Introduction to R

Amir B. Ferreira Neto

Contents

Introduction **2**

Understanding this guide . . . 2

1. Basic R **2**

1.1 Download and set-up . . . 3

1.2 Some best-practices . . . 4

1.3 Using R . . . 4

2. Data Manipulation **9**

2.1 Import/Export . . . 10

2.2 Representation of data . . . 11

2.3 Pipes . . . 12

2.4 Data Manipulation/Wrangling . . . 12

2.4.1 Select & Filter . . . 13

2.4.2 Mutate & Summarise . . . 15

2.4.3 Group_by . . . 18

2.4.4 Working with multiple objects . . . 18

2.4.5 Reshaping data . . . 21

3. Visualization **24**

4. RMarkdown **37**

4.1 Getting Started . . . 37

4.2 The RMarkdown File . . . 38

4.2.1 Header . . . 38

4.2.2 Text . . . 38

4.2.3 Code Chunk . . . 38

4.2.4 Compiling your RMarkdown . . . 38

References **39**

Introduction

There are several softwares/programming languages you can use to run your statistical analysis. In this guide I will introduce you to R.[1] The material here is mostly based on the book *R for Data Science* (Grolemund and Wickham 2017) and other resources in the internet[2].

With this guide you will to be able to download and install *R* and *RStudio*, and use start to use it for your own statistical analysis. I am breaking down this guide into 4 parts:

1. Basic R
2. Data Manipulation
3. Visualization
4. Regression Analysis

Before we start I want to highlight why you should choose to learn R:

- Like any language, the more you learn the easier it is to pick up new ones
- Unlike *Stata* that is mainly used by economists, R is consistent with broader computer science world; this means there is a larger network of users
  - closer to stats/machine learning community
  - great for spatial analysis
- R has better visualizations and is easier for writing customs packages and functions
- Several tools with compelling value added such as: ggplot2, dplyr, RMarkdown, among others

Throughout this guide, one of your best friend will be Google and StackOverflow.

Understanding this guide

In this guide I will include small pieces of codes and their outcome in R for illustration purpose. Code will appear inside a gray box like this:

```
# This is a R code.
# When we introduce the character "#" everything after it becomes a comment. This means
# that it will not be read by R as part of the code. This is extremely useful and
# helpful to maintain a well organized codebook.
```

The outcome of R will be displayed as after two "#" characters followed by squared bracket numbers (e.g. [1]). For example, an output of will be shown as this:

```
# This is the output for the function print(), which is used to display something.
print("Hello world")
```

```
## [1] "Hello world"
```

1. Basic R

In this part I will go over the basics of R such as setting it up and understanding its fundamental commands/functions.

[1] You are free to use *Stata* if you prefer to do so in this class, but I will not be able to help you as much in coding, etc.
[2] Quick R, r4stats, Phillips (2018), and Prof. Grant McDermott's lectures

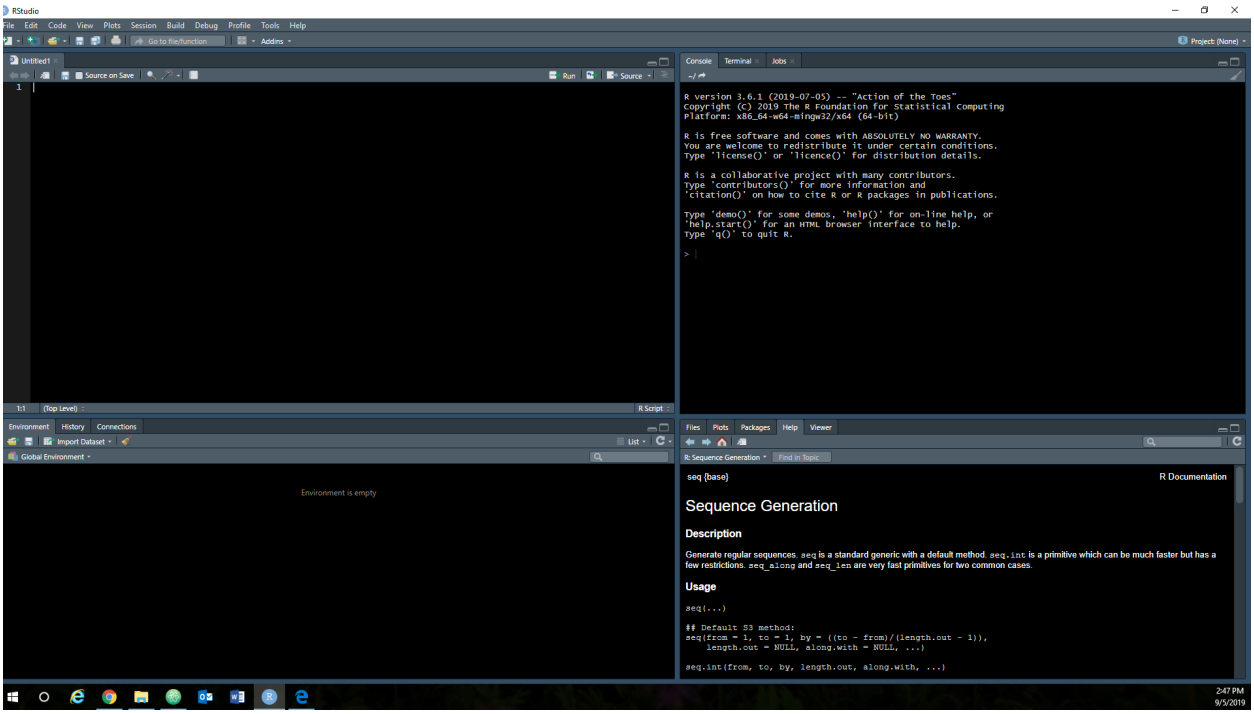
1.1 Download and set-up

First, let's download and install R:

- We download R from the r-project website. As this guide is being written, the most current version of R is *R.4.1.2* and you can download it either if you are a Windows or Mac user in this link: download R
- Follow the prompts on your screen to install it

To make it easier to use R for data analysis we make use of an IDE (integrated development environment) – *RStudio*.[3] In other words, RStudio allows for a better experience and for those who have used *Stata* before, it resembles it very much! Let's download and set up RStudio:

- We download RStudio from the rstudio website. The most current version of RStudio, as I write this guide, is *2021.09.1-372*. You can select the appropriate installer depending on your operating system in this link: download RStudio
- Follow the prompts on your screen to install it
- Set-up: go to your RStudio Preferences (this will vary for Mac/Windows users)
  - Appearance Tab: Editor theme = monokai. (feel free to choose the one you feel most comfortable with)
  - Panel Layout Tab: Top Left = *Source* ; Top Right = *Console* ; Bottom Left = *Environment,...* ; Bottom Right = *Files,...*



It is important then to make sense of each panel in RStudio:

- **Source**: This is where we write up our code; similar to Stata's *do-file*
- **Console**: This is where your code from *Source* is evaluated. (You can type in small pieces of code directly and it will run, but not save the code typed)
- **Environment**: You can see the objects in your working space
- **Files/.../Help**: You can see the files in your working directory, view your plots, access the Help

[3] As a side note, given its popularity, RStudio is becoming much more than a simple IDE for R.

1.2 Some best-practices

Before we delve into R, I want to introduce you to some best practices that will be very useful as you progress in your career and develop your coding skills. You can read more on this here and here. The most relevant for us are:

1. Always comment your code. Do **not** rely on your memory to remember what you are doing, why you are doing it, etc.
2. Be consistent throughout your code (with names, indentation, etc.)
3. Limit line length
4. Be organized
5. Avoid deep nesting

Remember, in the your ultimate end-goal is to have a code that produces the desired outcome, is reproducible and can be followed by anyone.

1.3 Using R

1.3.1 Calculator

You can use your console as calculator.

```
# Addition
2+2
```

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

```
# Subtraction
5-2
```

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

```
#Multiplication
3*3
```

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

```
#Division
14/7
```

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

```
#Exponent
2^4
```

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

```
#Whole Division
100%/% 60
```

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

```
#Remainder
100%%60
```

```
## [1] 40
```

R follows the order of operations (PEMDAS):

```
# The result of this equation:
59 + 73 + 2 / 3
```

```
## [1] 132.6667
```

```
# is different from this one:
(59 + 73 + 2) / 3
```

```
## [1] 44.66667
```

1.3.2 Assignment

R is object-oriented. This means that **"Everything is an object and everything hsa a name"**.

There is two ways to assign values to objects, either using `<-` or `=`. However you should **always** avoid using the `=` for assignment to avoid issues in your code later on, such as within functions.

```
obj_one <- 3

obj_two = 3
```

We can test to see if they are the same:

```
# We use == (double equals) to test if variables/objects are the equal to each other;
# more on this later in this guide

obj_one == obj_two
```

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

1.3.3 Object

Again, R is object-oriented. Everything is an objects and have a name.

R has 6 basic data types:

- numeric (real or decimal)
- character
- integer
- date-time
- logical (TRUE/FALSE)
- complex

R has many data structures, including

- atomic vector
- matrix
- data frames
- tibbles (enhanced data frame from tidyverse)
- lists
- factors

Object names are *case sensitive* and the also *class sensitive*. There are different norms used in naming variables: usually you always use lower cases, and separate words by "_“,”." or Capital letter.

```
obj_one <- 3     # This is my preferred method
obj.one <- NA    # Using "." can lead into problems -- avoid it
objOne  <- "3"   # " " makes the value be a character
```

Using the function **str()** allow us to evaluate the class of each object

```
str(obj_one) # This is numeric
```

```
##  num 3
```

```
str(obj.one) # This is numeric
```

```
##  logi NA
```

```
str(objOne) # This is character
```

```
##  chr "3"
```

1.3.4 Vectors and Dataframes

Vectors are a matrix of either one row or one column. A vector is a combination of scalars (single numbers/values) stored as unique object. To create vectors we use the function `c()`:

```
# This is one possible vectors:

vec_one <- c(1,3,5,10)
print(vec_one)
```

```
## [1]  1  3  5 10
```

```
str(vec_one)
```

```
##  num [1:4] 1 3 5 10
```

```
# Let's create another vector combining the scalars created before
vec_two <- c(obj_one,obj.one,objOne)
str(vec_two)
```

```
##  chr [1:3] "3" NA "3"
```

Note that `obj_one` and `obj.one` were coerced to character. This happens because **Vectors contain either numbers or characters, not both**. Using vectors of same lenght we are able to use the operations as well, and this will be done element by element (row by row).

```
vec_three <- c(3,5,7,10)

# Addition
vec_one+vec_three
```

```
## [1]  4  8 12 20
```

```
# Subtraction
vec_one-vec_three
```

```
## [1] -2 -2 -2  0
```

```
#Multiplication
vec_one*vec_three
```

```
## [1]   3  15  35 100
```

```
#Division
vec_one/vec_three
```

```
## [1] 0.3333333 0.6000000 0.7142857 1.0000000
```

Data frames: are collection of vectors. Each vector is stored as one variable, with several elements. Think of dataframes as a spreadsheet in excel. To create a dataframe we can use the function `data.frame()`.

```
# For example we can combine the different vectors into a dataframe:
# First we modify vec_two so it also has 4 elements

vec_two <- c(vec_two, "a")
df <- data.frame(vec_one,vec_two,vec_three)

# The str() function shows the structure of each variable (vector)
str(df)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ vec_one  : num  1 3 5 10
##  $ vec_two  : chr  "3" NA "3" "a"
##  $ vec_three: num  3 5 7 10
```

