Introduction to Econometrics with R

Florian Oswald, Jean-Marc Robin and Vincent Viers

2019-08-16

Contents

Syllabus

Welcome to Introductory Econometrics for 2nd year undergraduates at ScPo! On this page we outline the course and present the Syllabus. 2018/2019 is the first time this course will be taught, so we are still in a *beta* release stage - you should expect a couple of loose ends here and there, but we think the overall experience is going to be pleasant!

Objective

This course aims to teach you the basics of data analysis needed in a Social Sciences oriented University like SciencesPo. We purposefully start at a level that assumes no prior knowledge about statistics whatsoever. Our objective is to have you understand and be able to interpret linear regression analysis. We will not rely on maths and statistics, but practical learning in order to teach the main concepts.

Syllabus and Requirements

You can find the topics we want to go over in the left panel of this page. The later chapters are optional and depend on the speed with which we will proceed eventually. Chapters 1-4 are the core material of the course.

The only requirement is that **you bring your own personal computer** to each session. We will be using the free statistical computing language `R` very intensively. Before coming to the first session, please install `R` and `RStudio` as explained at the beginning of chapter 1.

Course Structure

This course is taught in several different groups across various campuses of SciencesPo. All groups will go over the same material, do the same exercises, and will have the same assessments.

Figure 1:

Groups meet once per week for 2 hours. The main purpose of the weekly meetings is to clarify any questions, and to work together through tutorials. The little theory we need will be covered in this book, and **you are expected to read through this in your own time** before coming to class.

This Book and Other Material

What you are looking at is an online textbook. You can therefore look at it in your browser (as you are doing just now), on your mobile phone or tablet, but you can also download it as a `pdf` file or as an `epub` file for your ebook-reader. We don't have any ambition to actually produce and publish a *book* for now, so you should just see this as a way to disseminate our lecture notes to you. The second part of course material next to the book is an extensive suite of tutorials and interactive demonstrations, which are all contained in the `R` package that builds this book (and which you installed by issuing the above commands).

Open Source

The book and all other content for this course are hosted under an open source license on github. You can contribute to the book by just clicking on the appropriate *edit* symbol in the top bar of this page. Other teachers who want to use our material can freely do so, observing the terms of the license on the github repository.

Assessments

We will assess participation in class and conduct a final exam.

Communication

We will communicate exclusively on our `slack` group. You will get an invitation email to join from your instructor in due course.

Chapter 1

Introduction to R

1.1 Getting Started

R is both a programming language and software environment for statistical computing, which is *free* and *open-source*. To get started, you will need to install two pieces of software:

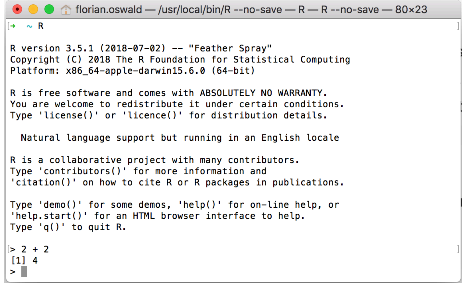
1. R, the actual programming language.
    - Chose your operating system, and select the most recent version.
2. RStudio, an excellent IDE for working with R.
    - Note, you must have R installed to use RStudio. RStudio is simply an interface used to interact with R.

The popularity of R is on the rise, and everyday it becomes a better tool for statistical analysis. It even generated this book!

The following few chapters will serve as a whirlwind introduction to R. They are by no means meant to be a complete reference for the R language, but simply an introduction to the basics that we will need along the way. Several of the more important topics will be re-stressed as they are actually needed for analyses.

This introductory R chapter may feel like an overwhelming amount of information. You are not expected to pick up everything the first time through. You should try all of the code from this chapter, then return to it a number of times as you return to the concepts when performing analyses. We only present the most basic aspects of R. If you want to know more, there are countless online tutorials, and you could start with the official CRAN sample session or have a look at the resources at Rstudio or on this github repo.

Figure 1.1: R GUI symbol and R in a MacOS Terminal



florian.oswald — /usr/local/bin/R --no-save — R — R --no-save — 80×23

```
➜  ~ R

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

> 2 + 2
[1] 4
> 
```

Figure 1.2: R GUI symbol and R in a MacOS Terminal

1.2 Starting R and RStudio

A key difference for you to understand is the one between `R`, the actual programming language, and `RStudio`, a popular interface to `R` which allows you to work efficiently and with greater ease with `R`.

The best way to appreciate the value of `RStudio` is to start using `R` *without* `RStudio`. To do this, double-click on the R GUI that you should have downloaded on your computer following the steps above (on windows or Mac), or start R in your terminal (on Linux or Mac) by just typing `R` in a terminal, see figure **??**. You've just opened the R **console** which allows you to start typing code right after the `>` sign, called *prompt*. Try typing `2 + 2` or `print("Your Name")` and hit the return key. And *voilà*, your first R commands!

Typing one command after the other into the console is not very convenient as our analysis becomes more involved. Ideally, we would like to collect all command statements in a file and run them one after the other, automatically. We can do this by writing so-called **script files** or just **scripts**, i.e. simple text files with extension `.R` or `.r` which can be *inserted* (or *sourced*) into an `R` session. RStudio makes this process very easy.

Open `RStudio` by clicking on the `RStudio` application on your computer, and notice how different the whole environment is from the basic `R` console – in fact, that *very same* `R` console is running in your bottom left panel. The upper-left panel is a space for you to write scripts – that is to say many lines of codes which you can run when you choose to. To run a single line of code, simply highlight it and hit `Command` + `Return`.

> We highly recommend that you use `RStudio` for everything related to this course (in particular, to launch our apps and tutorials).

RStudio has a large number of useful keyboard shortcuts. A list of these can be found using a keyboard shortcut – the keyboard shortcut to rule them all:

- On Windows: `Alt` + `Shift` + `K`
- On Mac: `Option` + `Shift` + `K`

The `RStudio` team has developed a number of "cheatsheets" for working with both `R` and `RStudio`. This particular cheatseet for Base `R` will summarize many of the concepts in this document. [1]

1.2.1 First Glossary

- `R`: a statistical programming language
- `RStudio`: an integrated development environment (IDE) to work with `R`
- *command*: user input (text or numbers) that `R` *understands*.
- *script*: a list of commands collected in a text file, each separated by a new line, to be run one after the other.

1.3 Basic Calculations

To get started, we'll use `R` like a simple calculator. Run the following code either directly from your RStudio console, or in RStudio by writting them in a script and running them using `Command` + `Return`.

[1] When programming, it is often a good practice to follow a style guide. (Where do spaces go? Tabs or spaces? Underscores or CamelCase when naming variables?) No style guide is "correct" but it helps to be aware of what others do. The more import thing is to be consistent within your own code. Here are two guides: Hadley Wickham Style Guide, and the Google Style Guide. For this course, our main deviation from these two guides is the use of `=` in place of `<-`. For all practical purposes, you should think `=` whenever you see `<-`.

Addition, Subtraction, Multiplication and Division

Math	R code	Result
$3+2$	`3 + 2`	5
$3-2$	`3 - 2`	1
$3 \cdot 2$	`3 * 2`	6
$3/2$	`3 / 2`	1.5

Exponents

Math	R code	Result
3^2	`3 ^ 2`	9
$2^{(-3)}$	`2 ^ (-3)`	0.125
$100^{1/2}$	`100 ^ (1 / 2)`	10
$\sqrt{100}$	`sqrt(100)`	10

Mathematical Constants

Math	R code	Result
π	`pi`	3.1415927
e	`exp(1)`	2.7182818

Logarithms

Note that we will use ln and log interchangeably to mean the natural logarithm. There is no `ln()` in `R`, instead it uses `log()` to mean the natural logarithm.

Math	R code	Result
$\log(e)$	`log(exp(1))`	1
$\log_{10}(1000)$	`log10(1000)`	3
$\log_2(8)$	`log2(8)`	3
$\log_4(16)$	`log(16, base = 4)`	2

Trigonometry

Math	R code	Result
$\sin(\pi/2)$	`sin(pi / 2)`	1

Math	R code	Result
cos(0)	`cos(0)`	1

1.4 Getting Help

In using `R` as a calculator, we have seen a number of functions: `sqrt()`, `exp()`, `log()` and `sin()`. To get documentation about a function in `R`, simply put a question mark in front of the function name, or call the function `help(function)` and RStudio will display the documentation, for example:

```
?log
?sin
?paste
?lm
help(lm)   # help() is equivalent
help(ggplot,package="ggplot2")  # show help from a certain package
```

Frequently one of the most difficult things to do when learning `R` is asking for help. First, you need to decide to ask for help, then you need to know *how* to ask for help. Your very first line of defense should be to Google your error message or a short description of your issue. (The ability to solve problems using this method is quickly becoming an extremely valuable skill.) If that fails, and it eventually will, you should ask for help. There are a number of things you should include when contacting an instructor, or posting to a help website such as Stack Overflow.

- Describe what you expect the code to do.
- State the end goal you are trying to achieve. (Sometimes what you expect the code to do, is not what you want to actually do.)
- Provide the full text of any errors you have received.
- Provide enough code to recreate the error. Often for the purpose of this course, you could simply post your entire `.R` script or `.Rmd` to `slack`.
- Sometimes it is also helpful to include a screenshot of your entire RStudio window when the error occurs.

If you follow these steps, you will get your issue resolved much quicker, and possibly learn more in the process. Do not be discouraged by running into errors and difficulties when learning `R`. (Or any other technical skill.) It is simply part of the learning process.

1.5 Installing Packages

`R` comes with a number of built-in functions and datasets, but one of the main strengths of `R` as an open-source project is its package system. Packages add additional functions and data. Frequently if you want to do something in R, and it is not available by default, there is a good chance that there is a package that will fulfill your needs.

To install a package, use the install.packages() function. Think of this as buying a recipe book from the store, bringing it home, and putting it on your shelf (i.e. into your library):

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

Once a package is installed, it must be loaded into your current R session before being used. Think of this as taking the book off of the shelf and opening it up to read.

```
library(ggplot2)
```

Once you close R, all the packages are closed and put back on the imaginary shelf. The next time you open R, you do not have to install the package again, but you do have to load any packages you intend to use by invoking library().

1.6 Code vs Output in this Book

A quick note on styling choices in this book. We had to make a decision how to visually separate R code and resulting output in this book. We decided to prefix all output lines with #OUT> to make the distinction. A typical code snippet with output is thus going to look like this:

```
1 + 3
```

```
#OUT> [1] 4
```

```
# everything after a # is a comment, i.e. R disregards it.
```

where you see on the first line the R code, and on the second line the output. As mentioned, that line starts with #OUT> to say *this is an output*, followed by [1] (indicating this is a vector of length *one* - more on this below!), followed by the actual result - 1 + 3 = 4!

Notice that you can simply copy and paste all the code you see into your R console. In fact, you are *strongly* encouraged to actually do this and try out **all the code** you see in this book.

Finally, please note that this way of showing output is fully our choice in this textbook, and that you should expect other output formats elsewhere. For example, in my RStudio console, the above code and output looks like this:

```
> 1 + 3
[1] 4
```

1.7 ScPoEconometrics Package

To fully take advantage of our course, please install the associated `R` package directly from its online code repository. You can do this by copy and pasting the following three lines into your `R` console:

```
if (!require("devtools")) install.packages("devtools")
library(devtools)
install_github(repo = "ScPoEcon/ScPoEconometrics")
```

In order to check whether everything works fine, you could load the library, and check it's current version:

```
library(ScPoEconometrics)
packageVersion("ScPoEconometrics")
```

```
#OUT> [1] '0.2.2'
```

1.8 Data Types

`R` has a number of basic *data types*. While `R` is not a *strongly typed language* (i.e. you can be agnostic about types most of the times), it is useful to know what data types are available to you:

- Numeric
  - Also known as Double. The default type when dealing with numbers.
  - Examples: `1`, `1.0`, `42.5`
- Integer
  - Examples: `1L`, `2L`, `42L`
- Complex
  - Example: `4 + 2i`
- Logical
  - Two possible values: `TRUE` and `FALSE`
  - You can also use `T` and `F`, but this is *not* recommended.
  - `NA` is also considered logical.
- Character
  - Examples: `"a"`, `"Statistics"`, `"1 plus 2."`
- Categorical or `factor`
  - A mixture of integer and character. A `factor` variable assigns a label to a numeric value.
  - For example `factor(x=c(0,1),labels=c("male","female"))` assigns the string *male* to the numeric values `0`, and the string *female* to the value `1`.

1.9 Data Structures

R also has a number of basic data *structures*. A data structure is either homogeneous (all elements are of the same data type) or heterogeneous (elements can be of more than one data type).

Dimension	**Homogeneous**	**Heterogeneous**
1	Vector	List
2	Matrix	Data Frame
3+	Array	nested Lists

1.9.1 Vectors

Many operations in R make heavy use of **vectors**. A vector is a *container* for objects of identical type (see 1.8 above). Vectors in R are indexed starting at 1. That is what the [1] in the output is indicating, that the first element of the row being displayed is the first element of the vector. Larger vectors will start additional rows with something like [7] where 7 is the index of the first element of that row.

Possibly the most common way to create a vector in R is using the c() function, which is short for "combine". As the name suggests, it combines a list of elements separated by commas. (Are you busy typing all of those examples into your R console? :-))

```
c(1, 3, 5, 7, 8, 9)
```

```
#OUT> [1] 1 3 5 7 8 9
```

Here R simply outputs this vector. If we would like to store this vector in a **variable** we can do so with the **assignment** operator =. In this case the variable x now holds the vector we just created, and we can access the vector by typing x.

```
x = c(1, 3, 5, 7, 8, 9)
x
```

```
#OUT> [1] 1 3 5 7 8 9
```

As an aside, there is a long history of the assignment operator in R, partially due to the keys available on the keyboards of the creators of the S language. (Which preceded R.) For simplicity we will use =, but know that often you will see <- as the assignment operator.

Because vectors must contain elements that are all the same type, R will automatically **coerce** (i.e. convert) to a single type when attempting to create a vector that combines multiple types.

```
c(42, "Statistics", TRUE)
```

```
#OUT> [1] "42"         "Statistics" "TRUE"
```

```
c(42, TRUE)
```

```
#OUT> [1] 42  1
```

Frequently you may wish to create a vector based on a sequence of numbers. The quickest and easiest way to do this is with the `:` operator, which creates a sequence of integers between two specified integers.

```
(y = 1:100)
```

```
#OUT>   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
#OUT>  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
#OUT>  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
#OUT>  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
#OUT>  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
#OUT>  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Here we see `R` labeling the rows after the first since this is a large vector. Also, we see that by putting parentheses around the assignment, `R` both stores the vector in a variable called `y` and automatically outputs `y` to the console.

Note that scalars do not exists in `R`. They are simply vectors of length `1`.

```
2
```

```
#OUT> [1] 2
```

If we want to create a sequence that isn't limited to integers and increasing by 1 at a time, we can use the `seq()` function.

```
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
#OUT>  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
#OUT> [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

We will discuss functions in detail later, but note here that the input labels `from`, `to`, and `by` are optional.

```
seq(1.5, 4.2, 0.1)
```

```
#OUT>  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
#OUT> [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

Another common operation to create a vector is `rep()`, which can repeat a single value a number of times.

```
rep("A", times = 10)
```

```
#OUT>  [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"
```

The `rep()` function can be used to repeat a vector some number of times.

```
rep(x, times = 3)
```

```
#OUT>  [1] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9
```

We have now seen four different ways to create vectors:

- `c()`
- `:`
- `seq()`
- `rep()`

So far we have mostly used them in isolation, but they are often used together.

```
c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
```

```
#OUT>  [1]  1  3  5  7  8  9  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9  1  2
#OUT> [24]  3 42  2  3  4
```

The length of a vector can be obtained with the `length()` function.

```
length(x)
```

```
#OUT> [1] 6
```

```
length(y)
```

```
#OUT> [1] 100
```

Let's try this out! **Your turn**:

1.9.1.1 Task 1

1. Create a vector of five ones, i.e. `[1,1,1,1,1]`
2. Notice that the colon operator `a:b` is just short for *construct a sequence **from** a **to** b*. Create a vector the counts down from 10 to 0, i.e. it looks like `[10,9,8,7,6,5,4,3,2,1,0]`!
3. the `rep` function takes additional arguments `times` (as above), and `each`, which tells you how often *each element* should be repeated (as opposed to the entire input vector). Use `rep` to create a vector that looks like this: `[1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3]`

1.9.1.2 Subsetting

To subset a vector, i.e. to choose only some elements of it, we use square brackets, `[]`. Here we see that `x[1]` returns the first element, and `x[3]` returns the third element:

```
x
```

```
#OUT> [1] 1 3 5 7 8 9
```

```
x[1]
```

```
#OUT> [1] 1
```

```
x[3]
```

```
#OUT> [1] 5
```

We can also exclude certain indexes, in this case the second element.

```
x[-2]
```

```
#OUT> [1] 1 5 7 8 9
```

Lastly we see that we can subset based on a vector of indices.

```
x[1:3]
```

```
#OUT> [1] 1 3 5
```

```
x[c(1,3,4)]
```

```
#OUT> [1] 1 5 7
```

All of the above are subsetting a vector using a vector of indexes. (Remember a single number is still a vector.) We could instead use a vector of logical values.

```
z = c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE)
z
```

```
#OUT> [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

```
x[z]
```

```
#OUT> [1] 1 3 7 8
```

`R` is able to perform many operations on vectors and scalars alike:

```
x = 1:10  # a vector
x + 1     # add a scalar
```

```
#OUT>  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x     # multiply all elements by 2
```

```
#OUT>  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2 ^ x     # take 2 to the x as exponents
```

```
#OUT>  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)   # compute the square root of all elements in x
```

```
#OUT>  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
#OUT>  [8] 2.828427 3.000000 3.162278
```

```
log(x)    # take the natural log of all elements in x
```

```
#OUT>  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
#OUT>  [8] 2.0794415 2.1972246 2.3025851
```

```
x + 2*x   # add vector x to vector 2x
```

```
#OUT>  [1]  3  6  9 12 15 18 21 24 27 30
```

We see that when a function like `log()` is called on a vector `x`, a vector is returned which has applied the function to each element of the vector `x`.

1.9.2 Logical Operators

Operator	Summary	Example	Result
`x < y`	`x` less than `y`	`3 < 42`	TRUE
`x > y`	`x` greater than `y`	`3 > 42`	FALSE
`x <= y`	`x` less than or equal to `y`	`3 <= 42`	TRUE
`x >= y`	`x` greater than or equal to `y`	`3 >= 42`	FALSE
`x == y`	`x`equal to `y`	`3 == 42`	FALSE
`x != y`	`x` not equal to `y`	`3 != 42`	TRUE
`!x`	not `x`	`!(3 > 42)`	TRUE
`x | y`	`x` or `y`	`(3 > 42) | TRUE`	TRUE
`x & y`	`x` and `y`	`(3 < 4) & ( 42 > 13)`	TRUE

In R, logical operators also work on vectors:

```
x = c(1, 3, 5, 7, 8, 9)
```

```
x > 3
```

```
#OUT> [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
x < 3
```

```
#OUT> [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
x == 3
```

```
#OUT> [1] FALSE  TRUE FALSE FALSE FALSE FALSE
```

```
x != 3
```

```
#OUT> [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
x == 3 & x != 3
```

```
#OUT> [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
x == 3 | x != 3
```

```
#OUT> [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

This is quite useful for subsetting.

```
x[x > 3]
```

```
#OUT> [1] 5 7 8 9
```

```
x[x != 3]
```

```
#OUT> [1] 1 5 7 8 9
```

```
sum(x > 3)
```

```
#OUT> [1] 4
```

```
as.numeric(x > 3)
```

```
#OUT> [1] 0 0 1 1 1 1
```

Here we saw that using the `sum()` function on a vector of logical `TRUE` and `FALSE` values that is the result of `x > 3` results in a numeric result: you just *counted* for how many elements of `x`, the condition `> 3` is `TRUE`. During the call to `sum()`, `R` is first automatically coercing the logical to numeric where `TRUE` is `1` and `FALSE` is `0`. This coercion from logical to numeric happens for most mathematical operations.

```
# which(condition of x) returns true/false
# each index of x where condition is true
which(x > 3)
```

```
#OUT> [1] 3 4 5 6
```

```
x[which(x > 3)]
```

```
#OUT> [1] 5 7 8 9
```

```
max(x)
```

```
#OUT> [1] 9
```

```
which(x == max(x))
```

```
#OUT> [1] 6
```

```
which.max(x)
```

```
#OUT> [1] 6
```

1.9.2.1 Task 2

1. Create a vector filled with 10 numbers drawn from the uniform distribution (hint: use function `runif`) and store them in `x`.
2. Using logical subsetting as above, get all the elements of `x` which are larger than 0.5, and store them in `y`.
3. using the function `which`, store the *indices* of all the elements of `x` which are larger than 0.5 in `iy`.
4. Check that `y` and `x[iy]` are identical.

1.9.3 Matrices

`R` can also be used for **matrix** calculations. Matrices have rows and columns containing a single data type. In a matrix, the order of rows and columns is important. (This is not true of *data frames*, which we will see later.)

Matrices can be created using the `matrix` function.

```
x = 1:9
x
```

```
#OUT> [1] 1 2 3 4 5 6 7 8 9
```

```
X = matrix(x, nrow = 3, ncol = 3)
X
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    1    4    7
#OUT> [2,]    2    5    8
#OUT> [3,]    3    6    9
```

Notice here that `R` is case sensitive (`x` vs `X`).

By default the `matrix` function fills your data into the matrix column by column. But we can also tell `R` to fill rows instead:

```
Y = matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    1    2    3
#OUT> [2,]    4    5    6
#OUT> [3,]    7    8    9
```

We can also create a matrix of a specified dimension where every element is the same, in this case `0`.

```
Z = matrix(0, 2, 4)
Z
```

```
#OUT>      [,1] [,2] [,3] [,4]
#OUT> [1,]    0    0    0    0
#OUT> [2,]    0    0    0    0
```

Like vectors, matrices can be subsetted using square brackets, `[]`. However, since matrices are two-dimensional, we need to specify both a row and a column when subsetting.

```
X
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    1    4    7
#OUT> [2,]    2    5    8
#OUT> [3,]    3    6    9
```

```
X[1, 2]
```

```
#OUT> [1] 4
```

Here we accessed the element in the first row and the second column. We could also subset an entire row or column.

```
X[1, ]
```

```
#OUT> [1] 1 4 7
```

```
X[, 2]
```

```
#OUT> [1] 4 5 6
```

We can also use vectors to subset more than one row or column at a time. Here we subset to the first and third column of the second row:

```
X[2, c(1, 3)]
```

```
#OUT> [1] 2 8
```

Matrices can also be created by combining vectors as columns, using `cbind`, or combining vectors as rows, using `rbind`.

```
x = 1:9
rev(x)
```

```
#OUT> [1] 9 8 7 6 5 4 3 2 1
```

```
rep(1, 9)
```

```
#OUT> [1] 1 1 1 1 1 1 1 1 1
```

```
rbind(x, rev(x), rep(1, 9))
```

```
#OUT>   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
#OUT> x    1    2    3    4    5    6    7    8    9
#OUT>      9    8    7    6    5    4    3    2    1
```

```
#OUT>      1    1    1    1    1    1    1    1    1
```

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 9))
```

```
#OUT>       col_1 col_2 col_3
#OUT>  [1,]     1     9     1
#OUT>  [2,]     2     8     1
#OUT>  [3,]     3     7     1
#OUT>  [4,]     4     6     1
#OUT>  [5,]     5     5     1
#OUT>  [6,]     6     4     1
#OUT>  [7,]     7     3     1
#OUT>  [8,]     8     2     1
#OUT>  [9,]     9     1     1
```

When using `rbind` and `cbind` you can specify "argument" names that will be used as column names.

`R` can then be used to perform matrix calculations.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    1    4    7
#OUT> [2,]    2    5    8
#OUT> [3,]    3    6    9
```

```
Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    9    6    3
#OUT> [2,]    8    5    2
#OUT> [3,]    7    4    1
```

```
X + Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]   10   10   10
#OUT> [2,]   10   10   10
#OUT> [3,]   10   10   10
```

```
X - Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]   -8   -2    4
#OUT> [2,]   -6    0    6
#OUT> [3,]   -4    2    8
```

```
X * Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    9   24   21
#OUT> [2,]   16   25   16
#OUT> [3,]   21   24    9
```

```
X / Y
```

```
#OUT>           [,1]      [,2]     [,3]
#OUT> [1,] 0.1111111 0.6666667 2.333333
#OUT> [2,] 0.2500000 1.0000000 4.000000
#OUT> [3,] 0.4285714 1.5000000 9.000000
```

Note that `X * Y` is **not** matrix multiplication. It is *element by element* multiplication. (Same for `X / Y`). Matrix multiplication uses `%*%`. Other matrix functions include `t()` which gives the transpose of a matrix and `solve()` which returns the inverse of a square matrix if it is invertible.

```
X %*% Y
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]   90   54   18
#OUT> [2,]  114   69   24
#OUT> [3,]  138   84   30
```

```
t(X)
```

```
#OUT>      [,1] [,2] [,3]
#OUT> [1,]    1    2    3
#OUT> [2,]    4    5    6
#OUT> [3,]    7    8    9
```

1.9.4 Arrays

A vector is a one-dimensional array. A matrix is a two-dimensional array. In `R` you can create arrays of arbitrary dimensionality `N`. Here is how:

```
d = 1:16
d3 = array(data = d,dim = c(4,2,2))
d4 = array(data = d,dim = c(4,2,2,3))  # will recycle 1:16
d3
```

```
#OUT> , , 1
#OUT>
#OUT>      [,1] [,2]
#OUT> [1,]    1    5
#OUT> [2,]    2    6
#OUT> [3,]    3    7
```

```
#OUT> [4,]    4    8
#OUT> 
#OUT> , , 2
#OUT> 
#OUT>      [,1] [,2]
#OUT> [1,]    9   13
#OUT> [2,]   10   14
#OUT> [3,]   11   15
#OUT> [4,]   12   16
```

You can see that `d3` are simply *two* (4,2) matrices laid on top of each other, as if there were *two pages*. Similary, `d4` would have two pages, and another 3 registers in a fourth dimension. And so on. You can subset an array like you would a vector or a matrix, taking care to index each dimension:

```
d3[ ,1,1]  # all elements from col 1, page 1
```

```
#OUT> [1] 1 2 3 4
```

```
d3[2:3, , ]  # rows 2:3 from all pages
```

```
#OUT> , , 1
#OUT> 
#OUT>      [,1] [,2]
#OUT> [1,]    2    6
#OUT> [2,]    3    7
#OUT> 
#OUT> , , 2
#OUT> 
#OUT>      [,1] [,2]
#OUT> [1,]   10   14
#OUT> [2,]   11   15
```

```
d3[2,2, ]  # row 2, col 2 from both pages.
```

```
#OUT> [1]  6 14
```

1.9.4.1 Task 3

1. Create a vector containing `1,2,3,4,5` called v.
2. Create a (2,5) matrix `m` containing the data `1,2,3,4,5,6,7,8,9,10`. The first row should be `1,2,3,4,5`.
3. Perform matrix multiplication of `m` with `v`. Use the command `%*%`. What dimension does the output have?
4. Why does `v %*% m` not work?

1.9.5 Lists

A list is a one-dimensional *heterogeneous* data structure. So it is indexed like a vector with a single integer value (or with a name), but each element can contain an element of any type. Lists are similar to a python or julia `Dict` object. Many `R` structures and outputs are lists themselves. Lists are extremely useful and versatile objects, so make sure you understand their useage:

```
# creation without fieldnames
list(42, "Hello", TRUE)
```

```
#OUT> [[1]]
#OUT> [1] 42
#OUT>
#OUT> [[2]]
#OUT> [1] "Hello"
#OUT>
#OUT> [[3]]
#OUT> [1] TRUE
```

```
# creation with fieldnames
ex_list = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 42) {print("Hello World!")},
  e = diag(5)
)
```

Lists can be subset using two syntaxes, the `$` operator, and square brackets `[]`. The `$` operator returns a named **element** of a list. The `[]` syntax returns a **list**, while the `[[]]` returns an **element** of a list.

- `ex_list[1]` returns a list contain the first element.
- `ex_list[[1]]` returns the first element of the list, in this case, a vector.

