Introduction to Econometrics with R

Florian Oswald and Jean-Marc Robin

2018-08-28

Contents

Syllabus

Welcome to Introductory Econometrics for 2nd year undergraduates at ScPo! On this page we outline the course and present the Syllabus. As of today, this is still **work in progress**!

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.

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).

Figure 1:

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.

Team

tbc

Communication

We will communicate exclusively on our `slack` app. You will get an invitation email to join 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:

- `R`, the actual programming language.
    - Chose your operating system, and select the most recent version, 3.5.0.
- 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! (A skill you will learn in this course.) There are many good resources for learning `R`.

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 present the bare basics in this chapter, some more details are in chapter 13.

1.2 Starting R and RStudio

A key difference for you to understand is that between `R`, the actual programming language, and RStudio which is a software that allows you to efficiently and easily work with the R language.

The best way to appreciate the value of RStudio is to start using R *without* RStudio. To do this, click on the R application that you should have downloaded on your computer (see above). You've just opened the R **console** which allows you to start typing code right after the red `>` sign. Try typing `2 + 2` or `print("Your Name")` and hit the return key. And *voilà*, your first R commands!

Typing each command after the other is however not very convenient, and we would like to be able to write all the lines of code beforehand and then run all of them in one go. We can do this by writing **scripts**, and for this we will need tools like RStudio!

Open RStudio by clicking on the RStudio application on your computer, and notice how different (and prettier!) the whole environment is from the basic R console – which you can still find and use in the bottom panel of RStudio. 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 line of code, simply highlight it and hit `Command` + `Return`.

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.

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.

- Hadley Wickham Style Guide from Advanced `R`
- 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 `<-`.

1.3 Basic Calculations

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

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
$\cos(0)$	`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 emailing 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 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`.

1.7 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

Dimension	**Homogeneous**	**Heterogeneous**
3+	Array	nested Lists

1.7.1 Vectors

Many operations in `R` make heavy use of **vectors**. 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. [you should type all of those examples into your `R` console!!]

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

```
## [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
```

```
## [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)
```

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

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

```
## [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)
```

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

```r
2
```

```
## [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.

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

```
##  [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
## [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.

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

```
##  [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
## [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.

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

```
##  [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.

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

```
##  [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.

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

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

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

```r
length(x)
```

```
## [1] 6
```

```r
length(y)
```

```
## [1] 100
```

1.7.1.1 Task 1

Let's try this out!

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]`

1.7.1.2 Subsetting

To subset a vector, i.e. to choose only some elements of it, we use square brackets, `[]`.

```
x
```

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

```
x[1]
```

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

```
x[3]
```

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

We see that `x[1]` returns the first element, and `x[3]` returns the third element.

```
x[-2]
```

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

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

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

```
## [1] 1 3 5
```

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

```
## [1] 1 5 7
```

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

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
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

```
x[z]
```

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

1.7.2 Vectorization

One of the biggest strengths of `R` is its use of vectorized operations. This means, operations which work on - and are optimized for - entire vectors.

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

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

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

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

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

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

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

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

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

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

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`.

1.7.3 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 are vectorized.

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

```r
x > 3
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```r
x < 3
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

```r
x == 3
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE FALSE
```

```r
x != 3
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

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

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

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

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

This is extremely useful for subsetting.

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

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

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

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

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

```
## [1] 4
```

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

```
## [1] 0 0 1 1 1 1
```

Here we see 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. `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.

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

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

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

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

```r
max(x)
```

```
## [1] 9
```

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

```
## [1] 6
```

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

```
## [1] 6
```

1.7.3.1 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.

1.7.4 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
```

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

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

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [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
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [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
```

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

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

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

```
## [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, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

```
## [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)]
```

```
## [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)
```

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

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

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

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

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

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

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

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

```
Y
```

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

```
X + Y
```

```
##      [,1] [,2] [,3]
```

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

```
X - Y
```

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

```
X * Y
```

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

```
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
## [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
```

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

```
t(X)
```

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

1.7.5 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
```

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

```
## [4,]    4    8
##
## , , 2
##
##      [,1] [,2]
## [1,]    9   13
## [2,]   10   14
## [3,]   11   15
## [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:

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

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

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

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

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

```
## [1]  6 14
```

1.7.5.1 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?

1.7.6 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:

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

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

```
## [1] 42
##
## [[2]]
## [1] "Hello"
##
## [[3]]
## [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
```

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

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

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
```

```
ex_list[1]
```

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

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

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

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

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

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

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

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

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

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

```
ex_list$d
```

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

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

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

1.7.6.1 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.

1.7.7 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`)!

```
example_data
```

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

```
##  [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))
```

```
## [1] TRUE
```

```
str(example_data)
```

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

```
nrow(example_data)
```

```
## [1] 10
```

```
ncol(example_data)
```

```
## [1] 3
```

```
dim(example_data)
```

```
## [1] 10  3
```

```
names(example_data)
```

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

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 into `R`, we would use the `read_csv()` function from the `readr` package. Note that `R` has a built in function `read.csv()` that operates very similarly. The `readr` function `read_csv()` has a number of advantages. 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.

```r
library(readr)  # you need `install.packages("readr")` once!
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")`

```r
example_data_from_disk
```

```
## # A tibble: 10 x 3
##        x y       z
##    <int> <chr>   <lgl>
##  1     1 Hello   TRUE
##  2     3 Hello   FALSE
##  3     5 Hello   TRUE
##  4     7 Hello   FALSE
##  5     9 Hello   TRUE
##  6     1 Hello   FALSE
##  7     3 Hello   TRUE
##  8     5 Hello   FALSE
##  9     7 Hello   TRUE
## 10     9 Goodbye FALSE
```

The `as_tibble()` function can be used to coerce a regular data frame to a tibble.

```r
library(tibble)
example_data = as_tibble(example_data)
example_data
```

```
## # A tibble: 10 x 3
##        x y       z
##    <dbl> <fct>   <lgl>
##  1     1 Hello   TRUE
##  2     3 Hello   FALSE
##  3     5 Hello   TRUE
##  4     7 Hello   FALSE
##  5     9 Hello   TRUE
##  6     1 Hello   FALSE
##  7     3 Hello   TRUE
##  8     5 Hello   FALSE
##  9     7 Hello   TRUE
## 10     9 Goodbye FALSE
```

Alternatively, we could use the "Import Dataset" feature in RStudio which can be found in the environment window. (By default, the top-right pane of RStudio.) Once completed, this process will automatically generate the code to import a file. The resulting code will be shown in the console window. In recent versions of RStudio, `read_csv()` is used by default, thus reading in a tibble.

Earlier we looked at installing packages, in particular the `ggplot2` package.

```r
library(ggplot2)
```

Inside the `ggplot2` package is a dataset called `mpg`. By loading the package using the `library()` function, we can now access `mpg`.

When using data from inside a package, 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 the data, we have two useful commands: `head()` and `str()`.

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

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

The function `head()` will display the first `n` observations of the data frame. 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
```

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

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(mpg)
```

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

```
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ 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
```

`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(mpg)
```

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

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

```
mpg$year
```

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

```
mpg$hwy
```

```
##   [1] 29 29 31 30 26 26 27 26 25 28 27 25 25 25 25 24 25 23 20 15 20 17 17
##  [24] 26 23 26 25 24 19 14 15 17 27 30 26 29 26 24 24 22 22 24 24 17 22 21
##  [47] 23 23 19 18 17 17 19 19 12 17 15 17 17 12 17 16 18 15 16 12 17 17 16
##  [70] 12 15 16 17 15 17 17 18 17 19 17 19 19 17 17 17 16 16 17 15 17 26 25
##  [93] 26 24 21 22 23 22 20 33 32 32 29 32 34 36 36 29 26 27 30 31 26 26 28
## [116] 26 29 28 27 24 24 24 22 19 20 17 12 19 18 14 15 18 18 15 17 16 18 17
## [139] 19 19 17 29 27 31 32 27 26 26 25 25 17 17 20 18 26 26 27 28 25 25 24
## [162] 27 25 26 23 26 26 26 26 25 27 25 27 20 20 19 17 20 17 29 27 31 31 26
## [185] 26 28 27 29 31 31 26 26 27 30 33 35 37 35 15 18 20 20 22 17 19 18 20
## [208] 29 26 29 29 24 44 29 26 29 29 29 29 23 24 44 41 29 26 28 29 29 29 28
## [231] 29 26 26 26
```

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

```
dim(mpg)
```

```
## [1] 234  11
```

```
nrow(mpg)
```

```
## [1] 234
```

```
ncol(mpg)
```

```
## [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 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")]
```

```
## # A tibble: 6 x 3
##   manufacturer model       year
##   <chr>        <chr>      <int>
## 1 honda        civic       2008
## 2 honda        civic       2008
## 3 toyota       corolla     2008
## 4 volkswagen   jetta       1999
## 5 volkswagen   new beetle  1999
## 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, we could use the `filter` and `select` functions from the `dplyr` package which introduces the *pipe operator* `%>%` from the `magrittr` package. 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. We will see more of this in chapter 2. For additional info on the pipe operator in R, you might be interested in this tutorial.

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

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

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

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

1.7.7.1 Task 5

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!).

1.8 Programming Basics

1.8.1 Control Flow

In `R`, the if/else syntax is:

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

For example,

```
x = 1
y = 3
if (x > y) {
  z = x * y
  print("x is larger than y")
} else {
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

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

```
z
```

```
## [1] 16
```

R also has a special function `ifelse()` which is very useful. It returns one of two specified values based on a conditional statement.

```
ifelse(4 > 3, 1, 0)
```

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

For Loops

Now a `for` loop example,

```
x = 11:15
for (i in 1:5) {
  x[i] = x[i] + i
}

x
```

```
## [1] 12 14 16 18 20
```

1.8.2 Functions

So far we have been using functions, but haven't actually discussed some of their details.

```
function_name(arg1 = 10, arg2 = 20)
```

To use a function, you simply type its name, followed by an open parenthesis, then specify values of its arguments, then finish with a closing parenthesis.

An **argument** is a variable which is used in the body of the function. Specifying the values of the arguments is essentially providing the inputs to the function.

We can also write our own functions in R. For example, we often like to "standardize" variables, that is, subtracting the sample mean, and dividing by the sample standard deviation.

$$z = \frac{x - \bar{x}}{s}$$

In R we would write a function to do this. When writing a function, there are three thing you must do.

- Give the function a name. Preferably something that is short, but descriptive.
- Specify the arguments using `function()`
- Write the body of the function within curly braces, `{}`.

```
standardize = function(x) {
  m = mean(x)
  std = sd(x)
  result = (x - m) / std
  result
}
```

Here the name of the function is `standardize`, and the function has a single argument `x` which is used in the body of function. Note that the output of the final line of the body is what is returned by the function. In this case the function returns the vector stored in the variable `result`.

To test our function, we will take a random sample of size `n = 10` from a normal distribution with a mean of `2` and a standard deviation of `5`.

```
(test_sample = rnorm(n = 10, mean = 2, sd = 5))
```

```
##  [1]  1.52076418 -0.05366843 10.36323088  0.08217924  5.54810662
##  [6]  1.02331620  5.13631856  3.61166574  8.57670479 10.07810107
```

```
standardize(x = test_sample)
```

```
##  [1] -0.7629021 -1.1544191  1.4359700 -1.1206377  0.2385844 -0.8866034
##  [7]  0.1361843 -0.2429539  0.9917113  1.3650663
```

This function could be written much more succinctly, simply performing all the operations on one line and immediately returning the result, without storing any of the intermediate results.

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

When specifying arguments, you can provide default arguments.

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

Let's look at a number of ways that we could run this function to perform the operation `10^2` resulting in `100`.

```
power_of_num(10)
```

```
## [1] 100
```

```
power_of_num(10, 2)
```

```
## [1] 100
```

```
power_of_num(num = 10, power = 2)
```

```
## [1] 100
```

```
power_of_num(power = 2, num = 10)
```

```
## [1] 100
```

Note that without using the argument names, the order matters. The following code will not evaluate to the same output as the previous example.

```
power_of_num(2, 10)
```

```
## [1] 1024
```

Also, the following line of code would produce an error since arguments without a default value must be specified.

```
power_of_num(power = 5)
```

To further illustrate a function with a default argument, we will write a function that calculates sample variance two ways.

By default, the function will calculate the unbiased estimate of σ^2, which we will call s^2.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x - \bar{x})^2$$

It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}^2$.

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x - \bar{x})^2$$

```
get_var = function(x, biased = FALSE) {
  n = length(x) - 1 * !biased
  (1 / n) * sum((x - mean(x)) ^ 2)
}
```

```
get_var(test_sample)
```

```
## [1] 16.17137
```

```
get_var(test_sample, biased = FALSE)
```

```
## [1] 16.17137
```

```
var(test_sample)
```

```
## [1] 16.17137
```

We see the function is working as expected, and when returning the unbiased estimate it matches `R`'s built in function `var()`. Finally, let's examine the biased estimate of σ^2.

```
get_var(test_sample, biased = TRUE)
```

```
## [1] 14.55424
```

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. *Make sure to have loaded the `ggplot2` library to be able to access the `mpg` dataset as introduced in section 1.7.7.*

```r
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.

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

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

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

```
##
##         4         f         r
## 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.

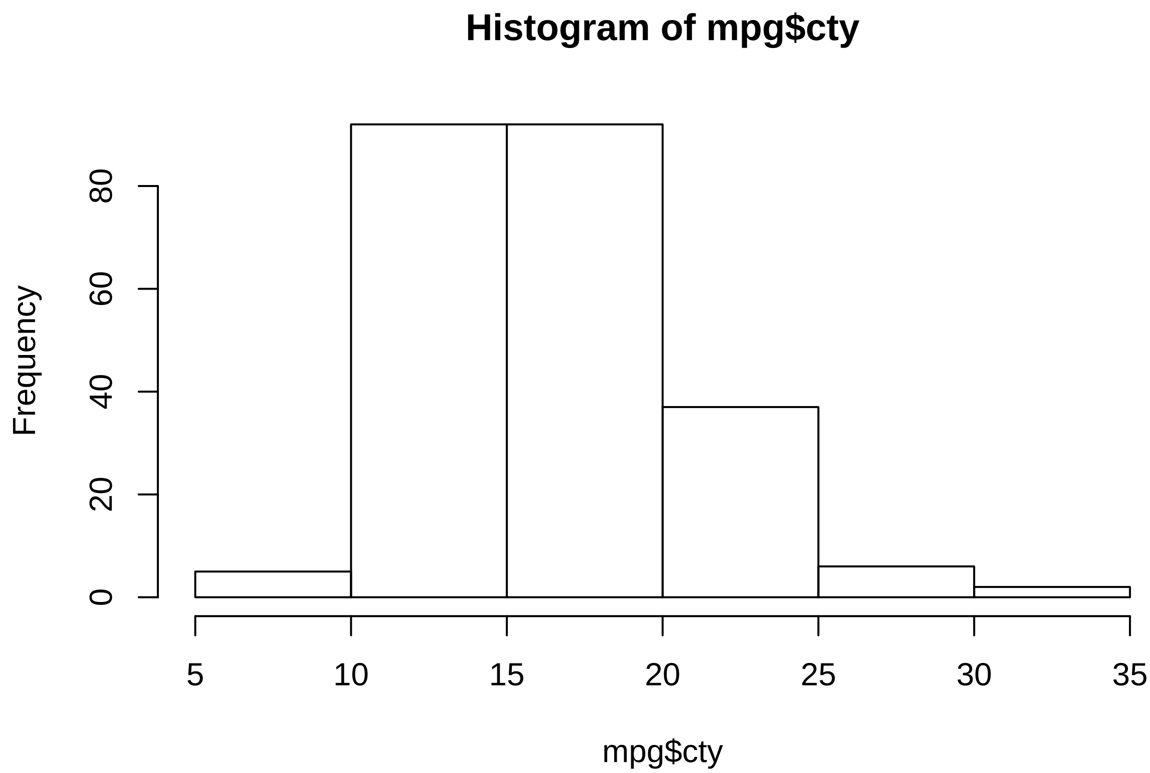
We will look at four methods of visualizing data by using the basic `plot` facilities built-in with `R`:

- Histograms
- Barplots
- Boxplots
- Scatterplots

2.2.1 Histograms

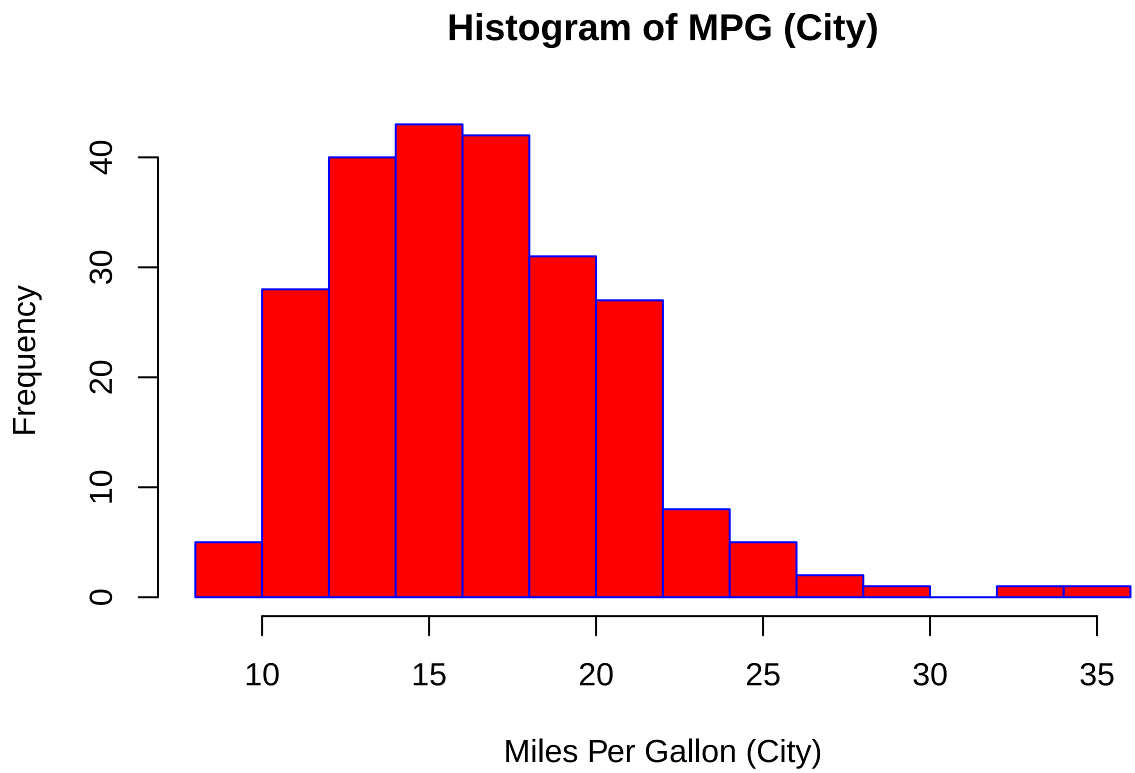
When visualizing a single numerical variable, a **histogram** is useful. It summarizes the *distribution* of values in a vector. In `R` you create one using the `hist()` function:

```
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")
```

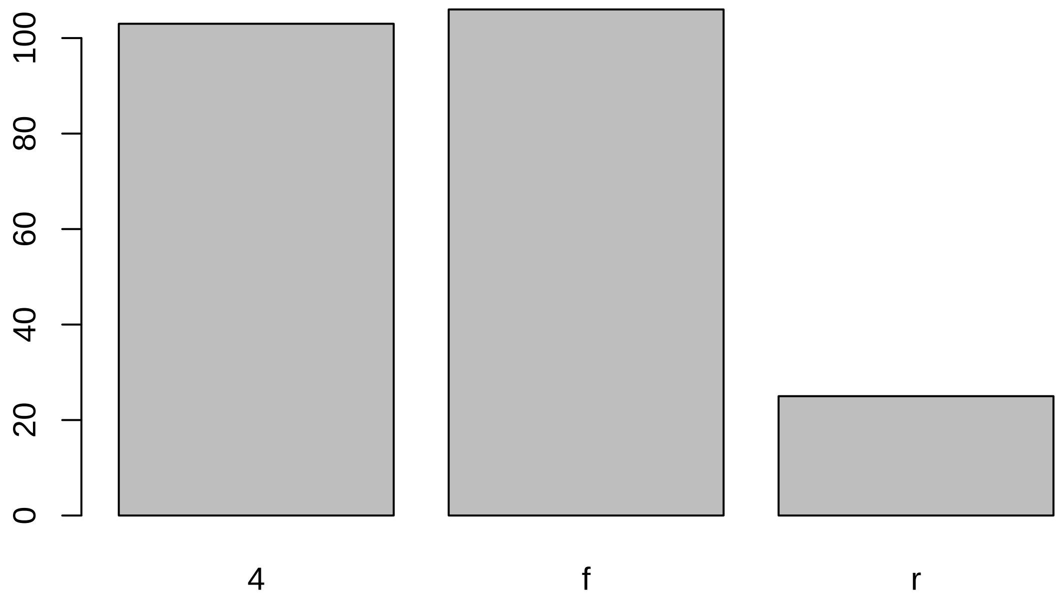


Importantly, you should always be sure to label your axes and give the plot a title. The argument `breaks` is specific to `hist()`. Entering an integer will give a suggestion to `R` for how many bars to use for the histogram. By default `R` will attempt to intelligently guess a good number of `breaks`,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

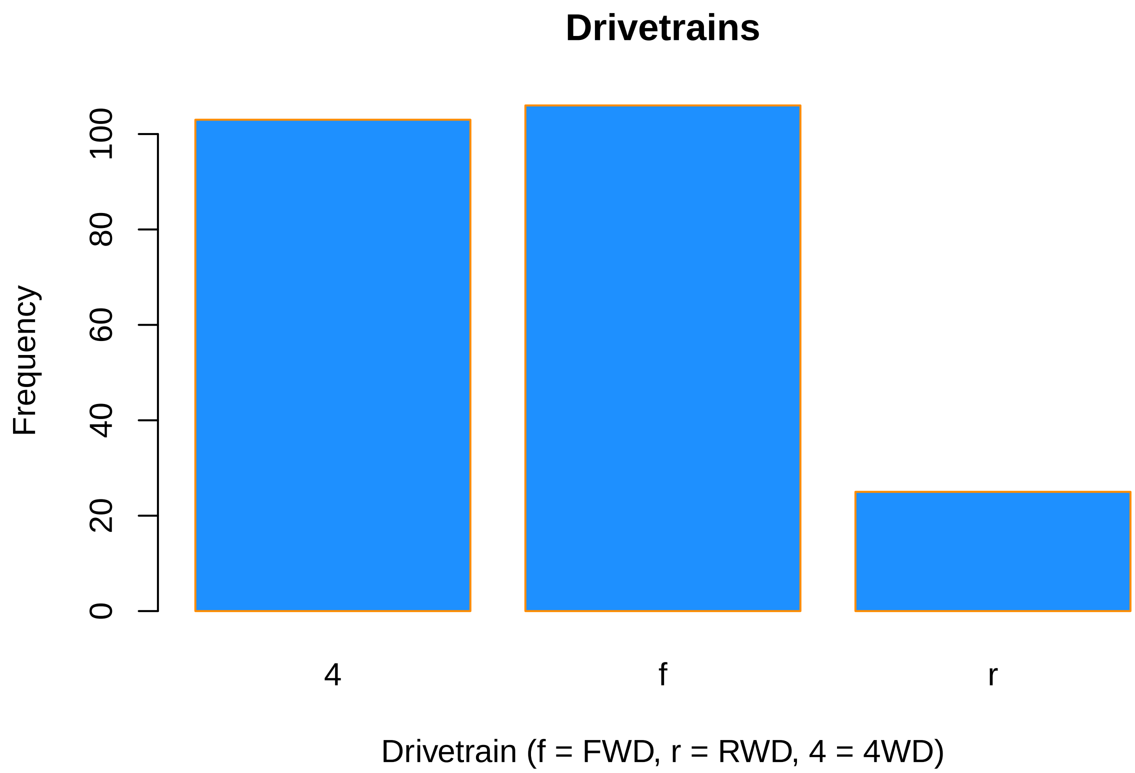
2.2.2 Barplots

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

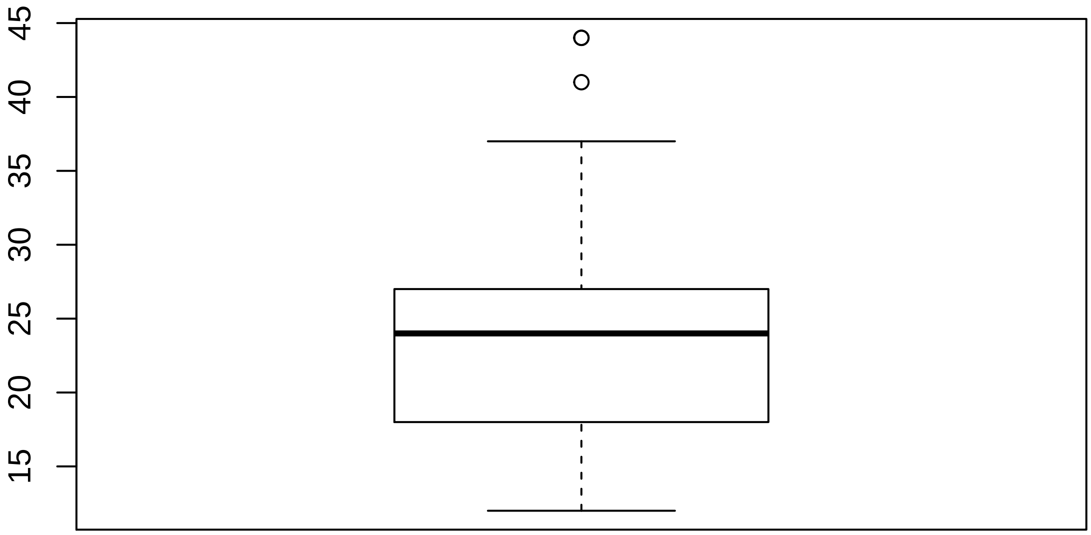
To visualize the relationship between a numerical and categorical variable, once could use a **boxplot**. In the `mpg` dataset, the `drv` variable takes a small, finite number of values. A car can only be front wheel drive, 4 wheel drive, or rear wheel drive.

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

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

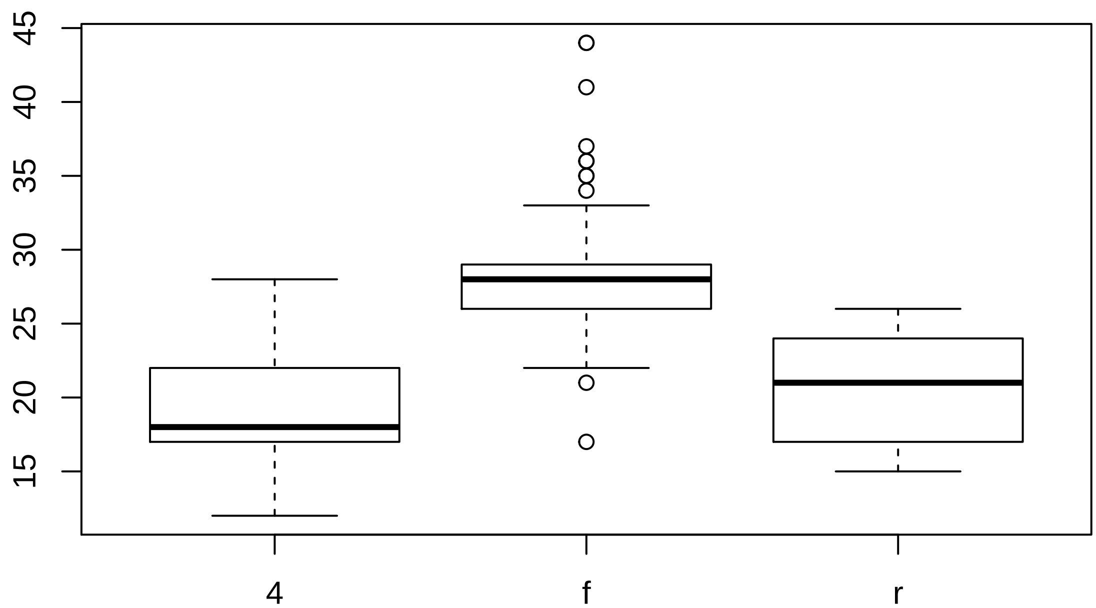
First note that we can use a single boxplot as an alternative to a histogram for visualizing a single numerical variable. To do so in `R`, we use the `boxplot()` function. The box shows the *interquartile range*, the solid line in the middle is the value of the median, the wiskers show 1.5 times the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

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



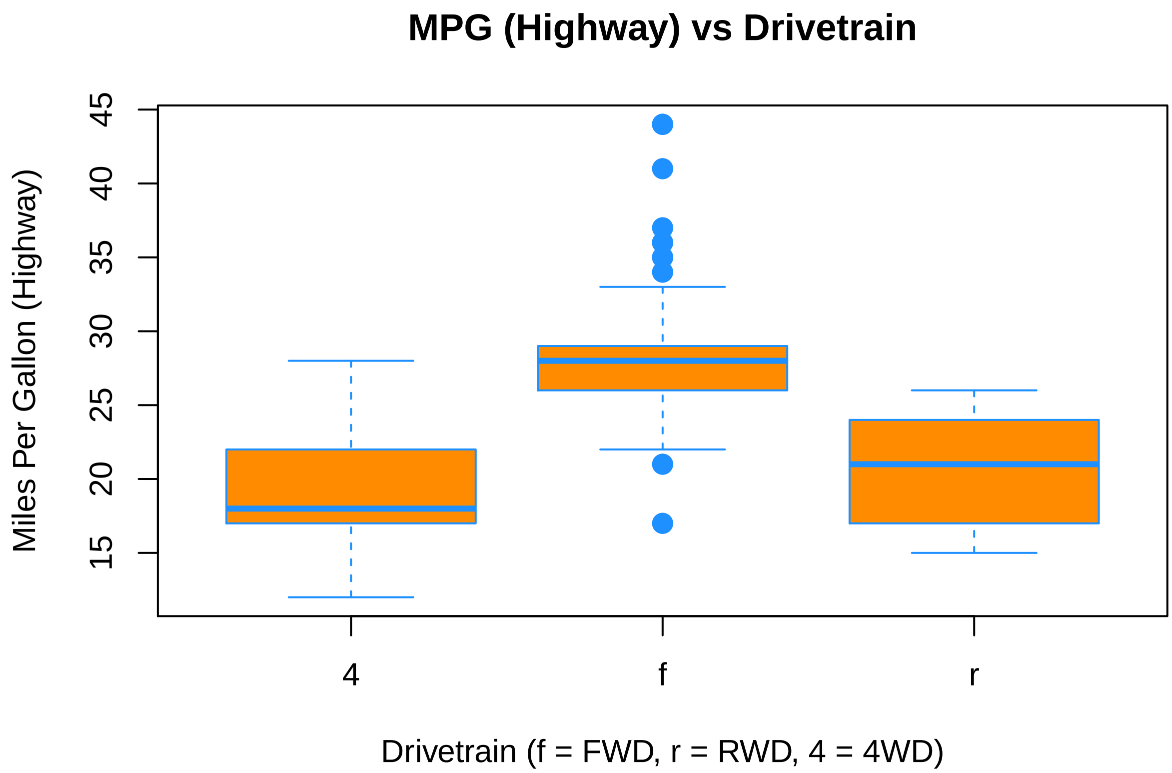
However, more often we will use boxplots to compare a numerical variable for different values of a categorical variable.

```
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.

```
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")
```

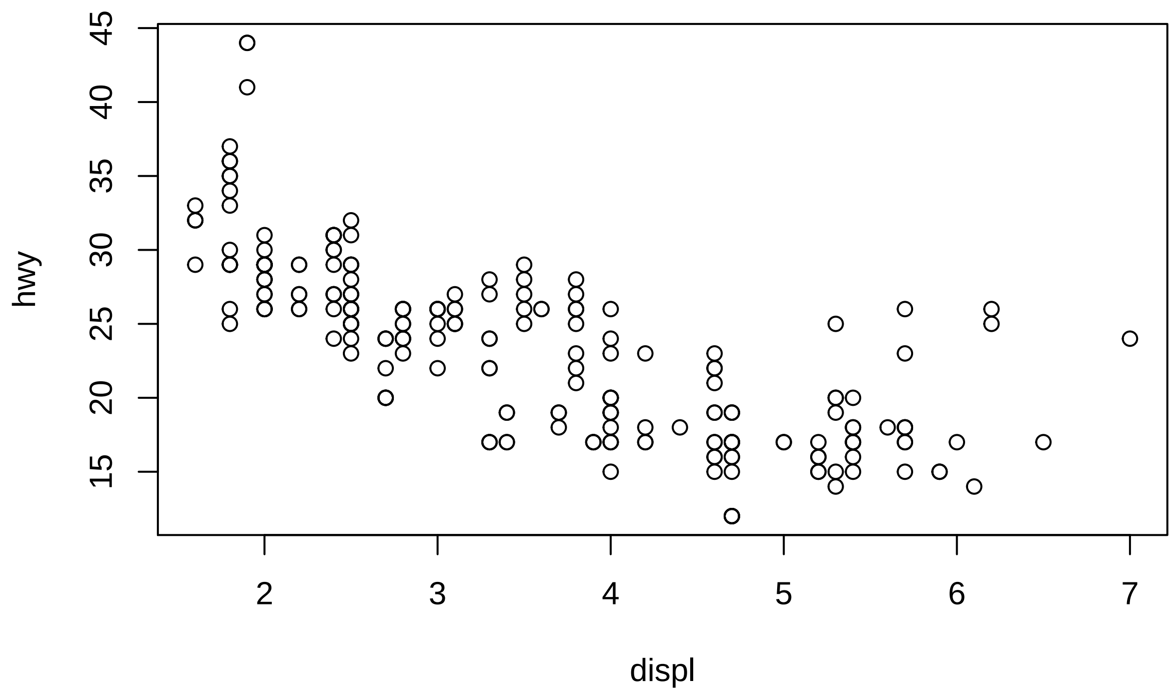


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

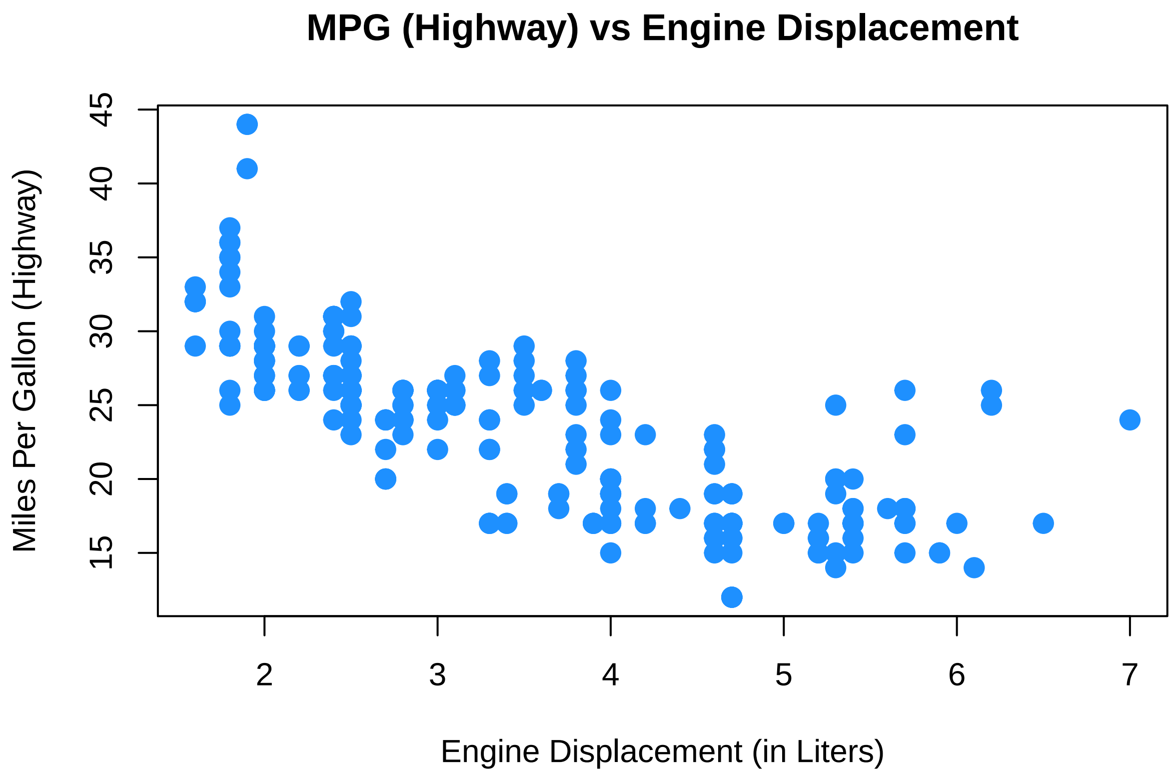
2.2.4 Scatterplots

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.)

