use. For instance, we can first define the function (let's call the function 'zvalue'):

```
zvalue <- function(x,y){
  return(x^2 + y^2 + 10*x + 12* y - 10)
}
```

Note that the `return()` is not necessary, as long as it is the last statement that you make.

And we invoke the function for $x = 2$ and $y = 2$, or for $x = 2$ and $y = 3$, or even for $x = 42$ and $y = 42$,as follows:

```
zvalue(2,2)
```

```
## [1] 42
```

```
zvalue(2,3) # Why not, why not
```

```
## [1] 59
```

```
zvalue(42,42) # The answer to everying squared!
```

```
## [1] 4442
```

In fact, we now see as well why functions are so useful. We can invoke them multiple times with different parameters.

3.6 Regression modeling

Before we start laying-out how to do regression modeling in R, we first need data. And for this purpose we will simulate our data by the following commands:

```
x <- runif(100, min = 0, max = 1) # create 100 uniformly distributed numbers in interval (0,1)
y <- 2 + 6*x + rnorm(100, mean = 0, sd = 1) # rnorm stands for the normal distribution
df <- data.frame(y,x) # Strictly not necessary but for the sake of the exposition
```

So in fact we have now created the following model:

$$y_i = 2 + 6x_i + \epsilon_i,$$

where ϵ_i is standard normally distributed (as I will explain later in Section 4.1 this is for convenience but **not** absolutely needed to do linear regression).

checking this with

```
head(df)
```

```
##          y          x
## 1 2.858058 0.20153805
## 2 6.507901 0.43083945
## 3 2.622260 0.18962624
## 4 3.245422 0.09814036
## 5 6.054300 0.42511502
## 6 6.407707 0.42129878
```

indeed shows the first 6 combination of x and y. If we now perform a linear regression, then we expect that the estimated intercept should be very close to 2 and the estimated slope parameter should be very close to 6. In R we have the command `lm()` (from linear model) to do this as follows:

```
linear_model <- lm(y~x, data = df)
```

We now have performed a regression of y onto x, using the data `df` and save the result in a variable called linear_model. First look at the first part of the `lm(,)` expression. `y~x` means that `y` is the left-hand side variable and `x` the righ-hand side variable (if we have more variables, the formula becomes something as`y~x+u+v+w+z`). The second part of the `lm(,)` expression denotes the specific dataframe to be used. Namely, in R you can have multiple dataframes so you have to specify which one is to be used.

Right, but now what? Well, we have now a variable called linear_model, just typing in `linear_model` only gets you the real basic results:

```
linear_model
```

```
##
## Call:
## lm(formula = y ~ x, data = df)
##
## Coefficients:
## (Intercept)            x
##       2.069        6.075
```

But, you want more, right? Standard errors, t-statistics, R-squares, the whole lot. For this, you need the `summary()` command, which gives you the following outcome