Note that the `vec_two` is now a factor of two levels: one level is "3" and the other is "a". If we don't want to combine vectors we can create the dataframe from scratch or import a dataframe created elsewhere (most common).

```r
df_new <- data.frame(
  height_cm = c(165,150,180,175,145),
  name = c("Al","Bob","Chris","Dan","El"),
  gender = c("M","M","F","F","F"),
  age = c(20,19,21,18,20),
  weight_kg = c(65,70,85,70,50),
  stringsAsFactors = F)           # This is a good common practice
str(df_new)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ height_cm: num  165 150 180 175 145
##  $ name     : chr  "Al" "Bob" "Chris" "Dan" ...
##  $ gender   : chr  "M" "M" "F" "F" ...
##  $ age      : num  20 19 21 18 20
##  $ weight_kg: num  65 70 85 70 50
```

Indexing

Indexing in R begins at 1. Not 0 like some languages (e.g. Python). To show how the indexing in R works let's walk through some examples:

```r
### We can use [ ] to index we create
a <- 1:10
a[4] # Get the 4th element of object "a"
```

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

```r
a[c(4, 6)] ## Get the 4th and 6th elements
```

```
## [1] 4 6
```

```r
### Which also works for matrices, lists and data frames
df_new[1,1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

```
## [1] 165
```

Lists are some complex and interesting object in R. They can contain different types of objects that don't share the same structure, class or shape (e.g. it can contain scalars, strings, a data frame, and lists of lists). Therefore for lists we need to use double square brackets `([[ ]])`:

```r
my_list <- list(
  a = "hello",
  b = c(1,2,3),
  c = data.frame(x=1:5, y=6:10)
)

# Return the 1st list object
my_list[[1]]
```

```
## [1] "hello"
```

```r
# Return the 3rd element of the 2nd list object
my_list[[2]][3]
```

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

```r
# Another possible indexing operator is the $
# Let's take a look when we print the list we created
my_list
```

```
## $a
## [1] "hello"
##
## $b
## [1] 1 2 3
##
## $c
##   x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

```r
# Note for each list object is demarcated with the $ operator
# Now we can use the $ to call these objects:
# Return list object "a"
my_list$a
```

```
## [1] "hello"
```

```r
# Return the 3rd element of list object "b"
my_list$b[3]
```

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

```r
# Return column "x" of list object "c"
my_list$c$x
```

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

Indexing is very important in data manipulation, and understand how to use the operators (brackets, double brackets, dollar sign) will help moving forward, especially since they work with different data structures.

2. Data Manipulation

Although R provides several functions that allow us to manipulate data and make statistical analysis, R has several packages or libraries that make our life easier by importing new functions. These libraries/packages can be installed using `install.package()` function, and called in R using `library()` function.

From now on, instead of showing different comparable functions, I will focus on the use of the library *tidyverse* which is part of *dplyr*. For more base R functions and appraches, see Phillips (2018). The *tidyverse* library is the focus of *R for Data Science* (2017) and provides clean and intuitive ways of dealing with data. Antoher powerful library worth exploring for data analysis is *data table*, but as mentioned we will focus on *tidyverse* here.

In the section we discussed dataframes. When using *tidyverse*, instead of dealing with dataframes, we will use *tibbles*. As described by Grolemund and Wickham (2017): "tibbles are data frames, but they tweak some older behaviours to make life a little easier".

Let's load the library *tidyverse*. As you can see below, it also uploads other dependencies, i.e., other libraries it makes use of:

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

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

2.1 Import/Export

Importing: to import data, we use the **readr** library. This library loads data as tibbles. The advantage of tibbles is that they are typically faster, they don't convert character vectors to factors, use rownames, or munge the column names. Also, they are more reproduceble.

- `read_csv()`: reads comma delimited files
- `read_fwf()`: reads fixed width files
- `read_rds()`: reads R's custom binary format

If you want to import stata files you can use the library **haven**:

- `read_dta()`: reads stata files

```
### Reading data into R
## First load necessary library
library(tidyverse)
library(haven)

# csv
df_csv <- read_csv("my_csv_file.csv")

# rds
df_rds <- read_rds("my_rds_file")

# stata
df_dta <- read_dta("my_stata_file.dta")
```

Exporting: to export data you can save then as csv, fwf, dta, etc. However, by doing so you may lose some information in terms of formatting, features of data, etc. I recommend using *.rds* extension:

- `write_csv()`: write comma delimited files
- `write_fwf()`: write fixed width files
- `write_rds()`: write R's custom binary format

Using the **haven** library, you can write stata files:

- `write_dta()`: reads stata files

```
### Writing data to other files
## First load necessary library
library(tidyverse)
library(haven)

# csv
write_csv(df, "my_csv_file.csv")

# rds
write_rds(df, "my_rds_file")

# stata
write_dta(df, "my_dta_file.dta")
```

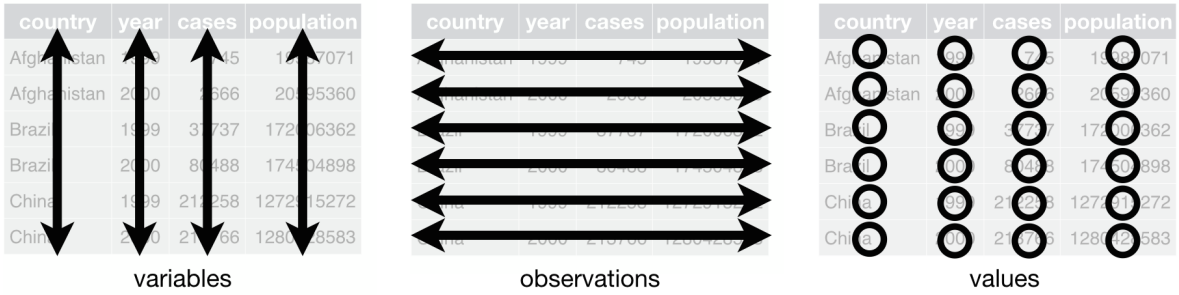
2.2 Representation of data

You can represent the same data in several ways. However, there are some rules/norms:

- Each variable must have its own column. This allows for better use of R's vectorized nature
- Each observation must have its own row
- Each value myst have its own cell

These rules lead to simples instructions:

1. Put each dataset in a tibble (dataframe)
2. Put each variable in a column



2.3 Pipes

tidyverse makes it easier for us to use the so-called **pipes**. Pipes are a powerful tool that clearly express the sequence of multiple operations. Pipes allow us to focus on the task (verb) and not on the outcome (noun). Newer versions of R have the native **pipes**, but I will rely on the `magrittr` one. Let us see some examples of how we could perform some tasks:

```
# Not using pipes
# Option one: through intermediate steps
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

This first step may create too many unnecessary (temporary) files. Another (not recommended) way can be:

```
# Not using pipes
# Option two : overwrite the original
foo_foo <- hop(foo _foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop (foo_foo, on = head)
```

Note now that using pipes, this becomes straight forward and intuitive:

```
# Using pipes
# Note that we use a pipe for each step.
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mice) %>%
  bop(on = head)
```

However, there are times we should avoid using pipes. Usually we want to avoid pipes that are too long and that have multiple inputs/outputs.

2.4 Data Manipulation/Wrangling

Set up

The first thing we want to do in R when using data is make sure that we can access our files. Therefore, it is critical that we tell R which folder it should be looking at. If you are not sure what folder (working directory) you are, you can simply type `getwd()`

```
getwd()
```

```
## [1] "/Users/amirneto/Dropbox/_FGCU/Teaching/_Guides/R"
```

Usually the working directory is the folder in which you have your R file saved. Note that you can change it using the `setwd()` function:

```
setwd("/Users/amirneto/Dropbox/_FGCU/Teaching/Undergraduate Seminar/_Guides/R")
```

To make sure we are in the correct folder we can check what files are there by using `list.files()` function

```
list.files()
```

```
##  [1] "bibfile.bib"           "codechunk.png"         "gather.png"
##  [4] "inner_join.png"        "intro_R.pdf"           "Intro_to_R.pdf"
##  [7] "Intro_to_R.Rmd"        "intro-R_data-only.pdf" "intro-R_data-only.Rmd"
## [10] "join-anti.png"         "join-one-to-many.png"  "join-semi.png"
## [13] "join-venn.png"         "join.png"              "outer_join.png"
## [16] "rStudio.png"           "separate.png"          "spread.png"
## [19] "tidy-1.png"            "unite.png"
```

For the following examples I will be using a pre-built dataset available in the *dplyr* library. Let's load this data and inspect it. In *tidyverse*, we can inspect the data by using `glimpse()` instead of `str()`, although the latter also works.

```
# Loading the pre-built data
data(starwars)

# Inspecting the data
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia...
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180...
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, ...
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown"...
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light"...
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blu...
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57....
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "femal...
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "fem...
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan",...
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "H...
## $ films      <list> [<"The Empire Strikes Back", "Revenge of the Sith", "Re...
## $ vehicles   <list> [<"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, ...
## $ starships  <list> [<"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced ...
```

This dataset has 13 variables (columns) and 87 observations (rows). The observations are the different starwars characters and the variables are different features of these characters.

2.4.1 Select & Filter

`select()` and `filter()` are two functions used to slice our data.`select()` is used to, as the name suggests, select the variables we want to keep in our dataset. This function, thus, affects only columns.

```
# For example, let's say we want to keep only name, height, mass and species variables:

df_sw <- starwars %>%              # assigning transformation of original to new dataset
  dplyr::select(name,height,mass,species)  # funtion + name of variables. Note here that
                                   # I used dplyr:: , this is used to make sure that the
                                   # select function comes from the dplyr package and not
```

```
                                    # some other package loaded into R

glimpse(df_sw)
```

```
## Rows: 87
## Columns: 4
## $ name    <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or...
## $ height  <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2...
## $ mass    <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77....
## $ species <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma...
```

`filter()`, on the other hand, is used to select observsations given some criteria. To filter variables selecting a subset of them we need to use some logical evaluation. Below there is a list of commands used for filtering

Command	Explanation
==	equal
!=	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
|	or (any condition)
!	not
&	and (all conditions)
%in%	in the set

Let me open a parenthesis in the logical operators above. Evaluating floating-points can be tricky. The reason comes from the way computers work. To get around this issue you can the function `all.equal()`

```
### all.equal()
# Let's take a look at a simple example:
0.1+0.2 == 0.3
```

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

```
# Now let's use all.equal
all.equal(0.1+0.2,0.3)
```

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

Now, given the logical operators above, we can work on subsetting our data.

```
# For example, let's say we want to further slice our starwars dataset and keep only
# humans with their name, height mass mass

df_sw_h <- df_sw %>%                  # assigning transformation of original to new dataset
  dplyr::filter(species == "Human")   # funtion + criteria ; Note that here we use the
                                      # == which evaluates whether or not the values
                                      # are the same

glimpse(df_sw_h)
```

```
## Rows: 35
## Columns: 4
## $ name    <chr> "Luke Skywalker", "Darth Vader", "Leia Organa", "Owen Lars"...
## $ height  <int> 172, 202, 150, 178, 165, 183, 182, 188, 180, 180, 170, 180,...
## $ mass    <dbl> 77.0, 136.0, 49.0, 120.0, 75.0, 84.0, 77.0, 84.0, NA, 80.0,...
## $ species <chr> "Human", "Human", "Human", "Human", "Human", "Human", "Huma...
```

2.4.2 Mutate & Summarise

Mutate

Most likely you will want to transform your data by creating new variables. Again, it is easy to see that we will use the function `mutate()` to transform our variables. Note that if we type the name of the dataframe it prints the data and it looks like a table. It shows only the 10 first observations (rows) and hides the remaining ones.