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



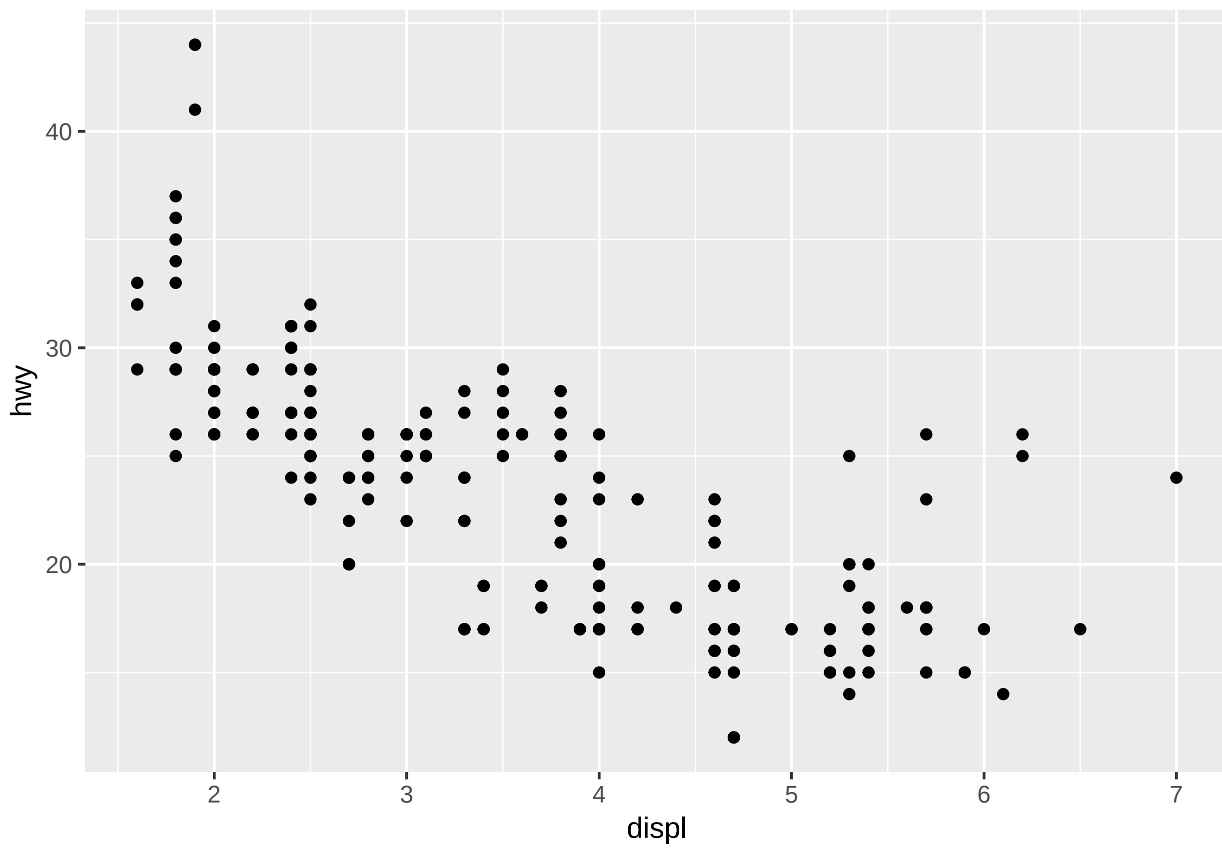

```
plot(hwy ~ displ, data = mpg,
     xlab = "Engine Displacement (in Liters)",
     ylab = "Miles Per Gallon (Highway)",
     main = "MPG (Highway) vs Engine Displacement",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
```



2.2.5 ggplot

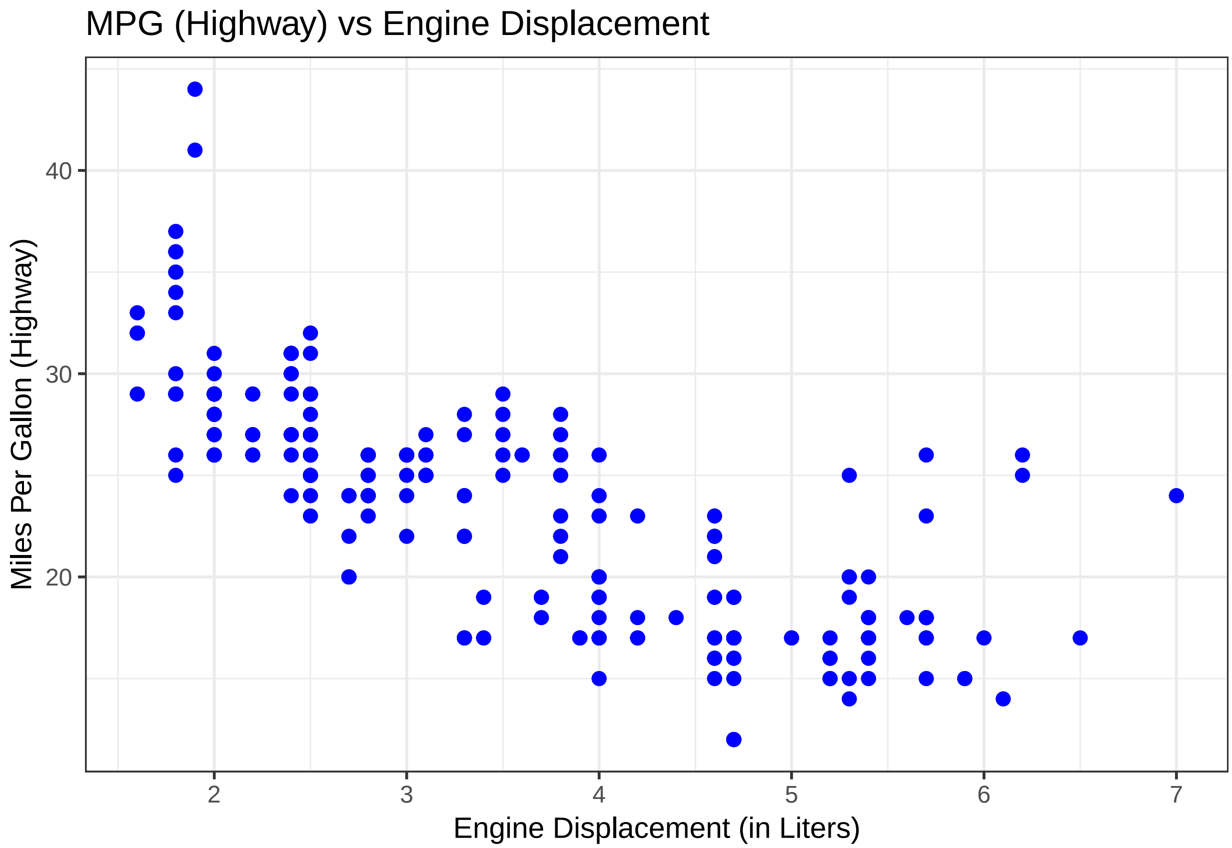
All of the above plots could also have been generated using the `ggplot` function from the already loaded `ggplot2` package. Which function you use is up to you, but sometimes a plot is easier to build in base R (like in the `boxplot` example maybe), sometimes the other way around.

```
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 try to familiarize yourself with it. Let's quickly demonstrate how one could 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
```



2.3 Summarizing Two Variables

We often are interested in how two (or more!) 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:

> **Note:**
>
> 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)
```

```
## [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 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**,

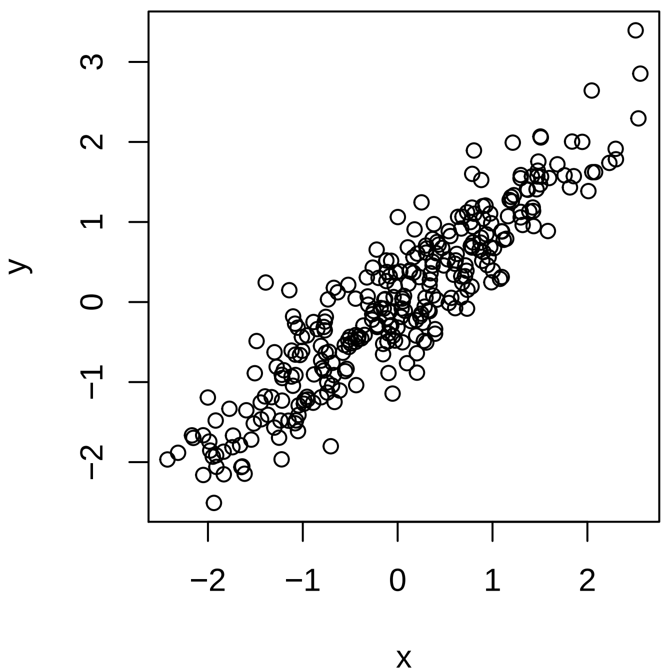


Figure 2.1: How are x and y related?

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)
```

```
## [1] 0.9142495
```

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

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 pioneered what he calls the *grammar of graphics* (hence, `gg`). 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 Tidy Example: Importing 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*:

```
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
```

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

This shows a `tibble`, which we encountered already in 1.7.7. 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.

2.4.1.1 `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`.

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

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

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

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

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

```r
tot_pop
```

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

2.4.1.2 `dplyr`

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

We already saw the `dplyr` package in action in chapter 1.7.7, but now we go further. With `dplyr` you can do the following operations on `data.frames` and `tibbles`:

- Choose observations based on a certain value: `filter()`
- Reorder rows: `arrange()`
- Select variables by name: `select()`

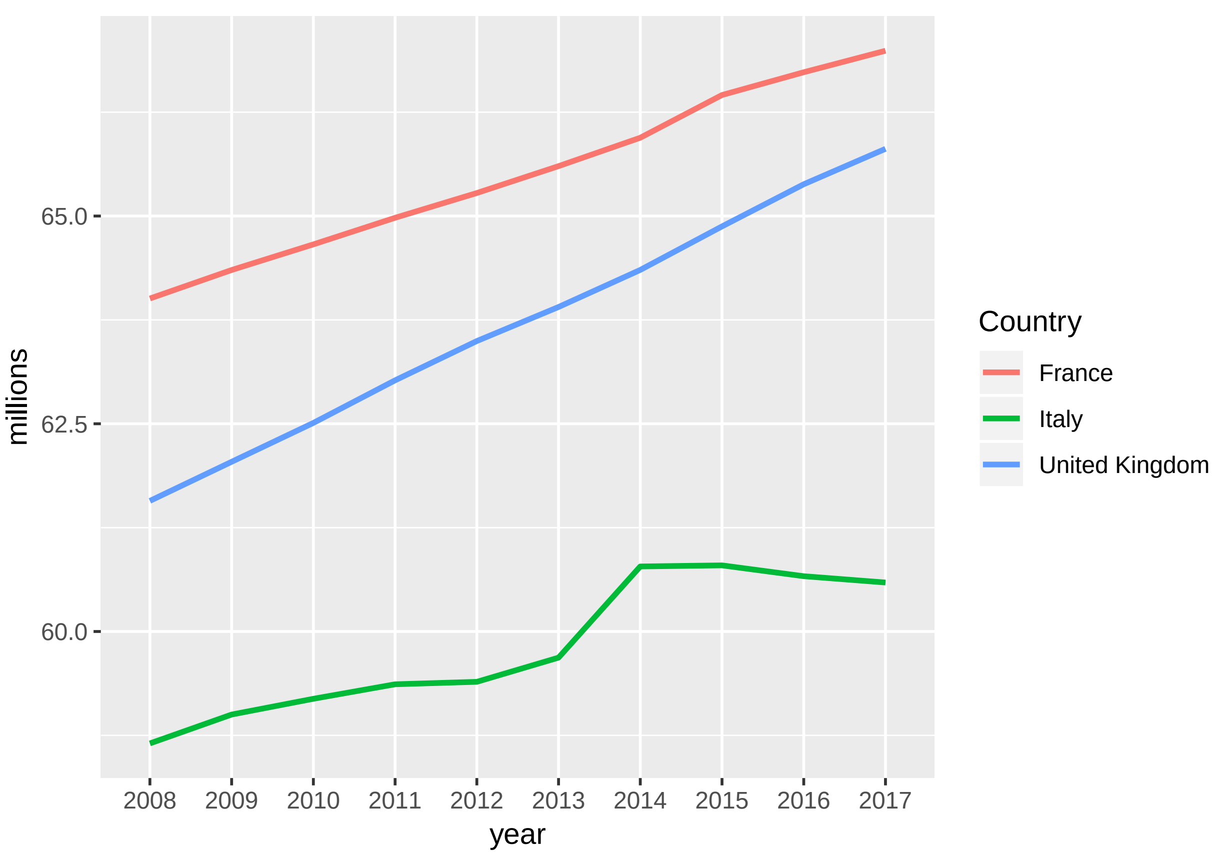
- Create new variables out of existing ones: `mutate()`
- Summarise variables: `summarise()`

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` as already mentioned in section 1.7.7.

```
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=1)
```



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
```

```
## # A tibble: 5 x 3
##   Country                                                   year   counts
##   <chr>                                                     <chr>   <int>
## 1 European Economic Area (EU28  - current composition, plu~ 2017   5.17e8
## 2 European Economic Area (EU28  - current composition, plu~ 2016   5.16e8
## 3 European Economic Area (EU28  - current composition, plu~ 2015   5.14e8
## 4 European Economic Area (EU27 -  before the accession of ~ 2017   5.13e8
## 5 European Economic Area (EU28  - current composition, plu~ 2014   5.12e8
```

```
# and bottom 5
bottom5
```

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

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

```
bottom5
```

```
## # A tibble: 5 x 2
##   Country       mean_count
##   <chr>              <dbl>
## 1 Luxembourg       563319.
## 2 Malta            440467.
## 3 Iceland          329501.
## 4 Liechtenstein     37353
## 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
```

```
## # A tibble: 59 x 11
```