```
# subsetting
ex_list$e
```

```
#OUT>      [,1] [,2] [,3] [,4] [,5]
#OUT> [1,]    1    0    0    0    0
#OUT> [2,]    0    1    0    0    0
#OUT> [3,]    0    0    1    0    0
#OUT> [4,]    0    0    0    1    0
#OUT> [5,]    0    0    0    0    1
```

```
ex_list[1:2]
```

```
#OUT> $a
```

```
#OUT> [1] 1 2 3 4
#OUT>
#OUT> $b
#OUT> [1] TRUE
```

```
ex_list[1]
```

```
#OUT> $a
#OUT> [1] 1 2 3 4
```

```
ex_list[[1]]
```

```
#OUT> [1] 1 2 3 4
```

```
ex_list[c("e", "a")]
```

```
#OUT> $e
#OUT>      [,1] [,2] [,3] [,4] [,5]
#OUT> [1,]    1    0    0    0    0
#OUT> [2,]    0    1    0    0    0
#OUT> [3,]    0    0    1    0    0
#OUT> [4,]    0    0    0    1    0
#OUT> [5,]    0    0    0    0    1
#OUT>
#OUT> $a
#OUT> [1] 1 2 3 4
```

```
ex_list["e"]
```

```
#OUT> $e
#OUT>      [,1] [,2] [,3] [,4] [,5]
#OUT> [1,]    1    0    0    0    0
#OUT> [2,]    0    1    0    0    0
#OUT> [3,]    0    0    1    0    0
#OUT> [4,]    0    0    0    1    0
#OUT> [5,]    0    0    0    0    1
```

```
ex_list[["e"]]
```

```
#OUT>      [,1] [,2] [,3] [,4] [,5]
#OUT> [1,]    1    0    0    0    0
#OUT> [2,]    0    1    0    0    0
#OUT> [3,]    0    0    1    0    0
#OUT> [4,]    0    0    0    1    0
#OUT> [5,]    0    0    0    0    1
```

```
ex_list$d
```

```
#OUT> function(arg = 42) {print("Hello World!")}
```

```
ex_list$d(arg = 1)
```

```
#OUT> [1] "Hello World!"
```

1.9.5.1 Task 4

1. Copy and paste the above code for `ex_list` into your R session. Remember that `list` can hold any kind of `R` object. Like…another list! So, create a new list `new_list` that has two fields: a first field called "this" with string content `"is awesome"`, and a second field called "ex_list" that contains `ex_list`.
2. Accessing members is like in a plain list, just with several layers now. Get the element `c` from `ex_list` in `new_list`!
3. Compose a new string out of the first element in `new_list`, the element under label `this`. Use the function `paste` to print `R is awesome` to your screen.

1.10 Data Frames

We have previously seen vectors and matrices for storing data as we introduced `R`. We will now introduce a **data frame** which will be the most common way that we store and interact with data in this course. A `data.frame` is similar to a python `pandas.dataframe` or a julia `DataFrame`. (But the `R` version was the first! :-))

```
example_data = data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                          y = c(rep("Hello", 9), "Goodbye"),
                          z = rep(c(TRUE, FALSE), 5))
```

Unlike a matrix, which can be thought of as a vector rearranged into rows and columns, a data frame is not required to have the same data type for each element. A data frame is a **list** of vectors, and each vector has a *name*. So, each vector must contain the same data type, but the different vectors can store different data types. Note, however, that all vectors must have **the same length** (differently from a `list`)!

> A **data.frame** is similar to a typical Spreadsheet. There are *rows*, and there are *columns*. A row is typically thought of as an *observation*, and each column is a certain *variable*, *characteristic* or *feature* of that observation.

Let's look at the data frame we just created above:

```
example_data
```

```
#OUT>    x       y     z
#OUT> 1  1   Hello  TRUE
#OUT> 2  3   Hello FALSE
#OUT> 3  5   Hello  TRUE
#OUT> 4  7   Hello FALSE
#OUT> 5  9   Hello  TRUE
#OUT> 6  1   Hello FALSE
#OUT> 7  3   Hello  TRUE
#OUT> 8  5   Hello FALSE
#OUT> 9  7   Hello  TRUE
#OUT> 10 9 Goodbye FALSE
```

Unlike a list, which has more flexibility, the elements of a data frame must all be vectors. Again, we access any given column with the `$` operator:

```
example_data$x
```

```
#OUT>  [1] 1 3 5 7 9 1 3 5 7 9
```

```
all.equal(length(example_data$x),
          length(example_data$y),
          length(example_data$z))
```

```
#OUT> [1] TRUE
```

```
str(example_data)
```

```
#OUT> 'data.frame': 10 obs. of  3 variables:
#OUT>  $ x: num  1 3 5 7 9 1 3 5 7 9
#OUT>  $ y: Factor w/ 2 levels "Goodbye","Hello": 2 2 2 2 2 2 2 2 2 1
#OUT>  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

```
nrow(example_data)
```

```
#OUT> [1] 10
```

```
ncol(example_data)
```

```
#OUT> [1] 3
```

```
dim(example_data)
```

```
#OUT> [1] 10  3
```

```
names(example_data)
```

```
#OUT> [1] "x" "y" "z"
```

1.10.1 Working with data.frames

The `data.frame()` function above is one way to create a data frame. We can also import data from various file types in into `R`, as well as use data stored in packages.

To read this data back into `R`, we will use the built-in function `read.csv`:

```
path = system.file(package="ScPoEconometrics","datasets","example-data.csv")
example_data_from_disk = read.csv(path)
```

This particular line of code assumes that you installed the associated R package to this book, hence you have this dataset stored on your computer at `system.file(package = "ScPoEconometrics","datasets","example-data.csv")`.

```
example_data_from_disk
```

```
#OUT>    x       y     z
#OUT> 1  1   Hello  TRUE
#OUT> 2  3   Hello FALSE
#OUT> 3  5   Hello  TRUE
#OUT> 4  7   Hello FALSE
#OUT> 5  9   Hello  TRUE
#OUT> 6  1   Hello FALSE
#OUT> 7  3   Hello  TRUE
#OUT> 8  5   Hello FALSE
#OUT> 9  7   Hello  TRUE
#OUT> 10 9 Goodbye FALSE
```

When using data, there are three things we would generally like to do:

- Look at the raw data.
- Understand the data. (Where did it come from? What are the variables? Etc.)
- Visualize the data.

To look at data in a `data.frame`, we have two useful commands: `head()` and `str()`.

```
# we are working with the built-in mtcars dataset:
mtcars
```

```
#OUT>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#OUT> Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#OUT> Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#OUT> Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#OUT> Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#OUT> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#OUT> Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#OUT> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
```

```
#OUT> Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#OUT> Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#OUT> Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#OUT> Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#OUT> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#OUT> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#OUT> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#OUT> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#OUT> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#OUT> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#OUT> Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#OUT> Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#OUT> Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#OUT> Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#OUT> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#OUT> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#OUT> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#OUT> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#OUT> Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#OUT> Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#OUT> Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#OUT> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#OUT> Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
#OUT> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
#OUT> Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

You can see that this prints the entire data.frame to screen. The function `head()` will display the first `n` observations of the data frame.

```
head(mtcars,n=2)
```

```
#OUT>               mpg cyl disp  hp drat    wt  qsec vs am gear carb
#OUT> Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
#OUT> Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
head(mtcars) # default
```

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

The function `str()` will display the “structure” of the data frame. It will display the number of **observations** and **variables**, list the variables, give the type of each variable, and show some elements of each variable. This information can also be found in the “Environment” window in RStudio.

```
str(mtcars)
```

```
#OUT> 'data.frame':	32 obs. of  11 variables:
#OUT>  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#OUT>  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#OUT>  $ disp: num  160 160 108 258 360 ...
#OUT>  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#OUT>  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#OUT>  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#OUT>  $ qsec: num  16.5 17 18.6 19.4 17 ...
#OUT>  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
#OUT>  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
#OUT>  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#OUT>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

In this dataset an observation is for a particular model of a car, and the variables describe attributes of the car, for example its fuel efficiency, or its weight.

To understand more about the data set, we use the `?` operator to pull up the documentation for the data.

```
?mtcars
```

`R` has a number of functions for quickly working with and extracting basic information from data frames. To quickly obtain a vector of the variable names, we use the `names()` function.

```
names(mtcars)
```

```
#OUT>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#OUT> [11] "carb"
```

To access one of the variables **as a vector**, we use the `$` operator.

```
mtcars$mpg
```

```
#OUT>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
#OUT> [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
#OUT> [29] 15.8 19.7 15.0 21.4
```

```
mtcars$wt
```

```
#OUT>  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
#OUT> [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520
#OUT> [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

We can use the `dim()`, `nrow()` and `ncol()` functions to obtain information about the dimension of the data frame.

```
dim(mtcars)
```

```
#OUT> [1] 32 11
```

```
nrow(mtcars)
```

```
#OUT> [1] 32
```

```
ncol(mtcars)
```

```
#OUT> [1] 11
```

Here `nrow()` is also the number of observations, which in most cases is the *sample size*.

Subsetting data frames can work much like subsetting matrices using square brackets, `[ , ]`. Here, we find vehicles with mpg over 25 miles per gallon and only display columns `cyl`, `disp` and `wt`.

```
# mpg[row condition, col condition]
mtcars[mtcars$mpg > 20, c("cyl", "disp", "wt")]
```

```
#OUT>                cyl  disp    wt
#OUT> Mazda RX4        6 160.0 2.620
#OUT> Mazda RX4 Wag    6 160.0 2.875
#OUT> Datsun 710       4 108.0 2.320
#OUT> Hornet 4 Drive   6 258.0 3.215
#OUT> Merc 240D        4 146.7 3.190
#OUT> Merc 230         4 140.8 3.150
#OUT> Fiat 128         4  78.7 2.200
#OUT> Honda Civic      4  75.7 1.615
#OUT> Toyota Corolla   4  71.1 1.835
#OUT> Toyota Corona    4 120.1 2.465
#OUT> Fiat X1-9        4  79.0 1.935
#OUT> Porsche 914-2    4 120.3 2.140
#OUT> Lotus Europa     4  95.1 1.513
#OUT> Volvo 142E       4 121.0 2.780
```

An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mtcars, subset = mpg > 25, select = c("cyl", "disp", "wt"))
```

1.10.1.1 Task 5

1. How many observations are there in `mtcars`?
2. How many variables?
3. What is the average value of `mpg`?

4. What is the average value of mpg for cars with more than 4 cylinders, i.e. with cyl>4?

1.11 Programming Basics

In this section we illustrate some general concepts related to programming.

1.11.1 Variables

We encountered the term *variable* already several times, but mainly in the context of a column of a data.frame. In programming, a variable is denotes an *object*. Another way to say it is that a variable is a name or a *label* for something:

```
x = 1
y = "roses"
z = function(x){sqrt(x)}
```

Here x refers to the value 1, y holds the string “roses”, and z is the name of a function that computes $\sqrt{x}$. Notice that the argument x of the function is different from the x we just defined. It is **local** to the function:

```
x
```

```
#OUT> [1] 1
```

```
z(9)
```

```
#OUT> [1] 3
```

1.11.2 Control Flow

Control Flow relates to ways in which you can adapt your code to different circumstances. Based on a condition being TRUE, your program will do one thing, as opposed to another thing. This is most widely known as an if/else statement. In R, the if/else syntax is:

```
if (condition = TRUE) {
  some R code
} else {
  some other R code
}
```

For example,

```
x = 1
y = 3
```

```
if (x > y) {  # test if x > y
  # if TRUE
  z = x * y
  print("x is larger than y")
} else {
  # if FALSE
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

```
#OUT> [1] "x is less than or equal to y"
```

```
z
```

```
#OUT> [1] 16
```

1.11.3 Loops

Loops are a very important programming construct. As the name suggests, in a *loop*, the programming *repeatedly* loops over a set of instructions, until some condition tells it to stop. A very powerful, yet simple, construction is that the program can *count how many steps* it has done already - which may be important to know for many algorithms. The syntax of a `for` loop (there are others), is

```
for (ix in 1:10){   # does not have to be 1:10!
  # loop body: gets executed each time
  # the value of ix changes with each iteration
}
```

For example, consider this simple `for` loop, which will simply print the value of the *iterator* (called `i` in this case) to screen:

```
for (i in 1:5){
  print(i)
}
```

```
#OUT> [1] 1
#OUT> [1] 2
#OUT> [1] 3
#OUT> [1] 4
#OUT> [1] 5
```

Notice that instead of `1:5`, we could have *any* kind of iterable collection:

```
for (i in c("mangos","bananas","apples")){
  print(paste("I love",i))  # the paste function pastes together strings
}
```

```
#OUT> [1] "I love mangos"
#OUT> [1] "I love bananas"
#OUT> [1] "I love apples"
```

We often also see *nested* loops, which are just what its name suggests:

```
for (i in 2:3){
  # first nest: for each i
  for (j in c("mangos","bananas","apples")){
    # second nest: for each j
    print(paste("Can I get",i,j,"please?"))
  }
}
```

```
#OUT> [1] "Can I get 2 mangos please?"
#OUT> [1] "Can I get 2 bananas please?"
#OUT> [1] "Can I get 2 apples please?"
#OUT> [1] "Can I get 3 mangos please?"
#OUT> [1] "Can I get 3 bananas please?"
#OUT> [1] "Can I get 3 apples please?"
```

The important thing to note here is that you can do calculations with the iterators *while inside a loop*.

1.11.4 Functions

So far we have been using functions, but haven't actually discussed some of their details. A function is a set of instructions that R executes for us, much like those collected in a script file. The good thing is that functions are much more flexible than scripts, since they can depend on *input arguments*, which change the way the function behaves. Here is how to define a function:

```
function_name <- function(arg1,arg2=default_value){
  # function body
  # you do stuff with arg1 and arg2
  # you can have any number of arguments, with or without defaults
  # any valid `R` commands can be included here
  # the last line is returned
}
```

And here is a trivial example of a function definition:

```
hello <- function(your_name = "Lord Vader"){
  paste("You R most welcome,",your_name)
  # we could also write:
  # return(paste("You R most welcome,",your_name))
}
```

```
# we call the function by typing it's name with round brackets
hello()
```

```
#OUT> [1] "You R most welcome, Lord Vader"
```

You see that by not specifying the argument `your_name`, `R` reverts to the default value given. Try with your own name now!

Just typing the function name returns the actual definition to us, which is handy sometimes:

```
hello
```

```
#OUT> function(your_name = "Lord Vader"){
#OUT>   paste("You R most welcome,",your_name)
#OUT>   # we could also write:
#OUT>   # return(paste("You R most welcome,",your_name))
#OUT> }
```

It's instructive to consider that before we defined the function `hello` above, `R` did not know what to do, had you called `hello()`. The function did not exist! In this sense, we *taught* `R` *a new trick*. This feature to create new capabilities on top of a core language is one of the most powerful characteristics of programming languages. In general, it is good practice to split your code into several smaller functions, rather than one long script file. It makes your code more readable, and it is easier to track down mistakes.

1.11.4.1 Task 6

1. Write a for loop that counts down from 10 to 1, printing the value of the iterator to the screen.
2. Modify that loop to write "i iterations to go" where `i` is the iterator
3. Modify that loop so that each iteration takes roughly one second. You can achieve that by adding the command `Sys.sleep(1)` below the line that prints "i iterations to go".

Chapter 2

Working With Data

In this chapter we will first learn some basic concepts that help summarizing data. Then, we will tackle a real-world task and read, clean, and summarize data from the web.

2.1 Summary Statistics

`R` has built in functions for a large number of summary statistics. For numeric variables, we can summarize data by looking at their center and spread, for example.

```r
# for the mpg dataset, we load:
library(ggplot2)
```

Central Tendency

Suppose we want to know the *mean* and *median* of all the values stored in the `data.frame` column `mpg$cty`:

Measure	`R`	Result
Mean	`mean(mpg$cty)`	16.8589744
Median	`median(mpg$cty)`	17

Spread

How do the values in that column *vary*? How far *spread out* are they?

Measure	R	Result
Variance	`var(mpg$cty)`	18.1130736
Standard Deviation	`sd(mpg$cty)`	4.2559457
IQR	`IQR(mpg$cty)`	5
Minimum	`min(mpg$cty)`	9
Maximum	`max(mpg$cty)`	35
Range	`range(mpg$cty)`	9, 35

Categorical

For categorical variables, counts and percentages can be used for summary.

```r
table(mpg$drv)
```

```
#OUT> 
#OUT>   4   f   r 
#OUT> 103 106  25
```

```r
table(mpg$drv) / nrow(mpg)
```

```
#OUT> 
#OUT>         4         f         r 
#OUT> 0.4401709 0.4529915 0.1068376
```

2.2 Plotting

Now that we have some data to work with, and we have learned about the data at the most basic level, our next tasks will be to visualize it. Often, a proper visualization can illuminate features of the data that can inform further analysis.

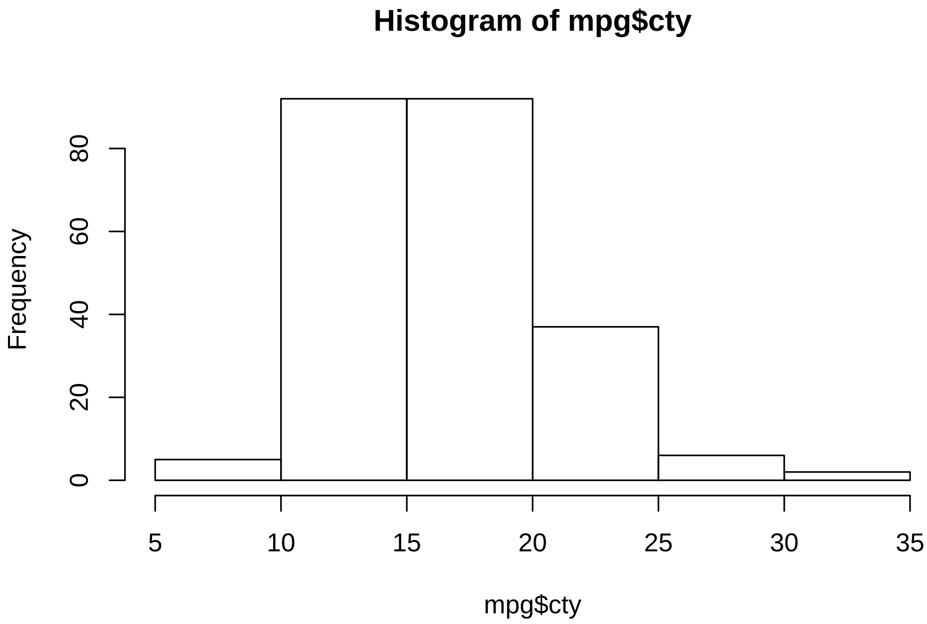
We will look at four methods of visualizing data by using the basic `plot` facilities built-in with `R`:

- Histograms
- Barplots
- Boxplots
- Scatterplots

2.2.1 Histograms

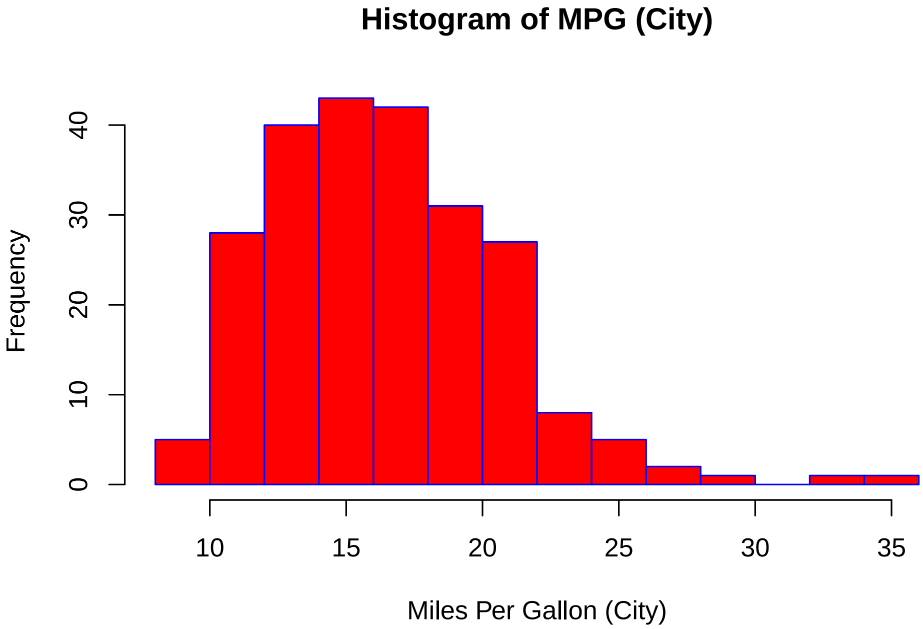
When visualizing a single numerical variable, a **histogram** is useful. It summarizes the *distribution* of values in a vector. In `R` you create one using the `hist()` function:

```
hist(mpg$cty)
```



The histogram function has a number of parameters which can be changed to make our plot look much nicer. Use the `?` operator to read the documentation for the `hist()` to see a full list of these parameters.

```
hist(mpg$cty,
     xlab   = "Miles Per Gallon (City)",
     main   = "Histogram of MPG (City)", # main title
     breaks = 12,   # how many breaks?
     col    = "red",
     border = "blue")
```

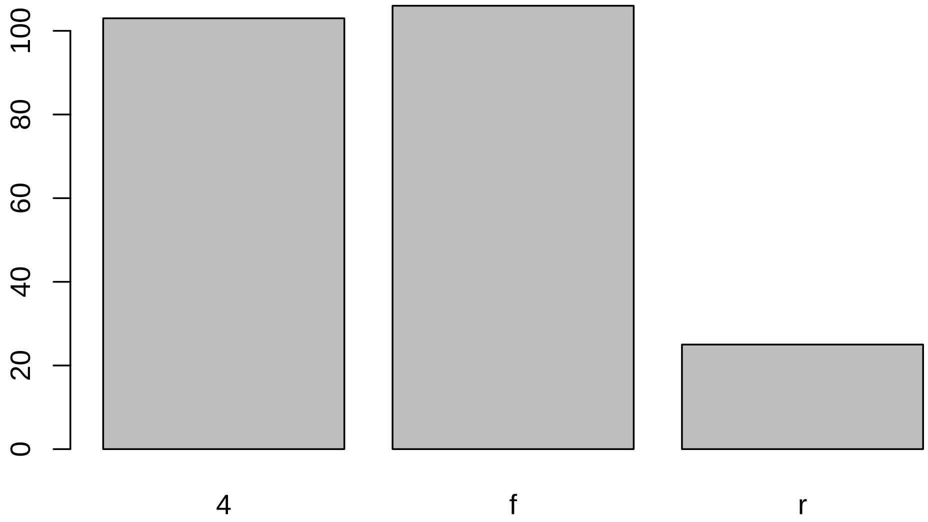


Importantly, you should always be sure to label your axes and give the plot a title. The argument `breaks` is specific to `hist()`. Entering an integer will give a suggestion to `R` for how many bars to use for the histogram. By default `R` will attempt to intelligently guess a good number of `breaks`, but as we can see here, it is sometimes useful to modify this yourself.

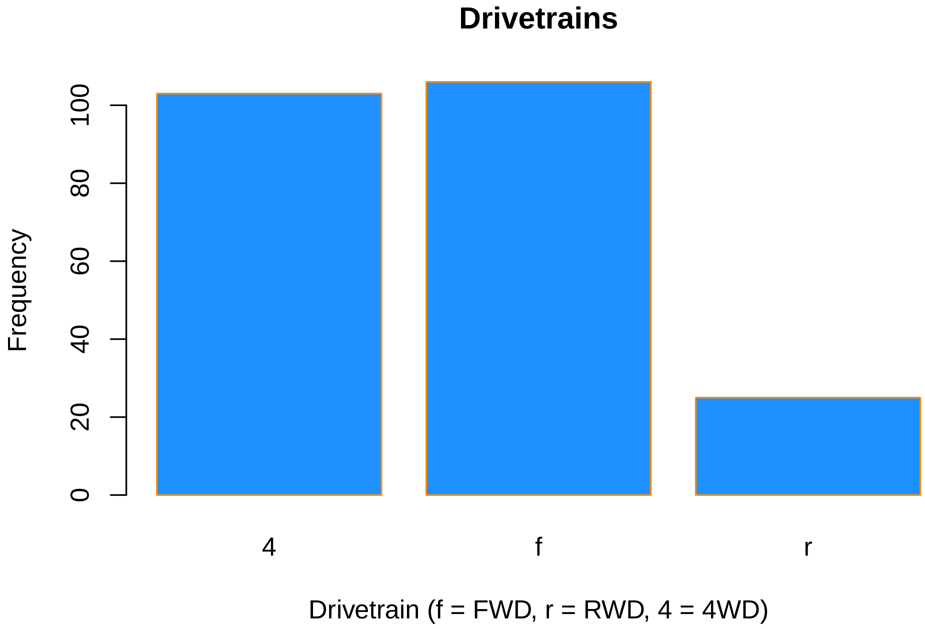
2.2.2 Barplots

Somewhat similar to a histogram, a barplot can provide a visual summary of a categorical variable, or a numeric variable with a finite number of values, like a ranking from 1 to 10.

```
barplot(table(mpg$drv))
```



```r
barplot(table(mpg$drv),
        xlab   = "Drivetrain (f = FWD, r = RWD, 4 = 4WD)",
        ylab   = "Frequency",
        main   = "Drivetrains",
        col    = "dodgerblue",
        border = "darkorange")
```



2.2.3 Boxplots

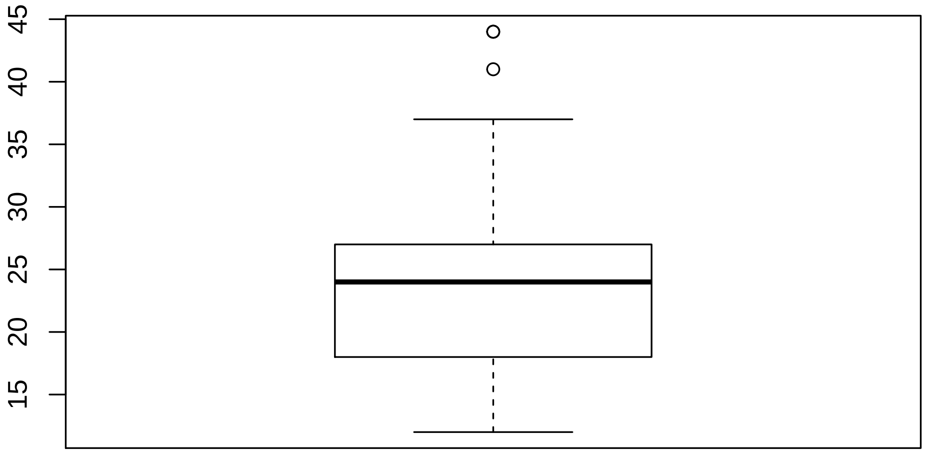
To visualize the relationship between a numerical and categorical variable, once could use a **boxplot**. In the `mpg` dataset, the `drv` variable takes a small, finite number of values. A car can only be front wheel drive, 4 wheel drive, or rear wheel drive.

```
unique(mpg$drv)
```

```
#OUT> [1] "f" "4" "r"
```

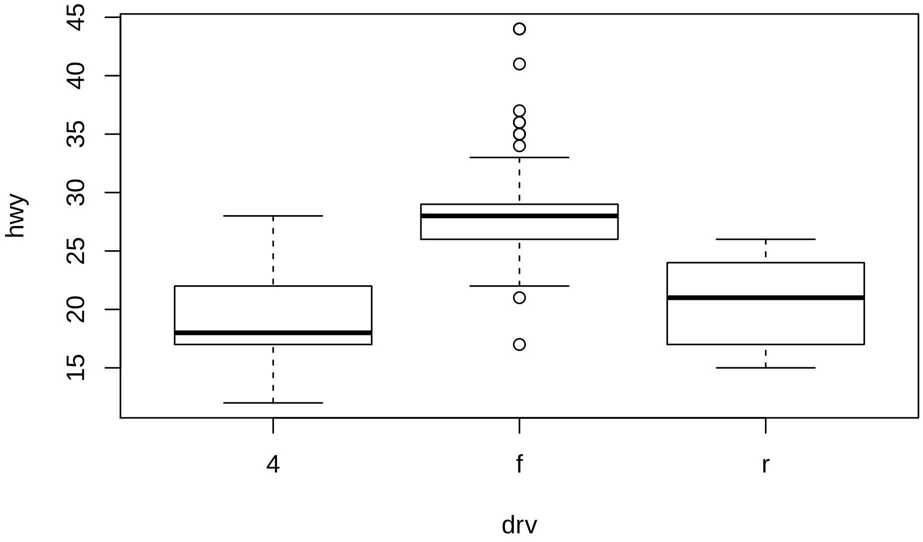
First note that we can use a single boxplot as an alternative to a histogram for visualizing a single numerical variable. To do so in `R`, we use the `boxplot()` function. The box shows the *interquartile range*, the solid line in the middle is the value of the median, the wiskers show 1.5 times the interquartile range, and the dots are outliers.

```
boxplot(mpg$hwy)
```



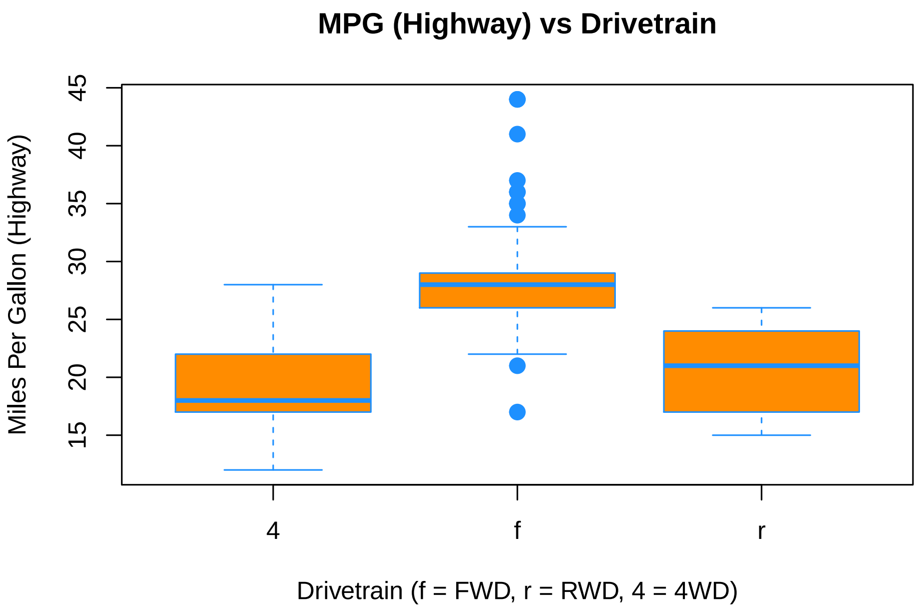
However, more often we will use boxplots to compare a numerical variable for different values of a categorical variable.

```
boxplot(hwy ~ drv, data = mpg)
```



Here used the `boxplot()` command to create side-by-side boxplots. However, since we are now dealing with two variables, the syntax has changed. The `R` syntax `hwy ~ drv, data = mpg` reads "Plot the `hwy` variable against the `drv` variable using the dataset `mpg`." We see the use of a `~` (which specifies a formula) and also a `data =` argument. This will be a syntax that is common to many functions we will use in this course.