```
# For example, let's say I want to calculate the Body Mass Index (BMI) for the human
# starwars characters

df_sw_h <- df_sw_h %>%  # assigning transformation of original to new dataset
  dplyr::mutate(height_m = height/100,      # creates height in meters
                bmi = mass / (height_m)^2)  # creates BMI

df_sw_h
```

```
## # A tibble: 35 x 6
##    name               height  mass species height_m   bmi
##    <chr>               <int> <dbl> <chr>      <dbl> <dbl>
##  1 Luke Skywalker        172    77 Human       1.72  26.0
##  2 Darth Vader           202   136 Human       2.02  33.3
##  3 Leia Organa           150    49 Human       1.5   21.8
##  4 Owen Lars             178   120 Human       1.78  37.9
##  5 Beru Whitesun lars    165    75 Human       1.65  27.5
##  6 Biggs Darklighter     183    84 Human       1.83  25.1
##  7 Obi-Wan Kenobi        182    77 Human       1.82  23.2
##  8 Anakin Skywalker      188    84 Human       1.88  23.8
##  9 Wilhuff Tarkin        180    NA Human       1.8   NA
## 10 Han Solo              180    80 Human       1.8   24.7
## # ... with 25 more rows
```

Dummy Variables: are indicator varibles that take the value of 1 or 0 that represent some criteria. Dummy variables are very useful in data analysis. There are at least two ways you can create a dummy variable: one way is to use the function `as.numeric()`, which returns logican (TRUE/FALSE) as numerical 1 or 0. The other way is to use the `if_else()` function which evaluates a statament and return values to it.

```
# For example, let's say we want to create two dummy variables: one for Humans, and
# another one for Humans who live in Tatooine

df_d <- starwars %>%
  dplyr::mutate(dummy1 = as.numeric(species=="Human"),  # create dummy base on species
                dummy2 = if_else(homeworld =="Tatooine" & dummy1==1, # must meet two
                                                                     # conditiona
                                 1,                                  # what returns if T
```

```
                                0)) %>%                                 # what returns if F
  dplyr::select(name,species,homeworld,dummy1,dummy2)

df_d
```

```
## # A tibble: 87 x 5
##    name               species homeworld dummy1 dummy2
##    <chr>              <chr>   <chr>      <dbl>  <dbl>
##  1 Luke Skywalker     Human   Tatooine       1      1
##  2 C-3PO              Droid   Tatooine       0      0
##  3 R2-D2              Droid   Naboo          0      0
##  4 Darth Vader        Human   Tatooine       1      1
##  5 Leia Organa        Human   Alderaan       1      0
##  6 Owen Lars          Human   Tatooine       1      1
##  7 Beru Whitesun lars Human   Tatooine       1      1
##  8 R5-D4              Droid   Tatooine       0      0
##  9 Biggs Darklighter  Human   Tatooine       1      1
## 10 Obi-Wan Kenobi     Human   Stewjon        1      0
## # ... with 77 more rows
```

Summarise

`summarise()` is used to collapses a dataframe to a single row. Usually we combine it with `group_by()`. This is very useful to create new dataframes that are aggregate observations. We can also use it to create summary statistics

```
# In this example I will not assign the data manipulation to a new dataframe
# As you can see it only prints the calculation and nothing is saved to the
# manipulated dataframe

df_d %>%
  summarise(
    d_mean = mean(dummy1, na.rm = T),      # average
    d_sd = sd(dummy1, na.rm = T),          # standard deviation
    d_median = median(dummy1, na.rm = T), # median
    d_min = min(dummy1, na.rm = T),        # minimum
    d_max = max(dummy1, na.rm = T),        # maximum
    d_n   = n(),                           # count
    d_sum = sum(dummy1, na.rm = T),        # sum of all heights
                                           # na.rm = T :: removes NA from calculation
  )
```

```
## # A tibble: 1 x 7
##   d_mean  d_sd d_median d_min d_max   d_n d_sum
##    <dbl> <dbl>    <dbl> <dbl> <dbl> <int> <dbl>
## 1  0.422 0.497        0     0     1    87    35
```

Summary Statistics: the easiest way to do it is to use the function `summary()`. However, the library *stargazer* allow us to print summary statistics table much easier as does the library *modelsummary*. Summary statistics are for numerical variables only.

```r
# Using the summary() we get this information
summary(df_d)
```

```
##      name              species           homeworld            dummy1
##  Length:87          Length:87          Length:87          Min.   :0.0000
##  Class :character   Class :character   Class :character   1st Qu.:0.0000
##  Mode  :character   Mode  :character   Mode  :character   Median :0.0000
##                                                           Mean   :0.4217
##                                                           3rd Qu.:1.0000
##                                                           Max.   :1.0000
##                                                           NA's   :4
##      dummy2
##  Min.   :0.00000
##  1st Qu.:0.00000
##  Median :0.00000
##  Mean   :0.09877
##  3rd Qu.:0.00000
##  Max.   :1.00000
##  NA's   :6
```

But a more friendly and aesthetically way is using stargazer. Note that stargazer does not recognize tibbles, so we must convert it into as dataframe using `as.data.frame()`

```r
# Here we use the stargazer() library
library(stargazer)
```

```
##
## Please cite as:

##  Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.

##  R package version 5.2.2. https://CRAN.R-project.org/package=stargazer
```

```r
# I add: omit.summary.stat = c("p25","p75") to omit percentiles 25 and 75
# from the list of statistics presented.
# I also add yype = "text" so the outcome is the one you see below.
# The default outcome is "latex" which makes it super easy to bring it
# to our manuscript
stargazer(as.data.frame(df_d), omit.summary.stat = c("p25","p75"), type = "text")
```

```
##
## ========================================
## Statistic N  Mean  St. Dev.  Min   Max
## ----------------------------------------
## dummy1    83 0.422  0.497   0.000 1.000
## dummy2    81 0.099  0.300   0.000 1.000
## ----------------------------------------
```

The recommended package will be `modelsummary` that allows for more exploratory analysis of data and results, producing nice tables easy to export as figure or to a TeX file. you can learn more on this website (link)

	Unique (#)	Missing (%)	Mean	SD	Min	Median	Max
dummy1	3	5	0.4	0.5	0.0	0.0	1.0
dummy2	3	7	0.1	0.3	0.0	0.0	1.0

```
library(modelsummary)

# for summary of the numeric variables in the dataset:
datasummary_skim(df_d,
                 histogram = F) # removes histogram from table
```

2.4.3 Group_by

As the name suggests, `group_by()` groups the variables by certain groups. This allows us to perform some analysis/calculation for these specific groups.

```
# For example, lets get the average height and mass for each gender
# combining group_by and summarise

starwars %>%
  group_by(gender) %>%
  summarise(
    height_mean = mean(height, na.rm = T),  # average
    mass_mean = sd(mass, na.rm = T),        # standard deviation
    n   = n()                               # count
  )
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 4
##   gender    height_mean mass_mean     n
##   <chr>           <dbl>     <dbl> <int>
## 1 feminine         165.      8.59    17
## 2 masculine        177.    185.      66
## 3 <NA>             181.     NA        4
```

This is one of my most used commands when cleaning and manipulating data and can be very powerful when combined with other functions like summarise, mutate, and filter.

2.4.4 Working with multiple objects

When working in research is very likely that you will need to use multiple databases. Thus, it is important to understand how to bring these different databases together.

Appending Data

Two important functions when using *data.frames* and *matrices* are the `cbind()` and `rbind()`. As the names suggest these functions combine vector or matrices together, with cbind combining them as columns and rbind as rows. The vectors being combined **must** have the same lenght!

```r
# Let us create some data sets. Let's suppose we survey 5 people:
df1 <- data.frame("index" = c(1,2,3,4,5),
                  "sex" = c("m","m","m","f","f"),
                  "age" = c(35,48,20,18,59),
                  "height" = c(178,180,160,155,165))

# Now say, we are able to interview three additional people
df2 <- data.frame("index" = c(6,7,8),
                  "sex" = c("m","f","f"),
                  "age" = c(80,30,25),
                  "height" = c(170,180,153))

# Let's bring these together. The columns are the same (variables), we just want to
#  add more rows (observations)

svy_df <- rbind(df1,df2)
glimpse(svy_df)
```

```
## Rows: 8
## Columns: 4
## $ index  <dbl> 1, 2, 3, 4, 5, 6, 7, 8
## $ sex    <chr> "m", "m", "m", "f", "f", "m", "f", "f"
## $ age    <dbl> 35, 48, 20, 18, 59, 80, 30, 25
## $ height <dbl> 178, 180, 160, 155, 165, 170, 180, 153
```

```r
# After some time we reach out to all of them and collect additional info
df3 <- data.frame("index" =  c(1,2,3,4,5,6,7,8),
                  "weight" = c(80,70,88,55,95,70,100,60))

# We are adding more variables (columns) to all observations (rows)

svy_df <- cbind(svy_df, df3)
glimpse(svy_df)
```

```
## Rows: 8
## Columns: 6
## $ index  <dbl> 1, 2, 3, 4, 5, 6, 7, 8
## $ sex    <chr> "m", "m", "m", "f", "f", "m", "f", "f"
## $ age    <dbl> 35, 48, 20, 18, 59, 80, 30, 25
## $ height <dbl> 178, 180, 160, 155, 165, 170, 180, 153
## $ index  <dbl> 1, 2, 3, 4, 5, 6, 7, 8
## $ weight <dbl> 80, 70, 88, 55, 95, 70, 100, 60
```

Note: if you are using rbind and cbind to append data, you must make sure that the order of variables (columns) and/or observations (rows) match!

Merging Data

Before we start mergin data we need to understand what a key is. Keys are variables used to connect pair of tables. A **primary key** *uniquely* identifies an observation in its own table and a **foregin key** *uniquely* identifies an observation in another table. Some times you will work with duplicate keys, but thinking about your primary key is very important. For more on this check (Grolemund and Wickham 2017).

```
#Let's think about two dataframes.