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

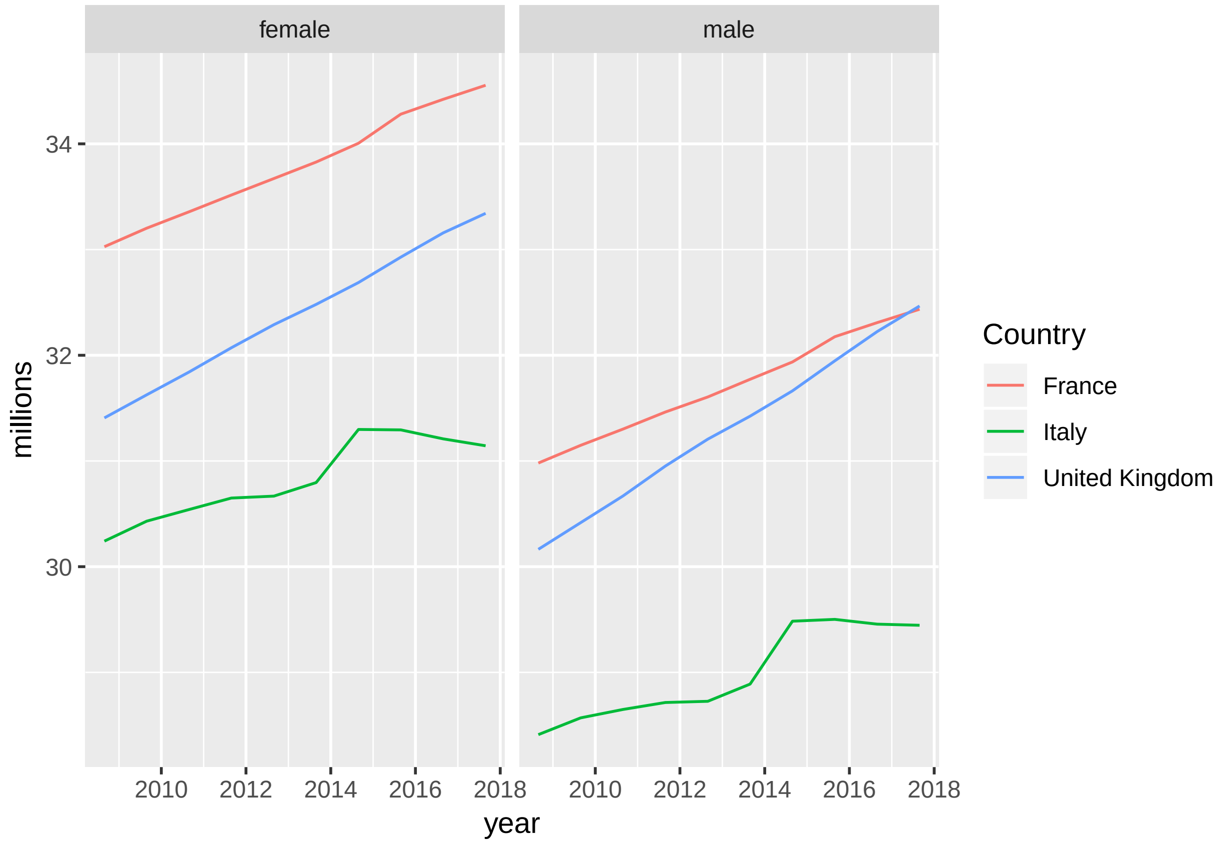
Let's try to tweak our above plot to show the same data in two separate panels: one for males and one for females. This is easiest to do with `ggplot` if we have all the data in one single `data.frame` (or `tibble`), and marked with a *group identifier*. Let's first add this to both datasets, and then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

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

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

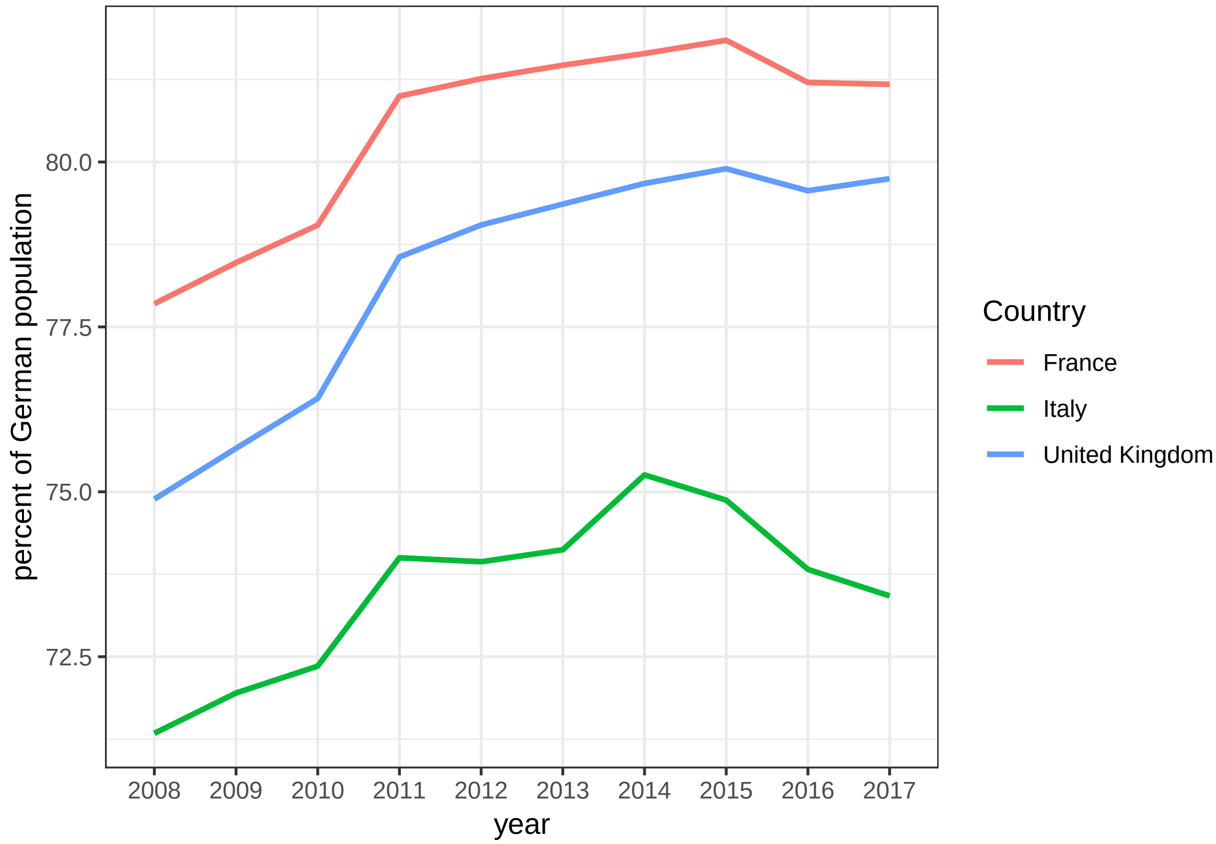
```
# remember that the pipe operator %>% takes the
# result of the previous operation and passes it
# as the *first* argument to the next function call
merge_GER <- tot_pop %>%
  # 1. subset to countries of interest
  filter(
    Country %in%
      c("France",
        "United Kingdom",
```

```
        "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
```

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

```
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?
```


percent of German population
80.0
77.5
75.0
72.5
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
year
Country
France
Italy
United Kingdom

Chapter 3

Linear Regression

3.1 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:

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 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

```r
dim(cars)
```

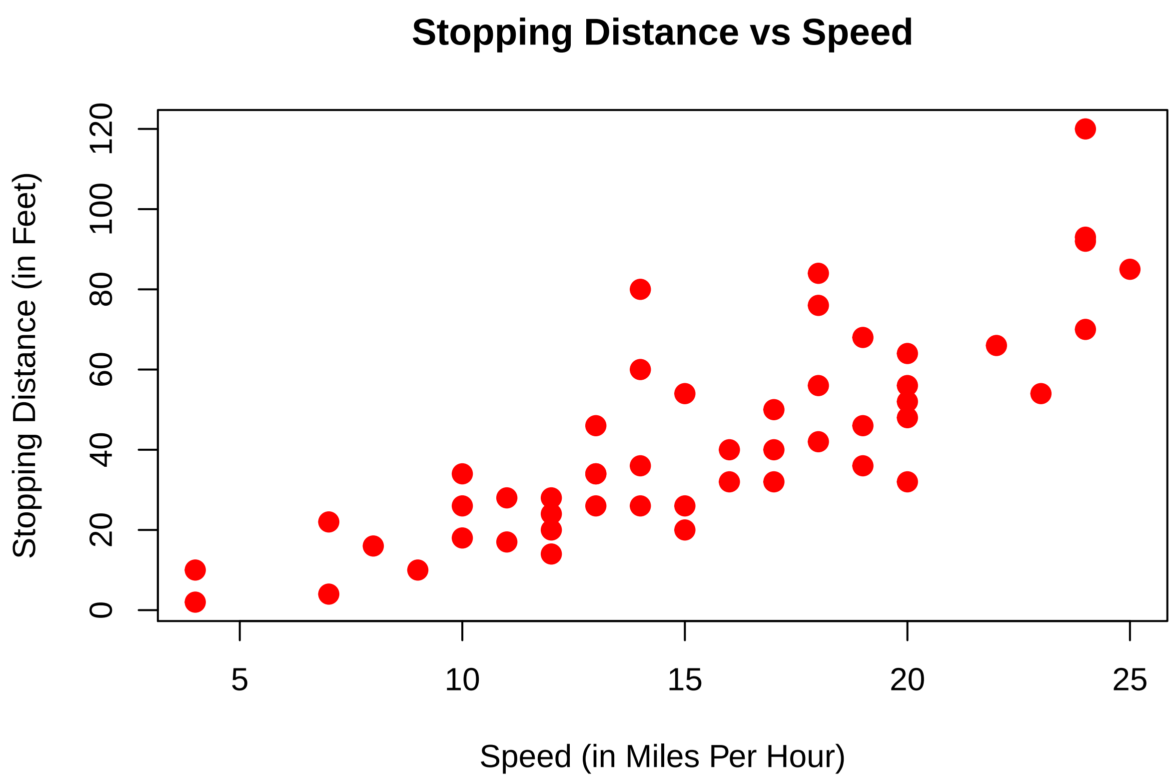
```
## [1] 50  2
```

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

3.1.1 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`:

```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")
```



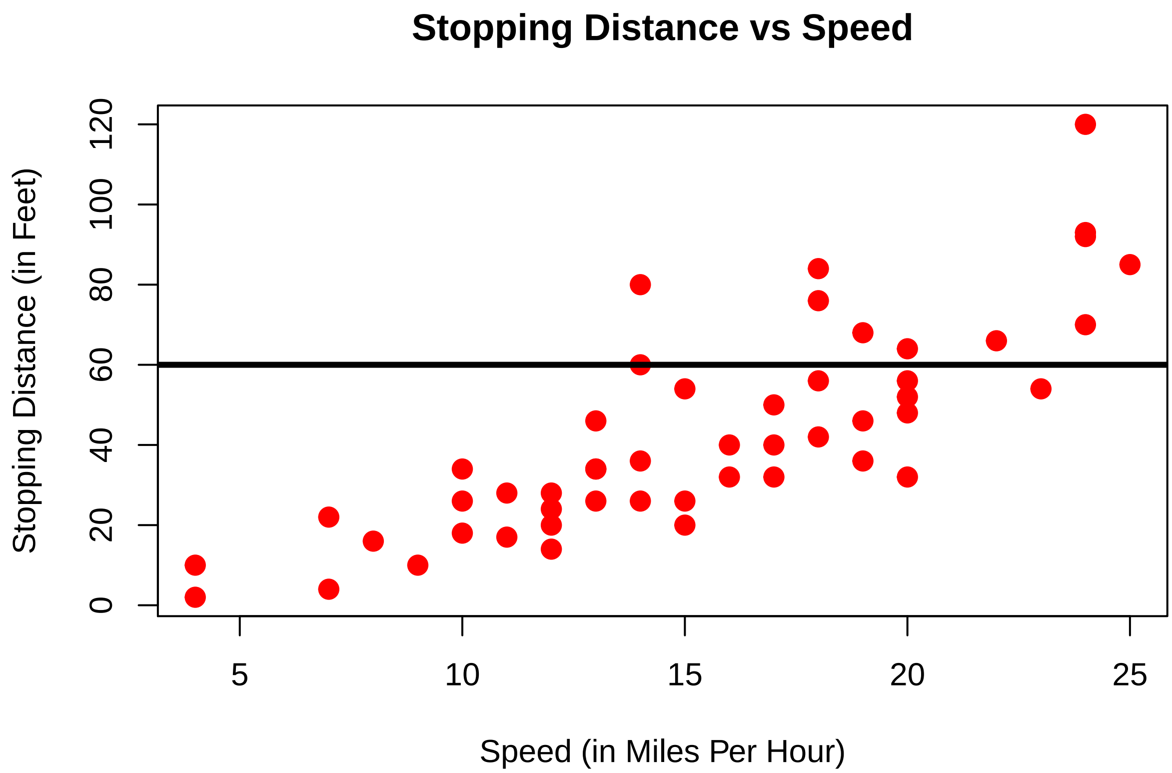
Here, each dot represents one observation. In this case, one particular measurement `speed` and `dist` for a car. Now, again:

Note:

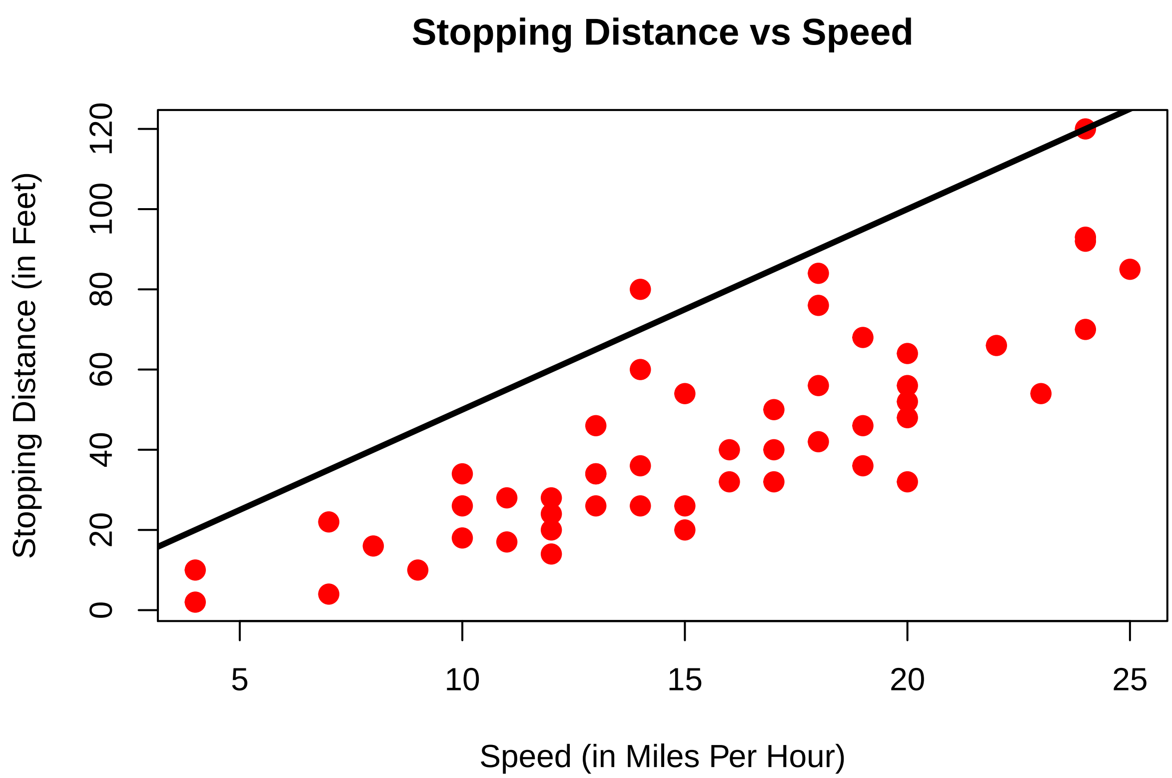
How are `speed` and `dist` related? How could one best *summarize* this relationship?