```r
boxplot(hwy ~ drv, data = mpg,
     xlab   = "Drivetrain (f = FWD, r = RWD, 4 = 4WD)",
     ylab   = "Miles Per Gallon (Highway)",
     main   = "MPG (Highway) vs Drivetrain",
     pch    = 20,
     cex    = 2,
     col    = "darkorange",
     border = "dodgerblue")
```

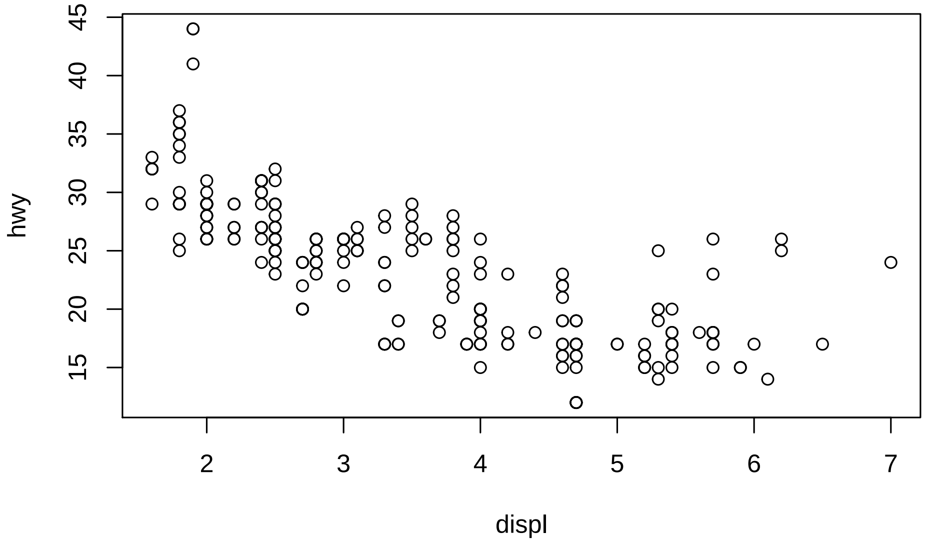


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

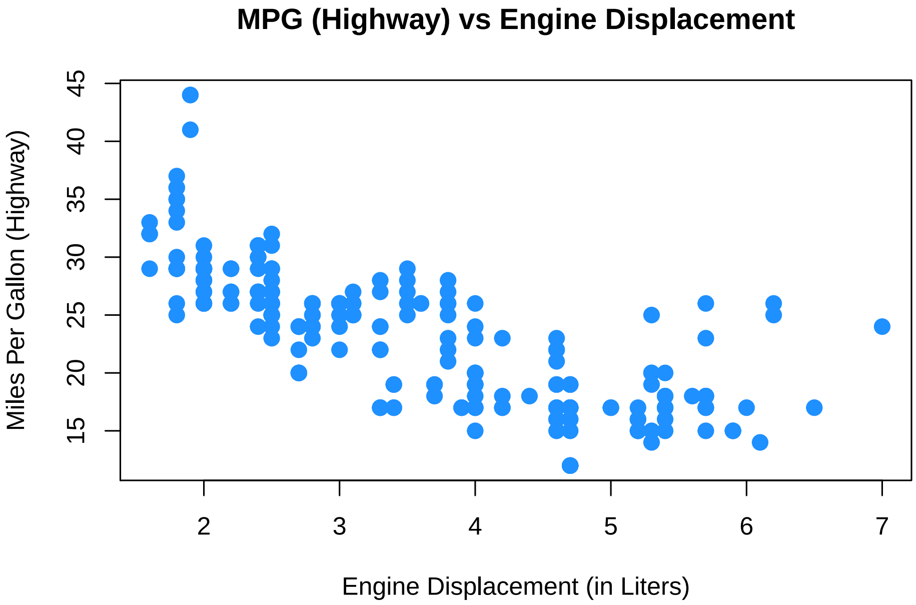
2.2.4 Scatterplots

Lastly, to visualize the relationship between two numeric variables we will use a **scatterplot**. This can be done with the `plot()` function and the `~` syntax we just used with a boxplot. (The function `plot()` can also be used more generally; see the documentation for details.)

```r
plot(hwy ~ displ, data = mpg)
```



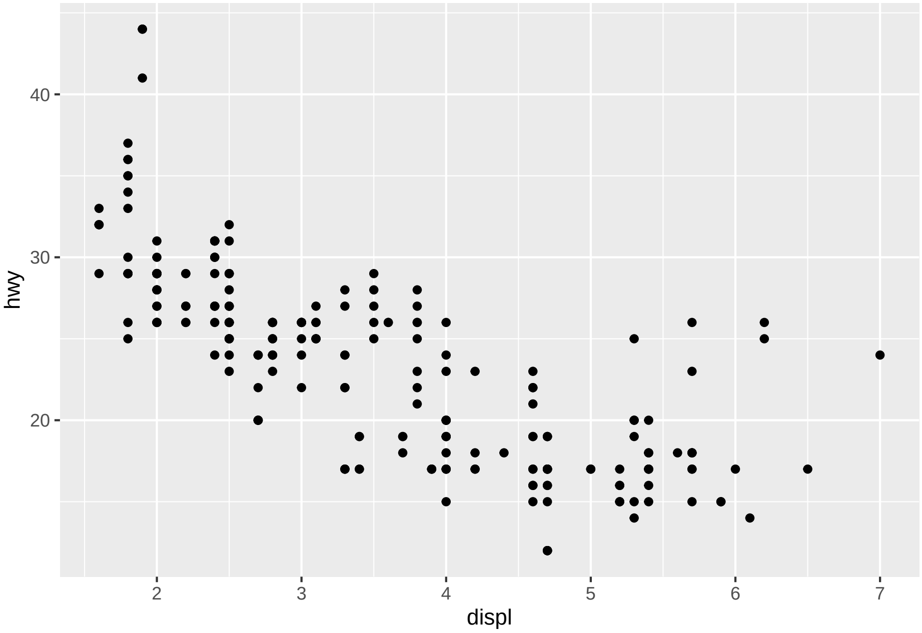
```r
plot(hwy ~ displ, data = mpg,
     xlab = "Engine Displacement (in Liters)",
     ylab = "Miles Per Gallon (Highway)",
     main = "MPG (Highway) vs Engine Displacement",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
```



2.2.5 ggplot

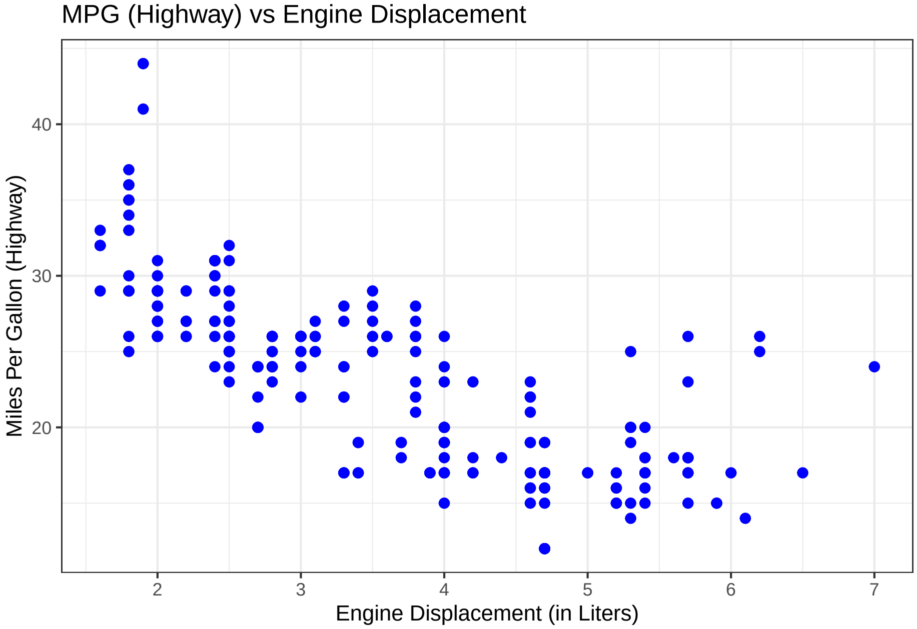
All of the above plots could also have been generated using the `ggplot` function from the already loaded `ggplot2` package. Which function you use is up to you, but sometimes a plot is easier to build in base R (like in the `boxplot` example maybe), sometimes the other way around.

```
ggplot(data = mpg,mapping = aes(x=displ,y=hwy)) + geom_point()
```



`ggplot` is impossible to describe in brief terms, so please look at the package's website which provides excellent guidance. We will from time to time use ggplot in this book, so you could familiarize yourself with it. Let's quickly demonstrate how one could further customize that first plot:

```
ggplot(data = mpg, mapping = aes(x=displ,y=hwy)) +   # ggplot() makes base plot
  geom_point(color="blue",size=2) +     # how to show x and y?
  scale_y_continuous(name="Miles Per Gallon (Highway)") +  # name of y axis
  scale_x_continuous(name="Engine Displacement (in Liters)") + # x axis
  theme_bw() +    # change the background
  ggtitle("MPG (Highway) vs Engine Displacement")    # add a title
```



If you want to see `ggplot` in action, you could start with this and then look at that very nice tutorial? It's fun!

2.3 Summarizing Two Variables

We often are interested in how two variables are related to each other. The core concepts here are *covariance* and *correlation*. Let's generate some data on `x` and `y` and plot them against each other:

Taking as example the data in this plot, the concepts *covariance* and *correlation* relate to the following type of question:

> Given we observe value of something like $x = 2$, say, can we expect a high or a low value of y, on average? Something like $y = 2$ or rather something like $y = -2$?

The answer to this type of question can be addressed by computing the covariance of both variables:

```
cov(x,y)
```

```
#OUT> [1] 1.041195
```

Here, this gives a positive number, 1.04, indicating that as one variable lies above it's average, the other one does as well. In other words, it indicates a **positive**

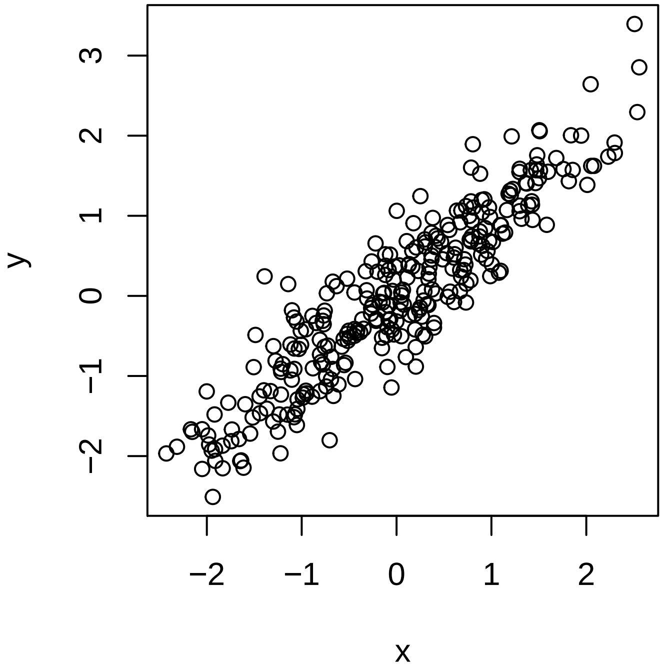


Figure 2.1: How are x and y related?

relationship. What is less clear, however, how to interpret the magnitude of 1.04. Is that a *strong* or a *weak* positive association?

In fact, we cannot tell. This is because the covariance is measured in the same units as the data, and those units often differ between both variables. There is a better measure available to us though, the **correlation**, which is obtained by *standardizing* each variable. By *standardizing* a variable x one means to divide x by its standard deviation σ_x:

$$z = \frac{x}{\sigma_x}$$

The *correlation coefficient* between x and y, commonly denoted $r_{x,y}$, is then defined as

$$r_{x,y} = \frac{cov(x,y)}{\sigma_x \sigma_y},$$

and we get rid of the units problem. In `R`, you can call directly

```
cor(x,y)
```

```
#OUT> [1] 0.9142495
```

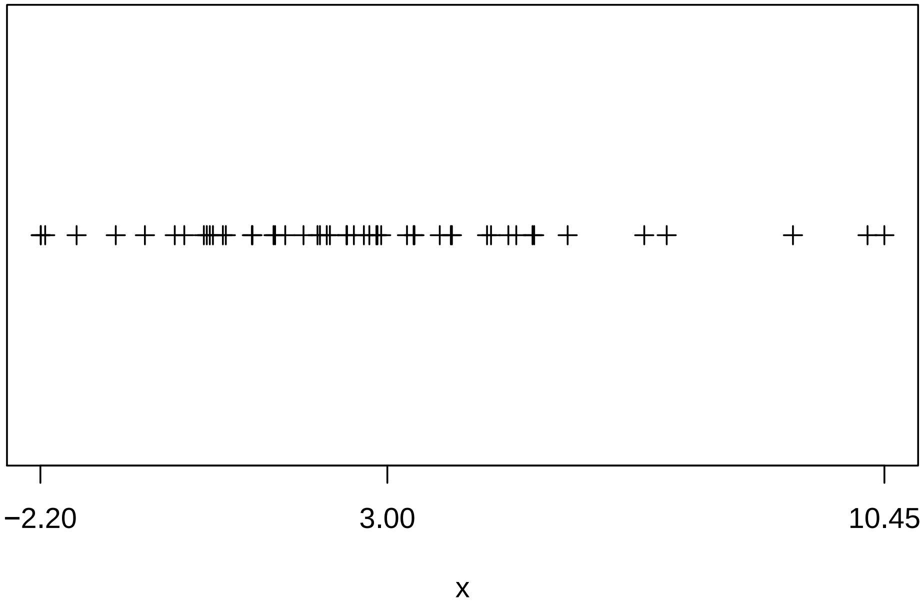


Figure 2.2: visual estimation on σ. The x-axis labels min and max as well as mean of x.

Now this is better. Given that the correlation has to lie in $[-1, 1]$, a value of 0.91 is indicative of a rather strong positive relationship for the data in figure 2.1

Note that x, y being drawn from a *continuous distribution* (they are joint normally distributed) had no implication for covariance and correlation: We can compute those measures also for discrete random variables (like the throws of two dice, as you will see in one of our tutorials).

2.3.1 Visually estimating σ

Sometimes it is useful to estimate the standard deviation of some data *without* the help of a computer (for example during an exam ;-)). If x is approximately normally distributed, 95% of its observations will lie within a range of $\bar{x}\pm$ two standard deviations of x. That is to say, *four* standard deviations of x cover 95% of its observations. Hence, a simple way to estimate the standard deviation for a variable is to look at the range of x, and simply divide that number by four.

This is illustrated in figure 2.2. Here we see that `range(x)/4` gives 3.16 which compares favourably to the actual standard deviation 3.

2.4 The tidyverse

Hadley Wickham is the author of R packages `ggplot2` and also of `dplyr` (and also a myriad of others). With `ggplot2` he introduced what is called the *grammar of graphics* (hence, `gg`) to `R`. Grammar in the sense that there are **nouns** and **verbs** and a **syntax**, i.e. rules of how nouns and verbs are to be put together to construct an understandable sentence. He has extended the *grammar* idea into various other packages. The `tidyverse` package is a collection of those packages.

`tidy` data is data where:

- Each variable is a column
- Each observation is a row
- Each value is a cell

Fair enough, you might say, that is a regular spreadsheet. And you are right! However, data comes to us *not* tidy most of the times, and we first need to clean, or `tidy`, it up. Once it's in `tidy` format, we can use the tools in the `tidyverse` with great efficiency to analyse the data and stop worrying about which tool to use.

2.4.1 Reading .csv data in the *tidy* way

We could have used the `read_csv()` function from the `readr` package to read our example dataset from the previous chapter. The `readr` function `read_csv()` has a number of advantages over the built-in `read.csv`. For example, it is much faster reading larger data. It also uses the `tibble` package to read the data as a tibble. **A `tibble` is simply a data frame that prints with sanity.** Notice in the output below that we are given additional information such as dimension and variable type.

```
library(readr)  # you need `install.packages("readr")` once!
path = system.file(package="ScPoEconometrics","datasets","example-data.csv")
example_data_from_disk = read_csv(path)
```

2.4.2 Tidy data.frames are tibbles

Let's grab some data from the `ggplot2` package:

```
data(mpg,package = "ggplot2")  # load dataset `mpg` from `ggplot2` package
head(mpg, n = 10)
```

```
#OUT> # A tibble: 10 x 11
#OUT>    manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
#OUT>    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
```

```
#OUT>  1 audi         a4      1.8  1999     4 auto~ f        18    29 p     comp~
#OUT>  2 audi         a4      1.8  1999     4 manu~ f        21    29 p     comp~
#OUT>  3 audi         a4      2    2008     4 manu~ f        20    31 p     comp~
#OUT>  4 audi         a4      2    2008     4 auto~ f        21    30 p     comp~
#OUT>  5 audi         a4      2.8  1999     6 auto~ f        16    26 p     comp~
#OUT>  6 audi         a4      2.8  1999     6 manu~ f        18    26 p     comp~
#OUT>  7 audi         a4      3.1  2008     6 auto~ f        18    27 p     comp~
#OUT>  8 audi         a4 q~   1.8  1999     4 manu~ 4        18    26 p     comp~
#OUT>  9 audi         a4 q~   1.8  1999     4 auto~ 4        16    25 p     comp~
#OUT> 10 audi         a4 q~   2    2008     4 manu~ 4        20    28 p     comp~
```

The function `head()` will display the first `n` observations of the data frame, as we have seen. The `head()` function was more useful before tibbles. Notice that `mpg` is a tibble already, so the output from `head()` indicates there are only 10 observations. Note that this applies to `head(mpg, n = 10)` and not `mpg` itself. Also note that tibbles print a limited number of rows and columns by default. The last line of the printed output indicates with rows and columns were omitted.

```
mpg
```

```
#OUT> # A tibble: 234 x 11
#OUT>    manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
#OUT>    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
#OUT>  1 audi         a4      1.8  1999     4 auto~ f        18    29 p     comp~
#OUT>  2 audi         a4      1.8  1999     4 manu~ f        21    29 p     comp~
#OUT>  3 audi         a4      2    2008     4 manu~ f        20    31 p     comp~
#OUT>  4 audi         a4      2    2008     4 auto~ f        21    30 p     comp~
#OUT>  5 audi         a4      2.8  1999     6 auto~ f        16    26 p     comp~
#OUT>  6 audi         a4      2.8  1999     6 manu~ f        18    26 p     comp~
#OUT>  7 audi         a4      3.1  2008     6 auto~ f        18    27 p     comp~
#OUT>  8 audi         a4 q~   1.8  1999     4 manu~ 4        18    26 p     comp~
#OUT>  9 audi         a4 q~   1.8  1999     4 auto~ 4        16    25 p     comp~
#OUT> 10 audi         a4 q~   2    2008     4 manu~ 4        20    28 p     comp~
#OUT> # ... with 224 more rows
```

Let's look at `str` as well to get familiar with the content of the data:

```
str(mpg)
```

```
#OUT> Classes 'tbl_df', 'tbl' and 'data.frame': 234 obs. of  11 variables:
#OUT>  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
#OUT>  $ model       : chr  "a4" "a4" "a4" "a4" ...
#OUT>  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
#OUT>  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
#OUT>  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
#OUT>  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
#OUT>  $ drv         : chr  "f" "f" "f" "f" ...
```

```
#OUT>  $ cty          : int  18 21 20 21 16 18 18 18 16 20 ...
#OUT>  $ hwy          : int  29 29 31 30 26 26 27 26 25 28 ...
#OUT>  $ fl           : chr  "p" "p" "p" "p" ...
#OUT>  $ class        : chr  "compact" "compact" "compact" "compact" ...
```

In this dataset an observation is for a particular model-year of a car, and the variables describe attributes of the car, for example its highway fuel efficiency.

To understand more about the data set, we use the `?` operator to pull up the documentation for the data.

```
?mpg
```

Working with tibbles is mostly the same as working with plain data.frames:

```
names(mpg)
```

```
#OUT>  [1] "manufacturer" "model"        "displ"        "year"
#OUT>  [5] "cyl"          "trans"        "drv"          "cty"
#OUT>  [9] "hwy"          "fl"           "class"
```

```
mpg$year
```

```
#OUT>   [1] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
#OUT>  [15] 2008 1999 2008 2008 2008 2008 2008 1999 2008 1999 1999 2008 2008 2008
#OUT>  [29] 2008 2008 1999 1999 1999 2008 1999 2008 2008 1999 1999 1999 1999 2008
#OUT>  [43] 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008 2008 1999
#OUT>  [57] 1999 1999 2008 2008 2008 1999 2008 1999 2008 2008 2008 2008 2008 2008
#OUT>  [71] 1999 1999 2008 1999 1999 1999 2008 1999 1999 1999 2008 2008 1999 1999
#OUT>  [85] 1999 1999 1999 2008 1999 2008 1999 1999 2008 2008 1999 1999 2008 2008
#OUT>  [99] 2008 1999 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008
#OUT> [113] 1999 1999 2008 1999 1999 2008 2008 2008 2008 2008 2008 2008 1999 1999
#OUT> [127] 2008 2008 2008 2008 1999 2008 2008 1999 1999 1999 2008 1999 2008 2008
#OUT> [141] 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 1999 1999 2008 2008
#OUT> [155] 1999 1999 1999 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 1999
#OUT> [169] 1999 2008 2008 2008 2008 1999 1999 1999 1999 2008 2008 1999 1999 2008
#OUT> [183] 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999 1999
#OUT> [197] 2008 2008 1999 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
#OUT> [211] 2008 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 1999 1999 1999
#OUT> [225] 1999 2008 2008 1999 1999 2008 2008 1999 1999 2008
```

```
mpg$hwy
```

```
#OUT>   [1] 29 29 31 30 26 26 27 26 25 28 27 25 25 25 25 24 25 23 20 15 20 17 17
#OUT>  [24] 26 23 26 25 24 19 14 15 17 27 30 26 29 26 24 24 22 22 24 24 17 22 21
#OUT>  [47] 23 23 19 18 17 17 19 19 12 17 15 17 17 12 17 16 18 15 16 12 17 17 16
#OUT>  [70] 12 15 16 17 15 17 17 18 17 19 17 19 19 17 17 17 16 16 17 15 17 26 25
#OUT>  [93] 26 24 21 22 23 22 20 33 32 32 29 32 34 36 36 29 26 27 30 31 26 26 28
#OUT> [116] 26 29 28 27 24 24 24 22 19 20 17 12 19 18 14 15 18 18 15 17 16 18 17
```

```
#OUT> [139] 19 19 17 29 27 31 32 27 26 26 25 25 17 17 20 18 26 26 27 28 25 25 24
#OUT> [162] 27 25 26 23 26 26 26 26 25 27 25 27 20 20 19 17 20 17 29 27 31 31 26
#OUT> [185] 26 28 27 29 31 31 26 26 27 30 33 35 37 35 15 18 20 20 22 17 19 18 20
#OUT> [208] 29 26 29 29 24 44 29 26 29 29 29 29 23 24 44 41 29 26 28 29 29 29 28
#OUT> [231] 29 26 26 26
```

Subsetting is also similar to dataframe. Here, we find fuel efficient vehicles earning over 35 miles per gallon and only display `manufacturer`, `model` and `year`.

```
# mpg[row condition, col condition]
mpg[mpg$hwy > 35, c("manufacturer", "model", "year")]
```

```
#OUT> # A tibble: 6 x 3
#OUT>   manufacturer model       year
#OUT>   <chr>        <chr>      <int>
#OUT> 1 honda        civic       2008
#OUT> 2 honda        civic       2008
#OUT> 3 toyota       corolla     2008
#OUT> 4 volkswagen   jetta       1999
#OUT> 5 volkswagen   new beetle  1999
#OUT> 6 volkswagen   new beetle  1999
```

An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"))
```

Lastly, and most *tidy*, we could use the `filter` and `select` functions from the `dplyr` package which introduces the *pipe operator* `f(x) %>% g(z)` from the `magrittr` package. This operator takes the output of the first command, for example `y = f(x)`, and passes it *as the first argument* to the next function, i.e. we'd obtain `g(y,z)` here.[1]

```
library(dplyr)
mpg %>%
  filter(hwy > 35) %>%
  select(manufacturer, model, year)
```

```
#OUT> # A tibble: 6 x 3
#OUT>   manufacturer model       year
#OUT>   <chr>        <chr>      <int>
#OUT> 1 honda        civic       2008
#OUT> 2 honda        civic       2008
#OUT> 3 toyota       corolla     2008
```

[1] A *pipe* is a concept from the Unix world, where it means to take the output of some command, and pass it on to another command. This way, one can construct a *pipeline* of commands. For additional info on the pipe operator in R, you might be interested in this tutorial.

```
#OUT> 4 volkswagen   jetta       1999
#OUT> 5 volkswagen   new beetle  1999
#OUT> 6 volkswagen   new beetle  1999
```

Note that the above syntax is equivalent to the following pipe-free command (which is much harder to read!):

```
library(dplyr)
select(filter(mpg, hwy > 35), manufacturer, model, year)
```

```
#OUT> # A tibble: 6 x 3
#OUT>   manufacturer model       year
#OUT>   <chr>        <chr>      <int>
#OUT> 1 honda        civic       2008
#OUT> 2 honda        civic       2008
#OUT> 3 toyota       corolla     2008
#OUT> 4 volkswagen   jetta       1999
#OUT> 5 volkswagen   new beetle  1999
#OUT> 6 volkswagen   new beetle  1999
```

All three approaches produce the same results. Which you use will be largely based on a given situation as well as your preference.

2.4.2.1 Task 1

1. Make sure to have the `mpg` dataset loaded by typing `data(mpg)` (and `library(ggplot2)` if you haven't!). Use the `table` function to find out how many cars were built by *mercury*?
2. What is the average year the audi's were built in this dataset? Use the function `mean` on the subset of column `year` that corresponds to `audi`. (Be careful: subsetting a `tibble` returns a `tibble` (and not a vector)!. so get the `year` column after you have subset the `tibble`.)
3. Use the `dplyr` piping syntax from above first with `group_by` and then with `summarise(newvar=your_expression)` to find the mean `year` by all manufacturers (i.e. same as previous task, but for all manufacturers. don't write a loop!).

2.4.3 Tidy Example: Importing Non-Tidy Excel Data

The data we will look at is from Eurostat on demography and migration. You should download the data yourself (click on previous link, then drill down to *database by themes > Population and social conditions > Demograph and migration > Population change - Demographic balance and crude rates at national level (demo_gind)*).

Once downloaded, we can read the data with the function `read_excel` from the package `readxl`, again part of the `tidyverse` suite.

It's important to know how the data is organized in the spreadsheet. Open the file with Excel to see:

- There is a heading which we don't need.
- There are 5 rows with info that we don't need.
- There is one table per variable (total population, males, females, etc)
- Each table has one row for each country, and one column for each year.
- As such, this data is **not tidy**.

Now we will read the first chunk of data, from the first table: *total population*:

```r
library(readxl)  # load the library
# Notice that if you installed the R package of this book,
# you have the .xls data file already at
# `system.file(package="ScPoEconometrics",
#                         "datasets","demo_gind.xls")`
# otherwise:
# * download the file to your computer
# * change the argument `path` to where you downloaded it
# you may want to change your working directory with `setwd("your/directory")
# or in RStudio by clicking Session > Set Working Directory

# total population in raw format
tot_pop_raw = read_excel(
                path = system.file(package="ScPoEconometrics",
                                    "datasets","demo_gind.xls"),
                sheet="Data", # which sheet
                range="A9:K68")  # which excel cell range to read
names(tot_pop_raw)[1] <- "Country"   # lets rename the first column
tot_pop_raw
```

```
#OUT> # A tibble: 59 x 11
#OUT>    Country `2008` `2009` `2010` `2011` `2012` `2013` `2014` `2015` `2016`
#OUT>    <chr>   <chr>  <chr>  <chr>  <chr>  <chr>  <chr>  <chr>  <chr>  <chr> 
#OUT>  1 Europe~ 50029~ 50209~ 50317~ 50296~ 50404~ 50516~ 50701~ 50854~ 51027~
#OUT>  2 Europe~ 43872~ 44004~ 44066~ 43994~ 44055~ 44125~ 44266~ 44366~ 44489~
#OUT>  3 Europe~ 49598~ 49778~ 49886~ 49867~ 49977~ 50090~ 50276~ 50431~ 50608~
#OUT>  4 Euro a~ 33309~ 33447~ 33526~ 33457~ 33528~ 33604~ 33754~ 33856~ 33988~
#OUT>  5 Euro a~ 32988~ 33128~ 33212~ 33152~ 33228~ 33307~ 33459~ 33563~ 33699~
#OUT>  6 Belgium 10666~ 10753~ 10839~ 11000~ 11075~ 11137~ 11180~ 11237~ 11311~
#OUT>  7 Bulgar~ 75180~ 74671~ 74217~ 73694~ 73272~ 72845~ 72456~ 72021~ 71537~
#OUT>  8 Czech ~ 10343~ 10425~ 10462~ 10486~ 10505~ 10516~ 10512~ 10538~ 10553~
#OUT>  9 Denmark 54757~ 55114~ 55347~ 55606~ 55805~ 56026~ 56272~ 56597~ 57072~
#OUT> 10 German~ 82217~ 82002~ 81802~ 80222~ 80327~ 80523~ 80767~ 81197~ 82175~
#OUT> # ... with 49 more rows, and 1 more variable: `2017` <chr>
```

This shows a `tibble`, which we encountered just above. The column names are `Country,2008,2009,...`, and the rows are numbered `1,2,3,...`. Notice, in particular, that *all* columns seem to be of type `<chr>`, i.e. characters - a string, not a number! We'll have to fix that, as this is clearly numeric data.

2.4.3.1 tidyr

In the previous `tibble`, each year is a column name (like `2008`) instead of all years being collected in one column `year`. We really would like to have several rows for each Country, one row per year. We want to `gather()` all years into a new column to tidy this up - and here is how:

1. specify which columns are to be gathered: in our case, all years (note that `paste(2008:2017)` produces a vector like `["2008", "2009", "2010",...]`)
2. say what those columns should be gathered into, i.e. what is the *key* for those values: we'll call it `year`.
3. Finally, what is the name of the new resulting column, containing the *value* from each cell: let's call it `counts`.

```
library(tidyr)   # for the gather function
tot_pop = gather(tot_pop_raw, paste(2008:2017),key="year", value = "counts")
tot_pop
```

```
#OUT> # A tibble: 590 x 3
#OUT>    Country                                      year  counts
#OUT>    <chr>                                        <chr> <chr>
#OUT>  1 European Union (current composition)         2008  500297033
#OUT>  2 European Union (without United Kingdom)      2008  438725386
#OUT>  3 European Union (before the accession of Croatia) 2008  495985066
#OUT>  4 Euro area (19 countries)                     2008  333096775
#OUT>  5 Euro area (18 countries)                     2008  329884170
#OUT>  6 Belgium                                      2008  10666866
#OUT>  7 Bulgaria                                     2008  7518002
#OUT>  8 Czech Republic                               2008  10343422
#OUT>  9 Denmark                                      2008  5475791
#OUT> 10 Germany (until 1990 former territory of the FRG) 2008  82217837
#OUT> # ... with 580 more rows
```

That's better! However, `counts` is still `chr`! Let's convert it to a number:

```
tot_pop$counts = as.integer(tot_pop$counts)
```

```
#OUT> Warning: NAs introduced by coercion
```

```
tot_pop
```

```
#OUT> # A tibble: 590 x 3
```

```
#OUT>    Country                                          year     counts
#OUT>    <chr>                                            <chr>     <int>
#OUT>  1 European Union (current composition)             2008  500297033
#OUT>  2 European Union (without United Kingdom)          2008  438725386
#OUT>  3 European Union (before the accession of Croatia) 2008  495985066
#OUT>  4 Euro area (19 countries)                         2008  333096775
#OUT>  5 Euro area (18 countries)                         2008  329884170
#OUT>  6 Belgium                                          2008   10666866
#OUT>  7 Bulgaria                                         2008    7518002
#OUT>  8 Czech Republic                                   2008   10343422
#OUT>  9 Denmark                                          2008    5475791
#OUT> 10 Germany (until 1990 former territory of the FRG) 2008   82217837
#OUT> # ... with 580 more rows
```

Now you can see that column `counts` is indeed `int`, i.e. an integer number, and we are fine. The `Warning: NAs introduced by coercion` means that `R` converted some values to `NA`, because it couldn't convert them into `numeric`. More below!