# Data 1      Data 2
# key var_x   key var_y
# 1   x1      1   y1
# 2   x2      2   y2
# 3   x3      4   y4
```

Basically there are four two of merging: inner and outer joins. The outer join can be broken into left, right and full. All joins will contain the key and values of x and y associated with the key

- Inner: keeps obs. in both Data 1 and Data 2 (1 and 2)
- Left: keeps all obs. in Data 1 (1,2 and 3) , (most common)
- Right: keeps all obs. in Data 2 (1,2 and 4)
- Full: keep all obs in Data 1 and Data 2 (1,2,3,4)

```
data1 <- tibble(key = c(1,2,3),
                var_x = c("x1","x2","x3"))
print(data1)
```

```
## # A tibble: 3 x 2
##     key var_x
##   <dbl> <chr>
## 1     1 x1
## 2     2 x2
## 3     3 x3
```

```
data2 <- tibble(key = c(1,2,4),
                var_y = c("y1","y2","y3"))
data2
```

```
## # A tibble: 3 x 2
##     key var_y
##   <dbl> <chr>
## 1     1 y1
## 2     2 y2
## 3     4 y3
```

```
# inner join
data1 %>%
  inner_join(data2, by = "key")
```

```
## # A tibble: 2 x 3
##     key var_x var_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

```
# left join
data1 %>%
  left_join(data2, by = "key")
```

```
## # A tibble: 3 x 3
##     key var_x var_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
```

```
# right join
data1 %>%
  right_join(data2, by = "key")
```

```
## # A tibble: 3 x 3
##     key var_x var_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     4 <NA>  y3
```

```
# full join
data1 %>%
  full_join(data2, by = "key")
```

```
## # A tibble: 4 x 3
##     key var_x var_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
```

2.4.5 Reshaping data

Some times we need to manipulate data, especially when we have longitudinal data (same units over time), to change what is considered a variable and what is considered an observation. This is called *reshaping*. For example we may want to have the year as an observation, or else, as new variables. Let's illustrate this:

```
## emp = avg employment and wage = avg wage from QCEW

# - Wide Data -
#
# county    emp_2018  emp_2019  wage_2018  wage_2019
# lee       220516    226803    42959      44486
# collier   134144    137774    48021      49547
# hendry    9720      9647      39103      40342
# ...

# - Long Data -
#
# county    year  emp     wage
# lee       2018  220516  42959
# lee       2019  226803  44486
# collier   2018  134144  48021
```

```
# collier   2019  137774  49547
# hendry    2018  9720    39103
# hendry    2019  9647    40342
# ...
```

To make these changes we need to use `pivot_wider()` and `pivot_longer()`. `pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`.

```
# For this example let's start saying we download individual data files from QCEW for
# counties in FL. (Here I am typing it in to make it easier)

df_2018 <- tibble(
  county = c("lee","collier","hendry"),
  emp = c(220516,134144,9720),
  wage = c(42959,48021,39103),
  year = c(2018,2018,2018)
)

df_2019 <- tibble(
  county = c("lee","collier","hendry"),
  emp = c(226803,137774,9647),
  wage = c(44486,49547,40342),
  year = c(2019,2019,2019)
)

# First we cshoud combine both objects. We can do that using rbind() or full_join()
# In this case, since variables (columns) are the same, we will just append the
# datasets, thus row-binding (rbind) them.

df <- rbind(df_2018,df_2019)
df
```

```
## # A tibble: 6 x 4
##   county     emp  wage  year
##   <chr>    <dbl> <dbl> <dbl>
## 1 lee     220516 42959  2018
## 2 collier 134144 48021  2018
## 3 hendry    9720 39103  2018
## 4 lee     226803 44486  2019
## 5 collier 137774 49547  2019
## 6 hendry    9647 40342  2019
```

```
# df here is the "Long" format. Let's transform it to "Wide". To do this we need
# first to convert columns in rows -- so to use pivot_longer()
dfw_1 <- df %>%
  pivot_longer(cols = c("emp", "wage"),  # columns we want to pivot
               names_to = "variable",    # variable we will store the names
               values_to = "value")      # variable we will store the values
dfw_1
```

```
## # A tibble: 12 x 4
```

```
##    county   year variable  value
##    <chr>   <dbl> <chr>     <dbl>
##  1 lee      2018 emp      220516
##  2 lee      2018 wage      42959
##  3 collier  2018 emp      134144
##  4 collier  2018 wage      48021
##  5 hendry   2018 emp        9720
##  6 hendry   2018 wage      39103
##  7 lee      2019 emp      226803
##  8 lee      2019 wage      44486
##  9 collier  2019 emp      137774
## 10 collier  2019 wage      49547
## 11 hendry   2019 emp        9647
## 12 hendry   2019 wage      40342
```

```
# Now we need to combine year and variables
dfw_2 <- dfw_1 %>%
  mutate(variable = paste(variable, year, sep = "_")) # paste combines variables, and sep
                                                      # how we separate the values being
                                                      # combined
dfw_2
```

```
## # A tibble: 12 x 4
##    county   year variable   value
##    <chr>   <dbl> <chr>      <dbl>
##  1 lee      2018 emp_2018  220516
##  2 lee      2018 wage_2018  42959
##  3 collier  2018 emp_2018  134144
##  4 collier  2018 wage_2018  48021
##  5 hendry   2018 emp_2018    9720
##  6 hendry   2018 wage_2018  39103
##  7 lee      2019 emp_2019  226803
##  8 lee      2019 wage_2019  44486
##  9 collier  2019 emp_2019  137774
## 10 collier  2019 wage_2019  49547
## 11 hendry   2019 emp_2019    9647
## 12 hendry   2019 wage_2019  40342
```

```
# Now we transform the variables intro columns so we need to use pivot_wider()

df_wide <- dfw_2 %>%
  pivot_wider(names_from = "variable", # where variables name come from
              values_from = "value",   # where variables values come from
              id_cols = "county")
df_wide
```

```
## # A tibble: 3 x 5
##   county  emp_2018 wage_2018 emp_2019 wage_2019
##   <chr>      <dbl>     <dbl>    <dbl>     <dbl>
## 1 lee       220516     42959   226803     44486
## 2 collier   134144     48021   137774     49547
## 3 hendry      9720     39103     9647     40342
```

```r
# Now let's do the opposite and go back to "Long" format

df_long <- df_wide %>%
  pivot_longer(cols = 2:5,                  # columns we want to pivot(could be names too)
               names_to = "variable",       # where we store names
               values_to = "value") %>%     # where we store values
  mutate(year = as.numeric(stringr::str_sub(variable, -4,-1)), # get the year
         variable = str_remove_all(variable, "_|[0-9]")) %>%   # remove "_yyyy" from variable
  pivot_wider(id_cols = c("county","year"), # id columns
              names_from = "variable",      # where variables name come from
              values_from = "value")        # where variables values come from
```

3. Visualization

R provides great aesthetics for visualization of data such as tables and graphs. In this part we will mainly focus in graphs, and leave tables for when we are printing our statistical analysis. One of the packages that provides lots of flexibility and great results is **ggplot2**.

According to RStudio "ggplot2 is based on the grammar of graphics, the idea that you can build every graph from the same few components: a data set, a set of geoms—visual marks that represent data points, and a coordinate system." One of the key as pointed out by Andy Grogan-Kaylor one of the main reasons ggplot2 is worth learning is that, because of its syntax "[it] forces you to think about the nature of your data, and the ideas that you are graphing."

Below I overview the synthax and go over some of the most used types of graphs. However, I encourage you to look at data-to-viz website for extra examples and resources.

Synthax

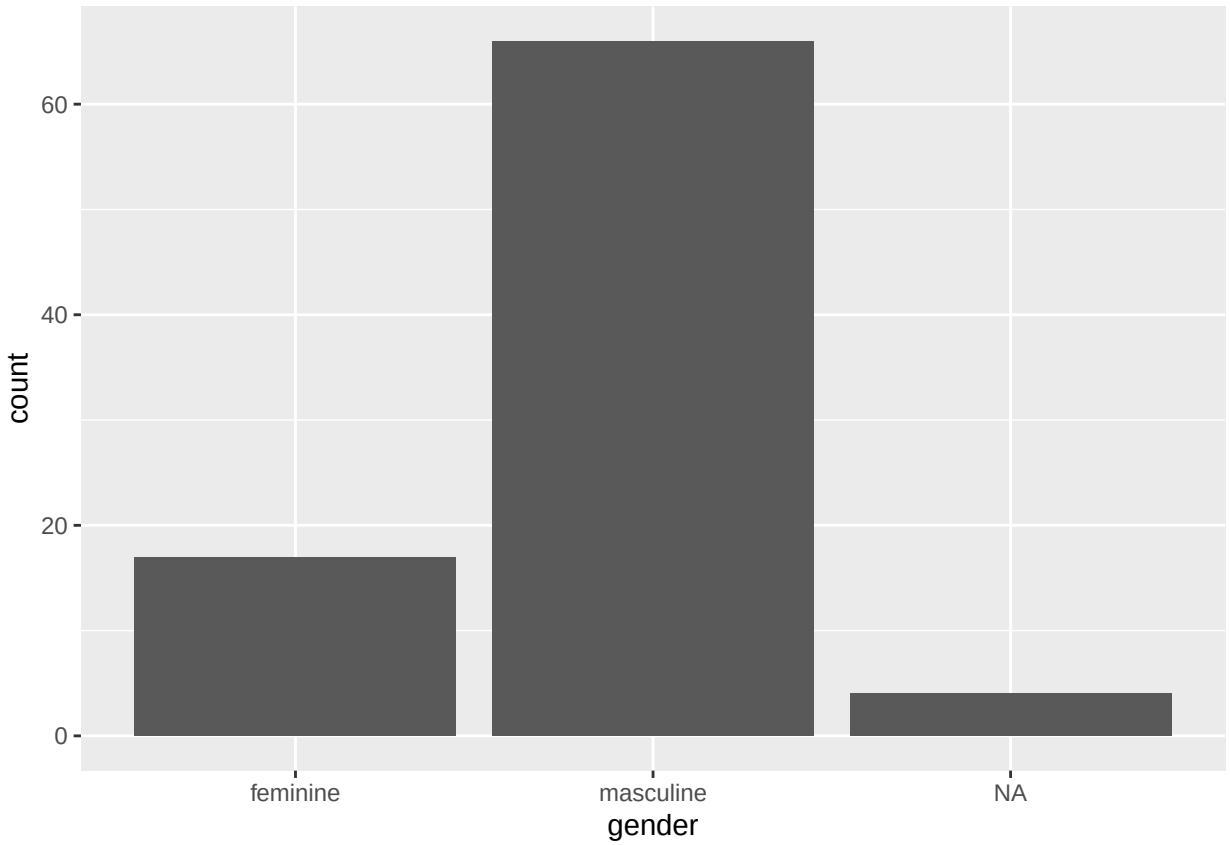
There are three components to every graph: data, coordinate and visual marks

```r
library(ggplot2)

# We start by creating the plot with data and coordinate system (variables)

ggplot(my_data,                 # data in ggplot
       aes(x = variable_x,      # coordinate system (variables)
           y = variable_y)) +   # the "+" indicates we will add geoms
                                # geom are layers added to the plot to
       geom_                    # choose data reprensentation, aesthetic properties, etc.
```