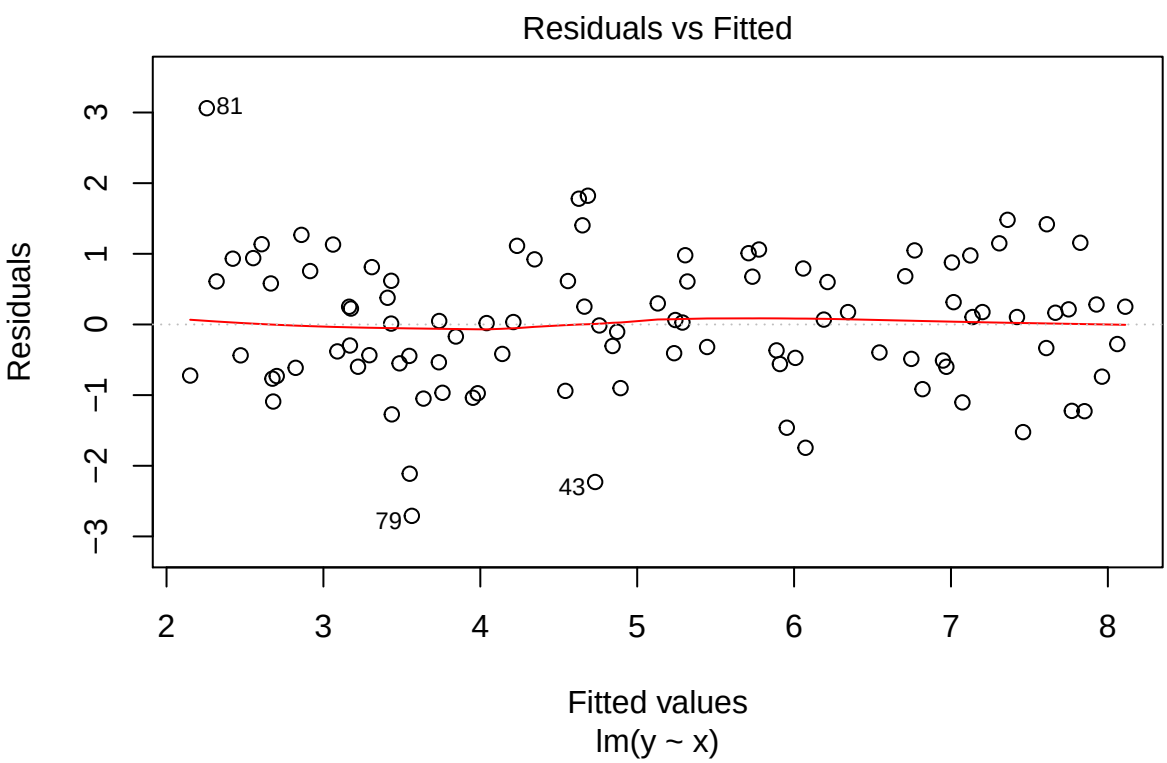
```
summary(linear_model)
```

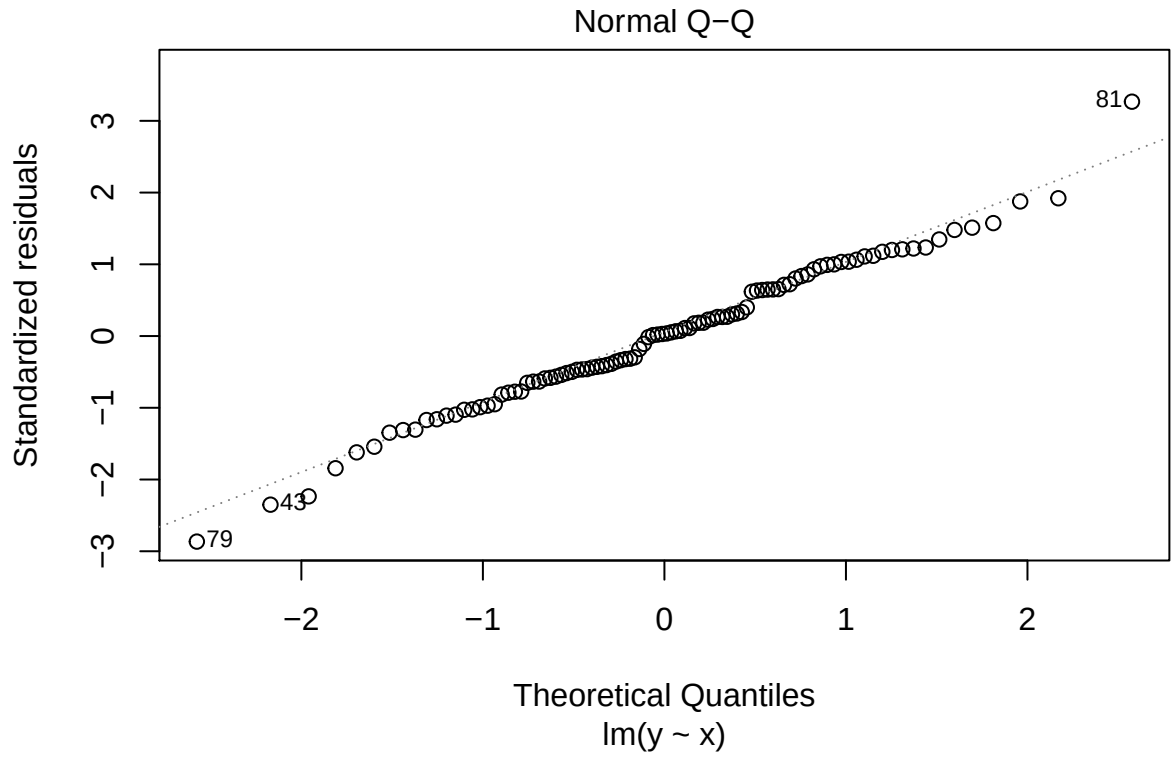
```
##
## Call:
```

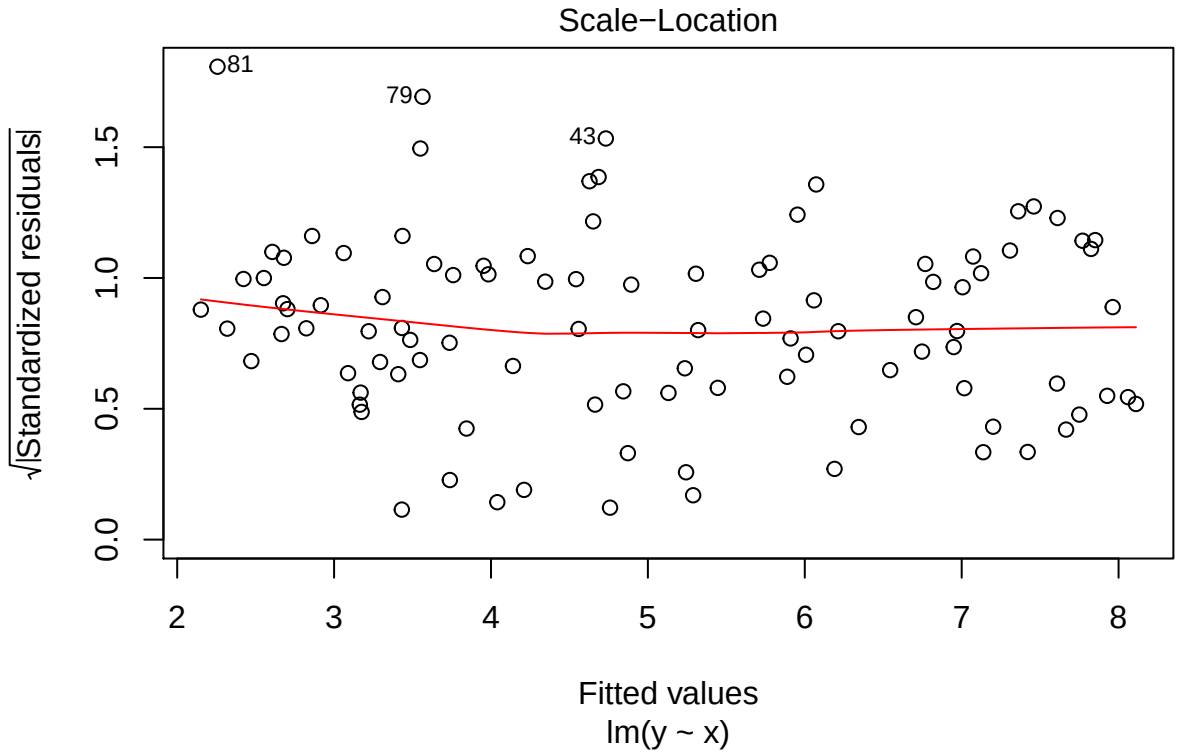
```
## lm(formula = y ~ x, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.70878 -0.56968  0.03074  0.67789  3.06108
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   2.0688     0.1858   11.14   <2e-16 ***
## x             6.0746     0.3265   18.61   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9535 on 98 degrees of freedom
## Multiple R-squared:  0.7794, Adjusted R-squared:  0.7771
## F-statistic: 346.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

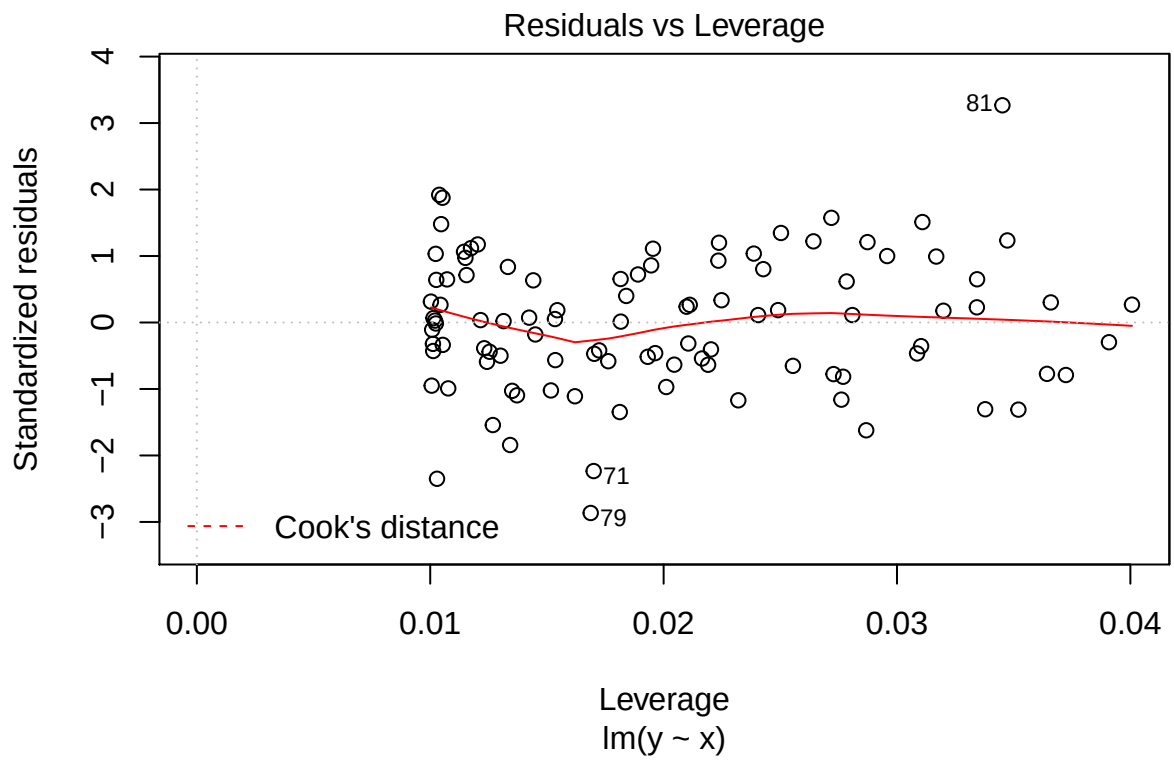
Interestingly, our parameters are close to 2 and 6, but not that close (which happens with a limited amount of observations, notice as well the relatively large standard errors). So, the command `summary()` gives all the needed statistical output, but what about regression diagnostics. For this you can ask for a plot of the variable linear_model

```
plot(linear_model)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(y ~ x)
81
43
79
−3
−2
−1
0
1
2
3


Scale−Location
√|Standardized residuals|
Fitted values
lm(y ~ x)
81
79
43
0.0
0.5
1.0
1.5
2
3
4
5
6
7
8

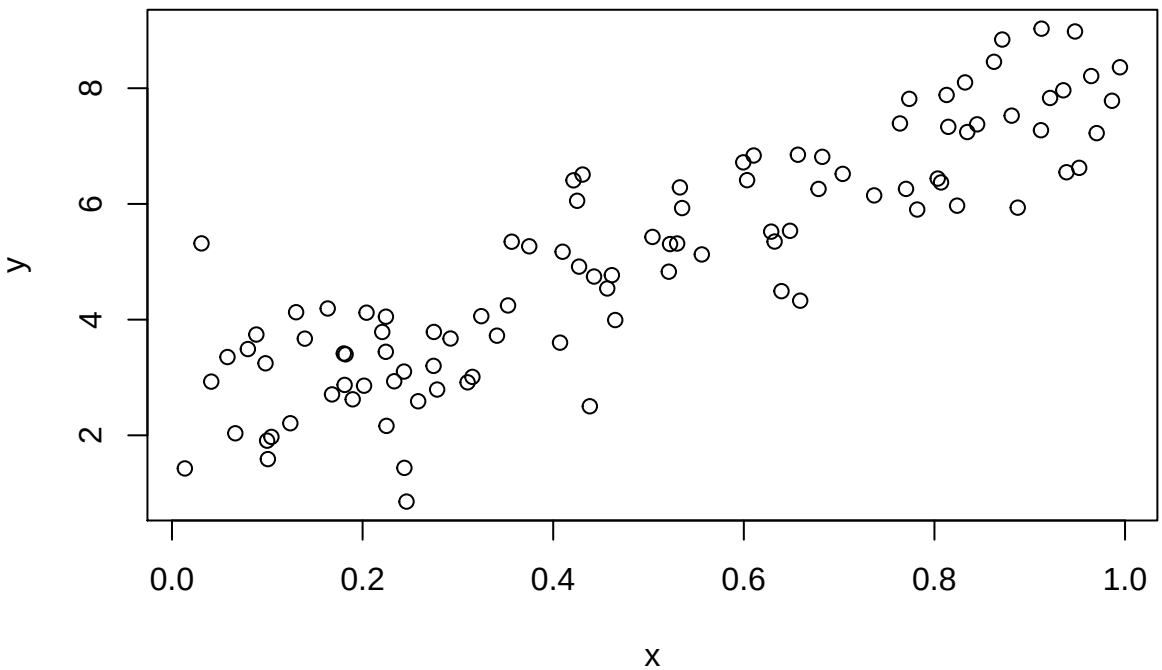


And that is basically it, at least for the basics. There is much more to say about the `lm()` command, but that is for later.