2.4.3.2 `dplyr`

> The transform chapter of Hadley Wickham's book is a great place to read up more on using `dplyr`.

With `dplyr` you can do the following operations on `data.frame`s and `tibble`s:

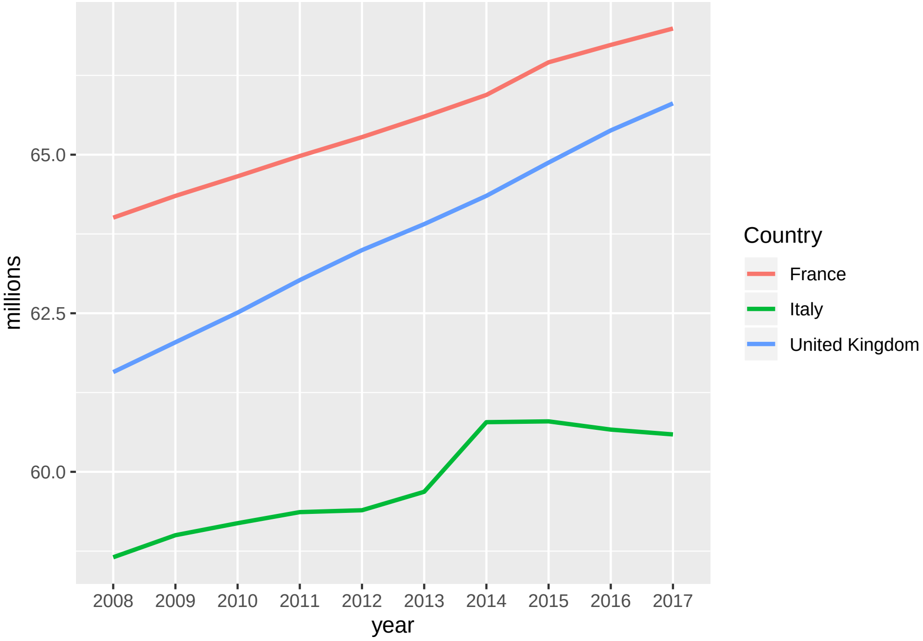
- Choose observations based on a certain value (i.e. subset): `filter()`
- Reorder rows: `arrange()`
- Select variables by name: `select()`
- Create new variables out of existing ones: `mutate()`
- Summarise variables: `summarise()`

All of those verbs can be used with `group_by()`, where we apply the respective operation on a *group* of the dataframe/tibble. For example, on our `tot_pop` tibble we will now

- filter
- mutate
- and plot the resulting values

Let's get a plot of the populations of France, the UK and Italy over time, in terms of millions of people. We will make use of the `piping` syntax of `dplyr` which we introduced just above.

```r
library(dplyr)  # for %>%, filter, mutate, ...
# 1. take the data.frame `tot_pop`
tot_pop %>%
  # 2. pipe it into the filter function
  # filter on Country being one of "France","United Kingdom" or "Italy"
  filter(Country %in% c("France","United Kingdom","Italy")) %>%
  # 3. pipe the result into the mutate function
  # create a new column called millions
  mutate(millions = counts / 1e6) %>%
  # 4. pipe the result into ggplot to make a plot
  ggplot(mapping = aes(x=year,y=millions,color=Country,group=Country)) + geom_line(size
```



Arrange a tibble

- What are the top/bottom 5 most populated areas?

```
top5 = tot_pop %>%
  arrange(desc(counts)) %>%  # arrange in descending order of col `counts`
  top_n(5)

bottom5 = tot_pop %>%
  arrange(desc(counts)) %>%
  top_n(-5)
# let's see top 5
top5
```

```
#OUT> # A tibble: 5 x 3
#OUT>   Country                                                          year    counts
```

```
#OUT>   <chr>                                                      <chr>    <int>
#OUT> 1 European Economic Area (EU28  - current composition, plus~ 2017    5.17e8
#OUT> 2 European Economic Area (EU28  - current composition, plus~ 2016    5.16e8
#OUT> 3 European Economic Area (EU28  - current composition, plus~ 2015    5.14e8
#OUT> 4 European Economic Area (EU27 -  before the accession of C~ 2017    5.13e8
#OUT> 5 European Economic Area (EU28  - current composition, plus~ 2014    5.12e8
```

```
# and bottom 5
bottom5
```

```
#OUT> # A tibble: 5 x 3
#OUT>   Country    year  counts
#OUT>   <chr>      <chr>  <int>
#OUT> 1 San Marino 2015   32789
#OUT> 2 San Marino 2014   32520
#OUT> 3 San Marino 2008   32054
#OUT> 4 San Marino 2011   31863
#OUT> 5 San Marino 2009   31269
```

Now this is not exactly what we wanted. It's always the same country in both top and bottom, because there are multiple years per country. Let's compute average population over the last 5 years and rank according to that:

```
topbottom = tot_pop %>%
  group_by(Country) %>%
  filter(year > 2012) %>%
  summarise(mean_count = mean(counts)) %>%
  arrange(desc(mean_count))

top5 = topbottom %>% top_n(5)
bottom5 = topbottom %>% top_n(-5)
top5
```

```
#OUT> # A tibble: 5 x 2
#OUT>   Country                                                        mean_count
#OUT>   <chr>                                                               <dbl>
#OUT> 1 European Economic Area (EU28  - current composition, plus IS,~ 514029320
#OUT> 2 European Economic Area (EU27 -  before the accession of Croat~ 509813491.
#OUT> 3 European Union (current composition)                           508502858.
#OUT> 4 European Union (before the accession of Croatia)               504287028.
#OUT> 5 European Union (without United Kingdom)                        443638309.
```

```
bottom5
```

```
#OUT> # A tibble: 5 x 2
#OUT>   Country       mean_count
#OUT>   <chr>              <dbl>
#OUT> 1 Luxembourg       563319.
#OUT> 2 Malta            440467.
```

```
#OUT> 3 Iceland          329501.
#OUT> 4 Liechtenstein     37353
#OUT> 5 San Marino        33014.
```

That's better!

Look for NAs in a tibble

Sometimes data is *missing*, and R represents it with the special value NA (not available). It is good to know where in our dataset we are going to encounter any missing values, so the task here is: let's produce a table that has three columns:

1. the names of countries with missing data
2. how many years of data are missing for each of those
3. and the actual years that are missing

```
missings = tot_pop %>%
  filter(is.na(counts)) %>% # is.na(x) returns TRUE if x is NA
  group_by(Country) %>%
  summarise(n_missing = n(),years = paste(year,collapse = ", "))
knitr:::kable(missings)  # knitr:::kable makes a nice table
```

Country	n_missing	years
Albania	2	2010, 2012
Andorra	2	2014, 2015
Armenia	1	2014
France (metropolitan)	4	2014, 2015, 2016, 2017
Georgia	1	2013
Monaco	7	2008, 2009, 2010, 2011, 2012, 2013, 2014
Russia	4	2013, 2015, 2016, 2017
San Marino	1	2010

Males and Females

Let's look at the numbers by male and female population. They are in the same xls file, but at different cell ranges. Also, I just realised that the special character : indicates *missing* data. We can feed that to read_excel and that will spare us the need to convert data types afterwards. Let's see:

```
females_raw = read_excel(
                path = system.file(package="ScPoEconometrics",
                                   "datasets","demo_gind.xls"),
                sheet="Data", # which sheet
                range="A141:K200",  # which excel cell range to read
                na=":" )   # missing data indicator
```

```
names(females_raw)[1] <- "Country"   # lets rename the first column
females_raw
```

```
#OUT> # A tibble: 59 x 11
#OUT>    Country `2008` `2009` `2010` `2011` `2012` `2013` `2014` `2015` `2016`
#OUT>    <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#OUT>  1 Europe~ 2.56e8 2.57e8 2.58e8 2.58e8 2.58e8 2.59e8 2.60e8 2.60e8 2.61e8
#OUT>  2 Europe~ 2.25e8 2.26e8 2.26e8 2.26e8 2.26e8 2.26e8 2.27e8 2.27e8 2.28e8
#OUT>  3 Europe~ 2.54e8 2.55e8 2.55e8 2.56e8 2.56e8 2.57e8 2.57e8 2.58e8 2.59e8
#OUT>  4 Euro a~ 1.71e8 1.71e8 1.72e8 1.72e8 1.72e8 1.72e8 1.73e8 1.73e8 1.74e8
#OUT>  5 Euro a~ 1.69e8 1.70e8 1.70e8 1.70e8 1.70e8 1.71e8 1.71e8 1.72e8 1.72e8
#OUT>  6 Belgium 5.44e6 5.48e6 5.53e6 5.60e6 5.64e6 5.67e6 5.69e6 5.71e6 5.74e6
#OUT>  7 Bulgar~ 3.86e6 3.83e6 3.81e6 3.78e6 3.76e6 3.74e6 3.72e6 3.70e6 3.68e6
#OUT>  8 Czech ~ 5.28e6 5.31e6 5.33e6 5.34e6 5.35e6 5.35e6 5.35e6 5.36e6 5.37e6
#OUT>  9 Denmark 2.76e6 2.78e6 2.79e6 2.80e6 2.81e6 2.82e6 2.83e6 2.85e6 2.87e6
#OUT> 10 German~ 4.19e7 4.18e7 4.17e7 4.11e7 4.11e7 4.11e7 4.12e7 4.14e7 4.17e7
#OUT> # ... with 49 more rows, and 1 more variable: `2017` <dbl>
```

You can see that `R` now correctly read the numbers as such, after we told it that the `:` character has the special *missing* meaning: before, it *coerced* the entire 2008 column (for example) to be of type `chr` after it hit the first `:`. We had to manually convert the column back to `numeric`, in the process automatically coercing the `:`s into `NA`. Now we addressed that issue directly. Let's also get the male data in the same way:

```
males_raw = read_excel(
                path = system.file(package="ScPoEconometrics",
                                    "datasets","demo_gind.xls"),
                sheet="Data", # which sheet
                range="A75:K134",  # which excel cell range to read
                na=":" )   # missing data indicator
names(males_raw)[1] <- "Country"   # lets rename the first column
```

Next step was to `tidy` up this data, just as before:

```
females = gather(females_raw, paste(2008:2017),key="year", value = "counts")
males = gather(males_raw, paste(2008:2017),key="year", value = "counts")
```

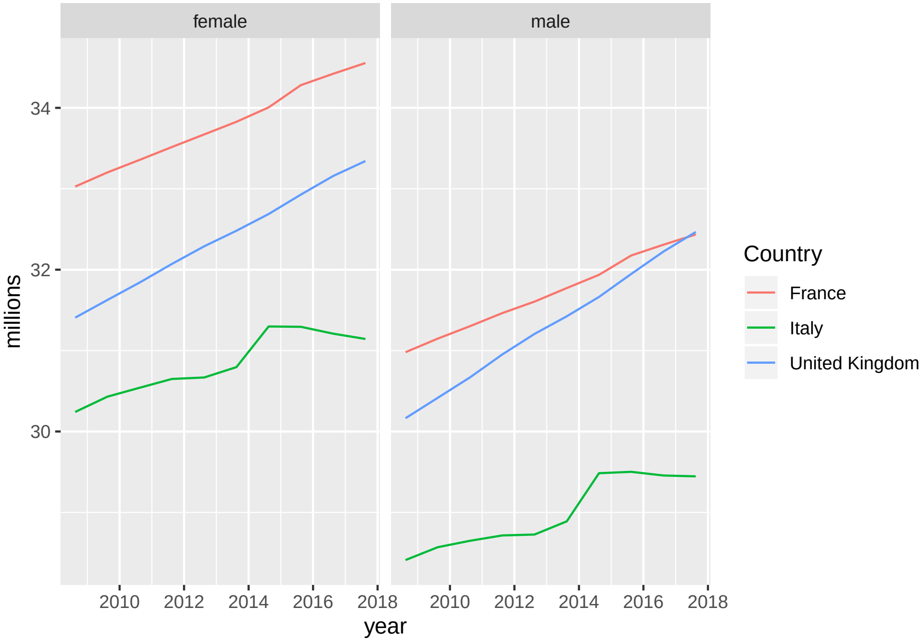
Let's try to tweak our above plot to show the same data in two separate panels: one for males and one for females. This is easiest to do with `ggplot` if we have all the data in one single `data.frame` (or `tibble`), and marked with a *group identifier*. Let's first add this to both datasets, and then let's just combine both into one:

```
females$sex = "female"
males$sex = "male"
sexes = rbind(males,females)   # "row bind" 2 data.frames
sexes
```

```
#OUT> # A tibble: 1,180 x 4
#OUT>    Country                                          year     counts sex  
#OUT>    <chr>                                            <chr>     <dbl> <chr>
#OUT>  1 European Union (current composition)             2008  243990548 male 
#OUT>  2 European Union (without United Kingdom)          2008  213826199 male 
#OUT>  3 European Union (before the accession of Croatia) 2008  241913560 male 
#OUT>  4 Euro area (19 countries)                         2008  162516883 male 
#OUT>  5 Euro area (18 countries)                         2008  161029464 male 
#OUT>  6 Belgium                                          2008    5224309 male 
#OUT>  7 Bulgaria                                         2008    3660367 male 
#OUT>  8 Czech Republic                                   2008    5065117 male 
#OUT>  9 Denmark                                          2008    2712666 male 
#OUT> 10 Germany (until 1990 former territory of the FRG) 2008   40274292 male 
#OUT> # ... with 1,170 more rows
```

Now that we have all the data nice and `tidy` in a `data.frame`, this is a very small change to our previous plotting code:

```
sexes %>%
  filter(Country %in% c("France","United Kingdom","Italy")) %>%
  mutate(millions = counts / 1e6) %>%
  ggplot(mapping = aes(x=as.Date(year,format="%Y"),  # convert to `Date`
                       y=millions,colour=Country,group=Country)) +
    geom_line() +
  scale_x_date(name = "year") + # rename x axis
  facet_wrap(~sex)   # make two panels, splitting by groups `sex`
```



Always Compare to Germany :-)

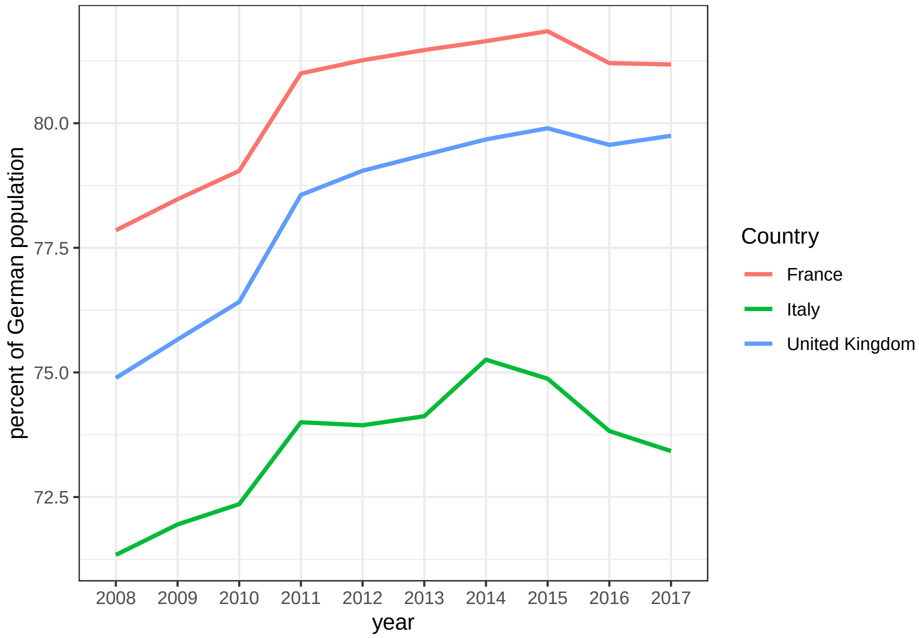
How do our three countries compare with respect to the biggest country in the EU in terms of population? What *fraction* of Germany does the French population make in any given year, for example?

```
# remember that the pipe operator %>% takes the
# result of the previous operation and passes it
# as the *first* argument to the next function call
merge_GER <- tot_pop %>%
  # 1. subset to countries of interest
  filter(
    Country %in%
      c("France",
        "United Kingdom",
        "Italy")
    ) %>%
  # 2. group data by year
  group_by(year) %>%
  # 3. add GER's count as new column *by year*
  left_join(
    # Germany only
    filter(tot_pop,
           Country %in% "Germany including former GDR"),
    # join back in `by year`
    by="year")
merge_GER
```

```
#OUT> # A tibble: 30 x 5
#OUT> # Groups:     year [10]
#OUT>    Country.x      year  counts.x Country.y                    counts.y
#OUT>    <chr>          <chr>    <int> <chr>                           <int>
#OUT>  1 France         2008  64007193 Germany including former GDR 82217837
#OUT>  2 Italy          2008  58652875 Germany including former GDR 82217837
#OUT>  3 United Kingdom 2008  61571647 Germany including former GDR 82217837
#OUT>  4 France         2009  64350226 Germany including former GDR 82002356
#OUT>  5 Italy          2009  59000586 Germany including former GDR 82002356
#OUT>  6 United Kingdom 2009  62042343 Germany including former GDR 82002356
#OUT>  7 France         2010  64658856 Germany including former GDR 81802257
#OUT>  8 Italy          2010  59190143 Germany including former GDR 81802257
#OUT>  9 United Kingdom 2010  62510197 Germany including former GDR 81802257
#OUT> 10 France         2011  64978721 Germany including former GDR 80222065
#OUT> # ... with 20 more rows
```

Here you see that the merge (or join) operation labelled `col.x` and `col.y` if both datasets contained a column called `col`. Now let's continue to compute what proportion of german population each country amounts to:

```r
names(merge_GER)[1] <- "Country"
merge_GER %>%
  mutate(prop_GER = 100 * counts.x / counts.y) %>%
  # 5. plot
  ggplot(mapping =
           aes(x = year,
               y = prop_GER,
               color = Country,
               group = Country)) +
  geom_line(size=1) +
  scale_y_continuous("percent of German population") +
  theme_bw()  # new theme for a change?
```



Chapter 3

Linear Regression

In this chapter we will learn an additional way how one can represent the relationship between *outcome*, or *dependent* variable variable y and an *explanatory* or *independent* variable x. We will refer throughout to the graphical representation of a collection of independent observations on x and y, i.e., a *dataset*.

3.1 How are `x` and `y` related?

3.1.1 Data on Cars

We will look at the built-in `cars` dataset. Let's get a view of this by just typing `View(cars)` in Rstudio. You can see something like this:

```
#OUT>   speed dist
#OUT> 1     4    2
#OUT> 2     4   10
#OUT> 3     7    4
#OUT> 4     7   22
#OUT> 5     8   16
#OUT> 6     9   10
```

We have a `data.frame` with two columns: `speed` and `dist`. Type `help(cars)` to find out more about the dataset. There you could read that

> The data give the speed of cars (mph) and the distances taken to stop (ft).

It's good practice to know the extent of a dataset. You could just type

```
dim(cars)
```

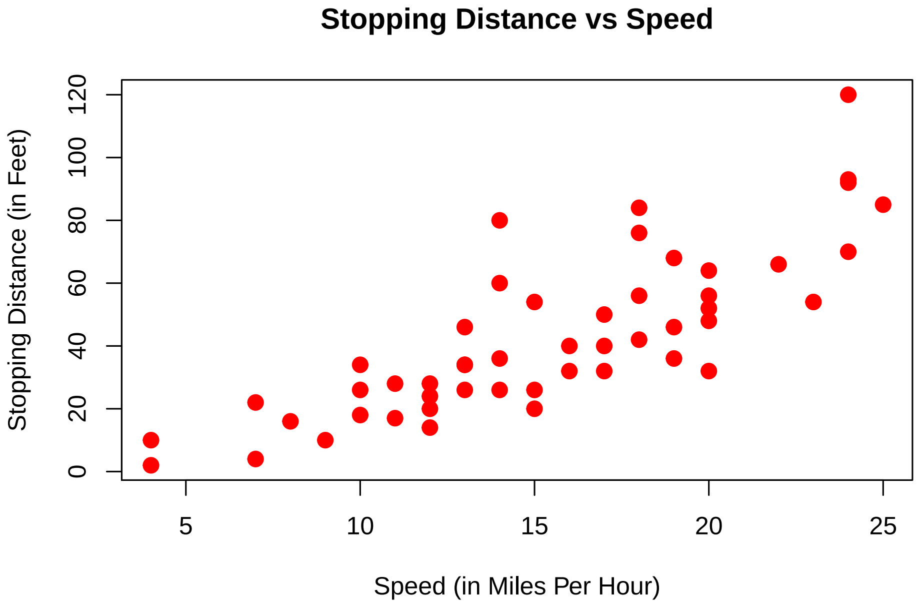
```
#OUT> [1] 50  2
```

to find out that we have 50 rows and 2 columns. A central question that we want to ask now is the following:

3.1.2 How are speed and dist related?

The simplest way to start is to plot the data. Remembering that we view each row of a data.frame as an observation, we could just label one axis of a graph `speed`, and the other one `dist`, and go through our table above row by row. We just have to read off the x/y coordinates and mark them in the graph. In `R`:

```
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 2,
     col  = "red")
```

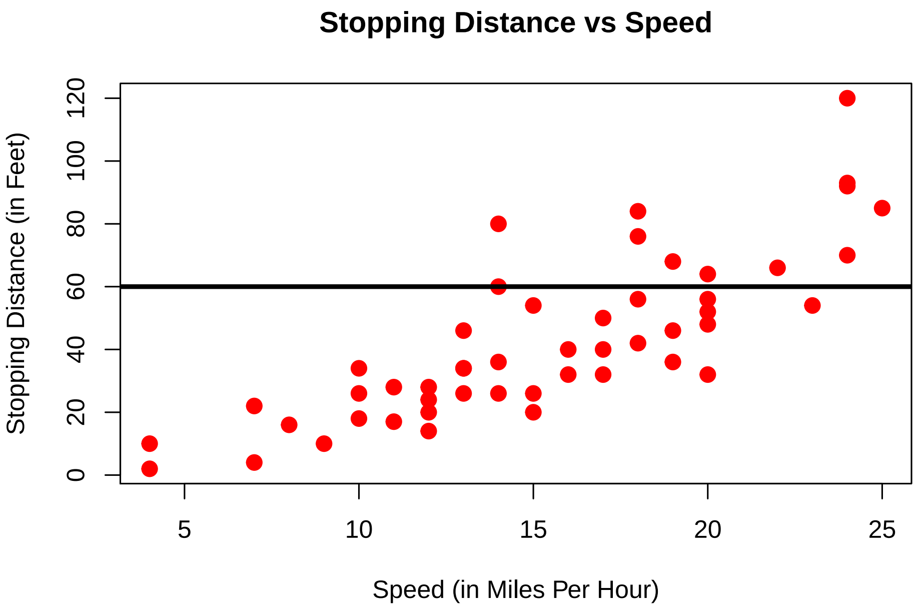


Here, each dot represents one observation. In this case, one particular measurement `speed` and `dist` for a car. Now, again:

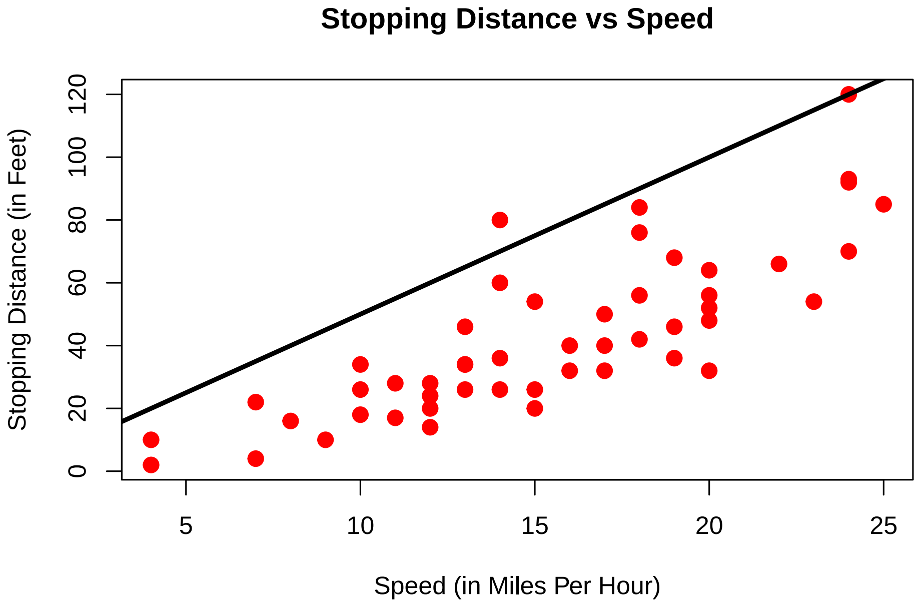
> How are `speed` and `dist` related? How could one best *summarize* this relationship?

One thing we could do, is draw a straight line through this scatterplot, like so:

```r
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 2,
     col  = "red")
abline(a = 60,b = 0,lw=3)
```



Now that doesn't seem a particularly *good* way to summarize the relationship. Clearly, a *better* line would be not be flat, but have a *slope*, i.e. go upwards:



That is slightly better. However, the line seems at too high a level - the point at which it crosses the y-axis is called the *intercept*; and it's too high. We just learned how to represent a *line*, i.e. with two numbers called *intercept* and *slope*. Let's write down a simple formula which represents a line where some outcome z is related to a variable x:

$$z = b_0 + b_1 x \tag{3.1}$$

Here b_0 represents the value of the intercept (i.e. z when $x = 0$), and b_1 is the value of the slope. The question for us is now: How to choose the number b_0 and b_1 such that the result is the **good** line?

3.1.3 Choosing the Best Line

In order to be able to reason about good or bad line, we need to denote the *output* of equation (3.1). We call the value $\hat{y}_i$ the *predicted value* for observation i, after having chosen some particular values b_0 and b_1:

$$\hat{y}_i = b_0 + b_1 x_i \tag{3.2}$$

In general it is likely that we won't be able to choose b_0 and b_1 in such as way as to provide a perfect prediction, i.e. one where $\hat{y}_i = y_i$ for all i. That is, we expect to make an *error* in our prediction $\hat{y}_i$, so let's denote this value e_i. If we acknowlegdge that we will make errors, let's at least make them as small as possible! Exactly this is going to be our task now.

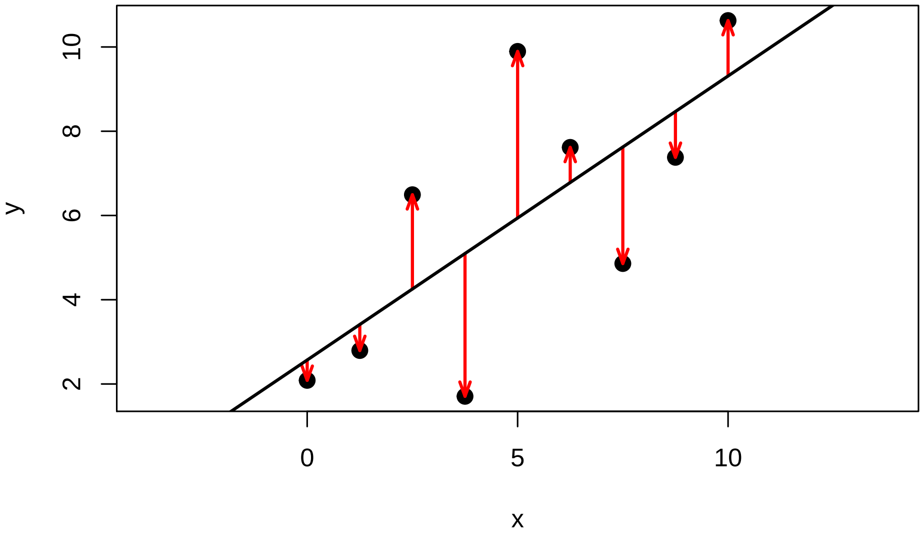


Figure 3.1: The best line and its errors

Suppose we have the following set of 9 observations on x and y, and we put the *best* straight line into it, that we can think of. It would look like this:

Here, the red arrows indicate the **distance** between the prediction (i.e. the black line) to each data point, in other words, each arrow is a particular e_i. An upward pointing arrow indicates a positive value of a particular e_i, and vice versa for downward pointing arrows. The erros are also called *residuals*, which comes from the way can write the equation for this relationship between two particular values (y_i, x_i) belonging to observation i:

$$y_i = b_0 + b_1 x_i + e_i \tag{3.3}$$

You realize of course that $\hat{y}_i = y_i - e_i$, which just means that our prediction is the observed value y_i minus any error e_i we make. In other words, e_i is what is left to be explained on top of the line $b_0 + b_1 x_i$, hence, it's a residual to explain y_i. Here are $y, \hat{y}$ and the resulting e which are plotted in figure 3.1:

x	y	y_hat	error
0.00	2.09	2.57	-0.48
1.25	2.79	3.41	-0.62
2.50	6.49	4.25	2.24
3.75	1.71	5.10	-3.39
5.00	9.89	5.94	3.95
6.25	7.62	6.78	0.83
7.50	4.86	7.63	-2.77
8.75	7.38	8.47	-1.09
10.00	10.63	9.31	1.32

If our line was a **perfect fit** to the data, all $e_i = 0$, and the column `error` would display `0` for each row - there would be no errors at all. (All points in figure 3.1 would perfectly line up on a straight line).

Now, back to our claim that this particular line is the *best* line. What exactly characterizes this best line? We now come back to what we said above - *how to make the errors as small as possible*? Keeping in mind that each residual e_i is $y_i - \hat{y}_i$, we have the following minization problem to solve:

$$e_i = y_i - \hat{y}_i = y_i - \underbrace{(b_0 + b_1 x_i)}_{\text{prediction}} \tag{3.4}$$

$$e_1^2 + \cdots + e_N^2 = \sum_{i=1}^{N} e_i^2 \equiv \text{SSR}(b_0, b_1) \tag{3.5}$$

$$(b_0, b_1) = \arg \min_{\text{int,slope}} \sum_{i=1}^{N} \left[y_i - (\text{int} + \text{slope } x_i) \right]^2 \tag{3.6}$$

The best line chooses b_0 and b_1 so as to minimize the sum of **squared residuals** (SSR).