Bar Graph

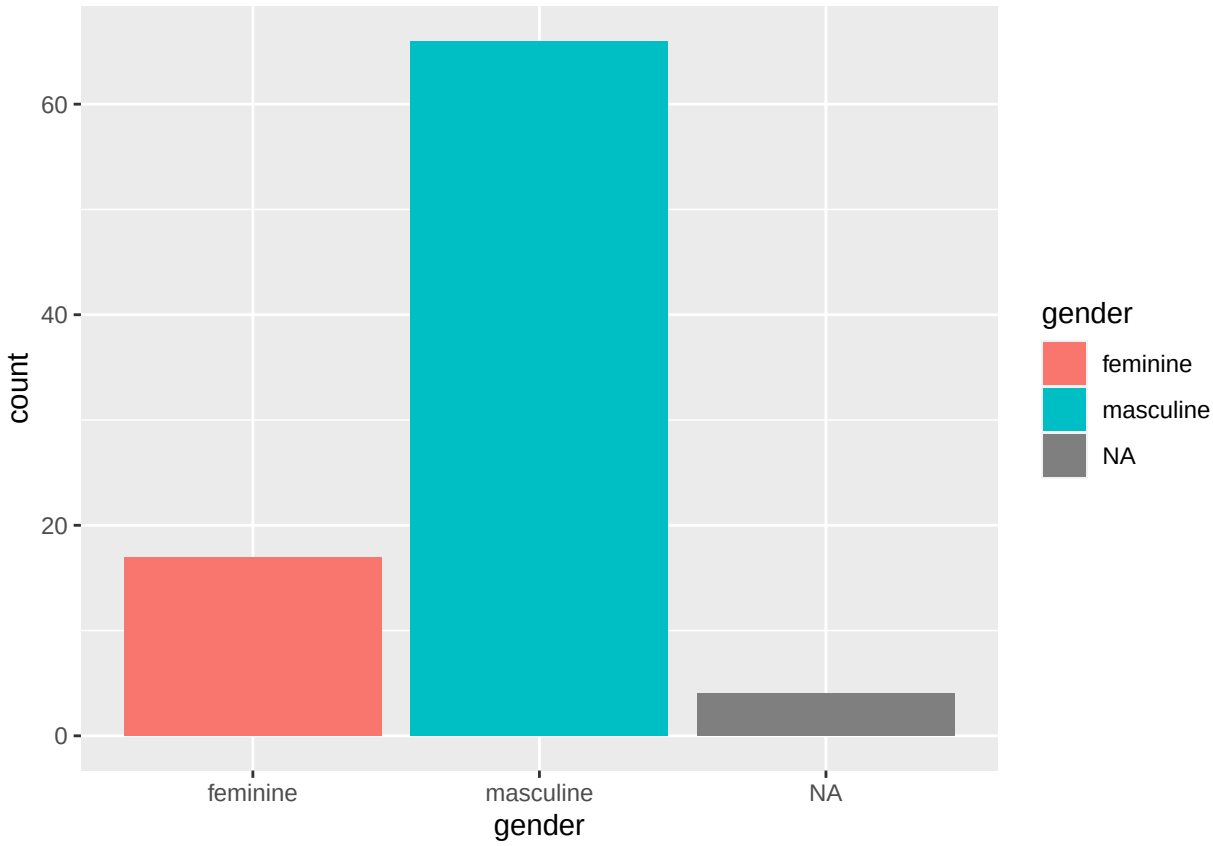
Bar plots show the relationship between a numeric and a categoric variable (gender, country, yes/no). The size of bar represent a bar and it's size represents the numeric value.

```r
# Example: gender of starwars character

ggplot(starwars,              # data
       aes(x = gender)) +     # categorical variable of interest
  geom_bar()                  # data representation type
```



In this example, we see that the colors of all bars are the same. We can change the *aesthetic* (aes) to add colors and distinguish the groups:

```
# Example: gender of starwars character

ggplot(starwars,              # data
       aes(x = gender,        # categorical variable of interest
           fill = gender)) + # category to distiguish
  geom_bar()                  # data representation type
```

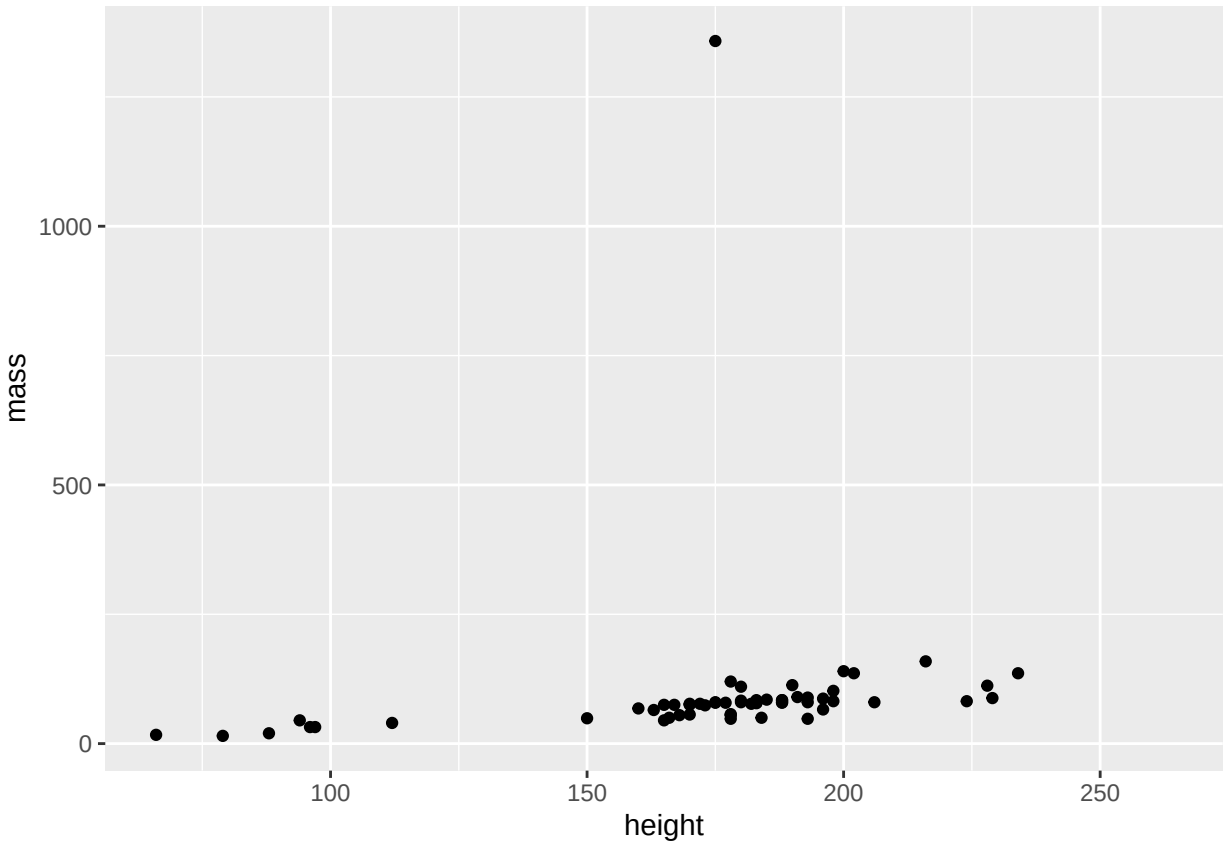


Scatter Plot

A scatter plot shows the relationship between two numeric variables. For each point.

```
# Example: height and weight of starwars characters

ggplot(starwars,              # data
       aes(x = height,        # variable 1
           y = mass)) +     # variable 2
  geom_point()                  # data representation type
```

```
## Warning: Removed 28 rows containing missing values (geom_point).
```



It is a good practice to add a linear trend to help the visualization of patterns. In this example it is clear there is an outlier, which could be problematic in our analysis.

```
# Example: height and weight of starwars characters

plot1 <- ggplot(starwars,         # data
                aes(x = height,   # variable 1
                    y = mass)) +  # variable 2
          geom_point()         +  # data representation type
          geom_smooth(method=lm , color="red", se=FALSE) # trend
          # se = confidence interval

plot2 <- ggplot(starwars %>% filter( mass<500), # data w/o outlier
                aes(x = height,   # variable 1
                    y = mass)) +  # variable 2
          geom_point()         +  # data representation type
          geom_smooth(method=lm , color="red", se=TRUE) # trend
          # se = confidence interval

# install.packages("gridExtra")
library(gridExtra)
```

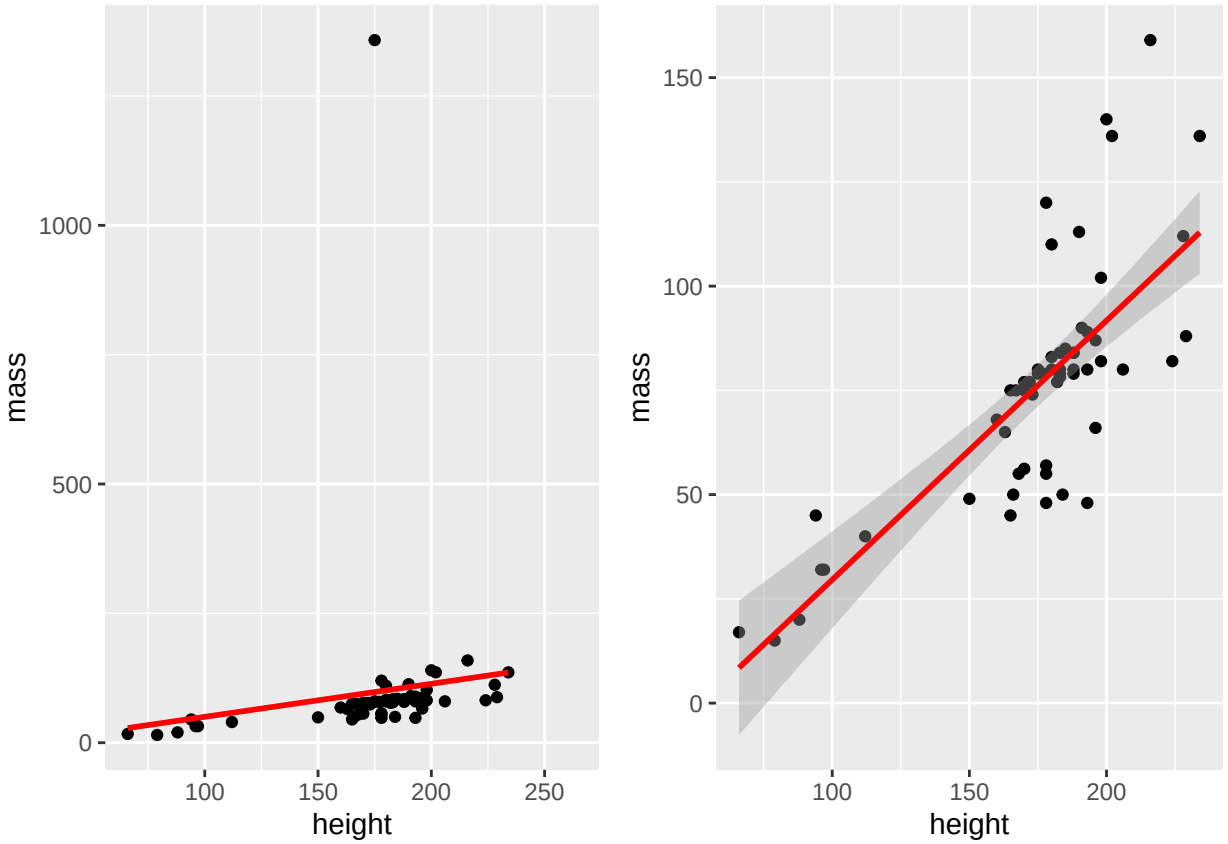
```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
```

```
##
##     combine
```

```
grid.arrange(plot1, plot2, ncol=2)
```