3.7 Making plots

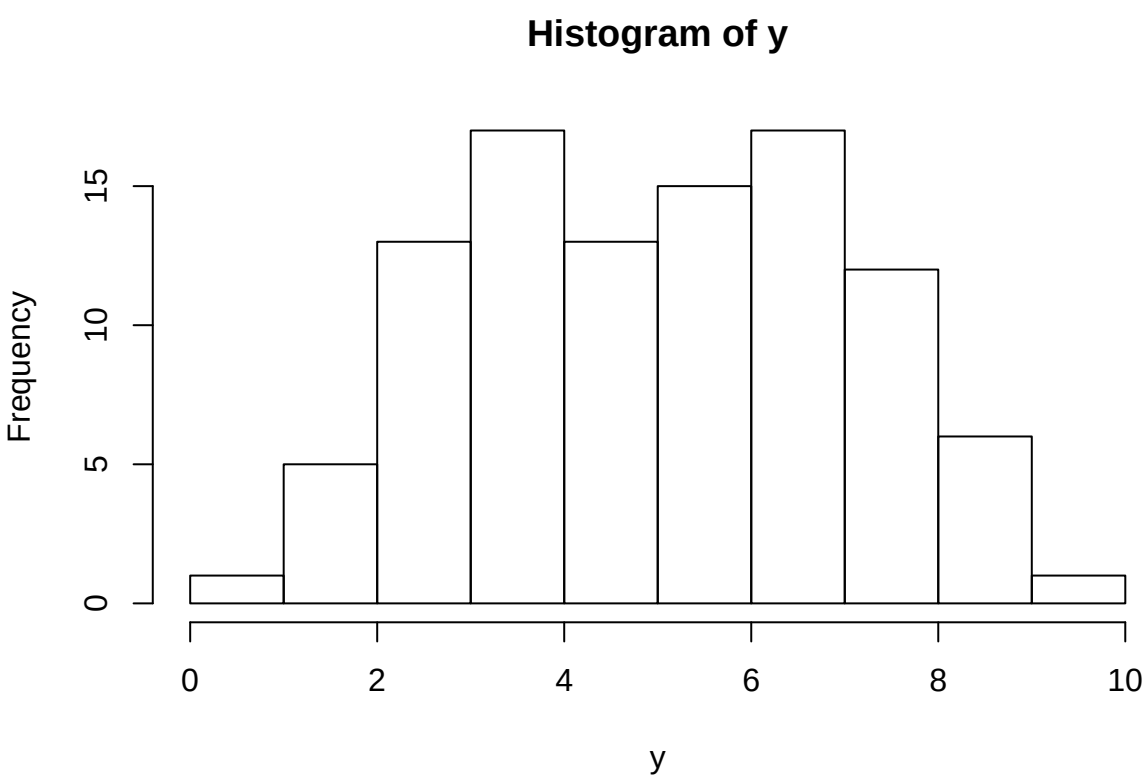
Where `R` truly shines is in making plots, diagrams, histograms, etcetera. The first thing with data you want to do is to make a scatterplot. With our previously defined x and y data this can be easily done by:

```
plot(x,y)
```



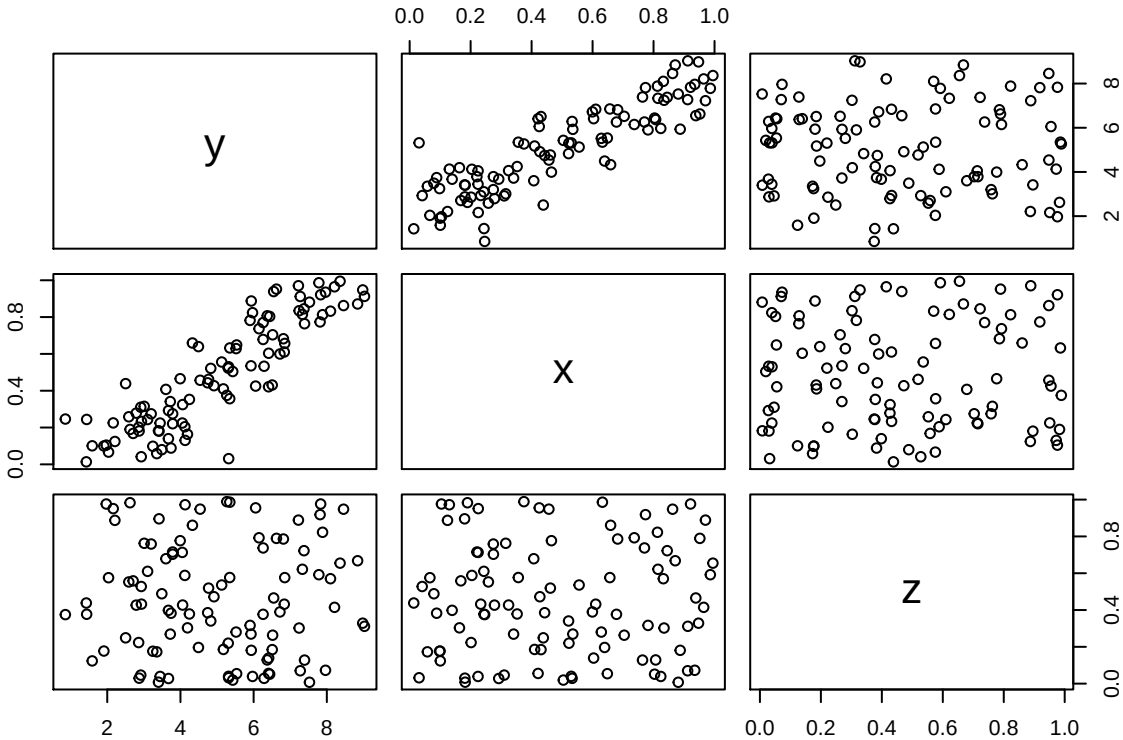
If you would like to create a histogram, just use `hist()` as

```
hist(y)
```



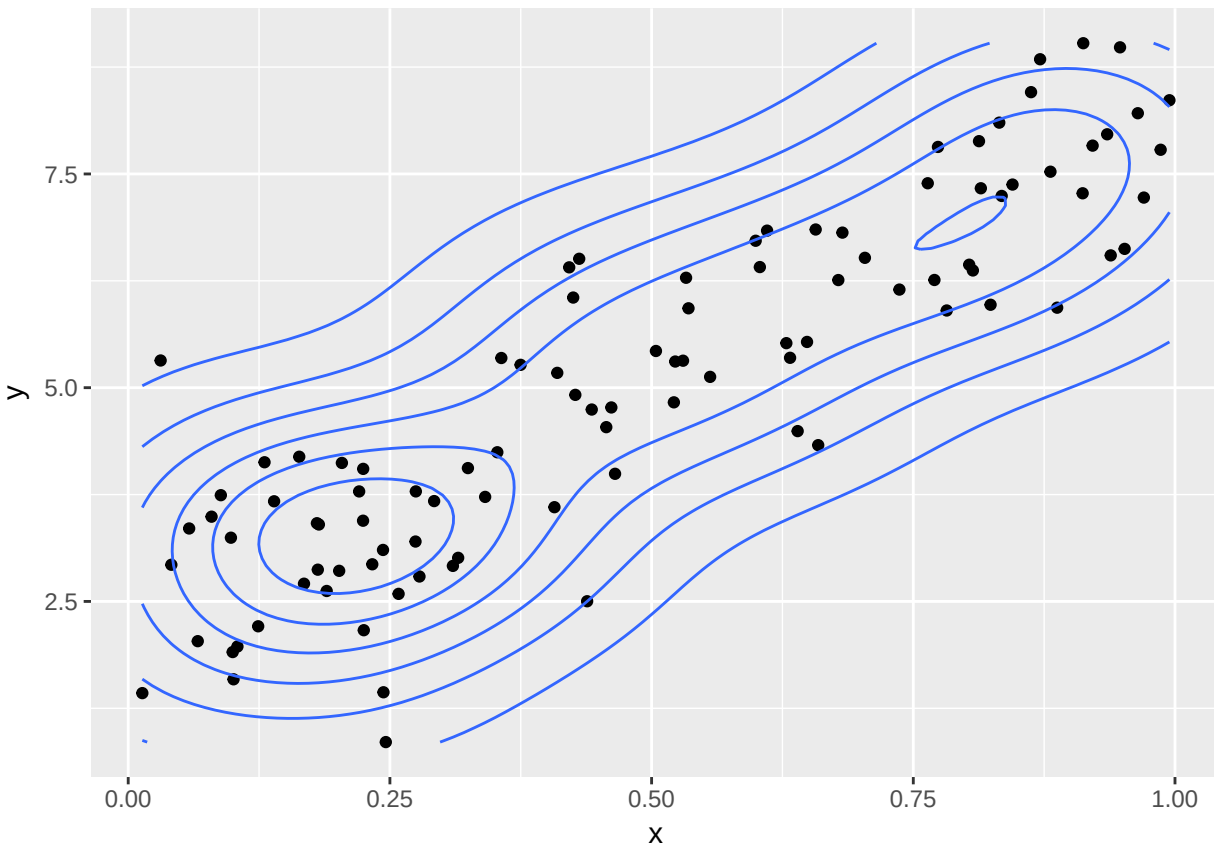
However, to go one step further you also make a plot of a dataframe. for our df dataframe this is not very insightful, but let's add another variable z uncorrelated with x and y and then plot the dataframe (the `$` indicate that `z` is a variable in dataframe `df`).

```r
df$z <- runif(100, 0,1)
plot(df)
```



And luckily, this plot confirms what we expect. x are correlated y by construction and z is not correlated with either x and y. These are all the so-called baseline plots. They are great (and already highly customizable), but lately there has been a new kid on the block called ggplot2. It goes to far to explain the details of ggplot2 (gg here stands for the grammar of graphics), but suffice to say that ggplot2 works with building blocks, so that every piece of the figure that you want (or can think of) can be constructed. Just as an example, let's redo our scatterplot but now using ggplot2 and say we want to add some density lines from our observations (just because we can). This can be done in the following way

```r
ggplot(df, aes(x,y))+geom_point() + geom_density2d()
```



3.8 Recap

For the absolute beginner `R` is huge and daunting. You need to learn by taking small steps and by practicing (a lot). Do not aim immediately at big and complex projects but start small and at the basics. You will then learn that you quickly make progress and at a certain time even become efficienter than when using mouse driven tools (clicking) as `Excel` or `SPSS`. In later chapters I dive in to some more detailed topics, and hopefully the material provides you with a background solid enough to understand and work with those topics using `R`.