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

```
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*. So how to choose the **best** line?

3.1.2 Choosing the Best Line

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 looks like this:

The red arrows indicate the **distance** of the line to each point and we call them *errors* or *residuals*, often written with the symbol ε. An upward pointing arrow indicates a positive value of a particular ε_i, and vice versa for downward pointing arrows. The name *residual* comes from the way we write an equation for this relationship between two particular values (y_i, x_i) belonging to observation i:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

Here β_0 is the intercept, and β_1 is the slope of our line, and ε_i is the value of the arrow (i.e. a positive or negative number) indicating the distance between the actual y_i and what is predicted by our line. In other words, ε_i is what is left to be explained on top of the line $\beta_0 + \beta_1 x_i$, hence, it's a residual to explain y_i. Now, back to our claim that this is the *best* line. What exactly characterizes the best line?

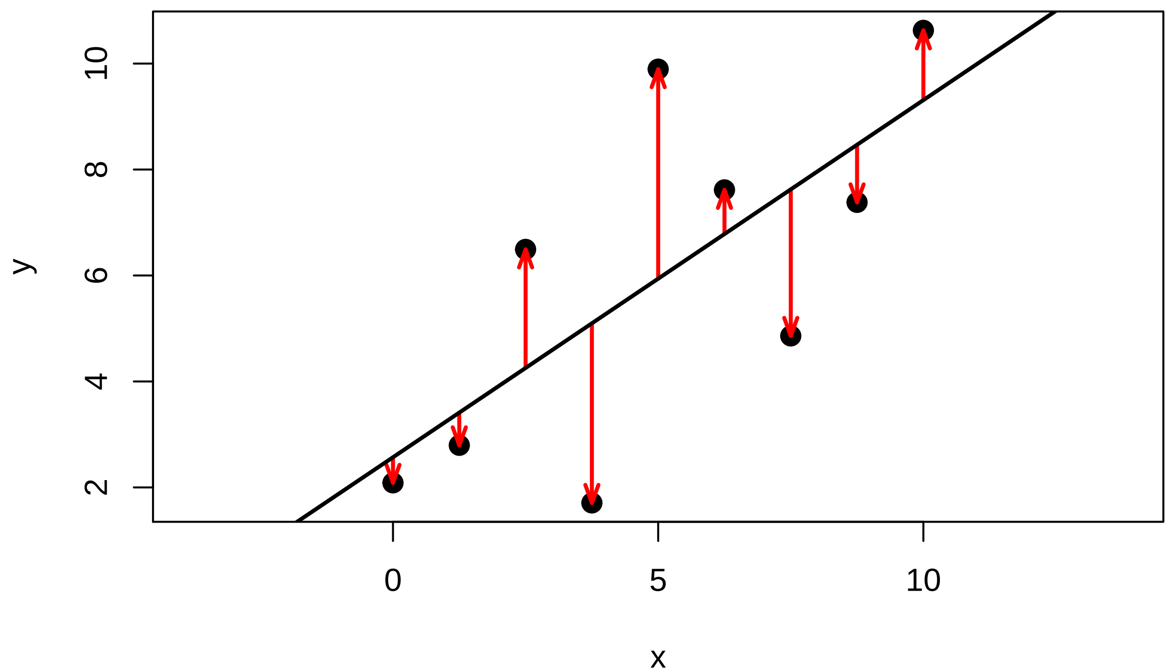


Figure 3.1: The best line and its errors

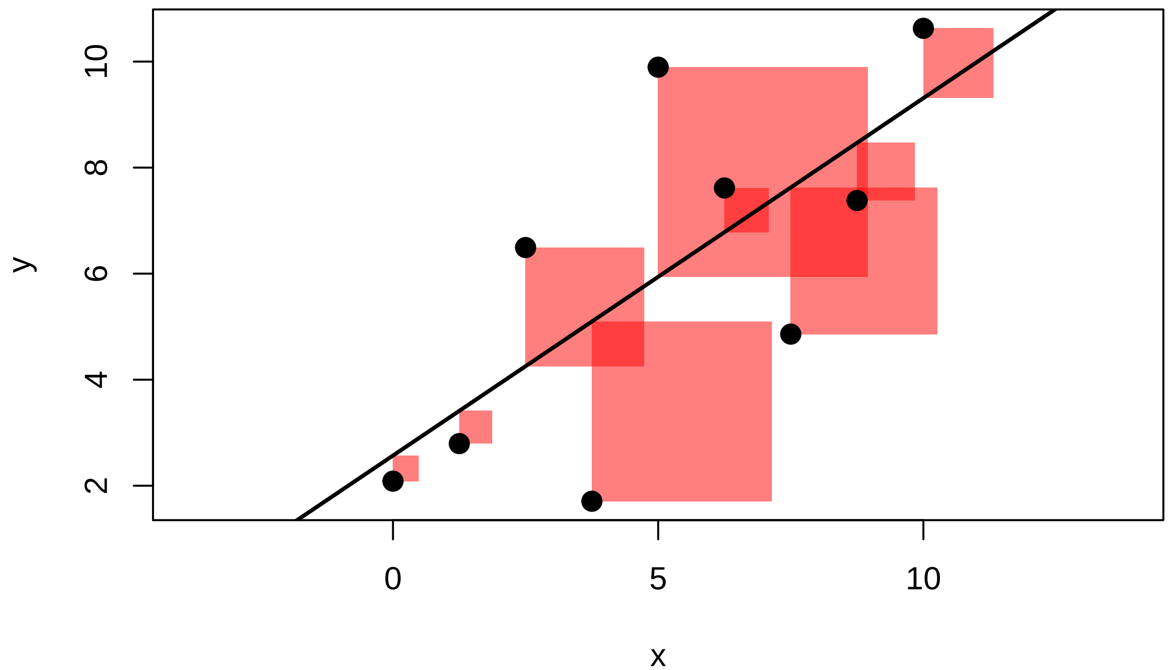


Figure 3.2: The best line and its SQUARED errors

Warning!

The best line minimizes the sum of **squared residuals**, i.e. it minimizes the SSR:

$$\varepsilon_1^2 + \cdots + \varepsilon_N^2 = \sum_{i=1}^{N} \varepsilon_i^2 \equiv \text{SSR}$$

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 ε_i for each observation i, now we draw a square with side ε_i, i.e. an area of ε_i^2. You will see in the practical sessions that choosing a different line to this one will increase the sum of squares.

3.1.3 Ordinary Least Squares (OLS) Coefficients

The method to estimate β_0 and β_1 we illustrated above is called *Ordinary Least Squares*, or OLS. There is a connection between the estimate for β_1 - denoted $\hat{\beta}_1$ - in equation (3.1.2) and the *covariance* of y and x - remember how we defined this in section 2.3. In the simple case shown in equation (3.1.2), the relationship is

$$\hat{\beta}_1 = \frac{cov(x, y)}{var(x)}.$$

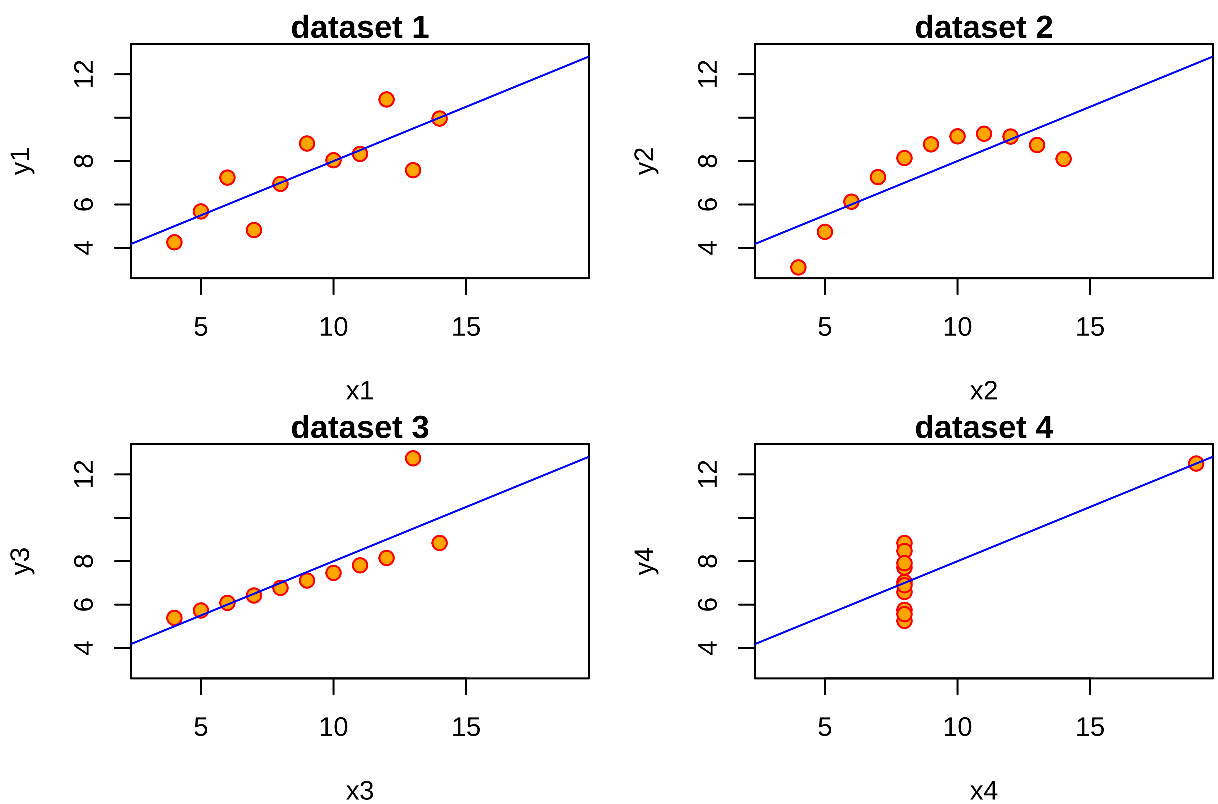
i.e. the estimate of the slope coefficient is the covariance between x and y divided by the variance of x. Similarly, the estimate for the intercept is given by

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

where $\bar{z}$ denotes the sample mean of variable z.

3.1.4 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.



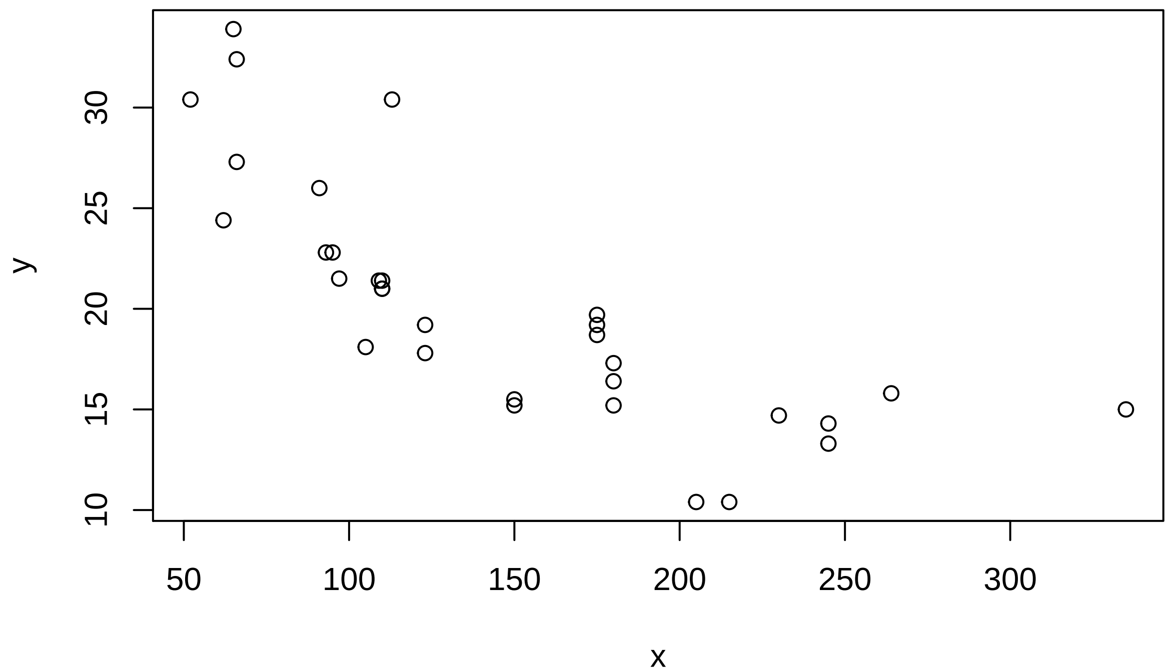
The important lesson from this example is the following:

Warning!

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.

3.1.5 Non-Linear Relationships in Data

Suppose our data now looks like this:



Putting our previous *best line* defined in equation (3.1.2) as $y = \beta_0 + \beta_1 x + u$, 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 (3.1.2) to read now

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i$$

This is a special case of *multiple regression*, which we will talk about in chapter 5. You can see that there are *multiple* slope coefficients. For now, let's just see how this performs:

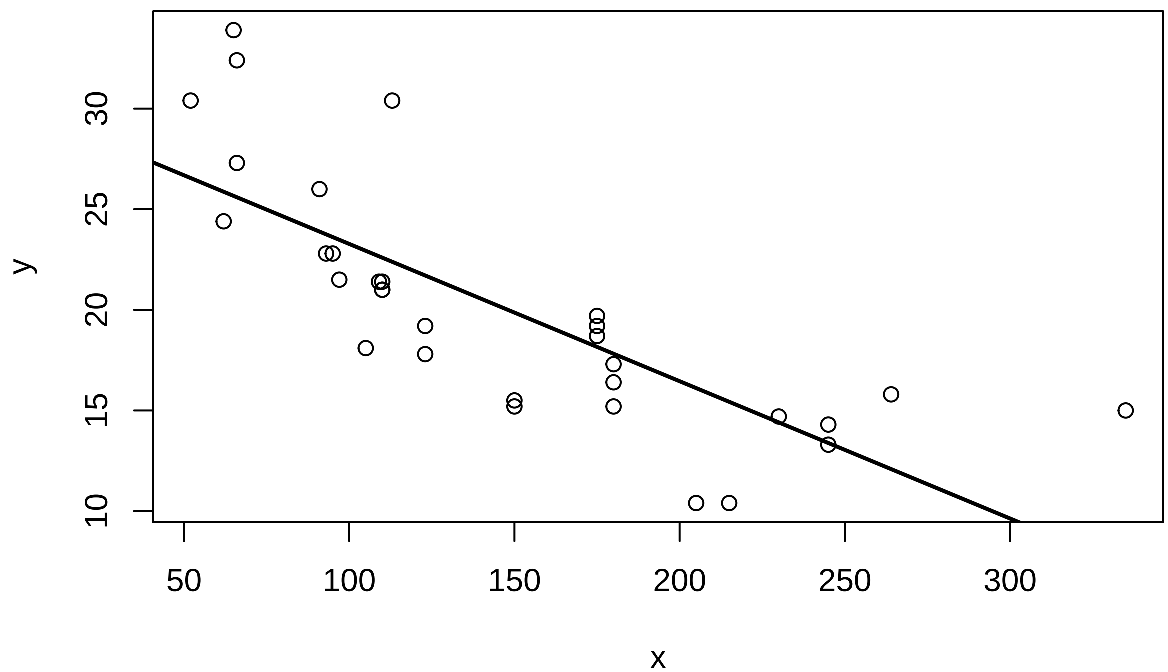


Figure 3.3: Best line with non-linear data?

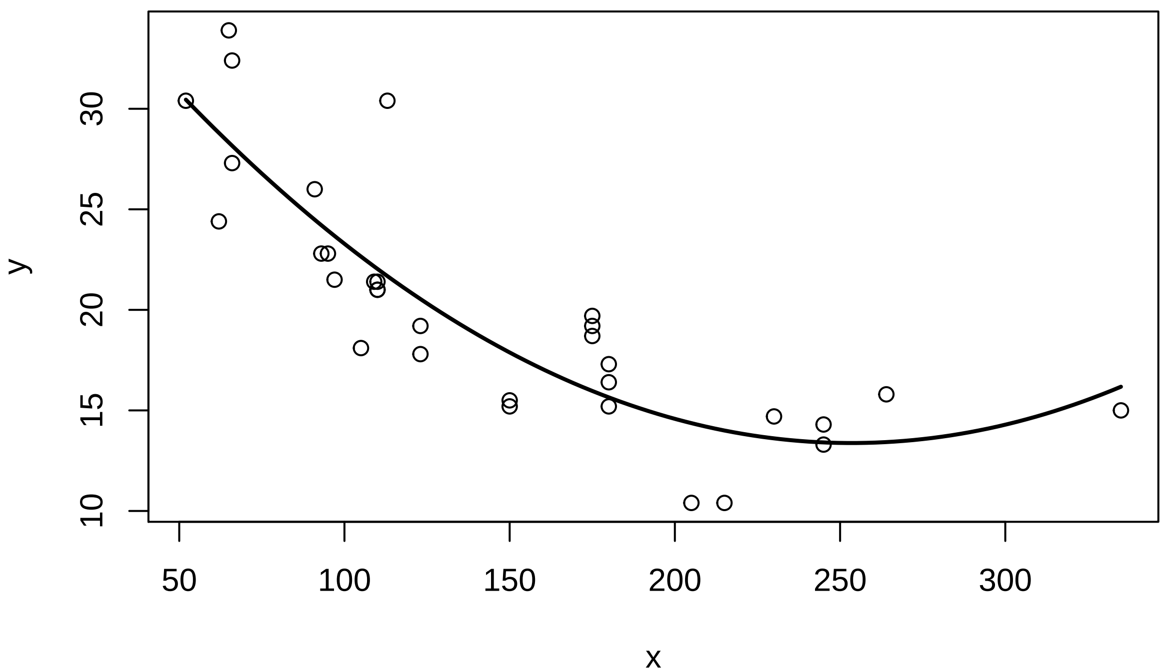


Figure 3.4: Better line with non-linear data!

3.2 DGP and Models

When we talk about a **model** in econometrics, we are making assumptions about how y and x are related in the data. For example, we have repeatedly seen the following equation,

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

which is a particular kind of model. What *generated* our data, on the other hand, is an unknown mechanism that we want to investigate: it's the **data generating process** (GDP), and our model is our assumption about how we think the GDP could look like. A natural question that comes to mind here, is *how to discriminate between models*, or in other words: which model to choose?

3.2.1 Assessing the *Goodness of Fit*

In our simple 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*. It is a statistic that makes use of a *benchmark* model, against which to compare any given model we may have in mind. Suppose we posit our standard representation of the best line:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

and let us write down the benchmark model as follows:

$$y_i = \beta_0 + \varepsilon_i$$

As you can see, the benchmark model in (3.2.1) is a model with an intercept only. You will see in one of our `apps` that this delivers an estimate of the mean of y. It is a benchmark because it does not include *any* explanatory variables, so we can compare against this other models which do in fact contain some x's. Back to our R^2 statistic: there are several equivalent definitions, and for our present case we will use the following.

Tip:

The **coefficient of determination** (R *squared*) is defined by

$$R^2 = 1 - \frac{\text{SSR our model}}{\text{SSR benchmark}}.$$

In the simple linear model, we have that $R^2 \in [0, 1]$, where $R^2 = 1$ would indicate that our model is a **very good** fit to the data, and vice versa for $R^2 = 0$. You can interpret the value of R^2 as the fraction of variation in outcome y that is accounted for by explanatory variable x.

The workings of this statistic are illustrated in the following figure 3.5. There, the left panel is our well-known depiction of the sum of squared residuals (SSR) of our model $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$. The right panel shows the SSR of $y_i = \beta_0 + \varepsilon_i$. Ideally, each red square would be small relative to its blue counterpart, indicating that our model has a small residual at a given observation.

3.3 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!

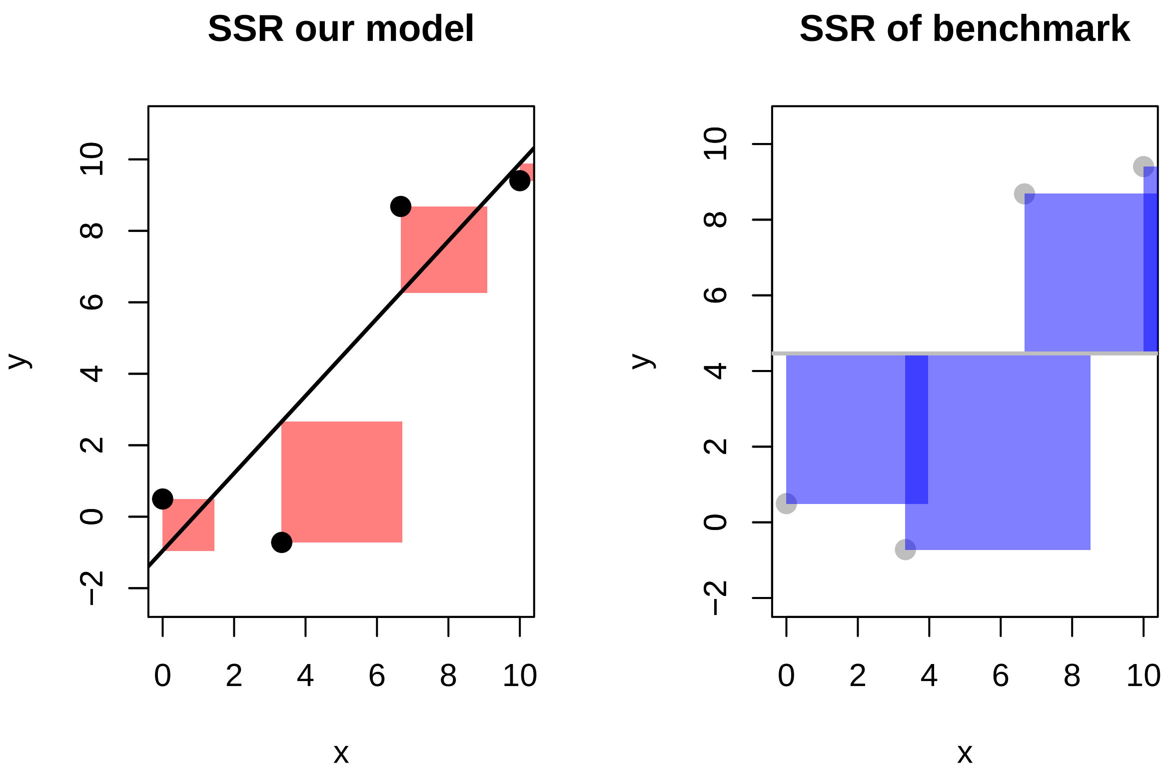


Figure 3.5: Left panel: SSR from our model. Right panel: SSR from benchmark (mean only) model. R^2 compares the size of each red square to each blue square.

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.

3.3.1 Loading and exploring Data

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

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

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

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

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

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

3.3.2 Fitting a linear model

Suppose a policymaker is interested in the following linear model:

$$testscr_i = \beta_0 + \beta_1 \times str_i + \epsilon_i$$

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. We can think of β_0 and β_1 as 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 ϵ_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 (don't worry if the syntax doesn't make much sense right now, we will come back to it very soon):

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

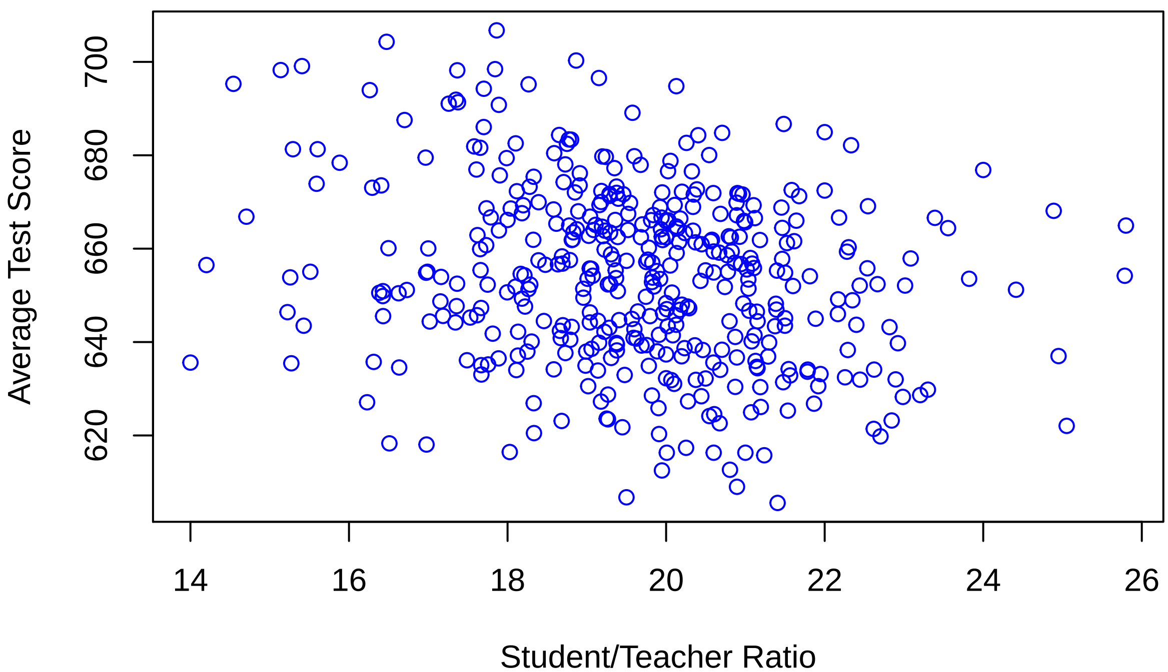


Figure 3.6: Student Teacher Ratio vs Test Scores

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`!

3.4 The `lm()` function

We will use the built-in `lm()` function to estimate the coefficients β_0 and β_1 using the data at hand. `lm` stands for *linear model*, which is what our representation in (3.1.2) 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 $income = \beta_0 + \beta_1 \cdot age$ can be written as `income ~ age`. You can also ask `R` to estimate a multivariate regression such as $income = \beta_0 + \beta_1 \cdot age + \beta_2 \cdot isWoman$ by simply separating all variables on the right-hand side of the equation with the `+` operator, like this : `income ~ age + isWoman`. 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:

```
lm(formula = testscr ~ str, data = Caschool)
```

```
##
## Call:
## lm(formula = testscr ~ str, data = Caschool)
##
## Coefficients:
## (Intercept)          str
##      698.93        -2.28
```

As we can see, `R` returns its estimates for the Intercept and Slope coefficients, $\hat{\beta}_0$ = 698.93 and $\hat{\beta}_1$ = -2.28. The estimated relationship between a school's Student/Teacher Ratio and its average test results is **negative**.

Running a linear regression in `R` is typically a two-steps process. You first assign the output of the `lm()` call to an object and **then** call a second function (for our purpose, mainly `summary()`) on the resulting object. In practice, this looks like this :

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

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

```
##
## Call:
## lm(formula = testscr ~ str, data = Caschool)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -47.727 -14.251   0.483  12.822  48.540
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) 698.9330     9.4675  73.825  < 2e-16 ***
## str          -2.2798     0.4798  -4.751 2.78e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.58 on 418 degrees of freedom
## Multiple R-squared:  0.05124,    Adjusted R-squared:  0.04897
## F-statistic: 22.58 on 1 and 418 DF,  p-value: 2.783e-06
```

Again, we recognize our intercept and slope estimates from before, alongside some other numbers and indications. This output is 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 not?

3.4.1 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_california, col = 'red') # add 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`:

```r
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")
```

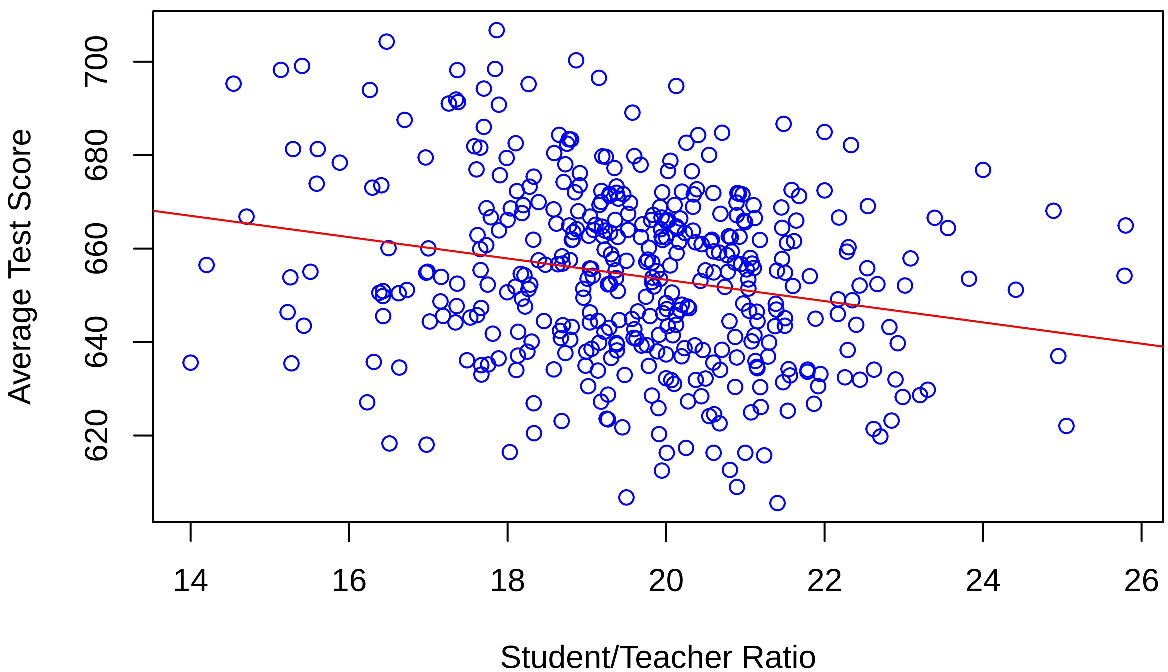
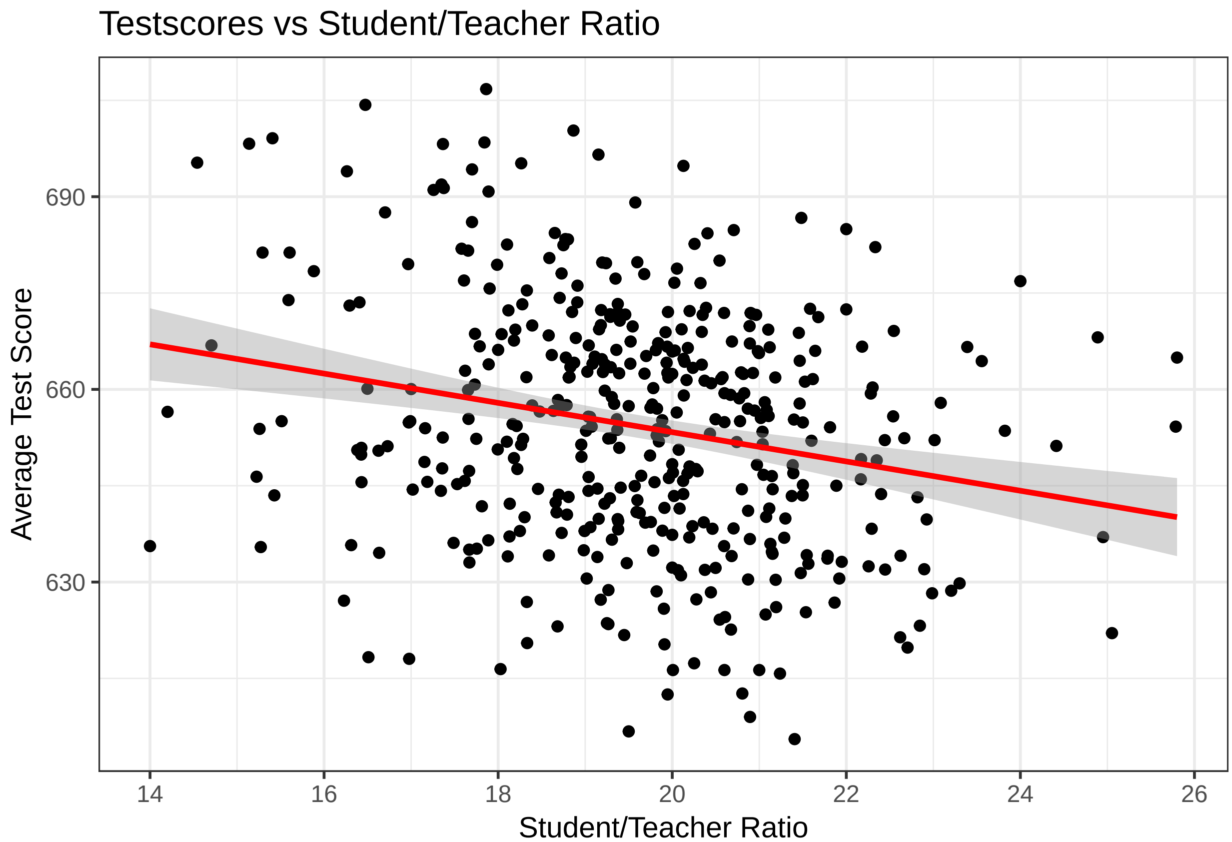


Figure 3.7: Test Scores with Regression Line



The shaded area around the red line shows the width of the 95% confidence interval around our estimate of the slope coefficient β_1. We will learn more about it in the next chapter.

Chapter 4

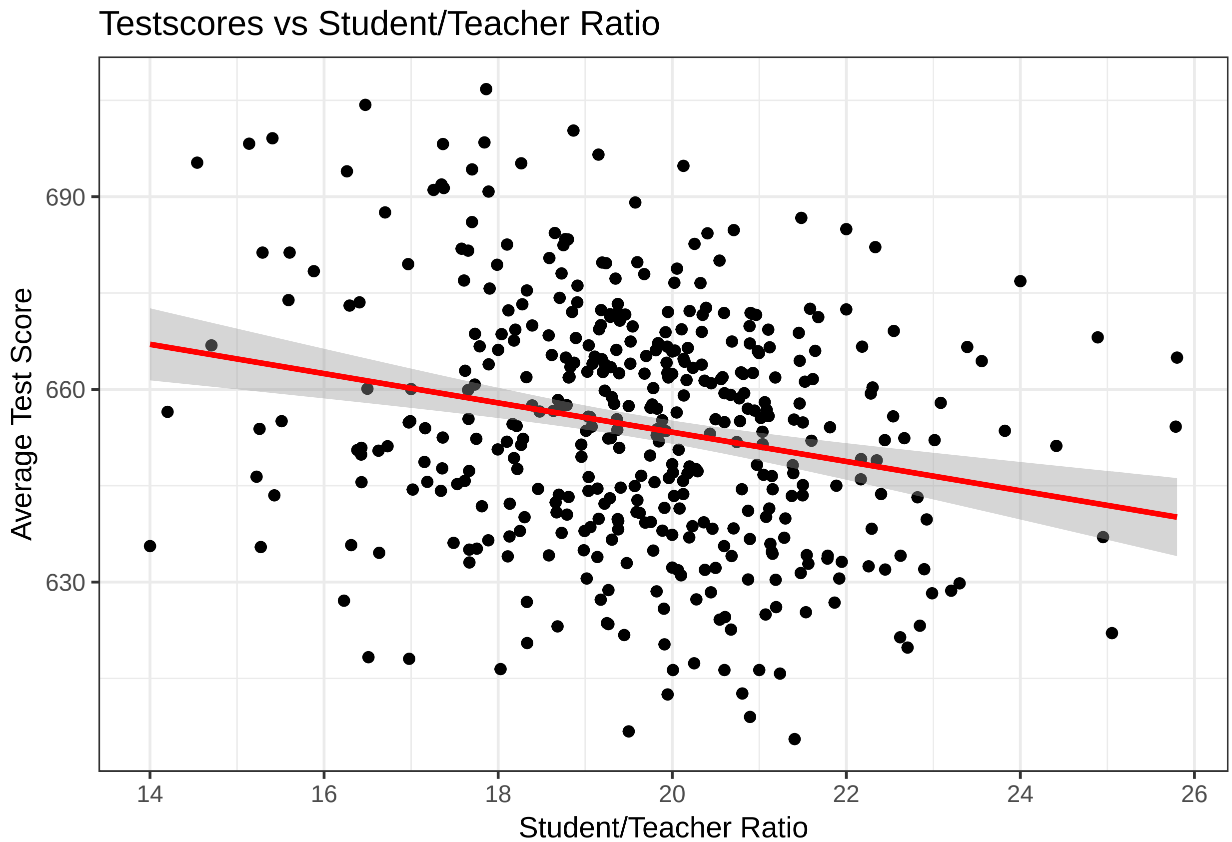
Standard Errors

In the previous chapter 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 $\hat{\beta}_0$ and $\hat{\beta}_1$ 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.

Note:

How *confident* should we be about the estimated values $\hat{\beta}_0$ and $\hat{\beta}_1$?

Let's remind ourselves of the example at the end of the previous chapter, and that we introduced the term *confidence interval*, shown here as the shaded area:



The shaded area shows us the region within which the **true** red line will lie with 95% probability. The fact that there is 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.

4.1 What is *true*?

We have a true data-generating process in mind. Let's bring back our simple model (3.1.2) from the previous chapter:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

First, we *assume* that this is the correct represenation of the DGP, which looks like the above equation. With that assumption in place, the values β_0 and β_1 are the *true parameter values* which generated the data. Notice, that both β_0 and β_1 don't have a "hat", which is widely used to indicate an estimate. Now, the fact that our data (y_i, x_i) are a sample from a larger population means that there will be *sampling variation* in our estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ - exactly like in the case of the sample mean estimating the population average as mentioned above.

4.2 Experiencing Standard Errors

We would like to make this point in a purely experiential way, i.e. we want you to experience what is going on. We invite you to spend some time with the following apps, before going into the associated tutorial:

```
library(ScPoEconometrics)
launchApp("standard_errors_simple") # start with this
launchApp("standard_errors_changeN")  # then do that
library(learnr)   # load the learner library
run_tutorial("standared-errors")   # WIP
```

4.3 Standard Errors in Theory

The precise formulae for the standard errors of regression coefficients depend critically on exactly which model we talk about. In other words, certain assumptions about the underlying DGP give rise to certain standard error formulas. What we continue to call the *simple linear regression model* is in fact a list of 5 assumptions. It is time to spell them out here:

1. The model is linear in parameters. I.e. it looks like (4.1) above.
2. (y_i, x_i) constitute a *random* sample from the underlying population: the sample is *representative*.
3. The mean of ε in (4.1) is zero, conditional on x. This means that ε and x should not be correlated.
4.

In our simple linear regression model the standard errors of the estimates have the following form

Chapter 5

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

$$mpg_i = \beta_0 + \beta_1 hp_i + \beta_2 wt_i + \varepsilon_i$$

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

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

```
##                      mpg  hp    wt
## Mazda RX4           21.0 110 2.620
## Mazda RX4 Wag       21.0 110 2.875
## Datsun 710          22.8  93 2.320
## Hornet 4 Drive      21.4 110 3.215
## Hornet Sportabout   18.7 175 3.440
## Valiant             18.1 105 3.460
## Duster 360          14.3 245 3.570
## Merc 240D           24.4  62 3.190
## Merc 230            22.8  95 3.150
## Merc 280            19.2 123 3.440
## Merc 280C           17.8 123 3.440
## Merc 450SE          16.4 180 4.070
## Merc 450SL          17.3 180 3.730
## Merc 450SLC         15.2 180 3.780
## Cadillac Fleetwood  10.4 205 5.250
## Lincoln Continental 10.4 215 5.424
## Chrysler Imperial   14.7 230 5.345
## Fiat 128            32.4  66 2.200
## Honda Civic         30.4  52 1.615
## Toyota Corolla      33.9  65 1.835
## Toyota Corona       21.5  97 2.465
## Dodge Challenger    15.5 150 3.520
## AMC Javelin         15.2 150 3.435
## Camaro Z28          13.3 245 3.840
```

Figure 5.1: Multiple Regression - a plane in 3D. The red lines indicate the residual for each observation.