Wait a moment, why *squared* residuals? This is easy to understand: suppose that instead, we wanted to just make the *sum* of the arrows in figure 3.1 as small as possible (that is, no squares). Choosing our line to make this number small would not give a particularly good representation of the data – given that errors of opposite sign and equal magnitude offset, we could have very long arrows (but of opposite signs), and a poor resulting line. Squaring each error avoids this (because now negative errors get positive values!)

We illustrate this in figure 3.2. This is the same data as in figure 3.1, but instead of arrows of length e_i for each observation i, now we draw a square with side e_i, i.e. an area of e_i^2. We have two apps for you at this point, one where you have to try and find the best line by choosing b_0 and b_1, only focusing on the sum of errors (and not their square), and a second one focusing on squared errors:

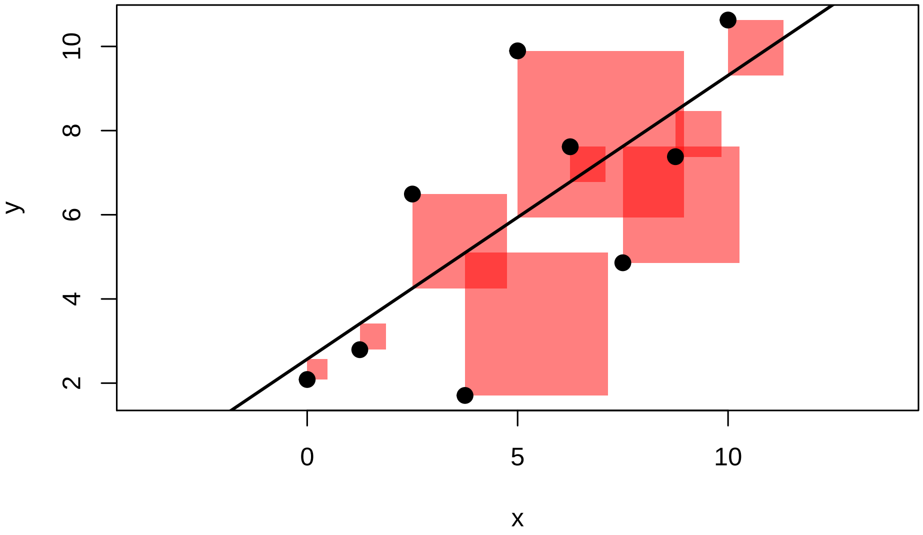


Figure 3.2: The best line and its SQUARED errors

```r
library(ScPoEconometrics)
launchApp("reg_simple_arrows")
launchApp("reg_simple") # with squared errors
launchApp("SSR_cone") # visualize the minimzation problem from above!
```

Most of our `apps` have an associated `about` document, which gives extra information and explanations. After you have looked at all three apps, we invite you thus to have a look at the associated explainers by typing

```r
aboutApp("reg_simple_arrows")
aboutApp("reg_simple")
aboutApp("SSR_cone")
```

3.2 Ordinary Least Squares (OLS) Estimator

The method to compute (or *estimate*) b_0 and b_1 we illustrated above is called *Ordinary Least Squares*, or OLS. b_0 and b_1 are therefore also often called the *OLS coefficients*. By solving problem (3.6) one can derive an explicit formula for them:

$$b_1 = \frac{cov(x,y)}{var(x)}, \tag{3.7}$$

i.e. the estimate of the slope coefficient is the covariance between x and y divided by the variance of x, both computed from our sample of data. With b_1 in hand, we can get the estimate for the intercept as

$$b_0 = \bar{y} - b_1\bar{x}. \tag{3.8}$$

where $\bar{z}$ denotes the sample mean of variable z. The interpretation of the OLS slope coefficient b_1 is as follows. Given a line as in $y = b_0 + b_1 x$,

- $b_1 = \frac{dy}{dx}$ measures the change in y resulting from a one unit change in x
- For example, if y is wage and x is years of education, b_1 would measure the effect of an additional year of education on wages.

There is an alternative representation for the OLS slope coefficient which relates to the *correlation coefficient* r. Remember from section 2.3 that $r = \frac{cov(x,y)}{s_x s_y}$, where s_z is the standard deviation of variable z. With this in hand, we can derive the OLS slope coefficient as

$$\begin{aligned} b_1 &= \frac{cov(x,y)}{var(x)} && (3.9) \\ &= \frac{cov(x,y)}{s_x s_x} && (3.10) \\ &= r\frac{s_y}{s_x} && (3.11) \end{aligned}$$

In other words, the slope coefficient is equal to the correlation coefficient r times the ratio of standard deviations of y and x.

3.2.1 Linear Regression without Regressor

There are several important special cases for the linear regression introduced above. Let's start with the most obvious one: What is the meaning of running a regression *without any regressor*, i.e. without a x? Our line becomes very simple. Instead of (3.1), we get

$$y = b_0. \tag{3.12}$$

This means that our minization problem in (3.6) *also* becomes very simple: We only have to choose b_0! We have

$$b_0 = \arg\min_{\text{int}} \sum_{i=1}^{N} \left[y_i - \text{int}\right]^2,$$

which is a quadratic equation with a unique optimum such that

$$b_0 = \frac{1}{N}\sum_{i=1}^{N} y_i = \overline{y}.$$

Least Squares **without regressor** x estimates the sample mean of the outcome variable y, i.e. it produces $\overline{y}$.

3.2.2 Regression without an Intercept

We follow the same logic here, just that we miss another bit from our initial equation and the minimisation problem in (3.6) now becomes:

$$b_1 = \arg\min_{\text{slope}} \sum_{i=1}^{N} \left[y_i - \text{slope } x_i\right]^2 \tag{3.13}$$

$$\mapsto b_1 = \frac{\frac{1}{N}\sum_{i=1}^{N} x_i y_i}{\frac{1}{N}\sum_{i=1}^{N} x_i^2} = \frac{\bar{x}\bar{y}}{\overline{x^2}} \tag{3.14}$$

Least Squares **without intercept** (i.e. with $b_0 = 0$) is a line that passes through the origin.

In this case we only get to choose the slope b_1 of this anchored line.[1] You should now try out both of those restrictions on our linear model by spending some time with

```
launchApp("reg_constrained")
```

3.2.3 Centering A Regression

By *centering* or *demeaning* a regression, we mean to substract from both y and x their respective averages to obtain $\tilde{y}_i = y_i - \bar{y}$ and $\tilde{x}_i = x_i - \bar{x}$. We then run a regression *without intercept* as above. That is, we use $\tilde{x}_i, \tilde{y}_i$ instead of x_i, y_i in (3.14) to obtain our slope estimate b_1:

[1] This slope is related to the angle between vectors $\mathbf{a} = (\overline{x}, \overline{y})$, and $\mathbf{b} = (\overline{x}, 0)$. Hence, it's related to the scalar projection of $\mathbf{a}$ on $\mathbf{b}$.

$$
\begin{aligned}
b_1 &= \frac{\frac{1}{N}\sum_{i=1}^N \tilde{x}_i \tilde{y}_i}{\frac{1}{N}\sum_{i=1}^N \tilde{x}_i^2} && (3.15)\\
&= \frac{\frac{1}{N}\sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{N}\sum_{i=1}^N (x_i - \bar{x})^2} && (3.16)\\
&= \frac{cov(x,y)}{var(x)} && (3.17)
\end{aligned}
$$

This last expression is *identical* to the one in (3.7)! It's the standard OLS estimate for the slope coefficient. We note the following:

> Adding a constant to a regression produces the same result as centering all variables and estimating without intercept. So, unless all variables are centered, **always** include an intercept in the regression.

To get a better feel for what is going on here, you can try this out now by yourself by typing:

```
launchApp("demeaned_reg")
```

3.2.4 Standardizing A Regression

Standardizing a variable z means to demean as above, but in addition to divide the demeaned value by its own standard deviation. Similarly to what we did above for *centering*, we define transformed variables $\breve{y}_i = \frac{y_i - \bar{y}}{\sigma_y}$ and $\breve{x}_i = \frac{x_i - \bar{x}}{\sigma_x}$ where σ_z is the standard deviation of variable z. From here on, you should by now be used to what comes next! As above, we use $\breve{x}_i, \breve{y}_i$ instead of x_i, y_i in (3.14) to this time obtain:

$$
\begin{aligned}
b_1 &= \frac{\frac{1}{N}\sum_{i=1}^N \breve{x}_i \breve{y}_i}{\frac{1}{N}\sum_{i=1}^N \breve{x}_i^2} && (3.18)\\
&= \frac{\frac{1}{N}\sum_{i=1}^N \frac{x_i - \bar{x}}{\sigma_x}\frac{y_i - \bar{y}}{\sigma_y}}{\frac{1}{N}\sum_{i=1}^N \left(\frac{x_i - \bar{x}}{\sigma_x}\right)^2} && (3.19)\\
&= \frac{Cov(x,y)}{\sigma_x \sigma_y} && (3.20)\\
&= Corr(x,y) && (3.21)
\end{aligned}
$$

After we standardize both y and x, the slope coefficient b_1 in the regression without intercept is equal to the **correlation coefficient**.

And also for this case we have a practical application for you. Just type this and play around with the app for a little while!

```
launchApp("reg_standardized")
```

3.3 Predictions and Residuals

Now we want to ask how our residuals e_i relate to the prediction $\hat{y}_i$. Let us first think about the average of all predictions $\hat{y}_i$, i.e. the number $\frac{1}{N}\sum_{i=1}^N \hat{y}_i$. Let's just take (3.2) and plug this into this average, so that we get

$$
\begin{aligned}
\frac{1}{N}\sum_{i=1}^N \hat{y}_i &= \frac{1}{N}\sum_{i=1}^N b_0 + b_1 x_i \qquad (3.22)\\
&= b_0 + b_1 \frac{1}{N}\sum_{i=1}^N x_i \qquad (3.23)\\
&= b_0 + b_1 \bar{x} \qquad (3.24)\\
& \qquad (3.25)
\end{aligned}
$$

But that last line is just equal to the formula for the OLS intercept (3.8), $b_0 = \bar{y} - b_1\bar{x}$! That means of course that

$$
\frac{1}{N}\sum_{i=1}^N \hat{y}_i = b_0 + b_1\bar{x} = \bar{y}
$$

in other words:

The average of our predictions $\hat{y}_i$ is identically equal to the mean of the outcome y. This implies that the average of the residuals is equal to zero.

Related to this result, we can show that the prediction $\hat{y}$ and the residuals are *uncorrelated*, something that is often called **orthogonality** between $\hat{y}_i$ and e_i. We would write this as

$$Cov(\hat{y}, e) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - \bar{y})(e_i - \bar{e}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - \bar{y}) e_i \tag{3.26}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \hat{y}_i e_i - \bar{y} \frac{1}{N} \sum_{i=1}^{N} e_i = 0 \tag{3.27}$$

It's useful to bring back the sample data which generate figure 3.1 at this point in order to verify these claims:

```
#OUT>       y y_hat error
#OUT> 1  2.09  2.57 -0.48
#OUT> 2  2.79  3.41 -0.62
#OUT> 3  6.49  4.25  2.24
#OUT> 4  1.71  5.10 -3.39
#OUT> 5  9.89  5.94  3.95
#OUT> 6  7.62  6.78  0.83
#OUT> 7  4.86  7.63 -2.77
#OUT> 8  7.38  8.47 -1.09
#OUT> 9 10.63  9.31  1.32
```

Let's check that these claims are true in this sample of data. We want that

1. The average of $\hat{y}_i$ to be the same as the mean of y
2. The average of the errors should be zero.
3. Prediction and errors should be uncorrelated.

```
# 1.
all.equal(mean(ss$error), 0)
```

```
#OUT> [1] TRUE
```

```
# 2.
all.equal(mean(ss$y_hat), mean(ss$y))
```

```
#OUT> [1] TRUE
```

```
# 3.
all.equal(cov(ss$error,ss$y_hat), 0)
```

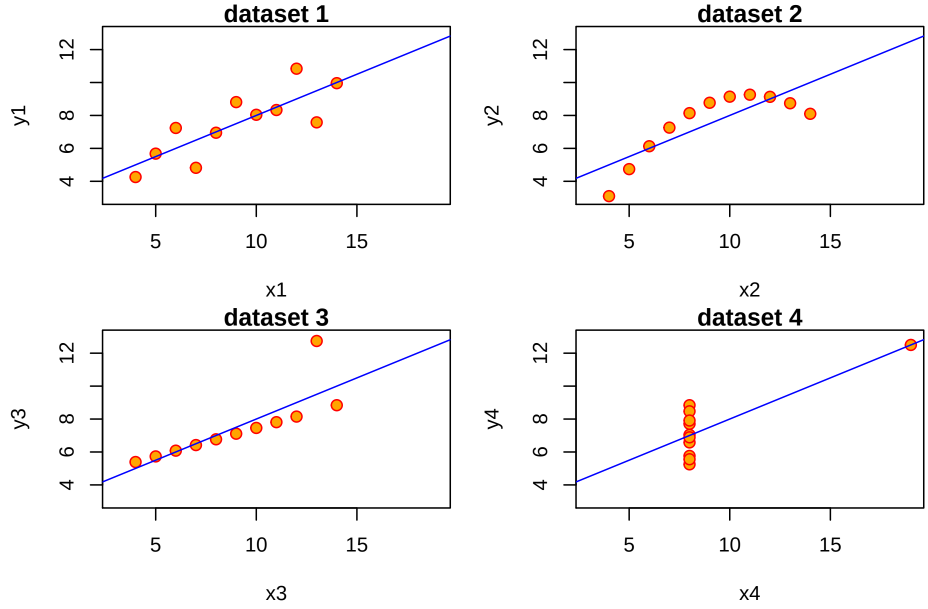
```
#OUT> [1] TRUE
```

So indeed we can confirm this result with our test dataset. Great!

3.4 Correlation, Covariance and Linearity

It is important to keep in mind that Correlation and Covariance relate to a *linear* relationship between `x` and `y`. Given how the regression line is estimated by

OLS (see just above), you can see that the regression line inherits this property from the Covariance. A famous exercise by Francis Anscombe (1973) illustrates this by constructing 4 different datasets which all have identical **linear** statistics: mean, variance, correlation and regression line *are identical*. However, the usefulness of the statistics to describe the relationship in the data is not clear.



The important lesson from this example is the following:

Always **visually inspect** your data, and don't rely exclusively on summary statistics like *mean, variance, correlation and regression line.* All of those assume a **linear** relationship between the variables in your data.

The mission of Anscombe has been continued recently. As a result of this we can have a look at the `datasauRus` package, which pursues Anscbombe's idea through a multitude of funny data sets, all with the same linear statistics. Don't just compute the covariance, or you might actually end up looking at a Dinosaur! What? Type this to find out:

```
launchApp("datasaurus")
aboutApp("datasaurus")
```

3.4.1 Non-Linear Relationships in Data

Suppose our data now looks like this:

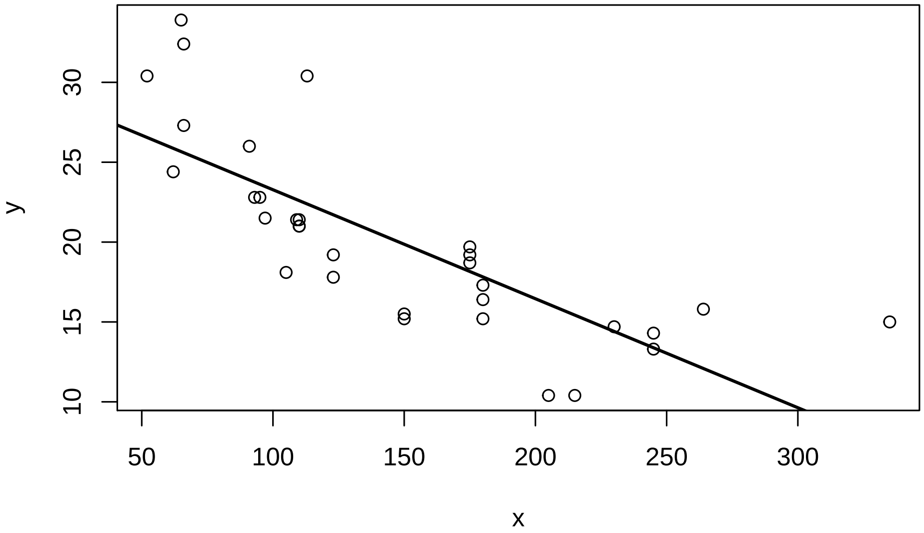
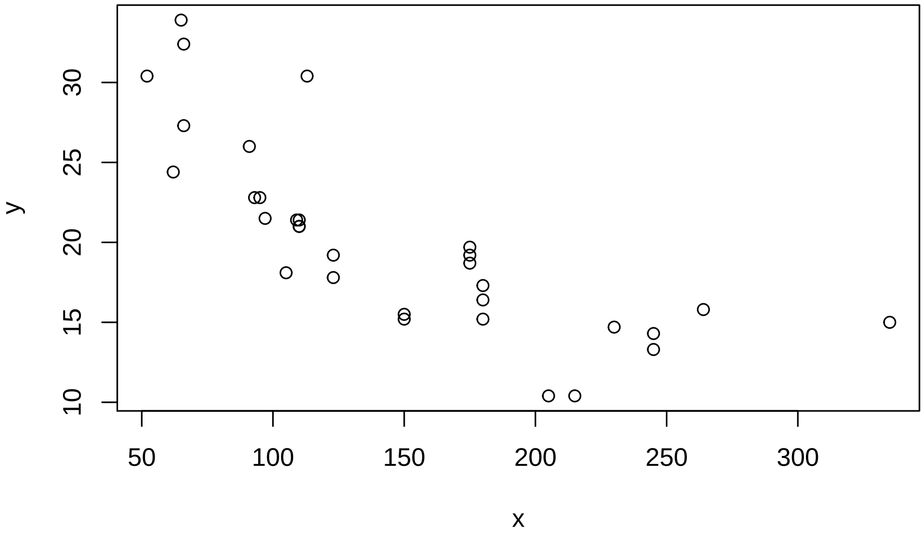


Figure 3.3: Best line with non-linear data?



Putting our previous *best line* defined in equation (3.3) as $y = b_0 + b_1 x + e$, we get something like this:

Somehow when looking at 3.3 one is not totally convinced that the straight line is a good summary of this relationship. For values $x \in [50, 120]$ the line seems to low, then again too high, and it completely misses the right boundary. It's easy to address this shortcoming by including *higher order terms* of an explanatory variable. We would modify (3.3) to read now

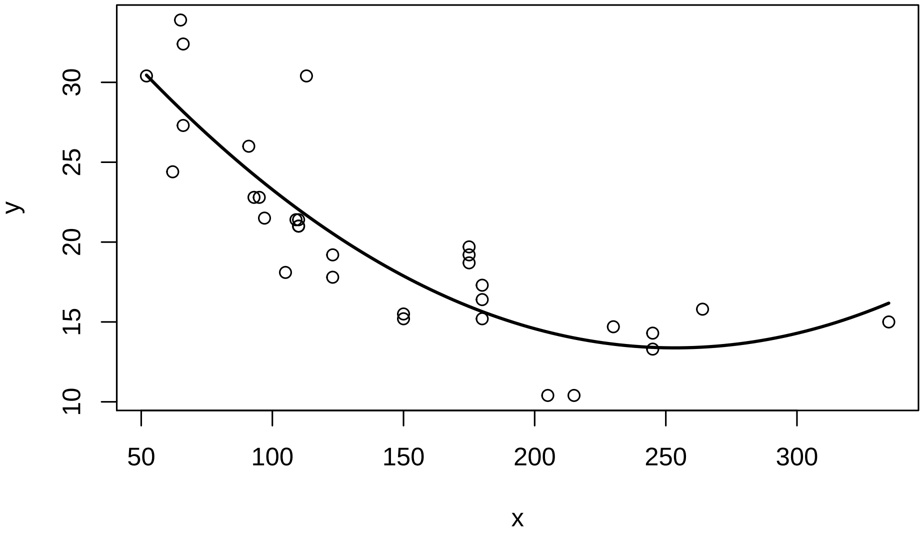


Figure 3.4: Better line with non-linear data!

$$y_i = b_0 + b_1 x_i + b_2 x_i^2 + e_i \tag{3.28}$$

This is a special case of *multiple regression*, which we will talk about in chapter 4. You can see that there are *multiple* slope coefficients. For now, let's just see how this performs:

3.5 Analysing $Var(y)$

Analysis of Variance (ANOVA) refers to a method to decompose variation in one variable as a function of several others. We can use this idea on our outcome y. Suppose we wanted to know the variance of y, keeping in mind that, by definition, $y_i = \hat{y}_i + e_i$. We would write

$$\begin{aligned} Var(y) &= Var(\hat{y} + e) && (3.29) \\ &= Var(\hat{y}) + Var(e) + 2Cov(\hat{y}, e) && (3.30) \\ &= Var(\hat{y}) + Var(e) && (3.31) \end{aligned}$$

We have seen above in 3.3 that the covariance between prediction $\hat{y}$ and error e is zero, that's why we have $Cov(\hat{y}, e) = 0$ in (3.31). What this tells us in words is that we can decompose the variance in the observed outcome y into a part that relates to variance as *explained by the model* and a part that comes from unexplained variation. Finally, we know the definition of *variance*, and can thus write down the respective formulae for each part:

- $Var(y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2$
- $Var(\hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - \bar{y})^2$, because the mean of $\hat{y}$ is $\bar{y}$ as we know. Finally,
- $Var(e) = \frac{1}{N}\sum_{i=1}^{N} e_i^2$, because the mean of e is zero.

We can thus formulate how the total variation in outcome y is aportioned between model and unexplained variation:

> The total variation in outcome y (often called SST, or *total sum of squares*) is equal to the sum of explained squares (SSE) plus the sum of residuals (SSR). We have thus **SST = SSE + SSR**.

3.6 Assessing the *Goodness of Fit*

In our setup, there exists a convenient measure for how good a particular statistical model fits the data. It is called R^2 (*R squared*), also called the *coefficient of determination.* We make use of the just introduced decomposition of variance, and write the formula as

$$R^2 = \frac{\text{variance explained}}{\text{total variance}} = \frac{SSE}{SST} = 1 - \frac{SSR}{SST} \in [0,1] \tag{3.32}$$

It is easy to see that a *good fit* is one where the sum of *explained* squares (SSE) is large relativ to the total variation (SST). In such a case, we observe an R^2 close to one. In the opposite case, we will see an R^2 close to zero. Notice that a small R^2 does not imply that the model is useless, just that it explains a small fraction of the observed variation.

3.7 An Example: California Student Test Scores

Luckily for us, fitting a linear model to some data does not require us to iteratively find the best intercept and slope manually, as you have experienced in our `apps`. As it turns out, `R` can do this much more precisely, and very fast!

Let's explore how to do this, using a real life dataset taken from the `Ecdat` package which includes many economics-related dataset. In this example, we will use the `Caschool` dataset which contains the average test scores of 420 elementary schools in California along with some additional information.

3.7.1 Loading and exploring Data

We can explore which variables are included in the dataset using the `names()` function:

```r
library("Ecdat") # Load the Ecdat library
names(Caschool) # Display the variables of the Caschool dataset
```

```
#OUT>  [1] "distcod"  "county"   "district" "grspan"   "enrltot"  "teachers"
#OUT>  [7] "calwpct"  "mealpct"  "computer" "testscr"  "compstu"  "expnstu"
#OUT> [13] "str"      "avginc"   "elpct"    "readscr"  "mathscr"
```

For each variable in the dataset, basic summary statistics can be obtained by calling `summary()`

```r
summary(Caschool[, c("testscr", "str", "avginc")])
```

```
#OUT>     testscr           str            avginc
#OUT>  Min.   :605.5   Min.   :14.00   Min.   : 5.335
#OUT>  1st Qu.:640.0   1st Qu.:18.58   1st Qu.:10.639
#OUT>  Median :654.5   Median :19.72   Median :13.728
#OUT>  Mean   :654.2   Mean   :19.64   Mean   :15.317
#OUT>  3rd Qu.:666.7   3rd Qu.:20.87   3rd Qu.:17.629
#OUT>  Max.   :706.8   Max.   :25.80   Max.   :55.328
```

3.7.2 Fitting a linear model

Suppose we are interested in the following linear model:

$$\text{testscr}_i = b_0 + b_1 \times \text{str}_i + e_i$$

Where testscr_i is the *average test score* for a given school i and str_i is the *Student/Teacher Ratio* (i.e. the average number of students per teacher) in the same school i. Again, b_0 and b_1 are the intercept and the slope of the regression line.

The subscript i indexes all unique elementary schools ($i \in \{1, 2, 3, \dots 420\}$) and e_i is the error, or *residual*, of the regression. (Remember that our procedure for finding the line of best fit is to minimize the *sum of squared residuals* (SSR)).

At this point you should step back and take a second to think about what you believe the relation between a school's test scores and student/teacher ratio will be. Do you believe that, in general, a high student/teacher ratio will be associated with higher-than-average test scores for the school? Do you think that the number of students per teacher will impact results in any way?

Let's find out! As always, we will start by plotting the data to inspect it visually:

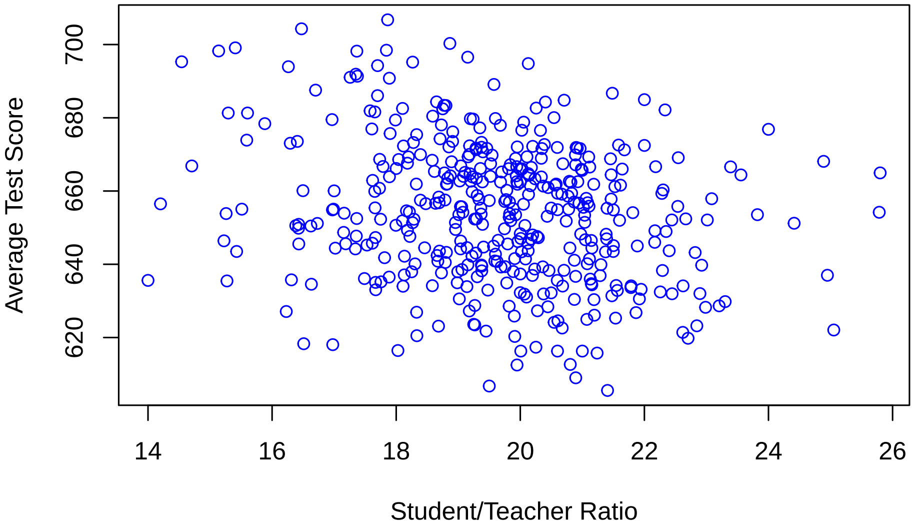


Figure 3.5: Student Teacher Ratio vs Test Scores

```r
plot(formula = testscr ~ str,
     data = Caschool,
     xlab = "Student/Teacher Ratio",
     ylab = "Average Test Score", pch = 21, col = 'blue')
```

Can you spot a trend in the data? According to you, what would the line of best fit look like? Would it be upward or downward slopping? Let's ask `R`!

3.7.3 The `lm()` function

We will use the built-in `lm()` function to estimate the coefficients b_0 and b_1 using the data at hand. `lm` stands for *linear model*, which is what our representation in (3.3) amounts to. This function typically only takes 2 arguments, `formula` and `data`:

```
lm(formula, data)
```

- `formula` is the description of our model which we want `R` to estimate for us. Its syntax is very simple: `Y ~ X` (more generally, `DependentVariable ~ Independent Variables`). You can think of the tilda operator `~` as the equal sign in your model equation. An intercept is included by default and so you do not have to ask for it in `formula`. For example, the simple model $income = b_0 + b_1 \cdot age$ can be written as `income ~ age`. A `formula` can sometimes be written between quotation marks: `"X ~ Y"`.
- `data` is simply the `data.frame` containing the variables in the model.

In the context of our example, the function call is therefore:

```r
# assign lm() output to some object `fit_cal`
fit_cal <- lm(formula = testscr ~ str, data = Caschool)

# ask R for the regression summary
summary(fit_cal)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = testscr ~ str, data = Caschool)
#OUT>
#OUT> Residuals:
#OUT>     Min      1Q  Median      3Q     Max
#OUT> -47.727 -14.251   0.483  12.822  48.540
#OUT>
#OUT> Coefficients:
#OUT>             Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 698.9330     9.4675  73.825  < 2e-16 ***
#OUT> str          -2.2798     0.4798  -4.751 2.78e-06 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 18.58 on 418 degrees of freedom
#OUT> Multiple R-squared:  0.05124, Adjusted R-squared:  0.04897
#OUT> F-statistic: 22.58 on 1 and 418 DF,  p-value: 2.783e-06
```

As we can see, `R` returns its estimates for the Intercept and Slope coefficients, b_0 = 698.93 and b_1 = -2.28. The estimated relationship between a school's Student/Teacher Ratio and its average test results is **negative**.

The output of the `summary` method for an `lm` object is commonly called a *regression table*, and you will be able to decypher it by the end of this course. You should be able to find an interpret the R^2 though: Are we explaining a lot of the variance in `testscr` with this simple model, or are we not?