4 Digression: Linear Regression and how to apply it

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

In the social sciences (in fact, in all sciences) linear regressions (also called OLS or ordinary least squares) or one of its relatives is the most *used* empirical research tool there is. Why? It is relatively simple, computationally fast, easy to interpret and relies on a relatively weak set of assumptions. Unfortunately, the assumptions needed to be able to apply linear regression, are often not well understood: both in the social sciences and beyond. Moreover, students in the social sciences typically get little guidance in how to apply linear regression in practice and how to interpret the results. Note that this does not have anything to do with the specific software students use, but more with that fact that regression techniques in a wide number of situations (courses) are just not given (apart from statistical or research methods courses). However, given the fact that data becomes increasingly more available, knowing when to use regression techniques, how to apply them and especially how to interpret and assess them is now becoming an issue of paramount importance. Or, perhaps more compelling, you need them to write your thesis.

In this chapter, I will first focus in section 4.1 on the *essential* theory behind regression analysis. I really keep it to the bare minimum. But if you understand these basics, I would argue that you understand more than 75% of the theorical background (the rest if just nitty-gritty). Section 4.2 will focus on applications of linear regression, specification issues (how many variables) and how to interpret the results.

4.1 Theoretical background

This subsection first deals with the model (what are you trying to explain), then about the three critical assumptions of (ordinary) least squares, discusses subsequently typical situations when these assumptions are violated, and finishes with a discussion about less important stuff (on which, alas, quite some attention is given in bachelor courses).

Before we start, I would like to make one important remark. In general, statistical models can be used for (*i*) finding **statistical** relations, finding (*ii*) **causal** relations and for (*iii*) **predicting**. All three uses require the same assumptions, but have different focuses. In statistics, generally the focus is on finding statistical relations, such as whether the Dutch population is *on average* taller then the German population. In economics the focus is very much on finding causal relations, so the need for explanatory power is not very large. Models that do not explain much (where the R^2's are low, say < 0.2) are just as good as models that explain quite a lot, as long as the least squares assumptions hold. In transportation science in general (and other disciplines that deals with making large models) predictions and thus explanatory power is key. Here it is now very important that you perfectly understand what causes what as long as out-of-sample predicting is good (say for predicting future commuting flows).

I usually have finding causal relations in mind when talking about least squares (already difficult enough), but note again that the same least squares assumptions, in some form or another, should hold when you want to predict or want to find statistical relations.

4.1.1 The model

Assume we are interested in the effect of the weight of a car on the fuel efficiency of the car (measured in miles per gallon). We state the following univariate regression model:

$$y_i = \alpha + \beta x_i + \epsilon_i,$$

where y denotes the fuel efficiency of the car, x denotes the weight of the car and subscript i stands for the i-th observation. α and β are the parameters of the model and they are **unknown** so they have to be **estimated**. ϵ is a so-called error term and denotes all the variation that is not captured by our two variables (α and β) and our weight of the car variable x.

The following observations are very important:

1. What is on the left hand side of the = sign is what is to be explained (in this case miles per gallon). What is on the right hand side is what we use to explain y; in this case being x, the weight of the car.
2. The parameters α en β constitute a **linear** relation between x and y
3. The parameter α is the constant and in the univariate setting denotes where the linear relation crosses the y-axis.
4. β gives the impact of x on y. Because it is a linear relation, the effect is simple. One unit change of x is associated with a β change of y. In general, we can say that β is equal to the marginal effect ($\frac{\partial y}{\partial x}$). Moreover, in a univariate setting β denotes the slope of the relation between x and y.
5. The regression error term ϵ gives all variation that is not captured by our model, so $y_i - (\alpha + \beta x_i) = \epsilon_i$. In this case, weight of the car most likely does not capture all variation in miles per gallon, so quite some variation is left in ϵ. Something else is captured as well by ϵ and that is the measurement error of y. So, if we have not measured miles per gallon precisely enough then that variation is captured as well by ϵ.

Now, let's assume that we want to incorporate another variable (say the number of cylinders, denoted by c), because we think that that variable is very important as well in explaining miles per gallon. Then we get the following *multivariate* regression model:

$$y_i = \alpha + \beta_1 x_i + \beta_2 c_i + \epsilon_i,$$

where we now have two variables on the right hand side (x_i and c_i) and three parameters (α, β_1 and β_2). In effect nothing changes with the intuition behind the model. Except for the interpretation of the parameter β_1 (and thus also β_2). Parameter β_1 now measures the impact of x on y *holding c constant*. So, multivariate regression models is nothing more (and less) than controlling for other factors. And we see later why that is very important.

4.1.2 The least squares assumptions

We are interested in the effect of the weight of the car on fuel efficiency and,therefore, our **estimate** of β_1 should be very close to the **true** β_1, especially when we have a large number of observations. Regression is great and utterly brilliant in finding this estimate, as long as the following three least squares assumptions hold:

1. There are no large outliers.
2. All left hand side (in this case y) and right side variables (in this case x and c) are *i.i.d.*
3. For the error term the following must hold: $E[\epsilon|X = x] = 0$.

The first one is easy to understand. OLS is very sensitive to large outliers in the dataset. It is therefore always good to look for outliers and think whether they are *real* observations or perhaps typo's (in Excel or something). Do not throw outliers immediately away but check whether they are correct.

The second is relatively easy to understand as well (but not that easy to uphold in practice). *i.i.d.* in this case stands for independently and identically distributed. This basically means that the observations in your dataset are independent from each other: in other words, the observations should have been correctly *sampled*.

The third looks the most horrible, and, to be quite honest, is so–both in theory as in reality. This is also the assumption that is least well understood; below, I will give some intuition what this assumptions stands for. And especially in assessing whether regression output is correct (is your estimate *really* close to β_1) this assumption is crucial.

4.1.3 Possible violations and how to spot them

A violation of the first assumption is usually easy to spot. There is a very strange outlier somewhere. But this also signifies the importance of analysing *descriptive statistics*, including means, maximums and minimums, scatterplots and histograms.

Whether your data is not *i.i.d.* can come because of a couple of reasons. The most straightforward is getting your data via snowballing (asking your friends and families using facebook to fill in a questionnaire and to ask their friends and families to do so as well). Usually, this means that you have a very specific sample and that the estimate you get is not close to the true value for the whole population. Observations might also be dependent upon each other, because of unobserved factors. In our case, it might be that a type of cars (American) are less fuel efficient than other cars (European).

Another typical violation of the *i.i.d.* assumption is in time-series, where what happened in the previous period might have a large effect on the current period.

In general, however, violations of the *i.i.d.* property are not that devastating for your model as long as you are only interested in finding the true β_1: namely, it usually only affects the precision of your estimate (the standard error) and not the estimate itself. When this assumption is violated, we therefore say that the

estimate is **inefficient**. When you want to predict, however, this assumption is crucial, as you would like your estimate to be *as precise* as possible.

When the third assumption is violated, we say that our estimate is **biased**, in other words **wrong**: our parameter estimate does not come close to the true parameter. And this happens more often than not. So, what does $E[\epsilon|X = x] = 0$ actually mean. Loosely speaking, the parameters on the right hand side (in our case x and c) on the one hand and the error term (ϵ) on the other hand are not allowed to be correlated. There are several ways how this might happen, from which the following are in our case the most relevant:

1) Reverse causality: x impacts y, but y might impact x as well. This is the classical chicken and egg problem. Do firms perform better because of good management, or do good managers go to the better performing firms?

2) Unobserved heterogeneity bias: there are factors that are not in the model but influence both x and y. For example, if american cars have both an influence on fuel efficiency and weight of the cars then the estimate that we find is not close to the true value of β_1.

3) Misspecification: we assume that our model is linear, but it actually is not. Then, again, our estimate that we find is not close to the true value of β_1.

There are other sources of violations, but in this case, these are the most important ones. Reverse causality is usually hard to correct for, but unobserved heterogeneity bias is luckily easier. Namely, we add *relevant* control variables (as we did with c). In this case we can *minimize* possible unobserved heterogeity bias. Misspefications are in general as well relatively easy to correct for. From our descriptive statistics and scatterplot we usually can infer the relation between x and y and control for possible nonlinearities by using (natural) logarithms and squared terms.

As a sidenote, natural logarithms are the ones most used for various reasons not discussed here. If we take the logarithm of both sides then for our univariate regression we get:

$$\ln(y_i) = \alpha + \beta \ln(x_i) + \epsilon_i,$$

and all the aforementioned rules and assumptions still apply. But there is something peculiar to this regression. Namely, if we are interested in the marginal effect ($\frac{\partial y}{\partial x}$), we get the following:

$$\frac{\partial y}{\partial x} = \frac{\partial e^{\ln(y_t)}}{\partial x} = \frac{\partial e^{(\alpha+\beta \ln(x_i)+\epsilon_i)}}{\partial x} = \frac{\beta}{x} e^{(\alpha+\beta \ln(x_i)+\epsilon_i)} = \frac{\beta}{x} y,$$

In other words:

$$\beta = \frac{\partial y}{\partial x} \frac{x}{y}$$

which is simply the **elasticity** between x and y. So, if there are logarithms on both the left and right hand side then the parameters (the β's) denotes elasticities.