```
## Pontiac Firebird    19.2 175 3.845
## Fiat X1-9           27.3  66 1.935
## Porsche 914-2       26.0  91 2.140
## Lotus Europa        30.4 113 1.513
## Ford Pantera L      15.8 264 3.170
## Ferrari Dino        19.7 175 2.770
## Maserati Bora       15.0 335 3.570
## 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 β_1 and β_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 5.1:

This visualization shows a couple of things: the data are shown with red points, the grey plane is the one resulting from OLS estimation of equation (5), and the red lines show the size of the error between estimated plane and observed data. You should realize that this is exactly the same story as told in figure 3.1 - just in three dimensions!

We can see from this 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

Tip:

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 5.1 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 (5), we want to know the following quantity:

$$\frac{\partial mpg_i}{\partial hp_i} = \beta_1$$

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 (5). We call the value of coefficient β_1 therefore also the *partial effect* of `hp` on `mpg`. In terms of our dataset, we use `R` to run the following **multiple regression**:

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

Warning!

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*.

5.1 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:

$$\text{testscr}_i = \beta_0 + \beta_1 \text{str}_i + \beta_2 \text{avginc}_i + \epsilon_i$$

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)
```

```
##
## Call:
## lm(formula = "testscr ~ str + avginc", data = Caschool)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.608  -9.052   0.707   9.259  31.898
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 638.72915    7.44908  85.746   <2e-16 ***
## str          -0.64874    0.35440  -1.831   0.0679 .
## avginc        1.83911    0.09279  19.821   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.35 on 417 degrees of freedom
## Multiple R-squared:  0.5115, Adjusted R-squared:  0.5091
## 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.

5.2 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. To measure such an effect, we would reformulate our model like this:

$$\text{testscr}_i = \beta_0 + \beta_1 \text{str}_i + \beta_2 \text{avginc}_i + \beta_3 (\text{str}_i \times \text{avginc}_i) + \epsilon_i$$

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 (5.2) with respect to `str`:

$$\frac{\partial \text{testscr}_i}{\partial \text{str}_i} = \beta_1 + \beta_3 \text{avginc}_i$$

> You should go back to equation (5) to remind yourself of what a *partial effect* was, and how exactly the present (5.2) differs from what we saw there.

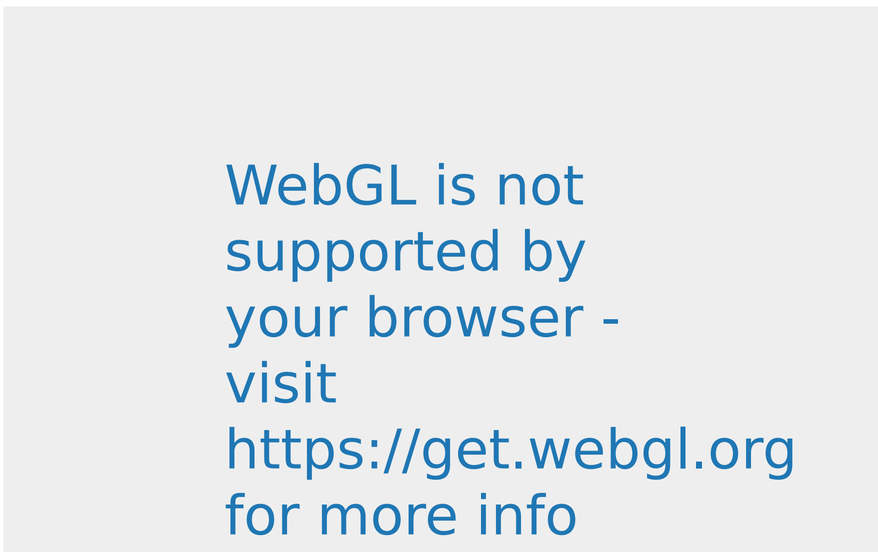


Figure 5.2: Californa Test Scores vs student/teach ratio and avg income.

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)
```

```
##
## Call:
## lm(formula = testscr ~ str + avginc + str * avginc, data = Caschool)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -41.346  -9.260   0.209   8.736  33.368
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 689.47473   14.40894  47.850  < 2e-16 ***
## str          -3.40957    0.75980  -4.487 9.34e-06 ***
## avginc       -1.62388    0.85214  -1.906   0.0574 .
## str:avginc    0.18988    0.04646   4.087 5.24e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.1 on 416 degrees of freedom
## Multiple R-squared:  0.5303, Adjusted R-squared:  0.527
## 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 β_3 for us. We observe also that the estimate of β_2 changes signs and becomes negative, while the interaction effect β_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 5.3 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 "5.3 looks nicer" ;-)). In particular, comparing both visualizations, can you explain why we observe this strange inversion of coefficient signs?

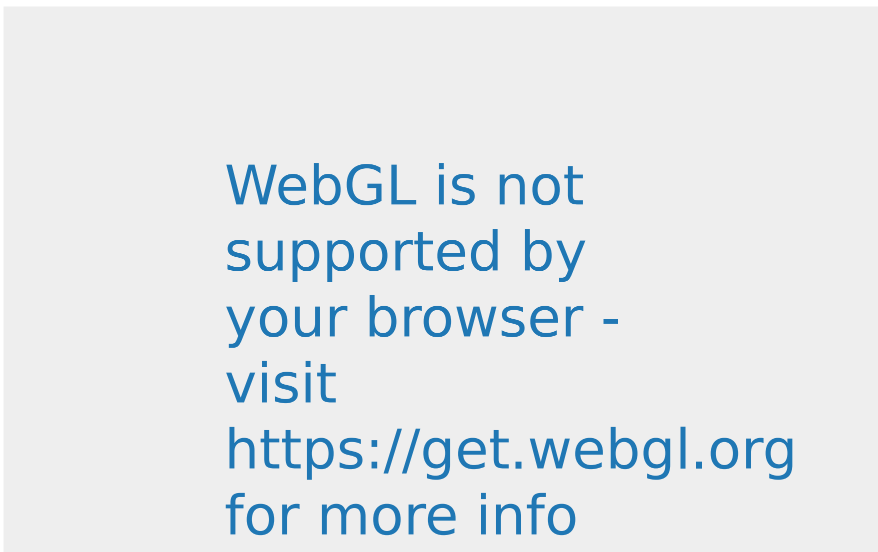


Figure 5.3: Californa Test Scores vs student/teach ratio and avg income plus interaction term

Chapter 6

Categorial Variables

TODO: excluded category vs intercept?

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`. 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, we could have a variable called `is.male` that is `TRUE` whenever i is male, and `FALSE` otherwise. A common way to represent this is with the so-called *indicator function* $\mathbf{1}$[condition],

$$\text{is.male}_i = \mathbf{1}[\text{sex}_i == \text{male}],$$

which would just mean

$$\mathbf{1}[\text{sex}_i == \text{male}] = \begin{cases} 1 & \text{if } i \text{ is male} \\ 0 & \text{if } i \text{ is female.} \end{cases}$$

Notice the use of `x == y` to represent the relationship ***x** is equal **y***, which is (very!) different from `x = y` meaning *assign **y** to **x***.

In terms of a regression formulation, this is our model when our regressor is binary:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, x_i \in \{0, 1\}$$

Let's run that regression here:

```
##
## Call:
## lm(formula = y ~ x, data = dta)
##
```

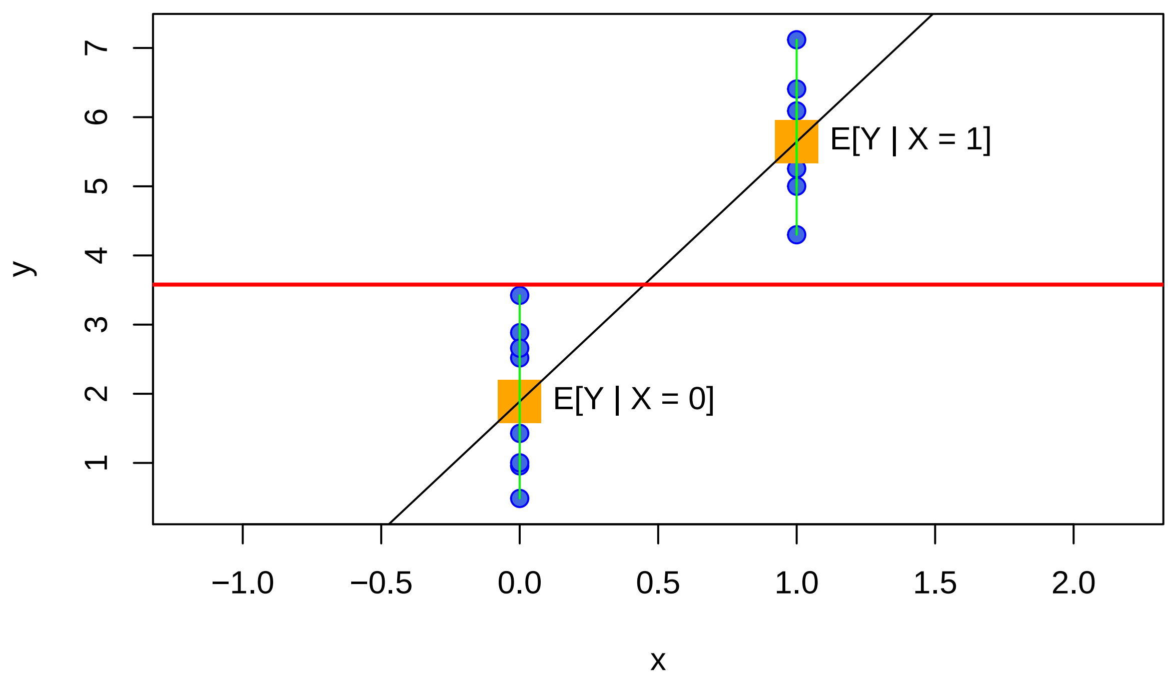


Figure 6.1: regressing $y \in \mathbb{R}$ on $x \in \{0, 1\}$. The red line is $E[y]$

```
## Coefficients:
## (Intercept)            x
##       1.889        3.756
```

We have seen in the `app` launched via

```
launchApp("reg_dummy")
```

that this regression simplifies to the straight line connecting the mean, or the *expected value* of y when $x = 0$, i.e. $E[y|x = 0]$, to the mean when $x = 1$, i.e. $E[y|x = 1]$. 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 red line. The red line will always lie in between both conditional means. Let's just refresh our memory by replicating the graph from the `app` here in figure 6.1:

Now suppose for a moment that in fact

$$x_i = \begin{cases} 1 & \text{if } i \text{ is male} \\ 0 & \text{if } i \text{ is female.} \end{cases}$$

and that y_i is a measure of i's annual labor income. The dummy variable version of the above regression is just

$$y_i = \beta_0 + \alpha \text{is.male}_i + \varepsilon_i$$

where

$$\text{is.male}_i = \mathbf{1}[x_i == 1],$$

and the resulting estimates of β_1 and α are in fact the same. Here we see the estimates from (6):

```
##
## Call:
## lm(formula = y ~ is.male, data = dta)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.40227 -0.50683 -0.09763  0.66287  1.53463
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.8888     0.2617   7.217 1.03e-06 ***
## is.male1      3.7558     0.3902   9.626 1.60e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.868 on 18 degrees of freedom
## Multiple R-squared:  0.8374, Adjusted R-squared:  0.8283
## F-statistic: 92.67 on 1 and 18 DF,  p-value: 1.599e-08
```

It is interesting to note that the estimate for $\alpha = 3.7558$ (and $\beta_1 = 3.7558!$) is the same as the difference in conditional means:

$$E[y|x = 1] - E[y|x = 0] = 3.7558.$$

6.1 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 1.6 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.male` for the above regression:

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

```
## [1] FALSE TRUE  TRUE  FALSE
## 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

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

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

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

6.1.1 More Levels

We can go *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:

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

```
## [1] HS      HS      someCol MSc     BA      MSc
## 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`.)

6.1.2 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.

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

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

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*. Let's initially say that a workers wage depends only on his *experience*, measured in the number of years he/she worked full-time:

$$\ln w_i = \beta_0 + \beta_1 exp_i + \varepsilon_i$$

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

```
##
## Call:
## lm(formula = lwage ~ exp, data = Wages)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.30153 -0.29144  0.02307  0.27927  1.97171
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 6.5014318  0.0144657  449.44   <2e-16 ***
## exp         0.0088101  0.0006378   13.81   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4513 on 4163 degrees of freedom
## Multiple R-squared:  0.04383,    Adjusted R-squared:  0.0436
## 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 $\exp(\hat{\beta}_1) = 1.0088491$. This means that hourly wages increase by roughly $100 * (\exp(\hat{\beta}_1) - 1) = 0.88$ percent with an additional year of experience. We can verify the positive relationship in figure 6.2.

Now let's investigate whether this relationship different for men and women.

$$\ln w_i = \beta_0 + \beta_1 exp_i + \beta_2 sex_i + \varepsilon_i$$

We can do this easily by using the `update` function as follows:

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

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

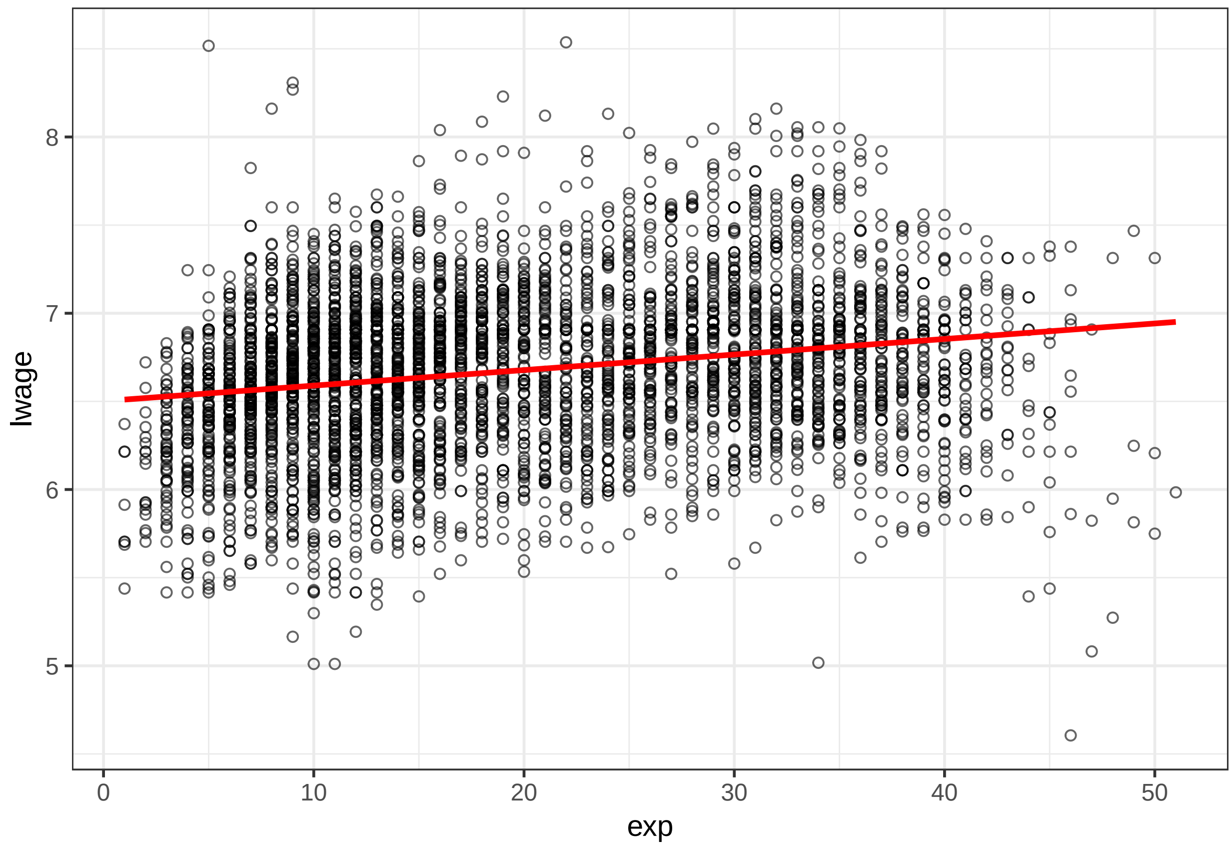


Figure 6.2: log wage vs experience. Red line shows the regression.

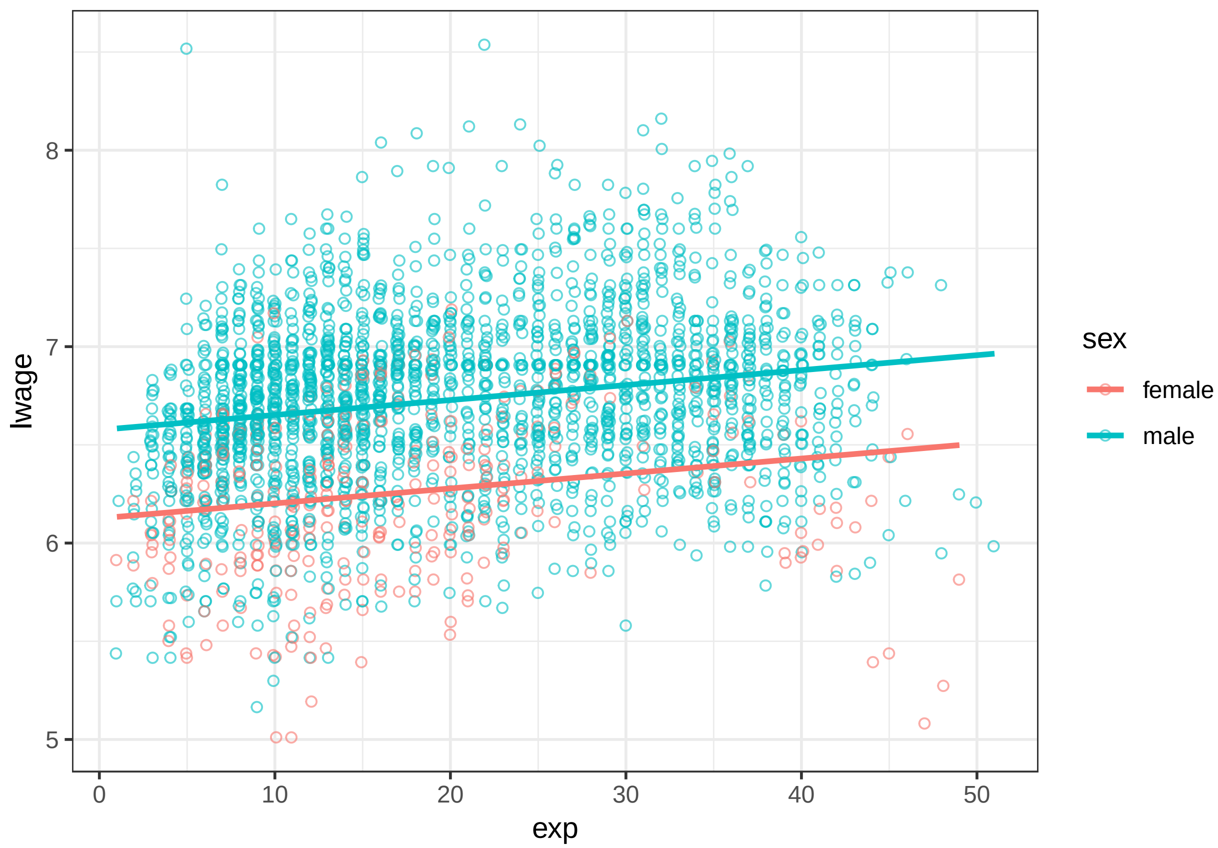


Figure 6.3: log wage vs experience with different intercepts by sex

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 β_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 6.3 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 $\beta_2 = 0.45$ is the size of the downward shift.

6.2 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 5.2, interactions allow that the *ceteris paribus* effect of a certain regressor, `exp` say, 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 6.3. Therefore we formulate this version of our model:

$$\ln w_i = \beta_0 + \beta_1 exp_i + \beta_2 sex_i + \beta_3 (sex_i \times exp_i) + \varepsilon_i$$

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 (6.2) with respect to `sex`:

$$\frac{\partial \ln w_i}{\partial sex_i} = \beta_2 + \beta_3 exp_i$$

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

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

```
##
## Call:
## lm(formula = lwage ~ exp * sex, data = Wages)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.82137 -0.26797  0.01781  0.26231  1.90757
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 6.169017   0.038165 161.643  < 2e-16 ***
## exp         0.005071   0.001918   2.644  0.00822 **
## sexmale     0.401116   0.040917   9.803  < 2e-16 ***
## exp:sexmale 0.002826   0.002022   1.397  0.16236
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4285 on 4161 degrees of freedom
## Multiple R-squared:  0.1385, Adjusted R-squared:  0.1379
## 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 $\beta_3 = 0$. (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 (6.2) 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:

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

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

```
## Coefficients:
##                                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)                       6.12378    0.06300  97.198  < 2e-16
## sexmale                           0.67057    0.06577  10.195  < 2e-16
## smsayes                           0.33424    0.06872   4.864 1.19e-06
## unionyes                          0.84284    0.16866   4.997 6.06e-07
## bluecolyes                       -0.34016    0.08423  -4.039 5.47e-05
## sexmale:smsayes                  -0.15917    0.07226  -2.203 0.027670
## sexmale:unionyes                 -0.92893    0.17816  -5.214 1.94e-07
## smsayes:unionyes                 -0.83927    0.17979  -4.668 3.14e-06
## sexmale:bluecolyes               -0.15046    0.08820  -1.706 0.088100
## smsayes:bluecolyes               -0.12471    0.09727  -1.282 0.199882
## unionyes:bluecolyes              -0.31819    0.22924  -1.388 0.165208
## sexmale:smsayes:unionyes          0.72672    0.19060   3.813 0.000139
## sexmale:smsayes:bluecolyes        0.25860    0.10327   2.504 0.012318
## sexmale:unionyes:bluecolyes       0.71906    0.23772   3.025 0.002503
## smsayes:unionyes:bluecolyes       0.50057    0.24862   2.013 0.044137
## sexmale:smsayes:unionyes:bluecolyes -0.58330    0.25899  -2.252 0.024361
##
## (Intercept)                      ***
## sexmale                          ***
## smsayes                          ***
## unionyes                         ***
## bluecolyes                       ***
## sexmale:smsayes                  *
## sexmale:unionyes                 ***
## smsayes:unionyes                 ***
## sexmale:bluecolyes               .
## smsayes:bluecolyes
## unionyes:bluecolyes
## sexmale:smsayes:unionyes         ***
## sexmale:smsayes:bluecolyes       *
## sexmale:unionyes:bluecolyes      **
## smsayes:unionyes:bluecolyes      *
## sexmale:smsayes:unionyes:bluecolyes *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3832 on 4149 degrees of freedom
## Multiple R-squared:  0.3129, Adjusted R-squared:  0.3105
## 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 7

Quantile Regression

1. before you were modelling the mean. the average link
2. now what happens to **outliers**? how robust is the mean to that
3. what about the entire distribution of this?

Chapter 8

Panel Data

8.1 fixed effects

8.2 DiD

8.3 RDD

8.4 Example

- scanner data on breakfast cereals, (Q_{it}, D_{it})
- why does D vary with Q
- pos relation ship
- don't observe the group identity!
- unobserved het alpha is correlated with Q
- within group estimator
- what if you don't have panel data?

Chapter 9

Instrumental Variables

- Measurement error
- Omitted Variable Bias
- Reverse Causality / Simultaneity Bias

are all called *endogeneity* problems.

9.1 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**:

$$\text{crime}_{it} = f(\text{police}_{it})$$
$$\text{police}_{it} = g(\text{crime}_{it})$$

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 10

Logit and Probit

Chapter 11

Principal Component Analysis

Chapter 12

Notes

this creates a library for the used R packages.

In order to install that package, you need to do

```
if (!require(devtools)){
  install.packages("devtools")
}
library(devtools)
install_github("floswald/ScPoEconometrics")
```

12.1 Book usage

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter **??**. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 3.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 12.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 12.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2018) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

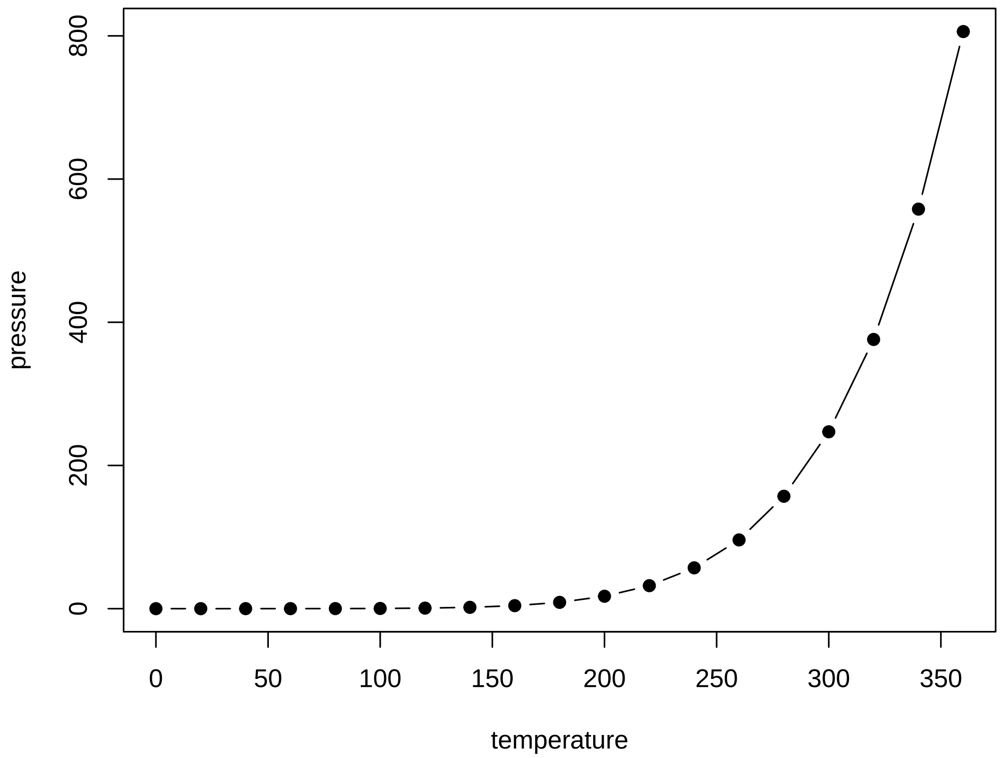


Figure 12.1: Here is a nice figure!

Table 12.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Chapter 13

Advanced R

This chapter continues with some advanced usage examples from chapter 1

13.1 More Vectorization

```
x = c(1, 3, 5, 7, 8, 9)
y = 1:100
```

```
x + 2
```

```
## [1]  3  5  7  9 10 11
```

```
x + rep(2, 6)
```

```
## [1]  3  5  7  9 10 11
```

```
x > 3
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
x > rep(3, 6)
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##   [1]   2   5   8  11  13  15   8  11  14  17  19  21  14  17  20  23  25
##  [18]  27  20  23  26  29  31  33  26  29  32  35  37  39  32  35  38  41
##  [35]  43  45  38  41  44  47  49  51  44  47  50  53  55  57  50  53  56
##  [52]  59  61  63  56  59  62  65  67  69  62  65  68  71  73  75  68  71
##  [69]  74  77  79  81  74  77  80  83  85  87  80  83  86  89  91  93  86
##  [86]  89  92  95  97  99  92  95  98 101 103 105  98 101 104 107
```

```
length(x)
```

```
## [1] 6
```

```
length(y)
```

```
## [1] 100
```

```
length(y) / length(x)
```

```
## [1] 16.66667
```

```
(x + y) - y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length

##   [1] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8
##  [36] 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7
##  [71] 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7
```

```
y = 1:60
x + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27 20 23 26 29 31
## [24] 33 26 29 32 35 37 39 32 35 38 41 43 45 38 41 44 47 49 51 44 47 50 53
## [47] 55 57 50 53 56 59 61 63 56 59 62 65 67 69
```

```
length(y) / length(x)
```

```
## [1] 10
```

```
rep(x, 10) + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27 20 23 26 29 31
## [24] 33 26 29 32 35 37 39 32 35 38 41 43 45 38 41 44 47 49 51 44 47 50 53
## [47] 55 57 50 53 56 59 61 63 56 59 62 65 67 69
```

```
all(x + y == rep(x, 10) + y)
```

```
## [1] TRUE
```

```
identical(x + y, rep(x, 10) + y)
```

```
## [1] TRUE
```

```
# ?any
# ?all.equal
```

13.2 Calculations with Vectors and Matrices

Certain operations in R, for example `%*%` have different behavior on vectors and matrices. To illustrate this, we will first create two vectors.

```
a_vec = c(1, 2, 3)
b_vec = c(2, 2, 2)
```

Note that these are indeed vectors. They are not matrices.

```
c(is.vector(a_vec), is.vector(b_vec))
```

```
## [1] TRUE TRUE
```

```
c(is.matrix(a_vec), is.matrix(b_vec))
```

```
## [1] FALSE FALSE
```

When this is the case, the `%*%` operator is used to calculate the **dot product**, also know as the **inner product** of the two vectors.

The dot product of vectors $\boldsymbol{a} = [a_1, a_2, \cdots a_n]$ and $\boldsymbol{b} = [b_1, b_2, \cdots b_n]$ is defined to be

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots a_n b_n.$$

```
a_vec %*% b_vec # inner product
```

```
##      [,1]
## [1,]   12
```

```
a_vec %o% b_vec # outer product
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

The `%o%` operator is used to calculate the **outer product** of the two vectors.

When vectors are coerced to become matrices, they are column vectors. So a vector of length n becomes an $n \times 1$ matrix after coercion.

```
as.matrix(a_vec)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

If we use the `%*%` operator on matrices, `%*%` again performs the expected matrix multiplication. So you might expect the following to produce an error, because the dimensions are incorrect.

```
as.matrix(a_vec) %*% b_vec
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

At face value this is a 3×1 matrix, multiplied by a 3×1 matrix. However, when `b_vec` is automatically coerced to be a matrix, `R` decided to make it a "row vector", a 1×3 matrix, so that the multiplication has conformable dimensions.

If we had coerced both, then `R` would produce an error.

```
as.matrix(a_vec) %*% as.matrix(b_vec)
```

Another way to calculate a *dot product* is with the `crossprod()` function. Given two vectors, the `crossprod()` function calculates their dot product. The function has a rather misleading name.

```
crossprod(a_vec, b_vec)  # inner product
```

```
##      [,1]
## [1,]   12
```

```
tcrossprod(a_vec, b_vec)  # outer product
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

These functions could be very useful later. When used with matrices X and Y as arguments, it calculates

$$X^\top Y.$$

When dealing with linear models, the calculation

$$X^\top X$$

is used repeatedly.

```r
C_mat = matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
D_mat = matrix(c(2, 2, 2, 2, 2, 2), 2, 3)
```

This is useful both as a shortcut for a frequent calculation and as a more efficient implementation than using `t()` and `%*%`.

```r
crossprod(C_mat, D_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```r
t(C_mat) %*% D_mat
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```r
all.equal(crossprod(C_mat, D_mat), t(C_mat) %*% D_mat)
```

```
## [1] TRUE
```

```r
crossprod(C_mat, C_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```r
t(C_mat) %*% C_mat
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```r
all.equal(crossprod(C_mat, C_mat), t(C_mat) %*% C_mat)
```

```
## [1] TRUE
```

13.3 Matrices

```
Z = matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

To verify that `solve(Z)` returns the inverse, we multiply it by `Z`. We would expect this to return the identity matrix, however we see that this is not the case due to some computational issues. However, `R` also has the `all.equal()` function which checks for equality, with some small tolerance which accounts for some computational issues. The `identical()` function is used to check for exact equality.

```
solve(Z) %*% Z
```

```
##              [,1]          [,2]         [,3]
## [1,] 1.000000e+00 -6.245005e-17 0.000000e+00
## [2,] 8.326673e-17  1.000000e+00 5.551115e-17
## [3,] 2.775558e-17  0.000000e+00 1.000000e+00
```

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
all.equal(solve(Z) %*% Z, diag(3))
```

```
## [1] TRUE
```

`R` has a number of matrix specific functions for obtaining dimension and summary information.

```
X = matrix(1:6, 2, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(X)
```

```
## [1] 2 3
```

```
rowSums(X)
```

```
## [1]  9 12
```

```
colSums(X)
```

```
## [1]  3  7 11
```

```r
rowMeans(X)
```

```
## [1] 3 4
```

```r
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

The `diag()` function can be used in a number of ways. We can extract the diagonal of a matrix.

```r
diag(Z)
```

```
## [1]  9  4 16
```

Or create a matrix with specified elements on the diagonal. (And `0` on the off-diagonals.)

```r
diag(1:5)
```

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

Or, lastly, create a square matrix of a certain dimension with `1` for every element of the diagonal and `0` for the off-diagonals.

```r
diag(5)
```

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

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.7.