3.7.4 Plotting the regression line

We can also use our `lm` fit to draw the regression line on top of our initial scatterplot, using the following syntax:

```r
plot(formula = testscr ~ str,
     data = Caschool,
     xlab = "Student/Teacher Ratio",
     ylab = "Average Test Score", pch = 21, col = 'blue')# same plot as before
abline(fit_cal, col = 'red') # add regression line
```

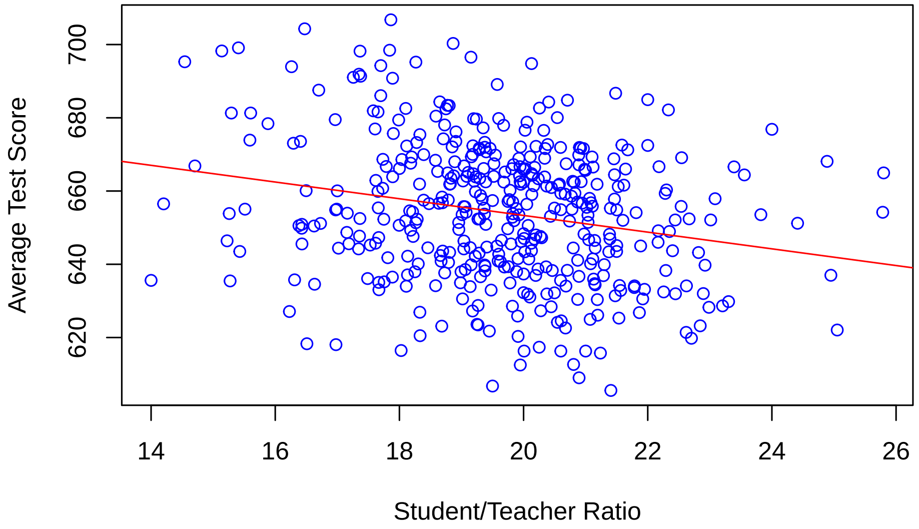
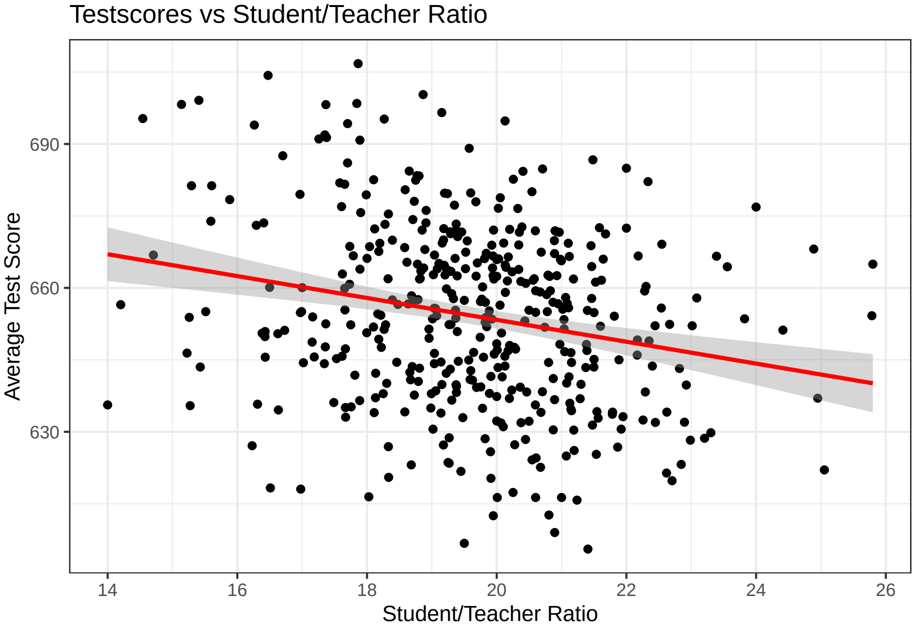


Figure 3.6: Test Scores with Regression Line

As you probably expected, the best line for schools' Student/Teacher Ratio and its average test results is downward sloping.

Just as a way of showcasing another way to make the above plot, here is how you could use `ggplot`:

```
library(ggplot2)
p <- ggplot(mapping = aes(x = str, y = testscr), data = Caschool) # base plot
p <- p + geom_point() # add points
p <- p + geom_smooth(method = "lm", size=1, color="red") # add regression line
p <- p + scale_y_continuous(name = "Average Test Score") +
         scale_x_continuous(name = "Student/Teacher Ratio")
p + theme_bw() + ggtitle("Testscores vs Student/Teacher Ratio")
```



The shaded area around the red line shows the width of the 95% confidence interval around our estimate of the slope coefficient b_1. We will learn more about it in chapter 6.

Chapter 4

Multiple Regression

We can extend the discussion from chapter 3 to more than one explanatory variable. For example, suppose that instead of only x we now had x_1 and x_2 in order to explain y. Everything we've learned for the single variable case applies here as well. Instead of a regression *line*, we now get a regression *plane*, i.e. an object representable in 3 dimenions: (x_1, x_2, y). As an example, suppose we wanted to explain how many *miles per gallon* (`mpg`) a car can travel as a function of its *horse power* (`hp`) and its *weight* (`wt`). In other words we want to estimate the equation

$$mpg_i = b_0 + b_1 hp_i + b_2 wt_i + e_i \tag{4.1}$$

on our built-in dataset of cars (`mtcars`):

```r
subset(mtcars, select = c(mpg,hp,wt))
```

```
#OUT>                      mpg  hp    wt
#OUT> Mazda RX4           21.0 110 2.620
#OUT> Mazda RX4 Wag       21.0 110 2.875
#OUT> Datsun 710          22.8  93 2.320
#OUT> Hornet 4 Drive      21.4 110 3.215
#OUT> Hornet Sportabout   18.7 175 3.440
#OUT> Valiant             18.1 105 3.460
#OUT> Duster 360          14.3 245 3.570
#OUT> Merc 240D           24.4  62 3.190
#OUT> Merc 230            22.8  95 3.150
#OUT> Merc 280            19.2 123 3.440
#OUT> Merc 280C           17.8 123 3.440
#OUT> Merc 450SE          16.4 180 4.070
#OUT> Merc 450SL          17.3 180 3.730
#OUT> Merc 450SLC         15.2 180 3.780
```

```
#OUT> Cadillac Fleetwood  10.4 205 5.250
#OUT> Lincoln Continental 10.4 215 5.424
#OUT> Chrysler Imperial   14.7 230 5.345
#OUT> Fiat 128            32.4  66 2.200
#OUT> Honda Civic         30.4  52 1.615
#OUT> Toyota Corolla      33.9  65 1.835
#OUT> Toyota Corona       21.5  97 2.465
#OUT> Dodge Challenger    15.5 150 3.520
#OUT> AMC Javelin         15.2 150 3.435
#OUT> Camaro Z28          13.3 245 3.840
#OUT> Pontiac Firebird    19.2 175 3.845
#OUT> Fiat X1-9           27.3  66 1.935
#OUT> Porsche 914-2       26.0  91 2.140
#OUT> Lotus Europa        30.4 113 1.513
#OUT> Ford Pantera L      15.8 264 3.170
#OUT> Ferrari Dino        19.7 175 2.770
#OUT> Maserati Bora       15.0 335 3.570
#OUT> Volvo 142E          21.4 109 2.780
```

How do you think `hp` and `wt` will influence how many miles per gallon of gasoline each of those cars can travel? In other words, what do you expect the signs of b_1 and b_2 to be?

With two explanatory variables as here, it is still possible to visualize the regression plane, so let's start with this as an answer. The OLS regression plane through this dataset looks like in figure 4.1:

This visualization shows a couple of things: the data are shown with red points and the grey plane is the one resulting from OLS estimation of equation (4.1). You should realize that this is exactly the same story as told in figure 3.1 - just in three dimensions!

Furthermore, *multiple* regression refers the fact that there could be *more* than two regressors. In fact, you could in principle have K regressors, and our theory developed so far would still be valid:

$$\hat{y}_i = b_0 + b_1 x_{1i} + b_2 x_{2i} + \cdots + b_K x_{Ki} \tag{4.2}$$

$$e_i = y_i - \hat{y}_i \tag{4.3}$$

Just as before, the least squares method chooses numbers $(b_0, b_1, \ldots, b_K)$ to as to minimize SSR, exactly as in the minimization problem for the one regressor case seen in (3.6).

WebGL is not supported by your browser - visit https://get.webgl.org for more info

Figure 4.1: Multiple Regression - a plane in 3D. The red lines indicate the residual for each observation.

4.1 All Else Equal

We can see from the above plot that cars with more horse power and greater weight, in general travel fewer miles per gallon of combustible. Hence, we observe a plane that is downward sloping in both the *weight* and *horse power* directions. Suppose now we wanted to know impact of `hp` on `mpg` *in isolation*, so as if we could ask

> Keeping the value of *wt* fixed for a certain car, what would be the impact on *mpg* be if we were to increase **only** its *hp*? Put differently, keeping **all else equal**, what's the impact of changing *hp* on *mpg*?

We ask this kind of question all the time in econometrics. In figure 4.1 you clearly see that both explanatory variables have a negative impact on the outcome of interest: as one increases either the horse power or the weight of a car, one finds that miles per gallon decreases. What is kind of hard to read off is *how negative* an impact each variable has in isolation.

As a matter of fact, the kind of question asked here is so common that it has got its own name: we'd say "*ceteris paribus*, what is the impact of `hp` on `mpg`?". *ceteris paribus* is latin and means *the others equal*, i.e. all other variables fixed. In terms of our model in (4.1), we want to know the following quantity:

$$\frac{\partial mpg_i}{\partial hp_i} = b_1 \tag{4.4}$$

This means: *keeping all other variables fixed, what is the effect of* ***hp*** *on* ***mpg****?*. In calculus, the answer to this is provided by the *partial derivative* as shown in (4.4). We call the value of coefficient b_1 therefore also the *partial effect* of `hp` on `mpg`. In terms of our dataset, we use `R` to run the following **multiple regression**:

```
#OUT>
#OUT> Call:
#OUT> lm(formula = mpg ~ wt + hp, data = mtcars)
#OUT>
#OUT> Residuals:
#OUT>    Min     1Q Median     3Q    Max
#OUT> -3.941 -1.600 -0.182  1.050  5.854
#OUT>
#OUT> Coefficients:
#OUT>             Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 37.22727    1.59879  23.285  < 2e-16 ***
#OUT> wt          -3.87783    0.63273  -6.129 1.12e-06 ***
#OUT> hp          -0.03177    0.00903  -3.519  0.00145 **
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 2.593 on 29 degrees of freedom
#OUT> Multiple R-squared:  0.8268,  Adjusted R-squared:  0.8148
#OUT> F-statistic: 69.21 on 2 and 29 DF,  p-value: 9.109e-12
```

From this table you see that the coefficient on `wt` has value -3.87783. You can interpret this as follows:

> Holding all other variables fixed at their observed values - or *ceteris paribus* - a one unit increase in *wt* implies a -3.87783 units change in *mpg*. Similarly, one more *hp* horse power implies a change in *mpg* of -0.03177 units, *all else (i.e. wt) equal*.

4.2 Multicolinearity

One important requirement for multiple regression is that the data be **not linearly dependent**: Each variable provides new information for the outcome, and it cannot be replicated as a linear combination of other variables. Suppose that in the example above, we had a variable `wtplus` defined as `wt + 1`, and we included this new variable together with `wt` in our regression. In this case, `wtplus` provides no new information. It's enough to know wt, and add 1 to it. In this sense, `wt_plus` is a redundant variable and should not be included in the model. Notice that this holds only for *linearly* dependent variables - *nonlinear* transformations (like for example wt^2) are exempt from this rule. Here is why:

$$
\begin{aligned}
y &= b_0 + b_1 \text{wt} + b_2 \text{wtplus} + e && (4.5)\\
&= b_0 + b_1 \text{wt} + b_2 (\text{wt} + 1) + e && (4.6)\\
&= (b_0 + b_2) + \text{wt}(b_1 + b_2) + e && (4.7)
\end{aligned}
$$

This shows that we cannot *identify* the regression coefficients in case of linearly dependent data. Variation in the variable `wt` identifies a different coefficient, say $\gamma = b_1 + b_2$, from what we actually wanted: separate estimates for b_1, b_2.

> All variables in a multiple regression should be *linearly independent*, or *not* colinear. This is known as the **rank condition**. In particular, this condition dictates that we need at least $N \geq K+1$, i.e. more observations than coefficients. The greater the degree of linear dependence amongst our explanatory variables, the less information we can get from them, and our estimates becomes *less precise*.

4.3 California Test Scores 2

Let us extend our example of student test scores from chapter 3 by adding families' average income to our previous model:

$$\text{testscr}_i = b_0 + b_1 \text{str}_i + b_2 \text{avginc}_i + e_i$$

We can incoporate this new variable to our model by simply adding it to our `formula`:

```r
library("Ecdat") # reload the data
fit_multivariate <- lm(formula = "testscr ~ str + avginc", data = Caschool)
summary(fit_multivariate)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = "testscr ~ str + avginc", data = Caschool)
#OUT>
#OUT> Residuals:
#OUT>     Min      1Q  Median      3Q     Max
```

WebGL is not supported by your browser - visit https://get.webgl.org for more info

Figure 4.2: Californa Test Scores vs student/teach ratio and avg income.

```
#OUT> -39.608  -9.052   0.707   9.259  31.898
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 638.72915    7.44908  85.746   <2e-16 ***
#OUT> str          -0.64874    0.35440  -1.831   0.0679 .
#OUT> avginc        1.83911    0.09279  19.821   <2e-16 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 13.35 on 417 degrees of freedom
#OUT> Multiple R-squared:  0.5115,  Adjusted R-squared:  0.5091
#OUT> F-statistic: 218.3 on 2 and 417 DF,  p-value: < 2.2e-16
```

Although it is quite cumbersome and not typical to visualize multivariate regressions, we can still do this with 2 explanatory variables using a *regression (2-dimensional) plane* [Interactive!].

While you explore this plot, ask yourself the following question: if you could only choose one of the two explanatory variables in our model (that is, either *str* or *avginc*) to predict the value of a given school's average test score, which one would you choose? Why? Discuss this with your classmates.

4.4 Interactions

Interactions allow that the *ceteris paribus* effect of a certain regressor, `str` say, depends also on the value of yet another regressor, `avginc` for example. In other words, do test scores depend differentially on the student teacher ratio, depending on wether the average income in an ares is high or low? Notice that `str` and `avginc` in isolation cannot answer that question (because the value of other variables is assumed *fixed*!). To measure such an effect, we would reformulate our model like this:

$$\text{testscr}_i = b_0 + b_1 \text{str}_i + b_2 \text{avginc}_i + b_3 (\text{str}_i \times \text{avginc}_i) + e_i \tag{4.8}$$

The inclusion of the *product* of `str` and `avginc` amounts to having different slopes with respect to `str` for different values of `avginc` (and vice versa). This is easy to see if we take the partial derivative of (4.8) with respect to `str`:

$$\frac{\partial \text{testscr}_i}{\partial \text{str}_i} = b_1 + b_3 \text{avginc}_i \tag{4.9}$$

> You should go back to equation (4.4) to remind yourself of what a *partial effect* was, and how exactly the present (4.9) differs from what we saw there.

Back in our `R` session, we can run the full interactions model like this:

```
lm_inter = lm(formula = testscr ~ str + avginc + str*avginc, data = Caschool)
# note that this would produce the same result:
# lm(formula = testscr ~ str*avginc, data = Caschool)
# R expands str*avginc for you in main effects + interactions
summary(lm_inter)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = testscr ~ str + avginc + str * avginc, data = Caschool)
#OUT>
#OUT> Residuals:
#OUT>     Min      1Q  Median      3Q     Max
#OUT> -41.346  -9.260   0.209   8.736  33.368
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 689.47473   14.40894  47.850  < 2e-16 ***
#OUT> str          -3.40957    0.75980  -4.487 9.34e-06 ***
#OUT> avginc       -1.62388    0.85214  -1.906   0.0574 .
#OUT> str:avginc    0.18988    0.04646   4.087 5.24e-05 ***
#OUT> ---
```

WebGL is not supported by your browser - visit https://get.webgl.org for more info

Figure 4.3: Californa Test Scores vs student/teach ratio and avg income plus interaction term

```
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 13.1 on 416 degrees of freedom
#OUT> Multiple R-squared:  0.5303,  Adjusted R-squared:  0.527
#OUT> F-statistic: 156.6 on 3 and 416 DF,  p-value: < 2.2e-16
```

We see here that the regression now estimates and additional coefficient b_3 for us. We observe also that the estimate of b_2 changes signs and becomes negative, while the interaction effect b_3 is positive. This means that an increase in `str` reduces average student scores (more students per teacher make it harder to teach effectively); that an increase in average district income in isolation actually reduces scores; and that the interaction of both increases scores (more students per teacher are actually a good thing for student performance in richer areas).

Looking at our visualization may help understand this result better. Figure 4.3 shows a plane that is no longer actually a *plane*. It shows a curved surface. You can see that the surface became more flexible in that we could kind of *bend* it more. Which model do you like better to explain this data? Discuss with your neighbor and give some reasons for your choice (other than “4.3 looks nicer” ;-)). In particular, comparing both visualizations, can you explain why we observe this strange inversion of coefficient signs?

Chapter 5

Categorial Variables

Up until now, we have encountered only examples with *continuous* variables x and y, that is, $x, y \in \mathbb{R}$, so that a typical observation could have been $(y_i, x_i) = (1.5, 5.62)$. There are many situations where it makes sense to think about the data in terms of *categories*, rather than continuous numbers. For example, whether an observation i is *male* or *female*, whether a pixel on a screen is *black* or *white*, and whether a good was produced in *France*, *Germany*, *Italy*, *China* or *Spain* are all categorical classifications of data.

Probably the simplest type of categorical variable is the *binary*, *boolean*, or just *dummy* variable. As the name suggests, it can take on only two values, `0` and `1`, or `TRUE` and `FALSE`.

5.1 The Binary Regressor Case

Even though this is an extremely parsimonious way of encoding that, it is a very powerful tool that allows us to represent that a certain observation i **is a member** of a certain category j. For example, let's imagine we have income data on males and females, and we would create a variable called `is.male` that is `TRUE` whenever i is male, `FALSE` otherwise, and similarly for women. For example, to encode whether subject i is male, one could do this:

$$\text{is.male}_i = \begin{cases} 1 & \text{if } i \text{ is male} \\ 0 & \text{if } i \text{ is not male.} \end{cases},$$

and similarly for females, we'd have

$$\text{is.female}_i = \begin{cases} 1 & \text{if } i \text{ is female} \\ 0 & \text{if } i \text{ is not female.} \end{cases}$$

By definition, we have just introduced a linear dependence into our dataset. It will always be true that $\text{is.male}_i + \text{is.female}_i = 1$. This is because dummy variables are based on data being mutually exclusively categorized - here, you are either male or female.[1] This should immediately remind you of section 4.2 where we introduced *multicolinearity*. A regression of income on both of our variables like this

$$y_i = b_0 + b_1 \text{is.female}_i + b_2 \text{is.male}_i + e_i$$

would be invalid because of perfect colinearity between is.female_i and is.male_i. The solution to this is pragmatic and simple:

> In dummy variable regressions, we remove one category from the regression (for example here: `is.male`) and call it the *reference category*. The effect of being *male* is absorbed in the intercept. The coefficient on the remaining categories measures the *difference* in mean outcome with respect to the reference category.

Now let's try this out. We start by creating the female indicator as above,

$$\text{is.female}_i = \begin{cases} 1 & \text{if } i \text{ is female} \\ 0 & \text{if } i \text{ is not female.} \end{cases}$$

and let's suppose that y_i is a measure of i's annual labor income. Our model is

$$y_i = b_0 + b_1 \text{is.female}_i + e_i \tag{5.1}$$

and here is how we estimate this in `R`:

```r
# x = sample(x = c(0, 1), size = n, replace = T)
dta$is.female = factor(x)  # convert x to factor
dummy_reg = lm(y~is.female,dta)
summary(dummy_reg)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = y ~ is.female, data = dta)
```

[1] There are transgender individuals where this example will not apply.

```
#OUT> 
#OUT> Residuals:
#OUT>     Min      1Q  Median      3Q     Max 
#OUT> -2.4253 -0.6551  0.1321  0.7217  2.8129 
#OUT> 
#OUT> Coefficients:
#OUT>             Estimate Std. Error t value Pr(>|t|)    
#OUT> (Intercept)   2.0568     0.2125   9.680 7.27e-13 ***
#OUT> is.female1   -3.0169     0.3203  -9.418 1.74e-12 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT> 
#OUT> Residual standard error: 1.124 on 48 degrees of freedom
#OUT> Multiple R-squared:  0.6489,	Adjusted R-squared:  0.6416 
#OUT> F-statistic:  88.7 on 1 and 48 DF,  p-value: 1.737e-12
```

Notice that `R` displays the *level* of the factor to which coefficient b_1 belongs here, i.e. `is.female1` means this coefficient is on level `is.female = 1` - the reference level is `is.female = 0`, and it has no separate coefficient. Also interesting is that b_1 is equal to the difference in conditional means between male and female

$$b_1 = E[y|\text{is.female} = 1] - E[y|\text{is.female} = 0] = -3.0169.$$

A dummy variable measures the difference or the *offset* in the mean of the response variable, $E[y]$, **conditional** on x belonging to some category - relative to a baseline category. In our artificial example, the coefficient b_1 informs us that women earn on average 3.756 units less than men.

It is instructive to reconsider this example graphically:

In figure 5.1 we see that this regression simplifies to the straight line connecting the mean, or the *expected value* of y when $\text{is.female}_i = 0$, i.e. $E[y|\text{is.female}_i = 0]$, to the mean when $\text{is.female}_i = 1$, i.e. $E[y|\text{is.female}_i = 1]$. It is useful to remember that the *unconditional mean* of y, i.e. $E[y]$, is going to be the result of regressing y only on an intercept, illustrated by the blue line. This line will always lie in between both conditional means. As indicated by the red arrow, the estimate of the coefficient on the dummy, b_1, is equal to the difference in conditional means for both groups. You should look at our app now to deepen your understanding of what's going on here:

```r
library(ScPoEconometrics)
launchApp("reg_dummy")
```

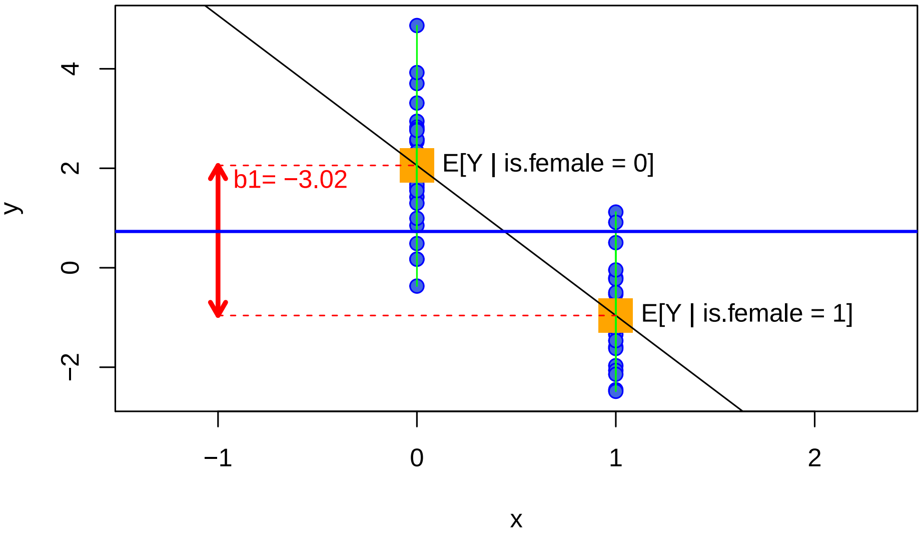


Figure 5.1: regressing $y \in \mathbb{R}$ on $\text{is.female}_i \in \{0, 1\}$. The blue line is $E[y]$, the red arrow is the size of b_1. Which is the same as the slope of the regression line in this case and the difference in conditional means!

5.2 Dummy and Continuous Variables

What happens if there are more predictors than just the dummy variable in a regression? For example, what if instead we had

$$y_i = b_0 + b_1 \text{is.female}_i + b_2 \text{exper}_i + e_i \tag{5.2}$$

where exper_i would measure years of experience in the labor market? As above, the dummy variable acts as an intercept shifter. We have

$$y_i = \begin{cases} b_0 + b_1 + b_2 \times \text{exper}_i + e_i & \text{if is.female=1} \\ b_0 + \quad\;\; + b_2 \times \text{exper}_i + e_i & \text{if is.female=0} \end{cases} \tag{5.3}$$

so that the intercept is $b_0 + b_1$ for women but b_0 for men. We will see this in the real-world example below, but for now let's see the effect of switching the dummy *on* and *off* in this app:

```r
library(ScPoEconometrics)
launchApp("reg_dummy_example")
```

5.3 Categorical Variables in `R`: `factor`

`R` has extensive support for categorical variables built-in. The relevant data type representing a categorical variable is called `factor`. We encountered them as basic data types in section 1.8 already, but it is worth repeating this here. We have seen that a factor *categorizes* a usually small number of numeric values by *labels*, as in this example which is similar to what I used to create regressor `is.female` for the above regression:

```r
is.female = factor(x = c(0,1,1,0), labels = c(FALSE,TRUE))
is.female
```

```
#OUT> [1] FALSE TRUE  TRUE  FALSE
#OUT> Levels: FALSE TRUE
```

You can see the result is a vector object of type `factor` with 4 entries, whereby `0` is represented as `FALSE` and `1` as `TRUE`. An other example could be if we wanted to record a variable *sex* instead, and we could do

```r
sex = factor(x = c(0,1,1,0), labels = c("male","female"))
sex
```

```
#OUT> [1] male   female female male
#OUT> Levels: male female
```

You can see that this is almost identical, just the *labels* are different.

5.3.1 More Levels

We can go beyond *binary* categorical variables such as `TRUE` vs `FALSE`. For example, suppose that x measures educational attainment, i.e. it is now something like $x_i \in \{\text{high school,some college,BA,MSc}\}$. In `R` parlance, *high school, some college, BA, MSc* are the **levels of factor** x. A straightforward extension of the above would dictate to create one dummy variable for each category (or level), like

$$
\begin{aligned}
\text{has.HS}_i &= \mathbf{1}[x_i == \text{high school}] \\
\text{has.someCol}_i &= \mathbf{1}[x_i == \text{some college}] \\
\text{has.BA}_i &= \mathbf{1}[x_i == \text{BA}] \\
\text{has.MSc}_i &= \mathbf{1}[x_i == \text{MSc}]
\end{aligned}
$$

but you can see that this is cumbersome. There is a better solution for us available:

```r
factor(x = c(1,1,2,4,3,4),labels = c("HS","someCol","BA","MSc"))
```

```
#OUT> [1] HS      HS      someCol MSc     BA      MSc
#OUT> Levels: HS someCol BA MSc
```

Notice here that `R` will apply the labels in increasing order the way you supplied it (i.e. a numerical value 4 will correspond to "MSc", no matter the ordering in `x`.)

5.3.2 `factor` and `lm()`

The above developed `factor` terminology fits neatly into `R`'s linear model fitting framework. Let us illustrate the simplest use by way of example.

```r
library(Ecdat)  # need to load this library
data("Wages")   # from Ecdat
str(Wages)   # let's examine this dataset!
```

```
#OUT> 'data.frame': 4165 obs. of  12 variables:
#OUT>  $ exp    : int  3 4 5 6 7 8 9 30 31 32 ...
#OUT>  $ wks    : int  32 43 40 39 42 35 32 34 27 33 ...
#OUT>  $ bluecol: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 2 2 2 ...
#OUT>  $ ind    : int  0 0 0 0 1 1 1 0 0 1 ...
#OUT>  $ south  : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 1 1 1 ...
#OUT>  $ smsa   : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
#OUT>  $ married: Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
#OUT>  $ sex    : Factor w/ 2 levels "female","male": 2 2 2 2 2 2 2 2 2 2 ...
#OUT>  $ union  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 2 ...
#OUT>  $ ed     : int  9 9 9 9 9 9 9 11 11 11 ...
#OUT>  $ black  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
#OUT>  $ lwage  : num  5.56 5.72 6 6 6.06 ...
```

Notice here that, conveniently, `sex` is already coded as type `factor`. Now assume that this is a single cross section for wages of US workers. The main outcome variable is `lwage` which stands for *logarithm of wage*.

5.3.3 Log Transformations

It is quite common to transform either outcome or explanatory or both variables by the natural logarithm. The primary motivation of this is to make the regression scale invariant. Suppose that factor α represented the *scale* of measurement of income, so that $\alpha = 1$ if we measure in Euros, or $\alpha = 1000$ if in thousands of Euros. With log-transforming regressor x, our equation would look like

$$y = b_0 + b_1 \log(\alpha x) = b_0 + \log \alpha + b_1 \log x$$

where the *scale* α moves into the intercept, and our slope coefficient becomes invariant to it. If both outcome and regressor are transformed, we have

$$\log y = b_0 + b_1 \log x$$

and the slope coefficient is

$$b_1 = \frac{d \log y}{d \log x} = \frac{dy/y}{dx/x}$$

which represents the **elasticity** of y with respect to x: what is the percentage change in y if x changes by one percent?

Finally, if *only the outcome* is log transformed, but not the regressor, we have a *semi-elasticity* formulation.

$$\log y = b_0 + b_1 x$$

and the slope coefficient becomes

$$b_1 = \frac{d \log y}{dx}$$

This means a one-unit change in x increases the logarithm of the outcome by b_1 units. For small changes in x, we can just exponentiate b_1 to get the effect of x the *level* of y.

Going back to our example, let's say that a workers wage depends only on his *experience*, measured in the number of years he/she worked full-time:

$$\ln w_i = b_0 + b_1 exp_i + e_i \tag{5.4}$$

```r
lm_w = lm(lwage ~ exp, data = Wages)
summary(lm_w)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = lwage ~ exp, data = Wages)
#OUT>
#OUT> Residuals:
#OUT>      Min       1Q   Median       3Q      Max
#OUT> -2.30153 -0.29144  0.02307  0.27927  1.97171
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 6.5014318  0.0144657  449.44   <2e-16 ***
#OUT> exp         0.0088101  0.0006378   13.81   <2e-16 ***
#OUT> ---
```

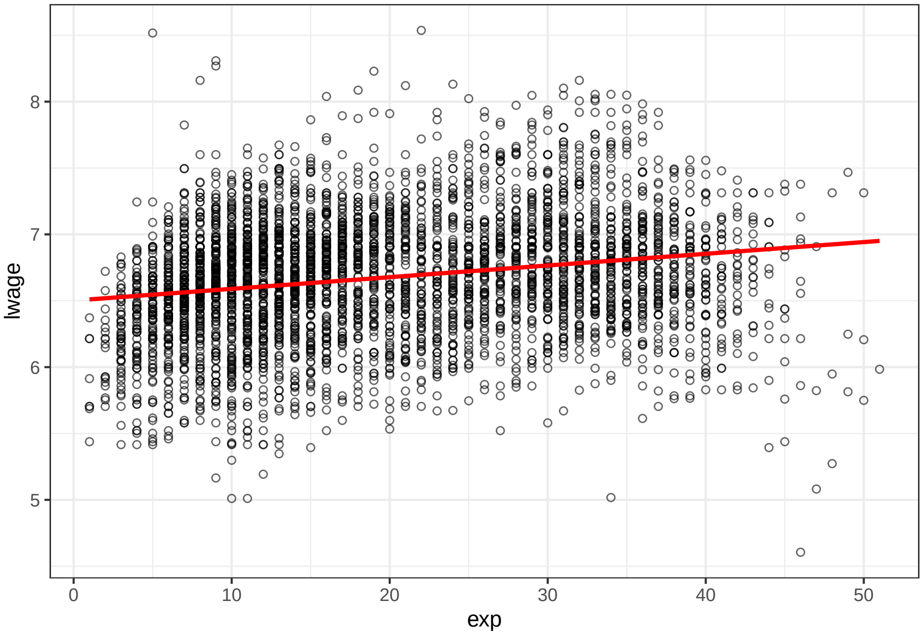


Figure 5.2: log wage vs experience. Red line shows the regression.

```
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 0.4513 on 4163 degrees of freedom
#OUT> Multiple R-squared:  0.04383, Adjusted R-squared:  0.0436
#OUT> F-statistic: 190.8 on 1 and 4163 DF,  p-value: < 2.2e-16
```

We see from this that an additional year of full-time work experience will increase the mean of $\ln w$ by 0.0088. Given the log transformation on wages, we can just exponentiate that to get an estimated effect on the (geometric!) mean of wages as $\exp(\hat{b}_1) = 1.0088491$. This means that hourly wages increase by roughly $100 * (\exp(b_1) - 1) = 0.88$ percent with an additional year of experience. We can verify the positive relationship in figure 5.2.