4.1.4 Normality, heteroskedasticity and multicollinearity

Until now, we have not discussed the concepts normality, heteroskedasticity and multicollinearity. That is simply because they are not that relevant (as long as we have enough observations, typically above 40). The validity of OLS hinges just upon the three assumptions mentioned above (and they are already difficult enough). In fact, if the three assumptions are satisfied, then, as an **outcome**, the parameters (β) are normally distributed. It goes to far to explain why (the theorems required for this are deeply fundamental to statistics), but in any case, normality is **not** a core OLS assumption. It would be nice if both y and x are normally distributed because then the standard errors are minimized, but again, whether the estimate of β you find is correct or not (biased or unbiased) does not depend on normality **assumptions**.

Heteroskedasticy (in other words your standard errors are not constant) as well leads to quite some confusion. In general heteroskedasticity only leads to inefficient estimates (so only affects the standard errors). Nothing

more, nothing less. And there are corrections for that (robust standard errors in `Stata` and similar procedures in `R`), so that nobody needs to care anymore about heteroskedasticity.

Finally, there is multicollinearity. And this comes in two flavours: perfect and imperfect multicollinearity. Perfect multicollinearity occurs, e.g., when your model contains two identical variables. Then, OLS can not decide which one to use and usually one of the variables is dropped, or your computer program gives an error (*computer says no*).

Imperfect multicollinearity occurs when two variables are highly (but not perfectly) correlated. This occurs less often than one may think. Variables that are highly correlated (say *age* and age^2) can be perfectly incorporated in a model. Only when the correlation becomes very high (say above 95% or even higher) then something strange happens: the standard errors get very large. Why? That is because of the definition mentioned above. Parameter β_1 measures the impact of x on y *holding* c *constant*. But if x and c are very highly correlated and you control for c, then not much variation is left over for x. So, c actually removes the variation *within* the variable x. This always happens, and there is a trade-off between adding more variables and leaving enough variation (note that there is always correlation between variables), but usually it all goes fine. Good judgement and sound thinking typically helps more than strange statistics (say VIF?).

4.2 Applications of linear regression

This section gives an application of regression analysis. Assume we are still interested in the effect of the weight of a car on the fuel efficiency of the car (measures in miles per gallon). We have found a dataset (internal in `R`), so the first thing we have to do is look at the descriptives of the dataset.

4.2.1 Descriptives

The build-in dataset `mtcars` has, besides several other variables, information on weight of a car (in 1000 lbs or in about 450 kilos) and miles per gallon. With the following command `head()` we can look at the first 6 observations.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Again, we are interested in the relation weight of a car (the variable `wt`) and miles per gallon (the variable `mpg`). Note, that in his case `mpg` is the first column and `wt` is the sixth column. (The column with car names above is not a real variable, but are the row names). Recall that the command `c()` combines stuff, so we can look at the summary statistics of the variables we are only interested in by:

```
summary(mtcars[,c(1,6)])
```

```
##       mpg              wt
##  Min.   :10.40   Min.   :1.513
##  1st Qu.:15.43   1st Qu.:2.581
##  Median :19.20   Median :3.325
##  Mean   :20.09   Mean   :3.217
##  3rd Qu.:22.80   3rd Qu.:3.610
##  Max.   :33.90   Max.   :5.424
```

There does not seem to be anything out of the ordinary here, but to be sure, we construct a scatterplot between weight of the car and miles per gallon.

```
plot(mpg~wt, mtcars)
```

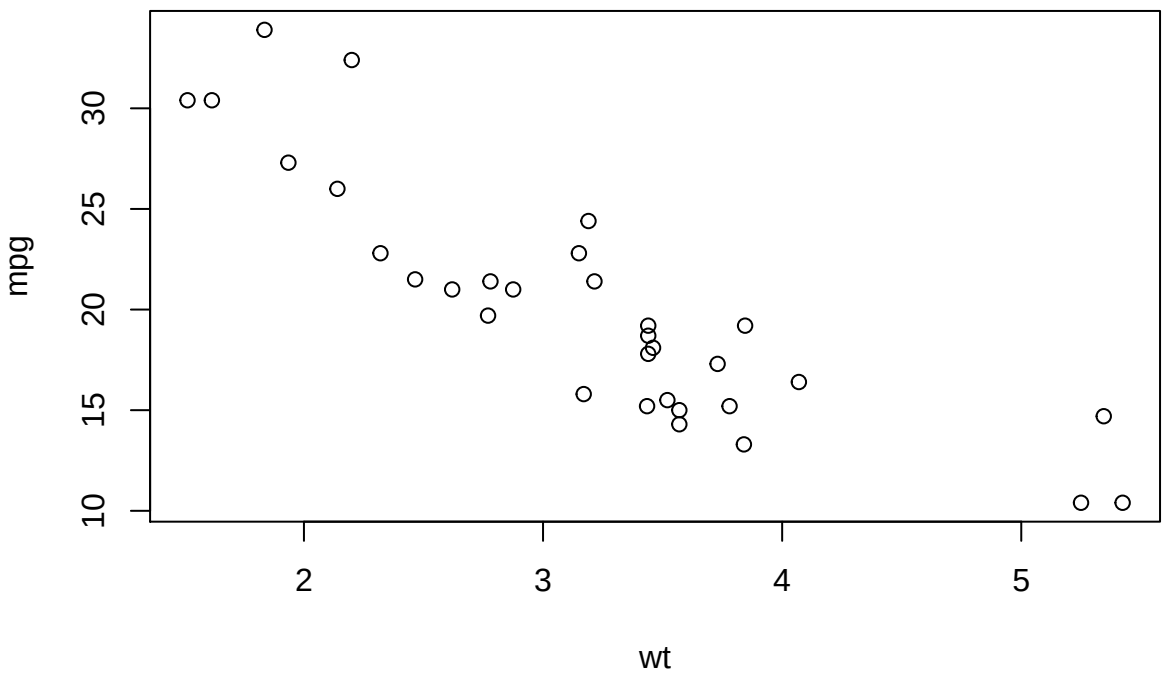


Figure 1: A scatterplog between miles per gallon and car weight.

So, there do not seem to be many outliers here. Moreover, as we would expect, there seems to be a downward sloping relation between weight of the car and miles per gallon (hopefully you agree, that this makes sense). To **quantify** this relation, the next subsection will perform a least squares estimation.

4.2.2 Baseline model

So, if we are only interested in the relation between weight of a car (the variable `wt`) and miles per gallon, we can easily perform the following regression (recall again that the command `lm()` performs a least squares estimation and that `<-` denotes an assignment to a variable:

```
baselinemodel <- lm(mpg~wt, mtcars)
summary(baselinemodel)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

At this point it is good to stop and see what we have got. We have the formula call (we know this one), some stuff about the residuals, stuff about the coefficients (most important for us), and some diagnostics (including the notorious R^2). We zoom in on the results about the coefficients. We have an estimation of the intercept (our α of above) and of `wt` (our β of above). For both, we have as well a standard error, a t-value, a probability and a bunch of stars. What do they all mean again? We focus on `wt` here (typically we are less interested in the constant or intercept).

The estimate is easy; that is β or the marginal effect of x on y. So, increasing x with 1 (or with a 1000 lbs) decreases the miles per gallon with 5.3445 (which is quite a lot).

The standard error denotes the *precision* of the estimate. As a rule of thumb: you know with 95% centainty that the true value of β lies within the interval [estimate $-2\times$ standard error, estimate $+2\times$ standard error]. The standard error is very important and is used to **test** possible values of the parameter. One of the most important tests is whether $\beta = 0$. Why? Because, if $\beta = 0$ then the variable `wt` does nothing on `mpg`. This specific test is always denoted by the t-value, its associated probability and the corresponding star thingies. In this case, the t-value is high in an absolute sense, so it is *very* improbable that the estimate could be zero (somehting like a probability of 0.0000000001 which is small indeed), and the stars neatly indicate that this probability (of $\beta = 0$) is smaller than 0.001.

One nice trick is to plot the regression line in the scatterplot above. One can do so by the command `abline()` or:

```r
plot(mpg~wt, mtcars)
abline(lm(mpg~wt, mtcars))
```

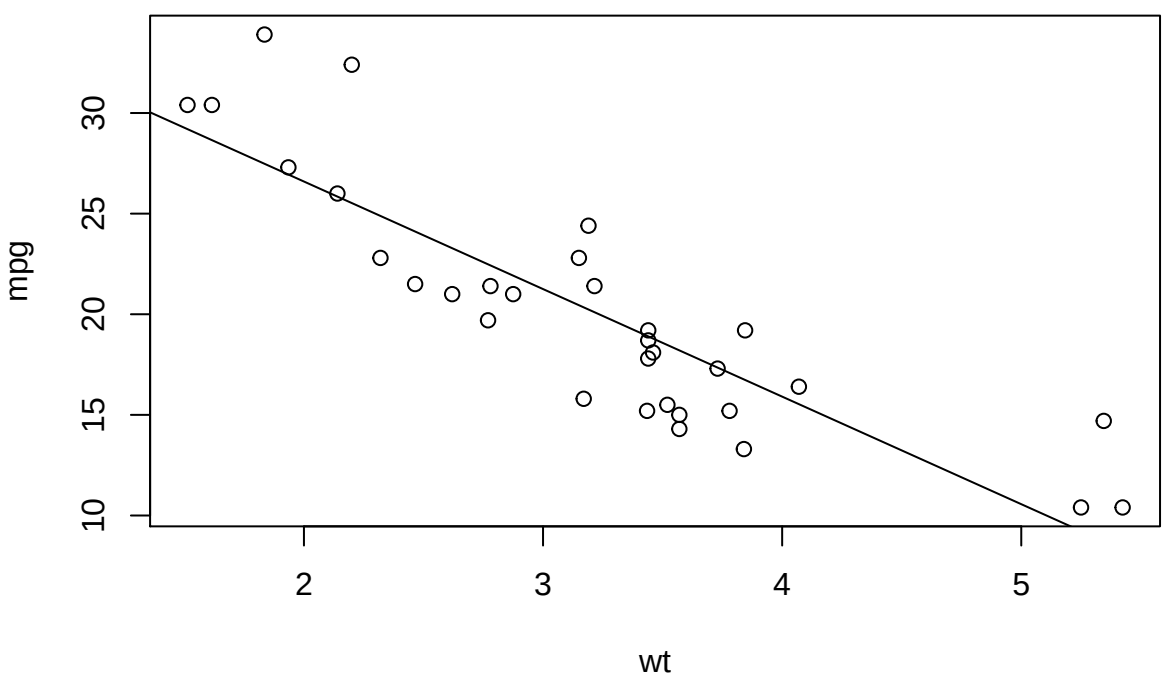


Figure 2: A scatterplog between miles per gallon and car weight with regression line.

4.2.3 Specificion issues

I can imagine that you are not very satisfied yet with the analysis. First of all, the relation between `mpg` and `wt` might be nonlinear and, secondly, you would like to include additional variables. First we look at the possible nonlinearity in the regression relation (note that we can use the regression formula as before, but that we can specify logarithmic relations by `log()`):

```
logmodel <- lm(log(mpg)~log(wt), mtcars)
summary(logmodel)
```

```
##
## Call:
## lm(formula = log(mpg) ~ log(wt), data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.18141 -0.10681 -0.02125  0.08109  0.26930
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.90181    0.08790   44.39  < 2e-16 ***
## log(wt)     -0.84182    0.07549  -11.15 3.41e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1334 on 30 degrees of freedom
## Multiple R-squared:  0.8056, Adjusted R-squared:  0.7992
## F-statistic: 124.4 on 1 and 30 DF,  p-value: 3.406e-12
```

We have now the same type of output as before, but I would like to focus on two things here. First, the R^2 has gone up and that is what you typically get in the social sciences. Logarithmically transformed variables usually **fit** better. Secondly, the interpretation of the estimate now differs. It has become an elasticity with size -0.84, which is quite high again. If the car doubles in weights, the fuel efficiency of the car goes down by 84%!

Secondly, you might want to include other variables, such as being a foreing car (opposite to a car from the US) (`vs`), the number of cylinders (`cyl`), the gross horsepower (`hp`) and how quick the car does over a quarter of a mile (`qsec`). Again, we are not interested in these additional variables or whether they crank up the R^2. The only thing we are interested is in whether the coefficient of `log(wt)` changes.

```
extendedmodel <- lm(log(mpg)~log(wt)+vs+cyl + hp + qsec, mtcars)
summary(extendedmodel)
```

```
##
## Call:
## lm(formula = log(mpg) ~ log(wt) + vs + cyl + hp + qsec, data = mtcars)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.201746 -0.065242 -0.009506  0.079809  0.205166
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.5280554  0.4610947   7.651 4.04e-08 ***
## log(wt)     -0.6514583  0.1509786  -4.315 0.000205 ***
## vs          -0.0149215  0.0863277  -0.173 0.864111
```

```
## cyl         -0.0160253  0.0314486  -0.510 0.614651
## hp          -0.0007670  0.0006861  -1.118 0.273786
## qsec         0.0212017  0.0267153   0.794 0.434603
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1121 on 26 degrees of freedom
## Multiple R-squared:  0.8811, Adjusted R-squared:  0.8582
## F-statistic: 38.53 on 5 and 26 DF,  p-value: 3.227e-11
```

Clearly, the coefficient of `log(wt)` changes with the inclusion of additional variables. So there was unobserved heterogeneity bias in our baseline model (and most likely there still is in our extended model), even though the addional variables are not siginficant.

4.2.4 Reporting

Hopefully you agree that the regression output of above looks **horrible** and that you do not need all these statistics and stuff. Therefore, you need to construct your own table for presentation format. The minimum what should be in these tables are the coefficient names, the estimates, the standard errors (and the stars would be nice as well), the R^2 and the number of observations used.

Unfortunately, creating tables of your own is a pain in the … Luckily, within `R` (and by the way in programs as `Stata` as well) you can do this automatically! In `R` there are several packages one can use. I prefer the package Stargazer and after using this package we get the following outcome for our logarithmic specification:[5]

```r
stargazer(logmodel, extendedmodel, header=FALSE, type = "html", omit.stat = c("rsq", "f"))
```

Dependent variable:

log(mpg)

(1)

(2)

log(wt)

-0.842***

-0.651***

(0.075)

(0.151)

vs

-0.015

(0.086)

cyl

-0.016

(0.031)

hp

-0.001

[5]To get nice tables directly into `Word`, you need something else (as you can imagine, because `Word` is not scriptable). With Stargazer you should give the command as such `stargazer(logmodel, extendedmodel, out = "table1.txt",omit.stat = c("rsq", "f"))` and thereafter you can read in and edit the table `table1.txt`in `Word`.

(0.001)

qsec

0.021

(0.027)

Constant

3.902***

3.528***

(0.088)

(0.461)

Observations

32

32

Adjusted R2

0.799

0.858

Residual Std. Error

0.133 (df = 30)

0.112 (df = 26)

Note:

p<0.1; ***p<0.05;*** p<0.01

Note that we now display both specifications, the first the univariate case and the second the multivariate case. This is now typically done in social science research. You start with a baseline and then add variables to check whether the variable of interest remains robust.

4.2.5 Internal validation

So, after done all modeling exercitions you are not done! Now, it is time to validate your results internally. Can you be confident of your results? Or is the parameter that you have found still suspect of possible biases. Here again come the three least squares assumptions:

1. No large outliers: we have checked that with our descriptive statistics and our scatterplot and it seems that this assumption is met.
2. Both miles per gallon and car weight should be *i.i.d.*. We can reasonably assume that the sampling has been done correctly.
3. The following condition should hold: $E[\epsilon|X = x] = 0$. This one is dubious. Probably there is no reverse causality (difficult to image that miles per gallon would influence car weight), but most likely there are other factors that impact car weight and miles per gallon which are omitted (where is the car build, what is the car brand, etcetera.). One possible strategy here is to add additional variables until the parameter of interest remains robust (does not change anymore).

Note that the number of observations in this case is only 32, so whether our standard errors are automatically normally distributed is highly questionable, but for the sake of the exposition (and the fact that small sample statistics are very complex) we leave it just with this observation.

5 Network analysis with R

```
library("igraph")
library("igraphdata")
library("networkD3")
```

5.1 Introduction

Network analysis is huge and applied in many disciplines, ranging from mathematics to sociology. In R there are many packages available. We will use the igraph package, which seems to be the most encompassing.[6] The next sections first starts with creating networks on our own (from scratch), then I show how to make structured networks and finally we create networks from data. Thereafter, I say something about creating interactive networks (those cool things, you sometimes see on internet). The last section deals with calculating network characteristics.

5.2 Creating networks

There are various ways to create a network: one can it oneself, one can create a structured network, and one can read a network from file or the internet. (The latter is typically done with very large and dynamics network data.)

5.2.1 Creating network yourself

First of all, I will show how to create a network "by hand". We first start with an undirected network. Suppose we have a network with four nodes (named 1,2, 3 and, you guessed it, 4). Then we can create a graph called g_4 by:

```
g_4 <- graph (edges = c(1,2,2,3,3,4, 4,1), n = 4, directed = FALSE)
```

Let us have a look at g_4

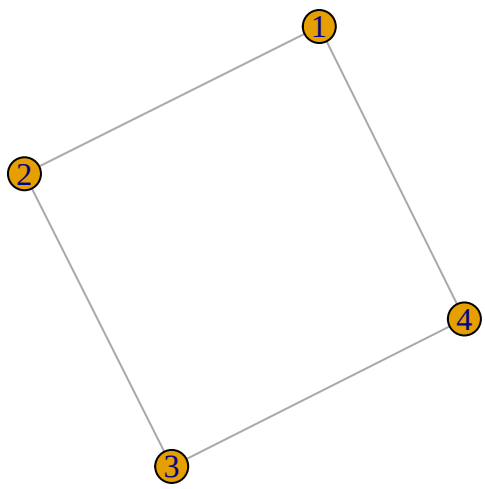
```
g_4
```