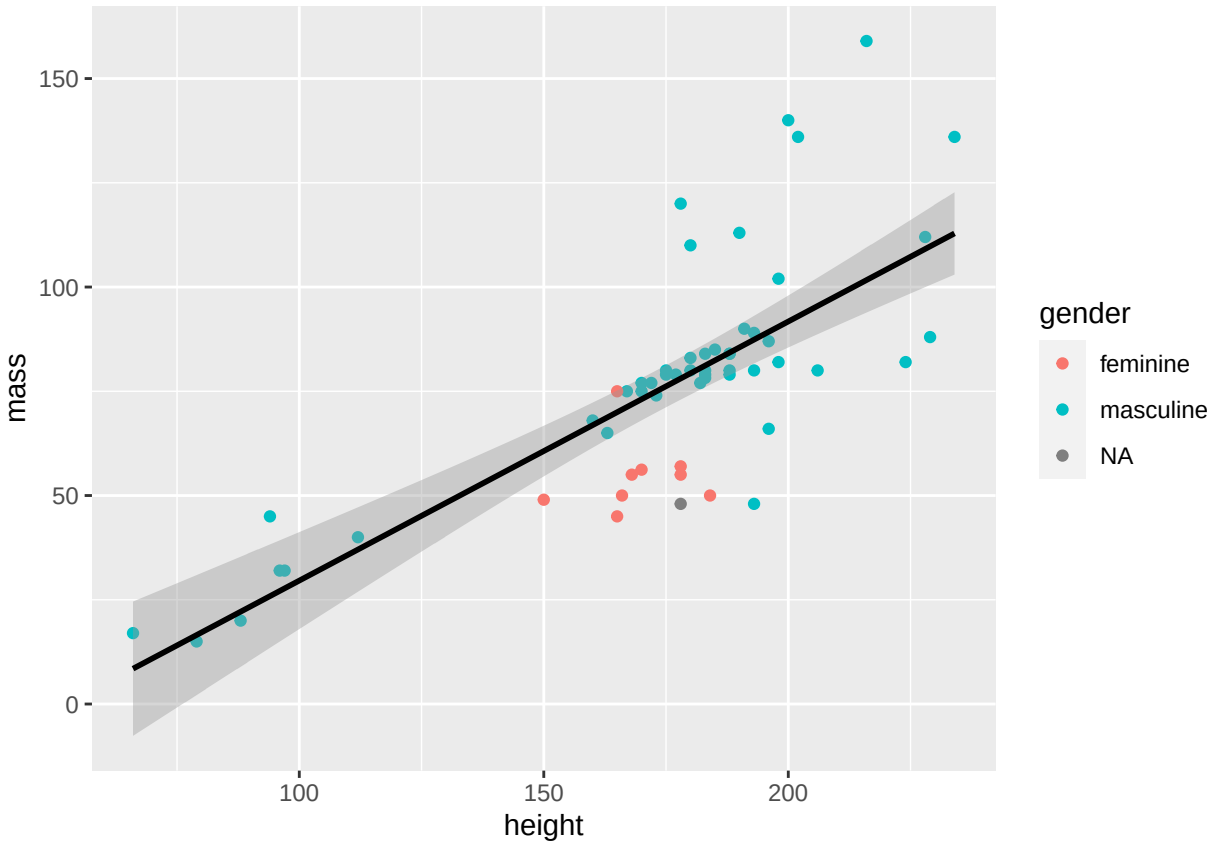
```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 28 rows containing non-finite values (stat_smooth).

## Warning: Removed 28 rows containing missing values (geom_point).

## `geom_smooth()` using formula 'y ~ x'
```



Here we can also differentiate categories if wanted.

```
# Example: height and weight of starwars characters

ggplot(starwars %>% filter( mass<500), # data w/o outlier
                aes(x = height,    # variable 1
                    y = mass,      # variable 2
                    color = gender ))  + # category
          geom_point()                  + # data representation type
          geom_smooth(method=lm , color="black", se=TRUE) # trend
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Histogram / Density

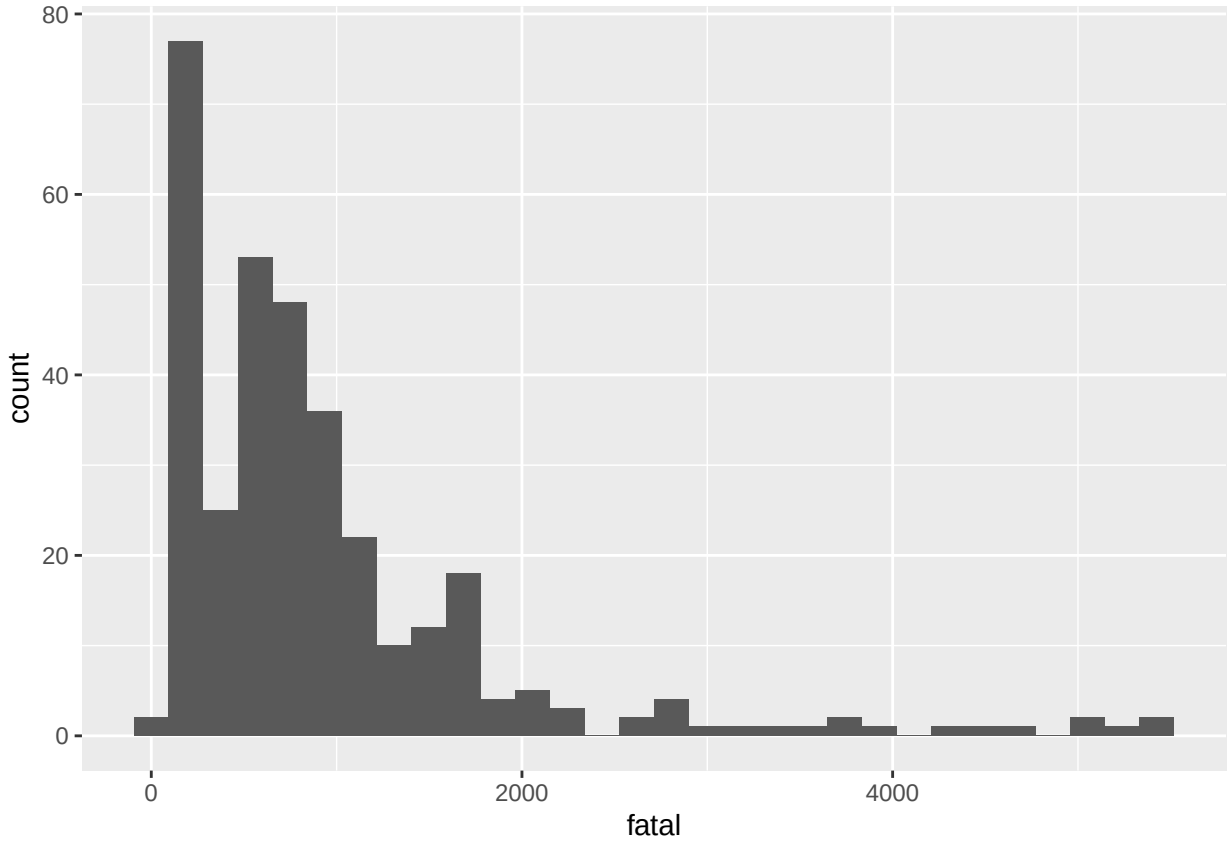
Histogram and Density graphs are used to show the distribution of a numeric variable. Histogram breaks the numeric variable into bins in which the bin's height show the number of observations in it. Let's start with **histograms**:

```
# For this example we will use another dataset. The package AER (Applied Econometrics
# with R) has some interesting functions as well as a rich number of databases

# For this example we will use Fatalities data, which contains, US traffic fatalities
# panel data for the "lower 48" US states (i.e., excluding Alaska and Hawaii), annually
# for 1982 through 1988.

# install.package(AER)
library(AER)
data("Fatalities")

# First we check the dataset
glimpse(Fatalities)
```

```
## Rows: 336
## Columns: 34
## $ state        <fct> al, al, al, al, al, al, al, az, az, az, az, az, az, az...
## $ year         <fct> 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1982, 1983, ...
## $ spirits      <dbl> 1.37, 1.36, 1.32, 1.28, 1.23, 1.18, 1.17, 1.97, 1.90, ...
## $ unemp        <dbl> 14.4, 13.7, 11.1, 8.9, 9.8, 7.8, 7.2, 9.9, 9.1, 5.0, 6...
```

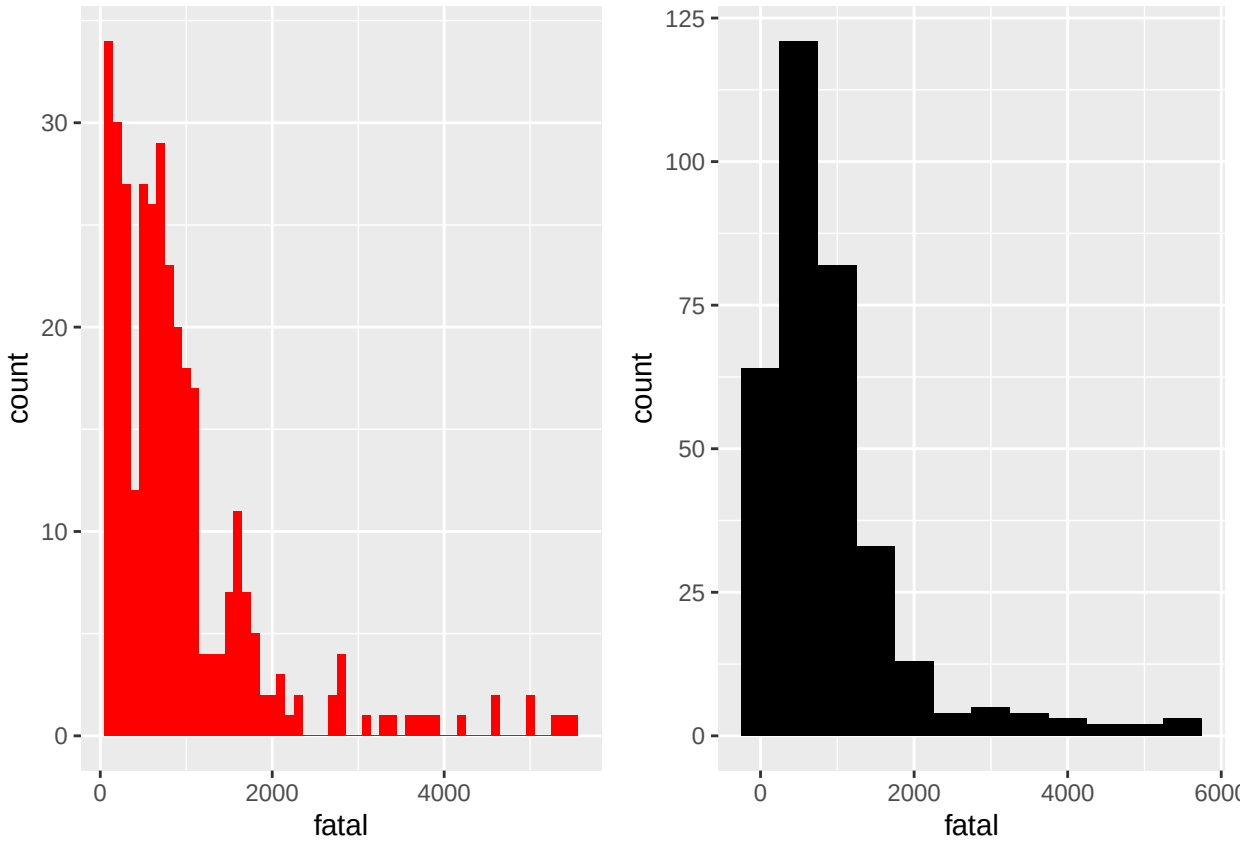
```
## $ income       <dbl> 10544.15, 10732.80, 11108.79, 11332.63, 11661.51, 1194...
## $ emppop       <dbl> 50.69204, 52.14703, 54.16809, 55.27114, 56.51450, 57.5...
## $ beertax      <dbl> 1.53937948, 1.78899074, 1.71428561, 1.65254235, 1.6099...
## $ baptist      <dbl> 30.3557, 30.3336, 30.3115, 30.2895, 30.2674, 30.2453, ...
## $ mormon       <dbl> 0.32829, 0.34341, 0.35924, 0.37579, 0.39311, 0.41123, ...
## $ drinkage     <dbl> 19.00, 19.00, 19.00, 19.67, 21.00, 21.00, 21.00, 19.00...
## $ dry          <dbl> 25.0063, 22.9942, 24.0426, 23.6339, 23.4647, 23.7924, ...
## $ youngdrivers <dbl> 0.211572, 0.210768, 0.211484, 0.211140, 0.213400, 0.21...
## $ miles        <dbl> 7233.887, 7836.348, 8262.990, 8726.917, 8952.854, 9166...
## $ breath       <fct> no, no, no, no, no, no, no, no, no, no, no, no, no, no...
## $ jail         <fct> no, no, no, no, no, no, no, yes, yes, yes, yes, yes, y...
## $ service      <fct> no, no, no, no, no, no, no, yes, yes, yes, yes, yes, y...
## $ fatal        <int> 839, 930, 932, 882, 1081, 1110, 1023, 724, 675, 869, 8...
## $ nfatal       <int> 146, 154, 165, 146, 172, 181, 139, 131, 112, 149, 150,...
## $ sfatal       <int> 99, 98, 94, 98, 119, 114, 89, 76, 60, 81, 75, 85, 87, ...
## $ fatal1517    <int> 53, 71, 49, 66, 82, 94, 66, 40, 40, 51, 48, 72, 50, 54...
## $ nfatal1517   <int> 9, 8, 7, 9, 10, 11, 8, 7, 7, 8, 11, 19, 16, 14, 5, 2, ...
## $ fatal1820    <int> 99, 108, 103, 100, 120, 127, 105, 81, 83, 118, 100, 10...
## $ nfatal1820   <int> 34, 26, 25, 23, 23, 31, 24, 16, 19, 34, 26, 30, 25, 14...
## $ fatal2124    <int> 120, 124, 118, 114, 119, 138, 123, 96, 80, 123, 121, 1...
## $ nfatal2124   <int> 32, 35, 34, 45, 29, 30, 25, 36, 17, 33, 30, 25, 34, 31...
## $ afatal       <dbl> 309.438, 341.834, 304.872, 276.742, 360.716, 368.421, ...
## $ pop          <dbl> 3942002, 3960008, 3988992, 4021008, 4049994, 4082999, ...
## $ pop1517      <dbl> 208999.6, 202000.1, 197000.0, 194999.7, 203999.9, 2049...
## $ pop1820      <dbl> 221553.4, 219125.5, 216724.1, 214349.0, 212000.0, 2089...
## $ pop2124      <dbl> 290000.1, 290000.2, 288000.2, 284000.3, 263000.3, 2589...
## $ milestot     <dbl> 28516, 31032, 32961, 35091, 36259, 37426, 39684, 19729...
## $ unempus      <dbl> 9.7, 9.6, 7.5, 7.2, 7.0, 6.2, 5.5, 9.7, 9.6, 7.5, 7.2,...
## $ emppopus     <dbl> 57.8, 57.9, 59.5, 60.1, 60.7, 61.5, 62.3, 57.8, 57.9, ...
## $ gsp          <dbl> -0.022124760, 0.046558253, 0.062797837, 0.027489973, 0...
```