Now let's investigate whether this relationship is different for men and women. We can do this by just including the `factor` variable `sex`:

$$\ln w_i = b_0 + b_1 exp_i + b_2 sex_i + e_i \tag{5.5}$$

In R we can do this easily by using the `update` function as follows:

```
lm_sex = update(lm_w, . ~ . + sex)  # update lm_w with same LHS, same RHS, but add sex
summary(lm_sex)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = lwage ~ exp + sex, data = Wages)
#OUT>
#OUT> Residuals:
#OUT>      Min       1Q   Median       3Q      Max
#OUT> -1.87081 -0.26688  0.01733  0.26336  1.90325
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 6.1257661  0.0223319  274.31   <2e-16 ***
#OUT> exp         0.0076134  0.0006082   12.52   <2e-16 ***
#OUT> sexmale     0.4501101  0.0210974   21.34   <2e-16 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 0.4286 on 4162 degrees of freedom
#OUT> Multiple R-squared:  0.1381,  Adjusted R-squared:  0.1377
#OUT> F-statistic: 333.4 on 2 and 4162 DF,  p-value: < 2.2e-16
```

What's going on here? Remember from above that `sex` is a `factor` with 2 levels *female* and *male*. We see in the above output that `R` included a regressor called `sexmale` $= \mathbf{1}[sex_i == male]$. This is a combination of the variable name `sex` and the level which was included in the regression. In other words, `R` chooses a *reference category* (by default the first of all levels by order of appearance), which is excluded - here this is `sex=="female"`. The interpretation is that b_2 measures the effect of being male *relative* to being female. `R` automatically creates a dummy variable for each potential level, excluding the first category. In particular, if `sex` had a third category `dont want to say`, there would be an additional regressor called `sexdontwanttosay`.

Figure 5.3 illustrates this. You can see that both male and female have the same upward sloping regression line. But you can also see that there is a parallel downward shift from male to female line. The estimate of $b_2 = 0.45$ is the size of the downward shift.

5.4 Saturated Models: Main Effects and Interactions

You can see above that we *restricted* male and female to have the same slope with repect to years of experience. This may or may not be a good assumption. Thankfully, the dummy variable regression machinery allows for a quick solution to this - so-called *interaction* effects. As already introduced in chapter 4.4, interactions allow that the *ceteris paribus* effect of a certain regressor, `exp` say,

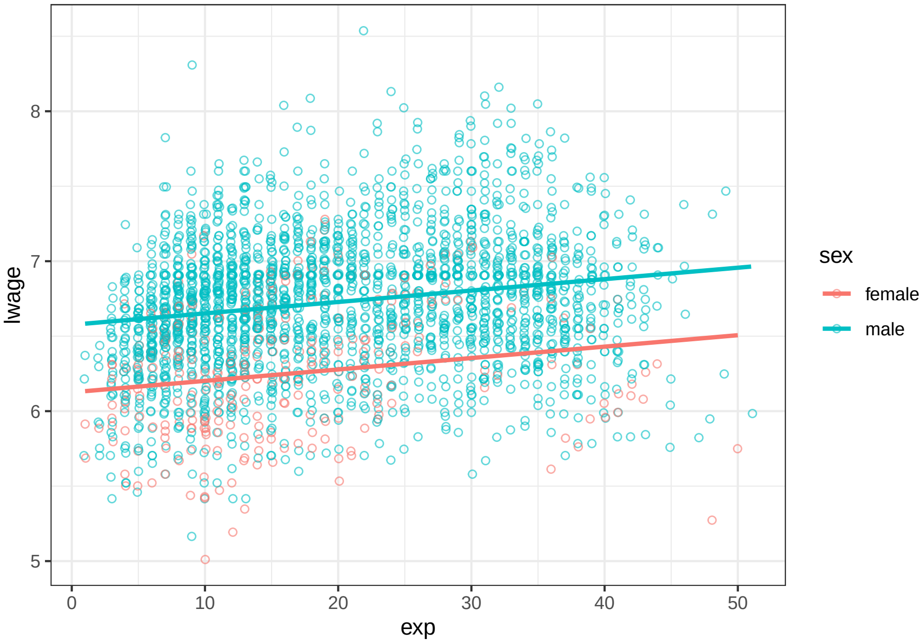


Figure 5.3: log wage vs experience with different intercepts by sex

depends also on the value of yet another regressor, `sex` for example. Suppose then we would like to see whether male and female not only have different intercepts, but also different slopes with respect to `exp` in figure 5.3. Therefore we formulate this version of our model:

$$\ln w_i = b_0 + b_1 exp_i + b_2 sex_i + b_3(sex_i \times exp_i) + e_i \tag{5.6}$$

The inclusion of the *product* of `exp` and `sex` amounts to having different slopes for different categories in `sex`. This is easy to see if we take the partial derivative of (5.6) with respect to `sex`:

$$\frac{\partial \ln w_i}{\partial sex_i} = b_2 + b_3 exp_i \tag{5.7}$$

Back in our `R` session, we can run the full interactions model like this:

```
lm_inter = lm(lwage ~ exp*sex, data = Wages)
summary(lm_inter)
```

```
#OUT>
#OUT> Call:
#OUT> lm(formula = lwage ~ exp * sex, data = Wages)
```

```
#OUT> 
#OUT> Residuals:
#OUT>      Min       1Q   Median       3Q      Max 
#OUT> -1.82137 -0.26797  0.01781  0.26231  1.90757 
#OUT> 
#OUT> Coefficients:
#OUT>             Estimate Std. Error t value Pr(>|t|)    
#OUT> (Intercept) 6.169017   0.038165 161.643  < 2e-16 ***
#OUT> exp         0.005071   0.001918   2.644  0.00822 ** 
#OUT> sexmale     0.401116   0.040917   9.803  < 2e-16 ***
#OUT> exp:sexmale 0.002826   0.002022   1.397  0.16236    
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT> 
#OUT> Residual standard error: 0.4285 on 4161 degrees of freedom
#OUT> Multiple R-squared:  0.1385,  Adjusted R-squared:  0.1379 
#OUT> F-statistic:   223 on 3 and 4161 DF,  p-value: < 2.2e-16
```

You can see here that `R` automatically expands `exp*sex` to include both *main effects*, i.e. `exp` and `sex` as single regressors as before, and their interaction, denoted by `exp:sexmale`. It turns out that in this example, the estimate for the interaction is not statistically significant, i.e. we cannot reject the null hypothesis that $b_3 = 0$. (If, for some reason, you wanted to include only the interaction, you could supply directly `formula = lwage ~ exp:sex` to `lm`, although this would be a rather difficult to interpret model.)

We call a model like (5.6) a *saturated model*, because it includes all main effects and possible interactions. What our little exercise showed us was that with the sample of data at hand, we cannot actually claim that there exists a differential slope for male and female, so the model with main effects only may be more appropriate here.

To finally illustrate the limits of interpretability when including interactions, suppose we run the fully saturated model for `sex`, `smsa`, `union` and `bluecol`, including all main and all interaction effects:

```
lm_full = lm(lwage ~ sex*smsa*union*bluecol,data=Wages)
summary(lm_full)
```

```
#OUT> 
#OUT> Call:
#OUT> lm(formula = lwage ~ sex * smsa * union * bluecol, data = Wages)
#OUT> 
#OUT> Residuals:
#OUT>      Min       1Q   Median       3Q      Max 
#OUT> -1.95214 -0.23409 -0.01681  0.25317  1.90450 
#OUT> 
#OUT> Coefficients:
```

```
#OUT>                                     Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept)                          6.12378    0.06300  97.198  < 2e-16
#OUT> sexmale                              0.67057    0.06577  10.195  < 2e-16
#OUT> smsayes                              0.33424    0.06872   4.864 1.19e-06
#OUT> unionyes                             0.84284    0.16866   4.997 6.06e-07
#OUT> bluecolyes                          -0.34016    0.08423  -4.039 5.47e-05
#OUT> sexmale:smsayes                     -0.15917    0.07226  -2.203 0.027670
#OUT> sexmale:unionyes                    -0.92893    0.17816  -5.214 1.94e-07
#OUT> smsayes:unionyes                    -0.83927    0.17979  -4.668 3.14e-06
#OUT> sexmale:bluecolyes                  -0.15046    0.08820  -1.706 0.088100
#OUT> smsayes:bluecolyes                  -0.12471    0.09727  -1.282 0.199882
#OUT> unionyes:bluecolyes                 -0.31819    0.22924  -1.388 0.165208
#OUT> sexmale:smsayes:unionyes             0.72672    0.19060   3.813 0.000139
#OUT> sexmale:smsayes:bluecolyes           0.25860    0.10327   2.504 0.012318
#OUT> sexmale:unionyes:bluecolyes          0.71906    0.23772   3.025 0.002503
#OUT> smsayes:unionyes:bluecolyes          0.50057    0.24862   2.013 0.044137
#OUT> sexmale:smsayes:unionyes:bluecolyes -0.58330    0.25899  -2.252 0.024361
#OUT>
#OUT> (Intercept)                         ***
#OUT> sexmale                             ***
#OUT> smsayes                             ***
#OUT> unionyes                            ***
#OUT> bluecolyes                          ***
#OUT> sexmale:smsayes                     *
#OUT> sexmale:unionyes                    ***
#OUT> smsayes:unionyes                    ***
#OUT> sexmale:bluecolyes                  .
#OUT> smsayes:bluecolyes
#OUT> unionyes:bluecolyes
#OUT> sexmale:smsayes:unionyes            ***
#OUT> sexmale:smsayes:bluecolyes          *
#OUT> sexmale:unionyes:bluecolyes         **
#OUT> smsayes:unionyes:bluecolyes         *
#OUT> sexmale:smsayes:unionyes:bluecolyes *
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 0.3832 on 4149 degrees of freedom
#OUT> Multiple R-squared:  0.3129,  Adjusted R-squared:  0.3105
#OUT> F-statistic:   126 on 15 and 4149 DF,  p-value: < 2.2e-16
```

The main effects remain clear to interpret: being a blue collar worker, for example, reduces average wages by 34% relative to white collar workers. One-way interactions are still ok to interpret as well: `sexmale:bluecolyes` indicates in addition to a wage premium over females of 0.67, and a penalty of being blue collar of -0.34, **male** blue collar workers suffer an additional wage loss of -0.15.

All of this is relative to the base category, which are female white collar workers who don't live in an smsa and are not union members. If we now add a third or even a fourth interaction, this becomes much harder to interpret, and in fact we rarely see such interactions in applied work.

Chapter 6

Standard Errors

In the previous chapters we have seen how the OLS method can produce estimates about intercept and slope coefficients from data. You have seen this method at work in `R` by using the `lm` function as well. It is now time to introduce the notion that given that b_0, b_1 and b_2 are *estimates* of some unkown *population parameters*, there is some degree of **uncertainty** about their values. An other way to say this is that we want some indication about the *precision* of those estimates.

> How *confident* should we be about the estimated values b?

Let's go back to the regression with one variable and remind ourselves of the example at the end of chapter 3. There we introduced the term *confidence interval*, shown as the shaded area in figure 6.1:

The shaded area shows us the region within which the **true** red line will lie with 95% probability. The fact that there is an unknown true line (i.e. a *true* slope coefficient β_1) that we wish to uncover from a sample of data should remind you immediately of our first tutorial. There, we wanted to estimate the true population mean from a sample of data, and we saw that as the sample size N increased, our estimate got better and better - fundamentally this is the same idea here.

6.1 What is *true*? What are Statistical Models?

A **statistical model** is simply a set of assumptions about how some data have been generated. As such, it models the data-generating process (DGP), as we have it in mind. Once we define a DGP, we could simulate data from it and see how this compares to the data we observe in the real world. Or, we could

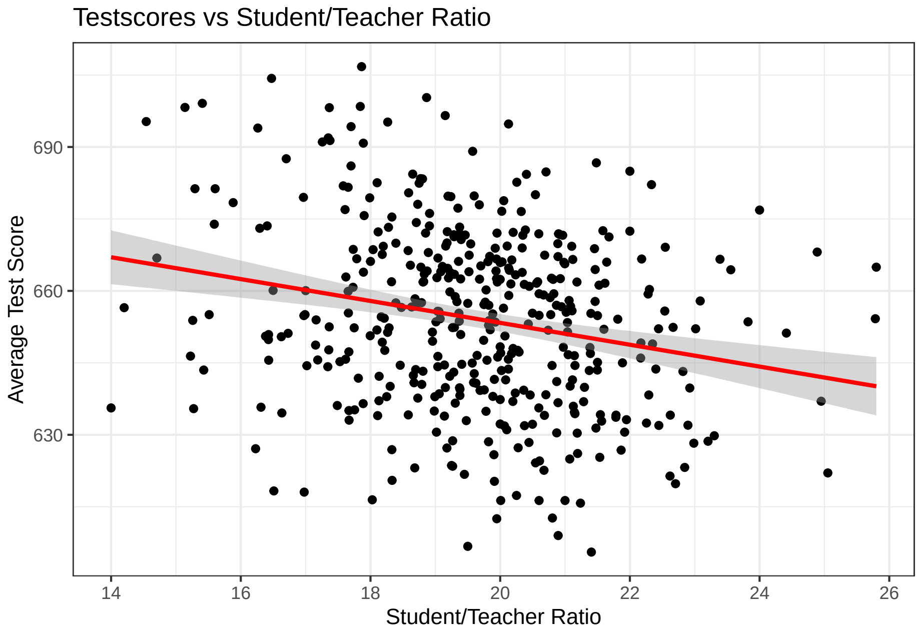


Figure 6.1: Confidence bands around a regression line.

change the parameters of the DGP so as to understand how the real world data *would* change, could we (or some policy) change the corresponding parameters in reality. Let us now consider one particular statistical model, which in fact we have seen so many times already.

6.2 The Classical Regression Model

Let's bring back our simple model (3.3) to explain this concept.

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \tag{6.1}$$

The smallest set of assumptions used to define the *classical regression model* as in (6.1) are the following:

1. The data are **not linearly dependent**: Each variable provides new information for the outcome, and it cannot be replicated as a linear combination of other variables. We have seen this in section 4.2. In the particular case of one regressor, as here, we require that x exhibit some variation in the data, i.e. $Var(x) \neq 0$.
2. The mean of the residuals conditional on x should be zero, $E[\varepsilon|x] = 0$. Notice that this also means that $Cov(\varepsilon, x) = 0$, i.e. that the errors and

our explanatory variable(s) should be *uncorrelated.* It is said that x should be **strictly exogenous** to the model.

These assumptions are necessary to successfully (and correctly!) run an OLS regression. They are often supplemented with an additional set of assumptions, which help with certain aspects of the exposition, but are not strictly necessary:

3. The data are drawn from a **random sample** of size n: observation (x_i, y_i) comes from the exact same distribution, and is independent of observation (x_j, y_j), for all $i \neq j$.
4. The variance of the error term ε is the same for each value of x: $Var(\varepsilon|x) = \sigma^2$. This property is called **homoskedasticity**.
5. The error is normally distributed, i.e. $\varepsilon \sim \mathcal{N}(0, \sigma^2)$

Invoking assumption 5. in particular defines what is commonly called the *normal* linear regression model.

6.2.1 b is not β!

Let's talk about the small but important modifications we applied to model (3.3) to end up at (6.1) above:

- β_0 and β_1 and intercept and slope parameters
- ε is the error term.

First, we *assumed* that (6.1) is the correct represenation of the DGP. With that assumption in place, the values β_0 and β_1 are the *true parameter values* which generated the data. Notice that β_0 and β_1 are potentially different from b_0 and b_1 in (3.3) for a given sample of data - they could in practice be very close to each other, but b_0 and b_1 are *estimates* of β_0 and β_1. And, crucially, those estimates are generated from a sample of data. Now, the fact that our data $\{y_i, x_i\}_{i=1}^N$ are a sample from a larger population, means that there will be *sampling variation* in our estimates - exactly like in the case of the sample mean estimating the population average as mentioned above. One particular sample of data will generate one particular set of estimates b_0 and b_1, whereas another sample of data will generate estimates which will in general be different - by *how much* those estimates differ across samples is the question in this chapter. In general, the more observations we have the greater the precision of our estimates, hence, the closer the estimates from different samples will lie together.

6.2.2 Standard Errors in Theory

The standard deviation of the OLS parameters is generally called *standard error*. As such, it is just the square root of the parameter's variance. Under assumptions 1. through 4. we can define the formula for the variance of our slope coefficient in the context of our single regressor model (6.1) as follows:

$$Var(b_1|x_i) = \frac{\sigma^2}{\sum_i^N (x_i - \bar{x})^2} \tag{6.2}$$

In pratice, we don't know the theoretical variance of ε, i.e. σ^2, but we form an estimate about it from our sample of data. A widely used estimate uses the already encountered SSR (sum of squared residuals), and is denoted s^2:

$$s^2 = \frac{SSR}{n-p} = \frac{\sum_{i=1}^n (y_i - b_0 - b_1 x_i)^2}{n-p} = \frac{\sum_{i=1}^n e_i^2}{n-p}$$

where $n-p$ are the *degrees of freedom* available in this estimation. p is the number of parameters we wish to estimate (here: 1). So, the variance formula would become

$$Var(b_1|x_i) = \frac{SSR}{(n-p)\sum_i^N (x_i - \bar{x})^2} \tag{6.3}$$

We most of the time work directly with the *standard error* of a coefficient, hence we define

$$SE(b_1) = \sqrt{Var(b_1|x_i)} = \sqrt{\frac{SSR}{(n-p)\sum_i^N (x_i - \bar{x})^2}} \tag{6.4}$$

You can clearly see that, as n increases, the denominator increases, and therefore variance and standard error of the estimate will decrease.

6.2.3 Standard Errors in Practice

We would like to further make this point in an experiential way, i.e. we want you to experience what is going on. We invite you to spend some time with the following apps. In particular, make sure you have a thorough understanding of `launchApp("estimate")`.

```
library(ScPoEconometrics)
launchApp("estimate")
launchApp("sampling")
launchApp("standard_errors_simple")
launchApp("standard_errors_changeN")
```

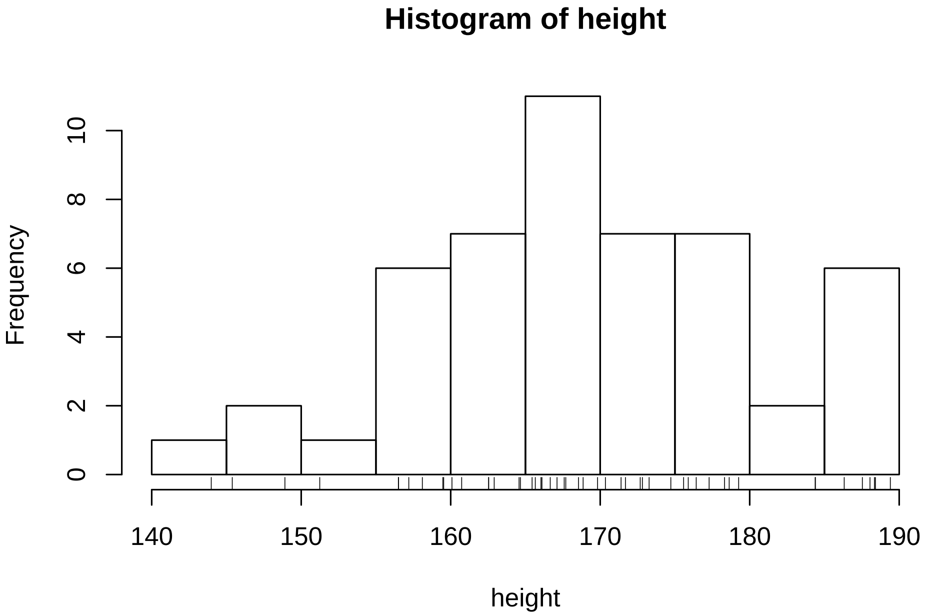


Figure 6.2: Our ficitious sample of SciencesPo students' body height. The small ticks indicate the location of each measurement.

6.3 Back to Sampling

Imagine we were tasked by the Director of our school to provide him with our best guess of the *mean body height* μ amongst all SciencesPo students in order to assess which height the new desks should have. Of course, we are econometricians and don't *guess* things: we **estimate** them! How would we go about this task and estimate μ?

You may want to ask: Why bother with this estimation business at all, and not just measure all students' height, compute μ, and that's it? That's a good question! In most cases, we cannot do this, either because we do not have access to the entire population (think of computing the mean height of all Europeans!), or it's too costly to measure everyone, or it's impractical. That's why we take *samples* from the wider population, to make inference. In our example, suppose we'd randomly measure students coming out of the SciencesPo building at 27 Rue Saint Guillaume until we have 50 measurements on any given Monday. Suppose further that we found a sample mean height $\bar{x} = 168.5$, and that the sample standard deviation was $s = 10$. In short, we found the data summarized in figure 6.2

What are we going to tell *Monsieur le Directeur* now, with those two numbers and figure 6.2 in hand? Before we address this issue, we need to make a short detour into *test statistics*.

6.3.1 Test Statistics

We have encountered many statistics already: think of the sample mean, or the standard deviation. Statistics are just functions of data. *Test* statistics are used to perform statistical tests.

Many test statistics rely on some notion of *standardizing* the sample data so that it becomes comparable to a theoretical distribution. We encountered this idea already in section 3.2.4, where we talked about a standardized regression. The most common standardization is the so-called *z-score*, which says that

$$\frac{x - \mu}{\sigma} \equiv z \sim \mathcal{N}(0, 1), \tag{6.5}$$

in other words, substracting the population mean from random variable x and dividing by it's population standard deviation yields a standard normally distributed random variable, commonly called z.

A very similar idea applies if we *don't know* the population variance (which is our case here!). The corresponding standardization gives rise to the *t-statistic*, and it looks very similar to (6.5):

$$\sqrt{n}\frac{\bar{x} - \mu}{s} \equiv T \sim t_{n-1} \tag{6.6}$$

Several things to note:

- We observe the same standardization as above: dividing by the sample standard deviation s brings $\bar{x} - \mu$ to a *unit free* scale.
- We use $\bar{x}$ and s instead of x and σ
- We multiply by $\sqrt{n}$ because we expect $\bar{x} - \mu$ to be a small number: we need to *rescale* it again to make it compatible with the t_{n-1} distribution.
- t_{n-1} is the Student's T distribution with $n - 1$ degrees of freedom. We don't have n degrees of freedom because we already had to estimate one statistic ($\bar{x}$) in order to construct T.

6.3.2 Confidence Intervals

Back to our example now! We are clearly in need of some measure of *confidence* about our sample statistic $\bar{x} = 168.5$ before we communicate our result. It seems reasonable to inform the Director about $\bar{x}$, but surely we also need to tell him that there was considerable *dispersion* in the data: Some people were as short as 143.98cm, while others were as tall as 189.41cm!

The way to proceed is to construct a *confidence interval* about the true population mean μ, based on $\bar{x}$, which will take this uncertainty into account. We will use the t statistic from above. We want to have a *symmetric interval* around

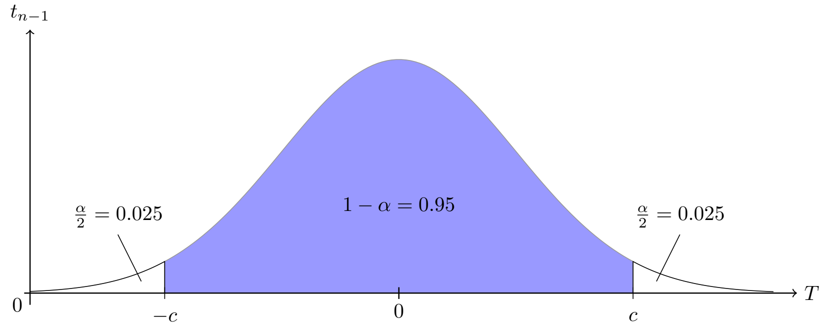


Figure 6.3: Confidence Interval Construction. The blue area is called *coverage region* which contains the true μ with probability $1 - \alpha$.

$\bar{x}$ which contains the true value μ with probability $1 - \alpha$. One very popular choice of α is 0.05, hence we cover μ with 95% probability. After computing our statistic T as defind in (6.6), this interval is defined as follows:

$$\Pr\left(-c \leq T \leq c\right) = 1 - \alpha \tag{6.7}$$

where c stands for *critical value*, which we need to choose. This is illustrated in figure 6.3.

Given the symmetry of the t distribution it's enough to find c at the upper tail: the point above which $\frac{\alpha}{2}$ of all probability mass of the t_{df} distribution comes to lie. In other words, if $\mathcal{T}_{df}$ is the CDF of the t distribution with df degrees of freedom, we find c as

$$\mathcal{T}_{df}(c) \equiv \Pr\left(T < c\right) = 1 - \frac{\alpha}{2} = 0.975 \tag{6.8}$$

$$c = \mathcal{T}_{df}^{-1}(\mathcal{T}_{df}(c)) = \mathcal{T}_{df}^{-1}(0.975) \tag{6.9}$$

Here $\mathcal{T}_{df}^{-1}$ stands for the *quantile function*, i.e. the inverse of the CDF. In our example with $df = 49$, you can find thus that $c = 2.01$ by typing `qt(0.975,df=49)` into your `R` session.[1] Now we only have to expand the definition of the T statistic from (6.6) inside (6.7) to obtain

[1] You often will see $c = 1.96$, which comes from the fact that one relies on the t distribution converging to the normal distribution with large n. Type `qnorm(0.975)` to confirm!

$$
\begin{aligned}
0.95 = 1-\alpha &= \Pr\left(-c \leq T \leq c\right) && (6.10)\\
&= \Pr\left(-2.01 \leq \sqrt{n}\frac{\bar{x}-\mu}{s} \leq 2.01\right) && (6.11)\\
&= \Pr\left(\bar{x} - 2.01\frac{s}{\sqrt{n}} \leq \mu \leq \bar{x} + 2.01\frac{s}{\sqrt{n}}\right) && (6.12)
\end{aligned}
$$

Finally, filling in our numbers for s etc, this implies that a 95% confidence interval about the location of the true average height of all SciencesPo students, μ, is given by:

$$CI = [165.658, 171.342] \qquad (6.13)$$

We would tell the director that with 95% probability, the true average height of all students comes to lie within those two bounds.

Finally, looking back at figure 6.1 above, the shaded area is just the 95% confidence interval *about the true value* β_1. We would say that *the true regression line* is contained within the shaded region with 95% probability. Very similarly to our example of $\bar{x}$, in that picture we have instead an estimate b_1, with an associated standard error $SE(b_1)$. The shaded area is called *confidence band*, and it is just plotting the confidence interval *for each value* x in the data. You can see how the band becomes narrower (i.e. the estimate becomes more precise) if there is more data associated to a certain x.

6.4 Hypothesis Testing

Now know by now how the standard errors of an OLS estimate are computed, and what they stand for. We can now briefly[2] discuss a very common usage of this information, in relation to which variables we should include in our regression. There is a statistical proceedure called *hypothesis testing* which helps us to make such decisions. In hypothesis testing, we have a baseline, or *null* hypothesis H_0, which we want to confront with a competing *alternative* hypthesis H_1. Continuing with our example of the mean height of SciencesPo students (μ), one potential hypothesis could be

$$
\begin{aligned}
H_0 :&\mu = 167 && (6.14)\\
H_1 :&\mu \neq 167 && (6.15)
\end{aligned}
$$

[2] We will not go into great detail here. Please refer back to your statistics course from last spring semester (chapters 8 and 9), or the short note I wrote while ago

Here we state that under the null hypthesis, $\mu = 167$, and under the alternative, it's not equal to that value. This would be called a *two-sided* test, because it tests deviations from H_0 below as well as above. An alternative formulation could use the *one-sided* test that

$$H_0 : \mu = 167 \tag{6.16}$$
$$H_1 : \mu > 167. \tag{6.17}$$

which would mean: under the null hypothesis, the average of all ScPo students' body height is 167cm. Under the alternative, it is larger. You can immediately see that this is very similar to confidence interval construction.

Suppose as above that we found $\bar{x} = 168.5$, and that the sample standard deviation is still $s = 10$. Would you regard this as strong or weak evidence against H_0 and in favor of H_1?

You should now remember what you saw when you did `launchApp("estimate")`. Look again at this app and set the slider to a sample size of 50, just as in our running example. You can see that the app draws one hundred (100) samples for you, locates their sample mean on the x-axis, and estimates the red density.

> The crucial thing to note here is that, given we are working with a **random sample** from a population with a certain distribution of *height*, our sample statistic $\bar{x}$ is **also a random variable**. Every new set of randomly drawn students would yield a different $\bar{x}$, and all of them together would follow the red density in the app. In reality we often only get to draw one single sample, and we can use knowledge about the sampling distribution to make inference.

Our task is now to decide if given that particular sampling distribution, given our estimate $\bar{x}$ and given an observed sample variance s^2, whether $\bar{x} = 168.5$ is *far away* from $\bar{x} = 167$, or not. The way to proceed is by computing a *test statistic*, which is to be compared to a *critical value*: if the test statistic exceeds that value, we reject H_0, otherwise we cannot. The critical value depends on the sampling distribution, and the size of the test. We talk about this next.

6.4.1 Making Errors

There are two types of error one can make when deploying such a test:

1. We might reject H_0, when in fact it is true! Here, upon observing $\bar{x} = 168.5$ we might conclude that indeed $\mu > 167$ and thus we'd reject. But we might have gotten unlucky and by chance have obtained an unusually tall sample of students. This is called **type one error**.