```
## IGRAPH bac3020 U--- 4 4 --
## + edges from bac3020:
## [1] 1--2 2--3 3--4 1--4
```

So, g_4 is an igraph sort of object which has four edges 1--2 2--3 3--4 1--4. Note that these coincide with c(1,2,2,3,3,4, 4,1), so an edge is always formed by two components (they do not have to be numbers). Now plotting g_4 leads to:

```
plot(g_4)
```

[6] Obviously I want to provide a self-contained manual, however, there are as well some pretty good and more extensive tutorials on internet: e.g., the one over here.



The connectivity matrix of the network can be directly given by:

```
g_4[]
```

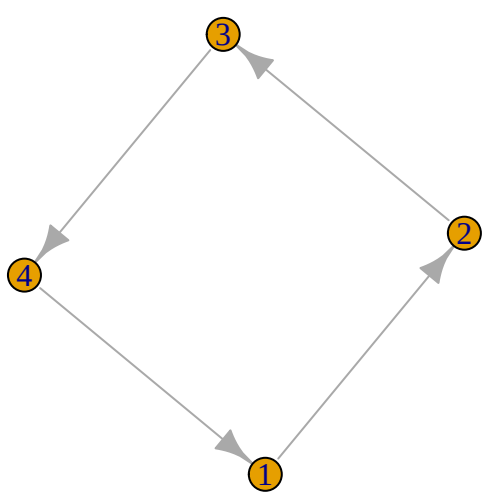
```
## 4 x 4 sparse Matrix of class "dgCMatrix"
##
## [1,] . 1 . 1
## [2,] 1 . 1 .
## [3,] . 1 . 1
## [4,] 1 . 1 .
```

Now, for more directed graphs we need to set `directed = TRUE`.

```
g_4 <- graph (edges = c(1,2,2,3,3,4, 4,1), n = 4, directed = TRUE)
```

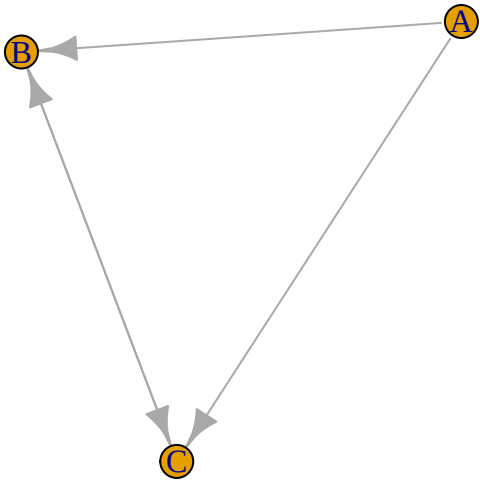
with as plot

```
plot(g_4)
```



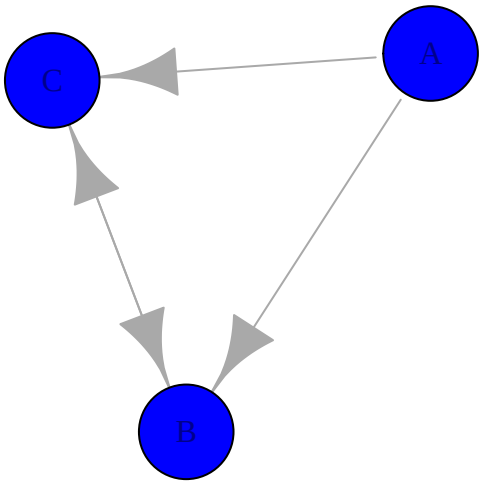
Graphs may also be generated by using `-`, `-+` and `+-+`, in combination with the command `graph_from_literal`. For example:

```
g_directed <- graph_from_literal(A-+B, B +-+ C, C+-A)
plot(g_directed)
```



Finally, we can influence the color and sizes (amongst others) of all the elements of the network, so:

```
plot(g_directed, edge.arrow.size=2,vertex.size = 50, vertex.color = "blue" )
```

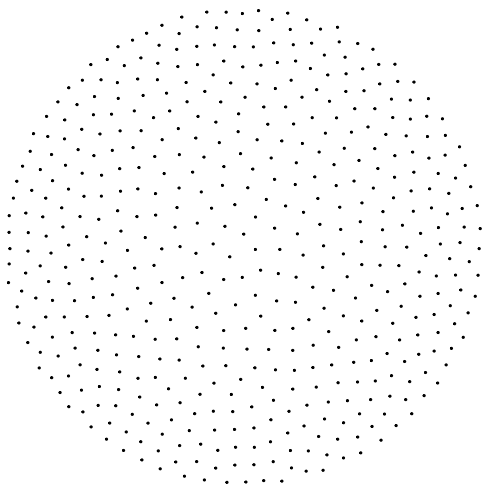


However, this way of creating of networks is not very efficient. We therefore offer some other ways as well, starting with created directly (well-structured networks).

5.2.2 Structured networks

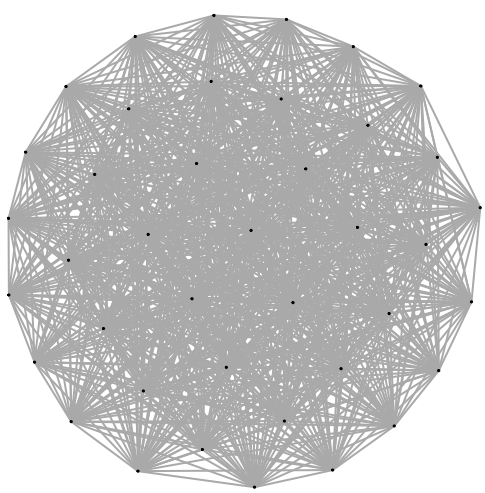
With the package `igraph` it is rather straighforward to create structured networks. Let's start with an empty network. We therefore use the command `make_empty_graph(500` in order to create 500 nodes and no links.

```
g_empty <- make_empty_graph(500)
plot(g_empty, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



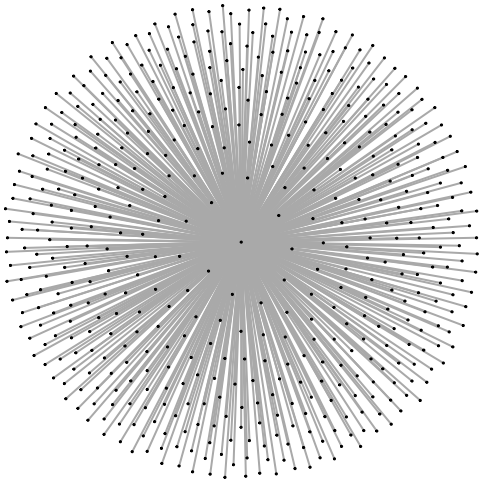
The other extreme is obviously the fully connected network. With the command `make_full_graph(40)` we can create a fully connected network between 40 nodes. (Please do not do 500 nodes; it takes a long time to create that network (with $500 \times 500 - 500$ links)).

```
g_full <- make_full_graph(40)
plot(g_full, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



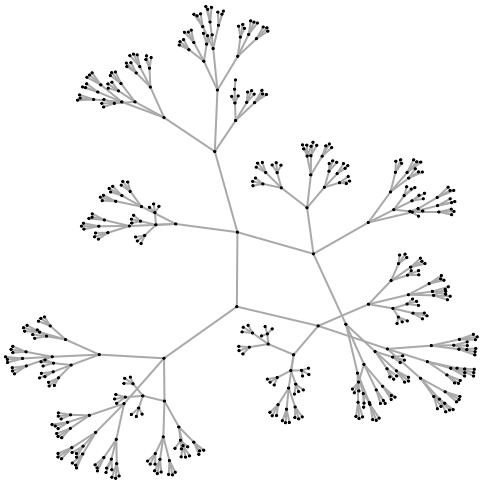
The `make_star(500)` command create a star network (or a hub-and-spoke network) with 500 nodes.

```
g_star <- make_star(500)
plot(g_star, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



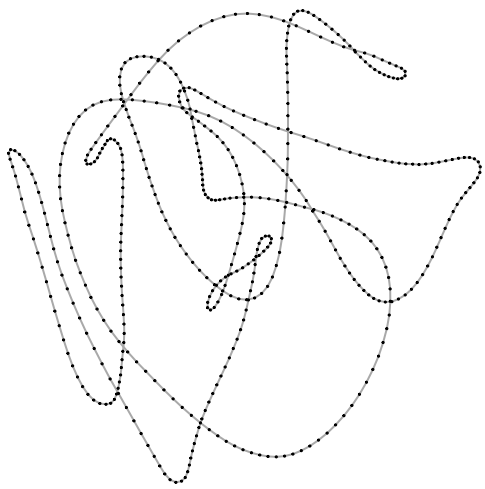
With the `make_tree(500, children = 3, mode="undirected` command, we create an undirected hierarchical tree with each time tree children

```
g_tree <- make_tree(500, children = 3, mode = "undirected")
plot(g_tree, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



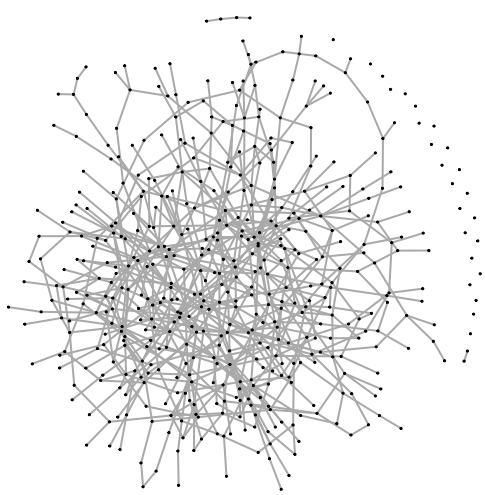
And the `graph.ring(500)` creates a ring (or line) structure where each node is connected to two other nodes in a sequential way