```r
# Histogram
ggplot(Fatalities,          # data
       aes(x = fatal)) +    # numeric variable
  geom_histogram()          # data representation type
```



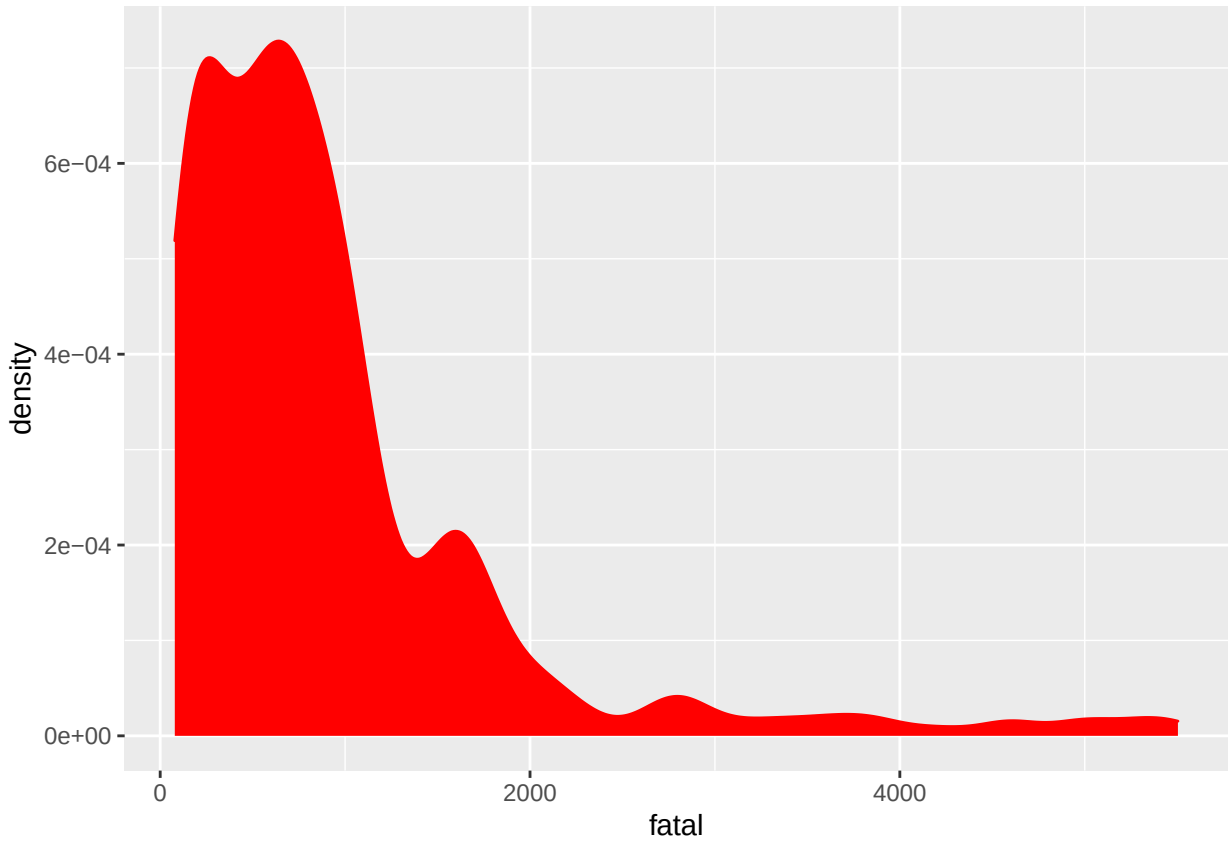
You can also change the bin size

```
# Example: height and weight of starwars characters

plot3 <- ggplot(Fatalities,          # data
                aes(x = fatal)) +  # numeric variable
                                   # data representation type
         geom_histogram(binwidth = 100,     # bin size
                        fill = "red")    # color

plot4 <- ggplot(Fatalities,          # data
                aes(x = fatal)) +  # numeric variable
                                   # data representation type
         geom_histogram(binwidth = 500,    # bin size
                        fill = "black")   # color

# install.packages("gridExtra")
# library(gridExtra)
grid.arrange(plot3, plot4, ncol=2)
```



Let's show the example of a **Density** graph now.

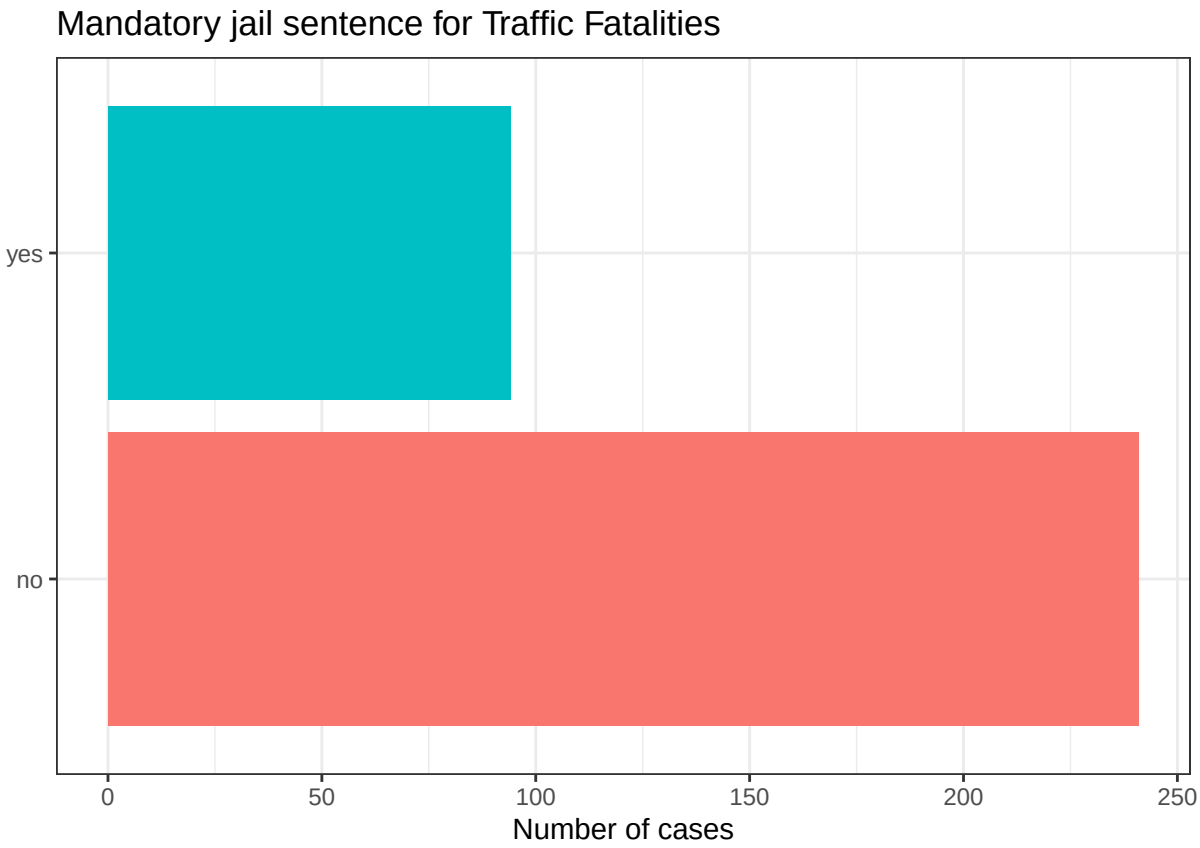
```
# Using the same dataset as for the Histograms:

ggplot(Fatalities,          # data
       aes(x = fatal)) +    # numeric variable
                            # data representation type
  geom_density(fill = "red",   # area color
               color = "red") # line color
```



Themes and Extra Features of the graph

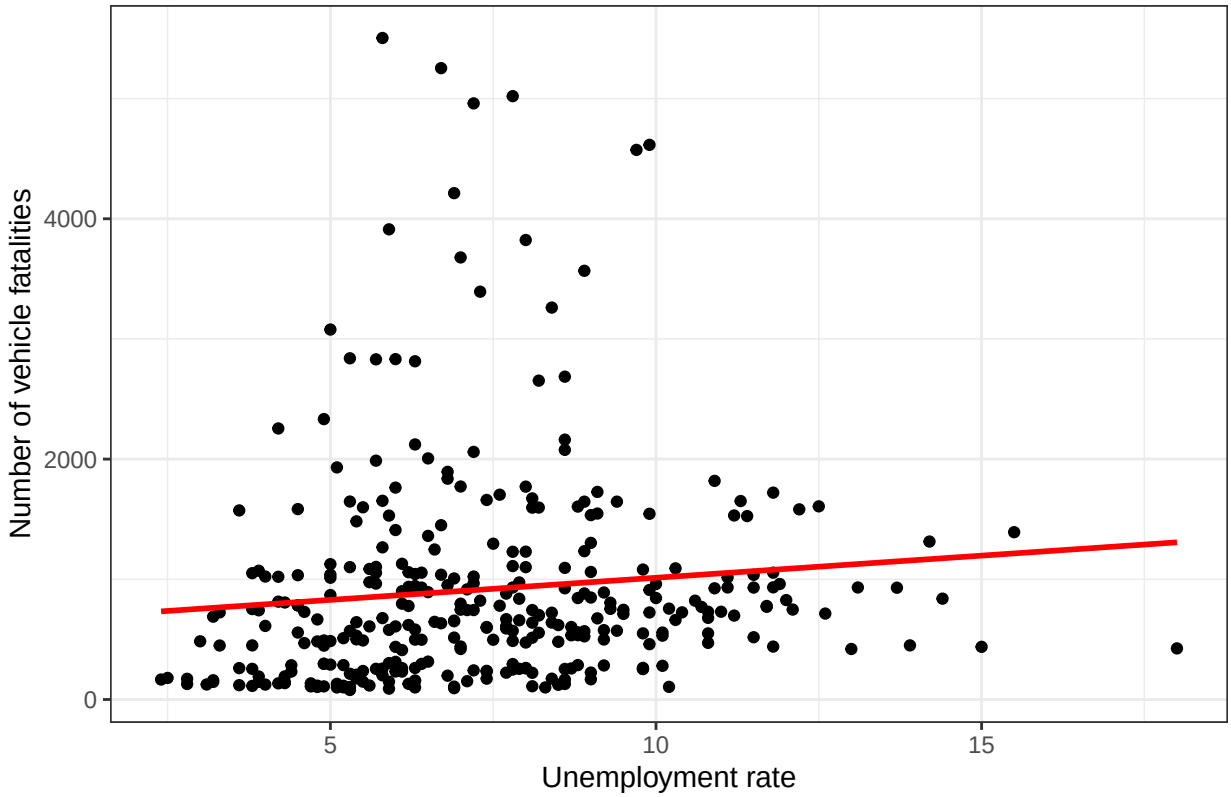
You can extra manipulate your graphs to customize your labels, legend, colors, etc. You can also use *ggthemes* package to imitate famous magazine charts. More on this here. I focus on modifying our **Barplot** and **Scatter plot** to show some of these features:

```
# Starting with our Barplot.
# Let's look at the fatalities and jail sentence

ggplot(Fatalities %>% filter(!is.na(jail)), # data with no NA
       aes(x = jail,        # categorical variable
           fill = jail)) +  # color of categories
  geom_bar() +              # data representation type
  coord_flip() +            # makes horizontal graph
  theme_bw() +              # white background
  theme(legend.position = "none") + # remove legend
  labs(title = "Mandatory jail sentence for Traffic Fatalities",   # Title of graph
       y = "Number of cases",   # X-axis label
       x = "")       # Y-axis label
```



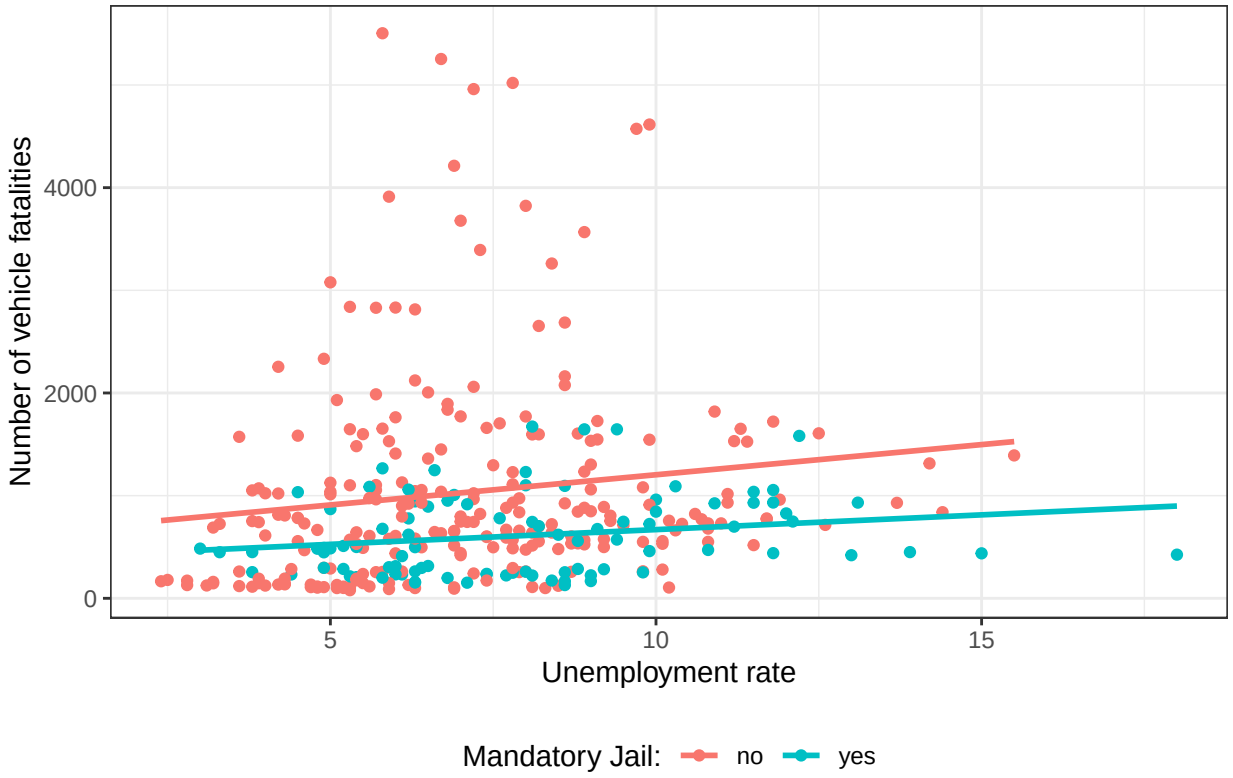

```
# Now let's look at a Scatter Plot
# Let's look at the fatalities and jail sentence

ggplot(Fatalities %>% filter(!is.na(jail)), # data with no NA
       aes(x = unemp ,     # unemployment
           y = fatal )) + # of fatalities
  geom_point()  + # data representation type
  geom_smooth(method=lm , color="red", se=FALSE) + # trend
  theme_bw()             +  # white background
  labs(title = "",    # No title in this graph
       y = "Number of vehicle fatalities",   # X-axis label
       x = "Unemployment rate")       # Y-axis label
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
# Let add more things to our scatter plot
ggplot(Fatalities %>% filter(!is.na(jail)), # data with no NA
       aes(x = unemp ,     # unemployment
           y = fatal ,     # of fatalities
           color = jail)) + # add category to each point
  geom_point()  + # data representation type
  geom_smooth(method=lm , se=F) + # linear trend per group
  theme_bw()             +  # white background
  theme(legend.position = "bottom",
          legend.box = "vertical") +
  labs(color = "Mandatory Jail:",  # Legend title
       title = "",   # Graph Title: empty
       y = "Number of vehicle fatalities",   # X-axis label
       x = "Unemployment rate")    # Y-axis label
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
# Let add more things to our scatter plot
ggplot(Fatalities %>% filter(!is.na(jail)), # data with no NA
       aes(x = unemp ,    # unemployment
           y = fatal ,    # of fatalities
           color = jail)) + # add category to each point
  geom_point()  + # data representation type
  geom_smooth(method=lm , se=F) + # linear trend per group
  facet_wrap(~jail)     + # create small graph per group
  theme_bw()            +  # white background
  theme(legend.position = "bottom",
        legend.box = "vertical") +
  scale_color_manual(values = c("red","blue")) + # Manually change colors
  labs(color = "Mandatory Jail:",  # Legend title
       title = "",   # Graph Title: empty
       y = "Number of vehicle fatalities",   # X-axis label
       x = "Unemployment rate")    # Y-axis label
```

```
## `geom_smooth()` using formula 'y ~ x'
```

4. RMarkdown

RStudio makes it very easy for us to use **RMarkdown**. What is **RMarkdown**? In short it is a way for you to create documents (reports or presentations) written in **RStudio** combining codes/results and textual analysis. For example, this document has been written using **RMarkdown**. Because of its complexity we will focus on just some key aspects of it so you can use it for your homework. You can more about **RMarkdown** on the following links: coding club, rstudio, towards data science, data quest, sage, and *R for Data Science* (Grolemund and Wickham 2017).

The first thing you want to do is to install some `LaTeX` repository. If you are using Windows use MiktTeX and if you are using a Mac use MacTeX.

4.1 Getting Started

To get RMarkdown working in RStudio install and load the package **rmarkdown** The RMarkdown file is the *(.Rmd)*.

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

Note: If you are using RMarkdown for your R analysis, for example a homework, all the data manipulation must be in the file as R will function as usual, looking for the object, modifying it and running the analysis as specified. The main difference here is that we can break the code into chuncks and add text in between so we can better document what we are doing as well as use the file as a report.

4.2 The RMarkdown File

4.2.1 Header

```
---
title: "Title of Analysis"
author: "First Name"
date: "12/09/2020"
output: pdf_document
---
```

4.2.2 Text

```
# Title 1

## Subtitle 2

### Subsubtitle 3

Some text describing the code to come or making some kind of analysis.
```

To customize your text you can use `*text*` for *italics* and `**text**` for **bold**.

4.2.3 Code Chunk

The code chunk is where you will make your analysis in R and save or print results.

You can add some rules to your chunk to display results, messages, warnings, the code itself, etc.

Rule	Default	Function
eval	TRUE	code run and the results included
include	TRUE	code and the results included
echo	TRUE	code displayed alongside the results
warning	TRUE	warning messages displayed
error	FALSE	error messages displayed
message	TRUE	messages displayed

4.2.4 Compiling your RMarkdown

To compile your RMarkdown you need to have access to a latex distribution. I suggest `tinytex`. Before compiling your RMarkdown file please run the folowing code on your console:

```
install.packages('tinytex')
tinytex::install_tinytex()
```

Now use the `Knit` button to decide if you will compile your file as `.html` or `.pdf`. You decide how your file is compiled in the header:

```
---
title: "Title of Analysis"
author: "First Name"
date: "12/09/2020"
output: pdf_document | html_document
---
```

References

Grolemund, Garrett, and Hadley Wickham. 2017. *R for Data Science*. O'Reilly Media.

Phillips, Nathaniel D. 2018. *YaRrr! The Pirate's Guide to R*. Published with bookdown.