2. We might *fail* to reject H_0 when in fact H_1 is true. This is called the **type two error**.

We design a test with a certain probability of *type one error* α in mind. In other words, we choose with which probability α we are willing to make a type one error. (Notice that the best tests also avoid making type two errors! The number $1 - \Pr(\text{type 2 error})$ is called *power*, hence we prefer tests with *high power*). A typical choice for α is 0.05, i.e. we are willing to make a type one error with probability 5%. α is commonly called the **level of significance** or the **size** of a test.

6.4.2 Performing the Test

We can stick to the following cookbook procedure, which is illustrated in figure 6.4.

1. Set up hypothesis and significance level:
    1. $H_0 : \mu = 167$
    2. $H_1 : \mu > 167$
    3. $\alpha = 0.05$
2. Test Statistic and test distribution:
    - We don't know the true population variance σ^2, hence we estimate it via s^2 from our sample.
    - The corresponding test statistic is the *t-statistic*, which follows the Student's T distribution.
    - That is, our statistic is $T = \frac{\bar{x}-\mu}{s/\sqrt{n}} \sim t_{49}$, where 49 is equal to the *degrees of freedom* in this case.
3. Rejection Region: We perform a one-sided test. We said we are happy with a 5% significance level, i.e. we are looking for the t value which corresponds *just* to $1-0.05 = 0.95$ mass under the pdf of the t distribution. More precisely, we are looking for the $1 - 0.05 = 0.95$ quantile of the t_{50} distribution.[3] This implies a critical value $c = 1.676$, which you can verify by typing `qt(0.95,df=50)` in `R`.
4. Calculate our test statistic: $\frac{\bar{x}-\mu}{s/\sqrt{n}} = \frac{168.5-167}{10/\sqrt{50}} = 1.061$
5. Decide: We find that $1.061 < 1.676$. Hence, we cannot reject H_0, because we only found weak evidence against it in our sample of data.

6.4.3 Testing Regression Coefficients

In Regression Analysis, we often want to test a very specific alternative hypothesis: We want to have a quick way to tell us whether a certain variable x_k is *relevant* in our statistical model or not. In hypothesis testing language, that would be

[3] See the previous footnote for an explanation of this!

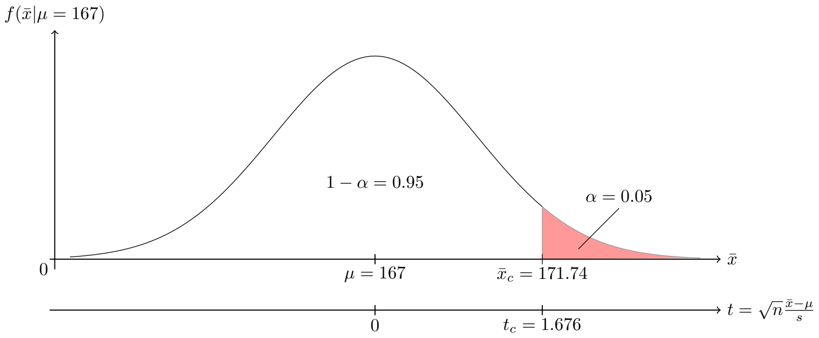


Figure 6.4: Cookbook Testing Proceedure. Subscripts c indicate *critical value*. There are two x-axis: one for values of $\bar{x}$, and one for the corresponding t statistic. The red area is the rejection area. If we observe a test statistic such that $t > t_c$, we feel reassured that our $\bar{x}$ is *sufficiently far away* from the hypothesized value μ, such that we feel comfortable with rejecting H_0. And vice versa: If our test statistic falls below t_c, we will not reject H_0

$$H_0 : \beta_k = 0 \tag{6.18}$$
$$H_1 : \beta_k \neq 0. \tag{6.19}$$

Clearly, if in the **true** regression model we find $\beta_k = 0$, this means that x_k has a zero partial effect on the outcome, hence it should be excluded from the regression. Notice that we are interested in β_k, not in b_k, which is the estimator that we compute from our sample (similarly to $\bar{x}$, which estimates μ above).

As such, this is a *two-sided test*. We can again illustrate this in figure 6.5. Notice how we now have two rejection areas.

The relevant test statistic for a regression coefficient is again the t distribution. In fact, this particular test is so important that all statistical packages report the t statistic corresponding to (6.19) automatically. Let's look at an example:

```
#OUT>
#OUT> Call:
#OUT> lm(formula = mpg ~ wt + hp + drat, data = mtcars)
#OUT>
#OUT> Residuals:
#OUT>     Min      1Q  Median      3Q     Max
#OUT> -3.3598 -1.8374 -0.5099  0.9681  5.7078
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 29.394934   6.156303   4.775 5.13e-05 ***
```

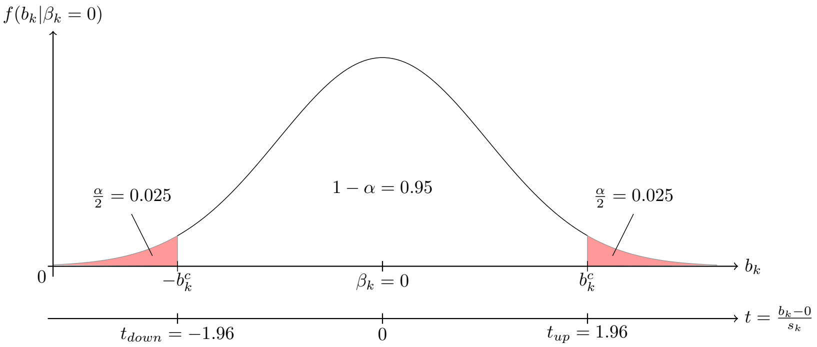


Figure 6.5: Testing whether coefficient b_k is *statistically significantly different* from zero. Now we have two red rejection areas. We relabel critical values with a superscript here. If we observe a test statistic falling in either red region, we reject, else we do not. Notice that the true value under H_0 is $\beta_k = 0$.

```
#OUT> wt          -3.227954   0.796398  -4.053 0.000364 ***
#OUT> hp          -0.032230   0.008925  -3.611 0.001178 **
#OUT> drat         1.615049   1.226983   1.316 0.198755
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 2.561 on 28 degrees of freedom
#OUT> Multiple R-squared:  0.8369,  Adjusted R-squared:  0.8194
#OUT> F-statistic: 47.88 on 3 and 28 DF,  p-value: 3.768e-11
```

The column `t value` is just `Estimate` divided by `Std. Error`. That is, `R` reports in the column `t value` the following number for us:

$$\text{t value} = \frac{b_k - 0}{s_k} \tag{6.20}$$

where s_k is the estimated standard error as introduced in 6.2.2, and where we test $H_0 : \beta_k = 0$. Notice that this particular t statistic is different from our previous formulation in (6.6): we don't have to scale by $\sqrt{n}$! This is so because `R` and other statistical software assumes the *normal* linear regression model (see 6.2). Normality of the regression error ε implies that the t statistic looks like in (6.20).

We have to choose a critical value for this test. Many people automatically choose the 0.975 quantile of the standard normal distribution, `qnorm(0.975)`, 1.96 in this case. This is fine for sample sizes greater than 100, say. In this regression, we only have 28 degrees of freedom, so we better choose the critical value from the t distribution as above. We get $t_{down} = -2.048$ and $t_{up} = 2.048$

as critical values. Let's test whether the coefficient on `wt` is statistically different from zero:

$$
\begin{aligned}
H_0 :& \beta_{wt} = 0 && (6.21)\\
H_1 :& \beta_{wt} \neq 0 && (6.22)
\end{aligned}
$$

We just take the `t value` entry, and see whether it lies above or below either critical value: Indeed, we see that $-4.053 < -2.048$, and we are happy to reject H_0.

On the other hand, when testing for statistical significance of `drat` that does not seem to be the case:

$$
\begin{aligned}
H_0 :& \beta_{drat} = 0 && (6.23)\\
H_1 :& \beta_{drat} \neq 0 && (6.24)
\end{aligned}
$$

Here we find that $1.316 \in [-2.048, 2.048]$, hence it does not lie in any rejection region, and we can *not* reject H_0. We would say that *coefficient β_{drat} is not statistically significant at the 5% level.* As such, we should not include it in our regression.

6.4.4 P-Values and Stars

`R` also reports two additional columns in its regression output. The so-called *p-value* in column `Pr(>|t|)` and a column with stars. P-values are an improvement over the dichotomy introduced in the standard reject/accept framework above. We never know if we *narrowly* rejected a H_0, or not. The p-value is defined as the particular level of significance α^*, up to which *all* H_0's would be rejected. If this is a very small number, we have overwhelming support to reject the null. If, on the contrary, α^* turns out to be rather large, we only found weak evidence against H_0.

We define the p-value as the sum of rejection areas for a given test statistic T^*. Notice that the symmetry of the t distribution implies that we multiply by two each of the two tail probabilities:

$$
\alpha^* = 2 \Pr(t > |T^*|) \qquad (6.25)
$$

The stars in the final column are a visualization of this information. They show a quick summary of the magnitude of each p-value. Commonly, `***` means an extremely small reference significance level $\alpha^* = 0$ (almost zero), `**` means

$\alpha^* = 0.001$, etc. In that case, up to a significance level of 0.1%, all H_0 would be rejected. You clearly see that all columns `Std. Error`, `t value` and `Pr(>|t|)` give a different type of the same information.

6.5 What's in my model? (And what is not?)

We want to revisit the underlying assumptions of the classical model outlined in 6.2. Right now we to talk a bit more about assumption number 2 of the above definition in 6.2. It said this:

> The mean of the residuals conditional on x should be zero, $E[\varepsilon|x] = 0$. This means that $Cov(\varepsilon, x) = 0$, i.e. that the errors and our explanatory variable(s) should be *uncorrelated*. We want x to be **strictly exogenous** to the model.

Great. But what does this *mean*? How could x be correlated with something we don't even observe?! Good questions - let's try with an example.

Imagine that we assume that

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \tag{6.26}$$

represents the DGP of impact the sales price of houses (y) as a function of number of bathrooms (x). We run OLS as

$$y_i = b_0 + b_1 x_i + e_i$$

You find a positive impact of bathrooms on houses:

```r
data(Housing, package="Ecdat")
hlm = lm(price ~ bathrms, data = Housing)
summary(hlm)
```

```
#OUT> 
#OUT> Call:
#OUT> lm(formula = price ~ bathrms, data = Housing)
#OUT> 
#OUT> Residuals:
#OUT>    Min     1Q Median     3Q    Max 
#OUT> -77225 -15271  -2510  11704 102729 
#OUT> 
#OUT> Coefficients:
#OUT>             Estimate Std. Error t value Pr(>|t|)    
#OUT> (Intercept)    32794       2694   12.17   <2e-16 ***
```

```
#OUT> bathrms        27477       1952   14.08   <2e-16 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 22880 on 544 degrees of freedom
#OUT> Multiple R-squared:  0.267,   Adjusted R-squared:  0.2657
#OUT> F-statistic: 198.2 on 1 and 544 DF,  p-value: < 2.2e-16
```

In fact, from this you conclude that each additional bathroom increases the sales price of a house by 27477 dollars. Let's see if our assumption $E[\varepsilon|x] = 0$ is satisfied:

```r
library(dplyr)
# add residuals to the data
Housing$resid <- resid(hlm)
Housing %>%
  group_by(bathrms) %>%
  summarise(mean_of_resid=mean(resid))
```

```
#OUT> # A tibble: 4 x 2
#OUT>   bathrms mean_of_resid
#OUT>     <dbl>         <dbl>
#OUT> 1       1          -118.
#OUT> 2       2           955.
#OUT> 3       3        -11195.
#OUT> 4       4         32298.
```

Oh, that doesn't look good. Even though the unconditional mean $E[e] = 0$ is *very* close to zero (type `mean(resid(hlm))`!), this doesn't seem to hold at all by categories of x. This indicates that there is something in the error term e which is *correlated* with `bathrms`. Going back to our discussion about *ceteris paribus* in section 4.1, we stated that the interpretation of our OLS slope estimate is that

> Keeping everything else fixed at the current value, what is the impact of x on y? *Everything* also includes things in ε (and, hence, e)!

It looks like our DGP in (6.26) is the *wrong model*. Suppose instead, that in reality sales prices are generated like this:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 z_i + \varepsilon_i \tag{6.27}$$

This would now mean that by running our regression, informed by the wrong DGP, what we estimate is in fact this:

$$y_i = b_0 + b_1 x_i + (b_2 z_i + e_i) = b_0 + b_1 x_i + u_i.$$

This is to say that by *omitting* variable z, we relegate it to a new error term, here called $u_i = b_2 z_i + e_i$. Our assumption above states that *all regressors need to be uncorrelated with the error term* - so, if $Corr(x, z) \neq 0$, we have a problem. Let's take this idea to our running example.

6.5.1 Omitted Variable Bias

What we are discussing here is called *Omitted Variable Bias*. There is a variable which we omitted from our regression, i.e. we forgot to include it. It is often difficult to find out what that variable could be, and you can go a long way by just reasoning about the data-generating process. In other words, do you think it's *reasonable* that price be determined by the number of bathrooms only? Or could there be another variable, omitted from our model, that is important to explain prices, and at the same time correlated with `bathrms`?

Let's try with `lotsize`, i.e. the size of the area on which the house stands. Intuitively, larger lots should command a higher price; At the same time, however, larger lots imply more space, hence, you can also have more bathrooms! Let's check this out:

```
#OUT>
#OUT> Call:
#OUT> lm(formula = price ~ bathrms + lotsize, data = Housing)
#OUT>
#OUT> Residuals:
#OUT>    Min     1Q Median     3Q    Max
#OUT> -60752 -12532  -1674  10514  92931
#OUT>
#OUT> Coefficients:
#OUT>              Estimate Std. Error t value Pr(>|t|)
#OUT> (Intercept) 1.008e+04  2.810e+03   3.588 0.000364 ***
#OUT> bathrms     2.281e+04  1.703e+03  13.397  < 2e-16 ***
#OUT> lotsize     5.575e+00  3.944e-01  14.136  < 2e-16 ***
#OUT> ---
#OUT> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#OUT>
#OUT> Residual standard error: 19580 on 543 degrees of freedom
#OUT> Multiple R-squared:  0.4642,  Adjusted R-squared:  0.4622
#OUT> F-statistic: 235.2 on 2 and 543 DF,  p-value: < 2.2e-16
```

Here we see that the estimate for the effect of an additional bathroom *decreased* from 27477 to 22811.5 by almost 5000 dollars! Well that's the problem then. We said above that one more bathroom is worth 27477 dollars - if **nothing else changes**! But that doesn't seem to hold, because we have seen that as we increase `bathrms` from `1` to `2`, the mean of the resulting residuals changes quite a bit. So there **is something in ε which does change**, hence, our conclusion

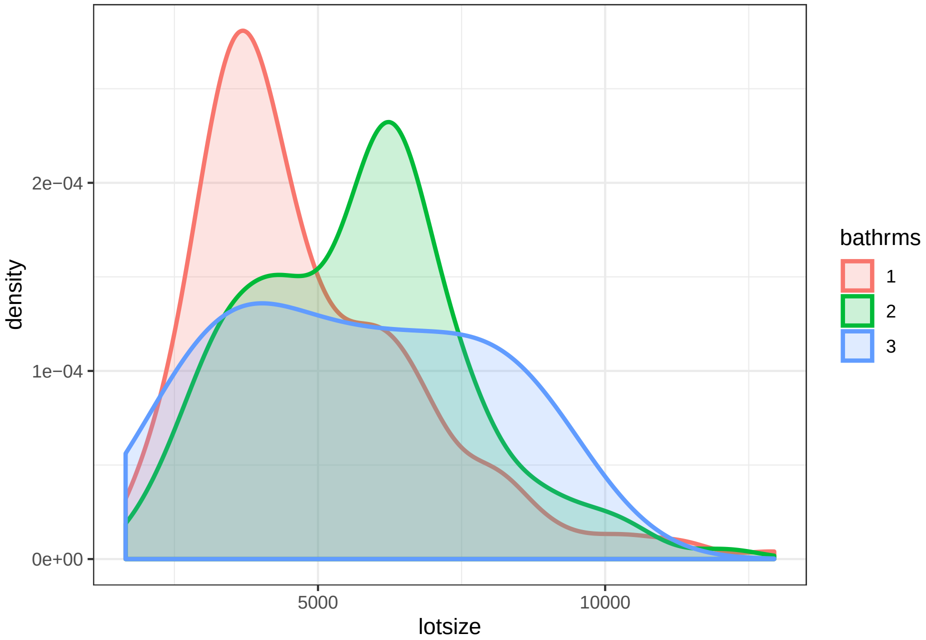


Figure 6.6: Distribution of 'lotsize' by 'bathrms'

that one more bathroom is worth 27477 dollars is in fact *invalid*!

The way in which `bathrms` and `lotsize` are correlated is important here, so let's investigate that:

This shows that lotsize and the number of bathrooms is indeed positively related. Larger lot of the house, more bathrooms. This leads to a general result:

Direction of Omitted Variable Bias
If there is an omitted variable z that is *positively* correlated with our explanatory variable x, then our estimate of effect of x on y will be too large (or, *biased upwards*). The correlation between x and z means that we attribute part of the impact of z on y mistakenly to x! And, of course, vice versa for *negatively* correlated omitted variables.

Chapter 7

Instrumental Variables

- Measurement error
- Omitted Variable Bias
- Reverse Causality / Simultaneity Bias

are all called *endogeneity* problems.

7.1 Simultaneity Bias

- Detroit has a large police force
- Detroit has a high crime rate
- Omaha has a small police force
- Omana has a small crime rate

Do large police forces **cause** high crime rates?

Absurd! Absurd? How could we use data to tell?

We have the problem that large police forces and high crime rates covary positively in the data, and for obvious reasons: Cities want to protect their citizens and therefore respond to increased crime with increased police. Using mathematical symbols, we have the following *system of linear equations*, i.e. two equations which are **jointly determined**:

$$
\begin{aligned}
\text{crime}_{it} &= f(\text{police}_{it}) \\
\text{police}_{it} &= g(\text{crime}_{it})
\end{aligned}
$$

We need a factor that is outside this circular system, affecting **only** the size of the police force, but not the actual crime rate. Such a factor is called an *instrumental variable.*

Chapter 8

Slides

- chapter 1
- chapter 2
- chapter 3
- chapter 4
- chapter 5
- chapter 6

8.1 Worked Tutorials

- Trade exercise

Chapter 9

Projects

9.1 Opportunity Atlas

In this empirical project you will have a close look at what is commonly known as *the American Dream* - the idea that in a society with few barriers, everybody can achieve upward social mobility and be better off than their parents, if only they work hard enough.[1]

The Opportunity Atlas is a website maintained by the US Census Bureau and fed with data from recent high-quality research on upward social mobility at the Census Tract level. Census Tracts are geographic areas which contain on average less than 4000 residents, and which cover the entire United States. The Atlas has been widely used in recent newspaper reporting, and you should start this project by reading the corresponding piece from the New York Times *Upshot* series.

This project focuses on developing your descriptive data analysis skills. In particular you will

1. use mapping tools to visualize geospatial data, and
2. compute simple descriptive statistics and report regression results to shed light on the relationship between two or more random variables.

In order to achieve those goals, you will interact with the Atlas website at https://www.opportunityatlas.org, and you will use the raw data behind the website to compute additional statistics.

[1] This project is a slightly modified version of the exercise designed by Raj Chetty and Gregory Bruich, available online here. Permission was granted to reuse their exercise.

9.1.1 Instructions

1. Your submission will be a statistical report addressing the below questions, produced from a single `R` markdown file (ending in `.Rmd`). You can create a simple `Rmd` template in Rstudio by clicking top left on the "new file" symbol, then selecting *R Markdown*. A quick but effective guide on using Markdown is available here.
2. Please submit two files: the `.Rmd` source file (which creates the report), and the report itself, in either HTML or PDF format (Rstudio dropdown menu *knit*)
3. You can do this exercise in teams of up to 3 people: make sure to edit the `author` field in the `Rmd` correspondingly.
4. Submit by sending a private message with the `Rmd` as attachment on slack.

9.1.2 Questions

1. Referring to the NYT article:

    1. Why does the Seattle Housing Authority give away vouchers to pay for rent in the area between 100th and 115th Streets, east of Meridian, west of 35th Avenue?
    2. The data here shows how easy it is for children of poor parents to escape poverty themselves. In explaining what makes a *good* neighborhood, how much of the variation we see is actually explained by things like school boundary lines and poverty levels alone? In other words, how *important* is the school you go to in isoluation of other factors, if you want to escape the low rank of your parents in the income distribution?
    3. Some select census tracts have received up to 500 million USD since 1990 in place-based neighborhood improvement policies. Do we know if and to what extent those investments were efficacious?

2. Go to https://www.opportunityatlas.org and look at the census tracts around where your home is, if you grew up in the United States. If you grew up outside the US, randomly choose one of the State Capitals from this list and select any census tract at random by zooming in. Make sure to immediately post your choice of city in the corresponding slack channel (e.g. "florian-atlas" for my group) to avoid having multiple reports on the same city. Let's call the chosen census tract your *home*.

    1. Create Figure 1 in your report, which should display the map shown by https://www.opportunityatlas.org for the census tracts around your home (you can download that map *as an image* bottom left).
    2. Your accompanying text should describe what is shown in that map, in particular what data are being visualized. Examine next the patterns for a number of different groups (e.g., lowest income

children, high income children) and outcomes (e.g., earnings in adulthood, incarceration rates). Only choose one or two of these to include in your report.

3. To answer the next question, read the Opportunity Atlas Manual: What period do the data you are analyzing come from? Are you concerned that the neighborhoods you are studying may have changed for kids now growing up there? What evidence do the authors of the manual provide suggesting that such changes are or are not important? What type of data could you use to test whether your neighborhood has changed in recent years?

4. Now turn to the `atlas.Rds` data set, which you can load as shown below. How does average upward mobility, pooling races and genders, for children with parents at the 25th percentile (`kfr_pooled_p25`) in your home Census tract compare to mean (population-weighted, using `count_pooled`) upward mobility in your state and in the U.S. overall? Do kids where you grew up have better or worse chances of climbing the income ladder than the average child in America? *Hint*: The Opportunity Atlas website will give you the tract, county, and state FIPS codes for your home address. For example, searching for “Lynwood Road, Verona, New Jersey” will display Tract 34013021000, Verona, NJ. The first two digits refer to the state code, the next three digits refer to the county code, and the last 6 digits refer to the tract code. In `R`, listing this observation can be done as follows:

```r
# load data
d = readRDS(system.file(package = "ScPoEconometrics","datasets","atlas.Rds"))
# subset data.frame
subset(d, kfr_pooled_p25 if state == 34 & county == 013 & tract == 021000)
```

5. What is the standard deviation of upward mobility (population-weighted) in your home county, and what does this number tell you? Is it larger or smaller than the standard deviation across tracts in your state (i.e. compare your county to all other counties in your state)? Across tracts in the entire country? What do you learn from these comparisons? Notice that you can compute a weighted standard deviation for vector `x` using weights `weight_variable` in `R` like this

```r
# you need the Hmisc package installed. if not:
# install.packages("Hmisc")
sqrt(Hmisc::wtd.var(x, weights = weight_variable))
```

6. Now let’s turn to downward mobility: repeat questions 3. and 4. looking at children who start with parents at the 75th and 100th percentiles. How do the patterns differ?

7. Using a linear regression, estimate the relationship between outcomes of

children at the 25th and 75th percentile for the Census tracts in your home county. Generate a scatter plot to visualize this regression. Do areas where children from low-income families do well generally have better outcomes for those from high-income families, too?

```
# explain lm
# explain scatter plot
```

8. Next, examine whether the patterns you have looked at above are similar by race. If there is not enough racial heterogeneity in the area of interest (i.e., data is missing for most racial groups), then choose a different area to examine.
9. Using the Census tracts in your home county, can you identify any covariates which help explain some of the patterns you have identified above? Some examples of covariates you might examine include housing prices, income inequality, fraction of children with single parents, job density, etc. For 2 or 3 of these, report estimated correlation coefficients along with their 95% confidence intervals.
10. Open question: formulate a hypothesis for why you see the variation in upward mobility for children who grew up in the Census tracts near your home and provide correlational evidence testing that hypothesis. For this question, many covariates have been provided to you in the `atlas.Rds` file, which are described in subsection 9.1.3.

9.1.3 Detailed Data Description

The data consist of $n = 73,278$ U.S. Census tracts. For more details on the construction of the variables included in this data set, please see Chetty, Raj, John Friedman, Nathaniel Hendren, Maggie R. Jones, and Sonya R. Porter. 2018. "The Opportunity Atlas: Mapping the Childhood Roots of Social Mobility." NBER Working Paper No. 25147

Variable name	Label	Obs.
tract	Tract FIPS Code (6-digit) 2010	73,278

Variable name	Label	Obs.
county	County FIPS Code (3-digit)	73,278
state	State FIPS Code (2-digit)	73,278
cz	Commuting Zone Identifier (1990 Definition)	72,473
hhinc_mean2000	Mean Household Income 2000	72,302
mean_commutetime2000	Average Commute Time of Working Adults in 2000	72,313

Variable name	Label	Obs.
frac_coll_plus2010	Fraction of Residents with a College Degree or More in 2010	72,930
foreign_share2010	Share of Population Born Outside the U.S.	72,279
med_hhinc2016	Median Household Income in 2016	72,763
med_hhinc1990	Median Household Income in 1999	72,913

Variable name	Label	Obs.
popdensity2000	Population Density (per square mile) in 2000	72,469
poor_share2010	Poverty Rate 2010	72,933
poor_share2000	Poverty Rate 2000	72,315
poor_share1990	Poverty Rate 1990	72,323
share_black2010	Share black 2010	73,111
share_hisp2010	Share Hispanic 2010	73,111
share_asian2010	Share Asian 2010	71,945
share_black2000	Share black 2000	72,368
share_white2000	Share white 2000	72,368
share_hisp2000	Share Hispanic 2000	72,368
share_asian2000	Share Asian 2000	71,050

Variable name	Label	Obs.
gsmn_math_g3_2013	Average School District Level Standardized Test Scores in 3rd Grade in 2013	72,390
rent_twobed2015	Average Rent for Two-Bedroom Apartment in 2015	58,607
singleparent_share2010	Share of Single-Headed Households with Children 2010	72,564

Variable name	Label	Obs.
singleparent_share1990	Share of Single-Headed Households with Children 1990	72,196
singleparent_share2000	Share of Single-Headed Households with Children 2000	72,285
traveltime15_2010	Share of Working Adults w/ Commute Time of 15 Minutes Or Less in 2010	72,039
emp2000	Employment Rate 2000	72,344

Variable name	Label	Obs.
mail_return_rate2010	Census Form Rate Return Rate 2010	72,547
ln_wage_growth_hs_grad	Log wage growth for HS Grad., 2005-2014	51,635
jobs_total_5mi_2015	Number of Primary Jobs within 5 Miles in 2015	72,311
jobs_highpay_5mi_2015	Number of High-Paying (>USD40,000 annually) Jobs within 5 Miles in 2015	72,311

Variable name	Label	Obs.
nonwhite_share2010	Share of People who are not white 2010	73,121
popdensity2010	Population Density (per square mile) in 2010	73,194
ann_avg_job_growth_2004_2013	Average Annual Job Growth Rate 2004-2013	70,664
job_density_2013	Job Density (in square miles) in 2013	72,463

Variable name	Label	Obs.
kfr_pooled_p25	Household income ($) at age 31-37 for children with parents at the 25th percentile of the national income distribution	73,311

Variable name	Label	Obs.
kfr_pooled_p75	Household income ($) at age 31-37 for children with parents at the 75th percentile of the national income distribution	73,312

Variable name	Label	Obs.
kfr_pooled_p100	Household income ($) at age 31-37 for children with parents at the 100th percentile of the national income distribution	71,968

Variable name	Label	Obs.
kfr_natam_p25	Household income ($) at age 31-37 for Native American children with parents at the 25th percentile of the national income distribution	1,733

Variable name	Label	Obs.
kfr_natam_p75	Household income ($) at age 31-37 for Native American children with parents at the 75th percentile of the national income distribution	1,728

Variable name	Label	Obs.
kfr_natam_p100	Household income ($) at age 31-37 for Native American children with parents at the 100th percentile of the national income distribution	1,514

Variable name	Label	Obs.
kfr_asian_p25	Household income ($) at age 31-37 for Asian children with parents at the 25th percentile of the national income distribution	12,434

Variable name	Label	Obs.
kfr_asian_p75	Household income ($) at age 31-37 for Asian children with parents at the 75th percentile of the national income distribution	157,360

Variable name	Label	Obs.
kfr_asian_p100	Household income ($) at age 31-37 for Asian children with parents at the 100th percentile of the national income distribution	13,480

Variable name	Label	Obs.
kfr_black_p25	Household income ($) at age 31-37 for Black children with parents at the 25th percentile of the national income distribution	34,086

Variable name	Label	Obs.
kfr_black_pooled_p75	Household income ($) at age 31-37 for Black children with parents at the 75th percentile of the national income distribution	37,049

Variable name	Label	Obs.
kfr_black_p100	Household income ($) at age 31-37 for Black children with parents at the 100th percentile of the national income distribution	320,536

Variable name	Label	Obs.
kfr_hisp_p25	Household income ($) at age 31-37 for Hispanic children with parents at the 25th percentile of the national income distribution	37,611

Variable name	Label	Obs.
kfr_hisp_p75	Household income ($) at age 31-37 for Hispanic children with parents at the 75th percentile of the national income distribution	35,579

Variable name	Label	Obs.
kfr_hisp_p100	Household income ($) at age 31-37 for Hispanic children with parents at the 100th percentile of the national income distribution	35,087

Variable name	Label	Obs.
kfr_white_p25	Household income ($) at age 31-37 for white children with parents at the 25th percentile of the national income distribution	63,078

Variable name	Label	Obs.
kfr_white_p75	Household income ($) at age 31-37 for white children with parents at the 75th percentile of the national income distribution	61,068

Variable name	Label	Obs.
kfr_white_p100	Household income ($) at age 31-37 for white children with parents at the 100th percentile of the national income distribution	57,027
count_pooled	Count of all children	72,451
count_white	Count of White children	72,451
count_black	Count of Black children	72,451
count_asian	Count of Asian children	72,451

Variable name	Label	Obs.
count_hisp	Count of Hispanic children	72,451
count_natam	Count of Native American children	72,451