```
g_ring <- graph.ring(500)
plot(g_ring, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



For the real-world network we can use the following commands. First, the command `erdos.renyi.game(500, 0.005)` creates a random network, where each node has a probability of 0.005 to be connected to each other node.

```
plot(erdos.renyi.game(500, 0.005), vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



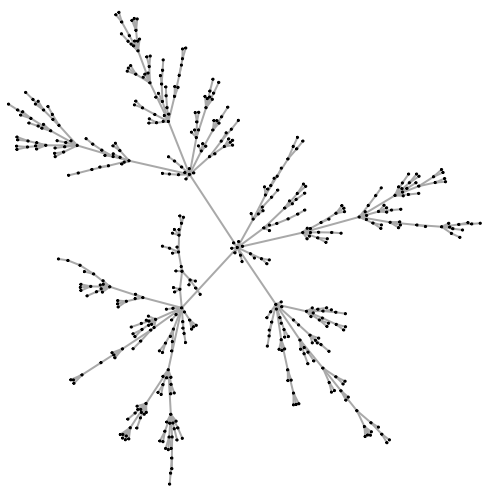
For the small-world network, we need to rewire the ring network. We can do by the command `rewire` as follows:

```
plot(rewire(g_ring,each_edge(prob = 0.5)), vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```

Note that the probability of rewiring is now 0.5.

Finally, for a power-law we need to invoke an algorithm. In this case the `barabasi.game`, where we now create a power-law type of network with the following underlying formula $P(k) = k^{-0.5}$.

```
g_power_law <- barabasi.game(500, 0.5)
plot(g_power_law, vertex.label= NA, edge.arrow.size=0.02,vertex.size = 0.5)
```



Note the tree-like structure.

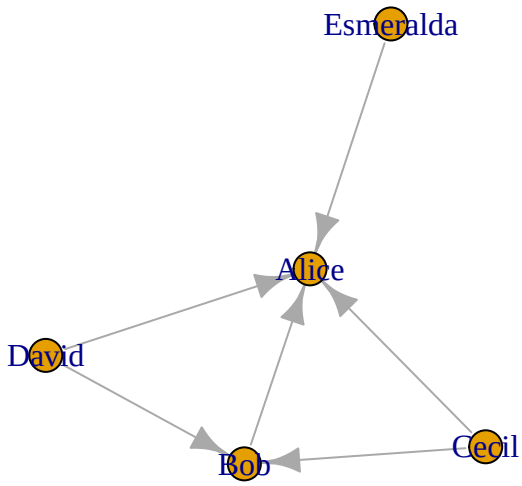
5.2.3 Reading in networks

Another good but smaller source is the `igraphdata` package which actually contains a network of american domestic flights.

Another possibility is to read in the data from a `.csv` file. Something like (I took and modified the example from http://stackoverflow.com/questions/23687806/creating-a-network-graph-using-igraph-in-r):

```
# Make up data
relations <- data.frame(from=c("Bob", "Cecil", "Cecil", "David", "David", "Esmeralda"),
                        to=c("Alice", "Bob", "Alice", "Alice", "Bob", "Alice"))
# Alternatively, you could read in the data from a similar CSV file as follows:
# relations <- read.csv("relations.csv")
```

```r
# Load (DIRECTED) graph from data frame
g_relations <- graph.data.frame(relations, directed=TRUE)
plot(g_relations)
```



Note that this only requires a from and to columns in a `.csv` file, which resembles trade, commuting, migration and transport networks (in a gravity-model type of way). The only thing is that we need to convert this data in a igraph structure with `graph.data.frame`.

5.2.4 Sankey networks

```r
# Read in the data in a dataframe called Data
Data <- read.csv("Migration_Corop_2003.csv", header=TRUE,stringsAsFactors=FALSE)
# We filter some of the data so that our sankey diagram will not be too large
Data <- filter(Data, To < 25)
Data <- filter(Data, From < 25)
# Create a variable for the names of the node (both on the left and right side)
DataNames <- rep(Data$NameFrom[1:8], times=2)
# Create a dataframe of this variable names
Names <- data.frame(DataNames, stringsAsFactors=FALSE)
# Revalue To and From such that they nicely count up to the number of nodes (Note, we start at index nu
Data$To <- rep(8:15, each=8)
Data$From <- rep(0:7, times=8)

# Now plot network
sankeyNetwork(Links = Data, # which dataframe to be used for the flows
              Nodes = Names, # which dataframe to be used for the names of the nodes
              Source = "From", # where do the flows come from (index number)
              Target = "To",  # where do the flows go to (index number)
              Value = "Migrants", # the variable that denote the flows from dataframe
              units = "Migrants", # Which units are the flows (not necessary)
              NodeID = "DataNames", # Variable that gives the names from the dataframe Names
              fontSize = 14,  # Make things look better
              nodeWidth = 40) # make things look good
```

5.3 Network characteristics

So, it is fine that you have a network, but what about the network characteristics. Well these are easily given once you have a network.

For density, or the completeness of the network, one can look at

6 {r} # edge_density(g_florence$PADGB)

7

8 Which give the proportion of the present edges from all possible edges in the network.

9

10 The diameter of a network can be retrieved as:

11

12 {r} # diameter(g_florence$PADGB)

somewhat more interesting is given by the degree function which gives the number of connections. For our power-law network this amounts to

13 {r} # deg <- degree(g_power_law) # deg

14

15 which is not that insightful, but a histogram is:

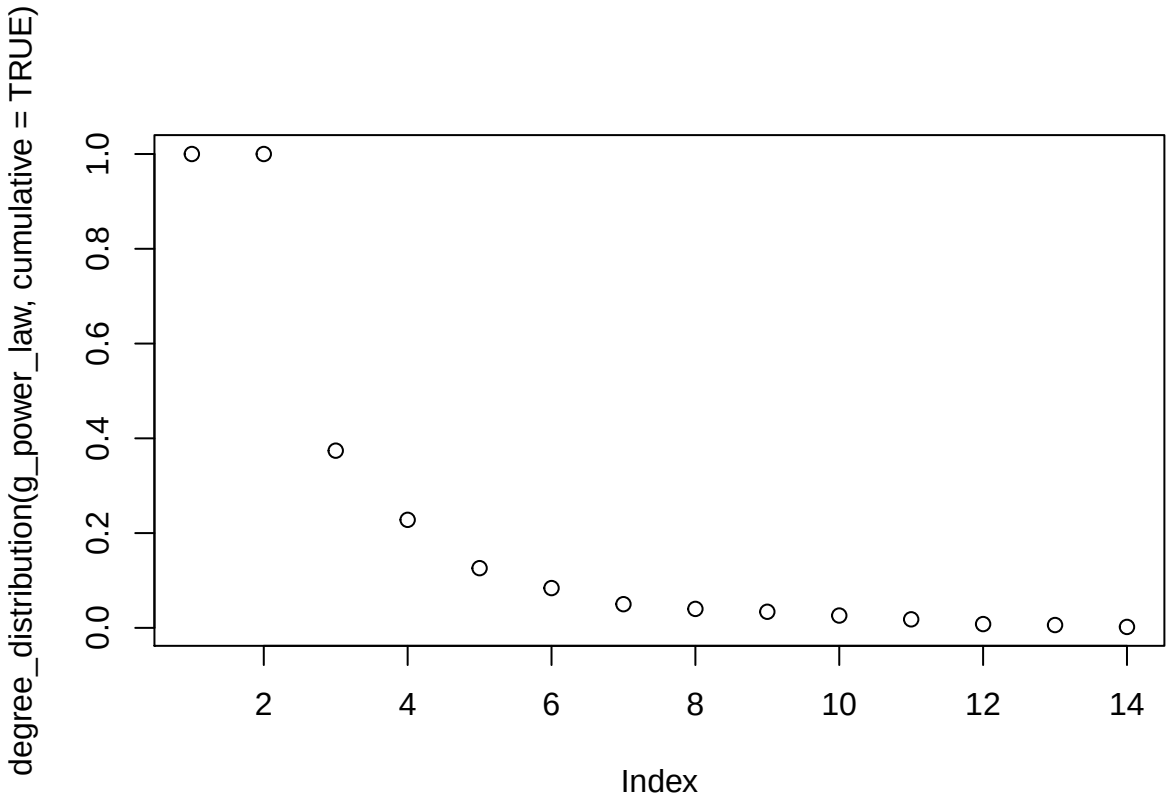
16

17 {r, message=FALSE} # hist(deg)

Which is typical for a power-law.

The `degree_distribution` is insightful here as well

```
plot(degree_distribution(g_power_law, cumulative = TRUE))
```



Finally, we may be interested in the shortest path. This can easily be retrieved using distances (with underlying Dijkstra's algorithm) as follows for the Florence network:

18 `{r} # distances(g_florence$PADGB) #`

Note that the Pucci family is not in the network and therefore the distances are set to infinity.

19 In conclusion

To be written