Collaborative Research Centre Transregio Munich - Frankfurt (TRR 267) Non-coding RNA in the cardiovascular system Make your paper figures professionally: Scientific data analysis and visualization with R

Fatemeh Behjati Ardakani, Vangelis Theodorakis, Julien Gagneur, Marcel Schulz

09 October, 2020

Contents

Chapter 1

Foreword

The aim of these series of lectures is to provide you with all the knowledge needed in order to make state of the art plots using the R programming language and Grammar of Graphics. The lectures cover the basic concepts of R, how to manipulate data and how to generate clear and meaningful plots for your target audience, using ggplot2 an implementation of Grammar of Graphics for R.

Chapter 2

Data visualisation introduction

Data Science is crucial for many applications in a variety of fields and is the language of collaboration between biology, computer science, bioinformatics, physics, and mathematics.

Goals of this course include to introduce you to the R and Rstudio, and provide you general analytic techniques to extract knowledge, patterns, and connections between samples and variables, and how to communicate these in an intuitive and clear way. Some of the examples will be expressed in the field of bioinformatics.

2.1 What is Data Science?

Data science is an interdisciplinary field about processes and systems to extract knowledge or insights from data in various forms, either structured or unstructured. It is a continuation of some of the data analysis fields such as statistics, data mining, and predictive analytics.

The Goals of Data Science include discovering new phenomena or trends from data, enabling decisions based on facts derived from data, and communicating findings from data.

Data science at its core is the heart of any scientific method. It starts with observations, makes hypotheses, experiments, and visualizes the results in such a way that you can make claims about the validity of the hypothesis. These skills are applicable not only to whatever research topic interests you, but are also some of the hottest skills you may develop for the job market.

2.2 What you'll learn (hopefully)

Opposed to many other statistics or machine learning lectures this course is designed to teach you the practical skills you need as a data scientist. We will not focus on the complex math that went into developing the high powered statistical tools, and machine learning techniques thought in many courses. We will focus on tidy data, visualizations, and data manipulation in R.

2.3 Our tool

- R- a language and program designed for data analytics. Not great for big applications, but an environment to work with data in an easy way based on scripts.
  - Scripting: work on the fly, define few functions and data structures but use existing packages to interpret your data along with your own intuition.
  - Package developer: robust, fast code, that copes with the inherent problems of the language.

2.4 Breakdown of topics

- R markdown: Notebooks that allow you to quickly publish and share your code informally so that you can see the results and the code that generated it. (This is an R markdown file)
- Tidy data: standard way of structuring data for efficient memory storage and efficient operations
- Visualization with ggplot2: flexible grammar and frame-work to visualize data in concise and easy ways.

Chapter 3

Introduction to R and Rstudio

- R is the language of this course. It shares some aspects with python, but the syntax is often unique compared to other languages. All R files are named with a .R extension (dot - R)

- Rstudio is an editor, it succinctly organizes your session into 4 windows each set up for you to do certain tasks in each window. Edit and write code (editor), run and execute code (console), list and name objects, and a window to show output in either function help, and plots.

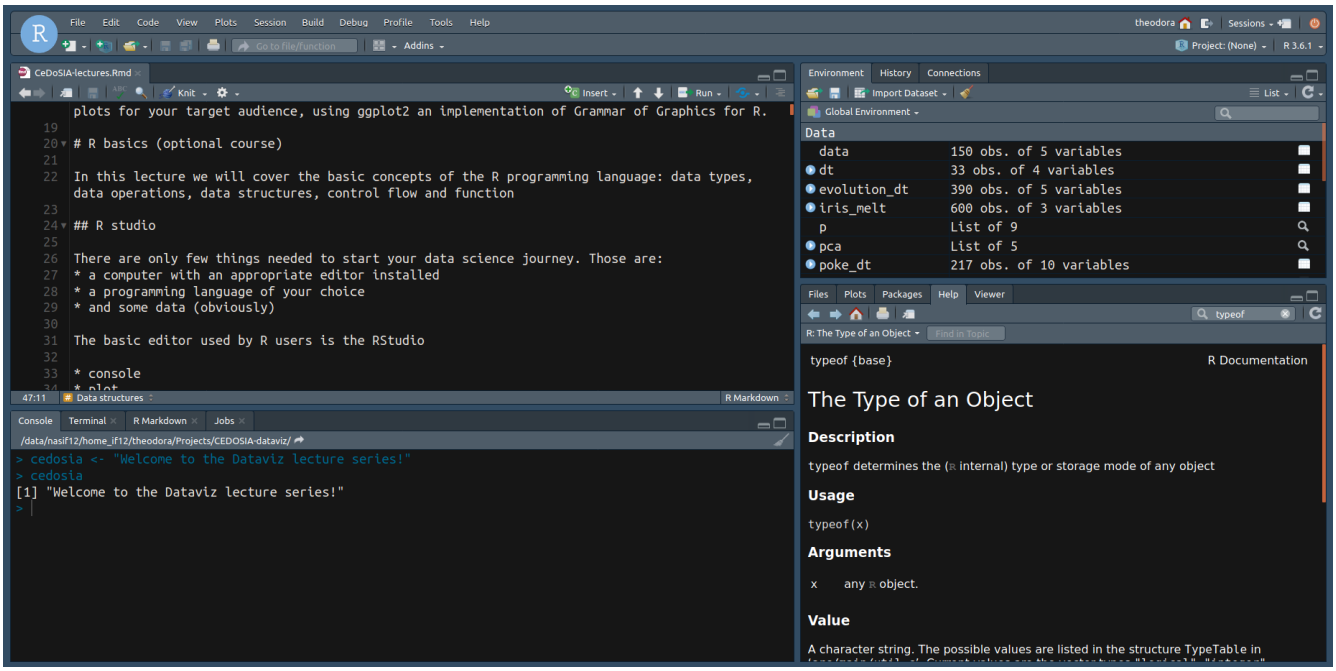


Figure 3.1: Rstudio

The most commonly used editor by R users is the RStudio. It consists of the following sections such as:

- the main script section, for writing scripts [top left section] (Ctrl+1 to focus)
- the console tab, for typing R commands directly [bottom left section as tab] (Ctrl+2 to focus)
- the terminal tab, for direct access to your system shell [bottom left section] (Shift+Alt+T to focus)
- the plot tab, where you see the last plot generated [bottom right section as tab]
- the help tab, with useful documentation of R functions [bottom right section as tab] (F1 on the name of a function or Ctrl+3)
- the history tab, with the list of the R commands used [top right section as tab]
- the environment tab, with the created variables and functions loaded [top right section as tab]

- and the packages tab, with the available/loaded R packages [bottom right section as tab]

Check the View menu to find out the rest of useful shortcuts!

3.1 Introduction to basic R syntax

- All code is run in the console (bottom left) you can write and alter code directly in here or in the editor above (top left) for easier editing/debugging
- All of your questions and issues can be investigated Here
- R examples:
  - basic operators

```
x <- 10
x
```

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

```
x + 5
```

```
## [1] 15
```

```
y <- x + 5
y
```

```
## [1] 15
```

```
x == y
```

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

```
x
```

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

  - built-in methods

```
x <- rnorm(10)  #returns a vector of random numbers based on the normal distribution
mean(x)
```

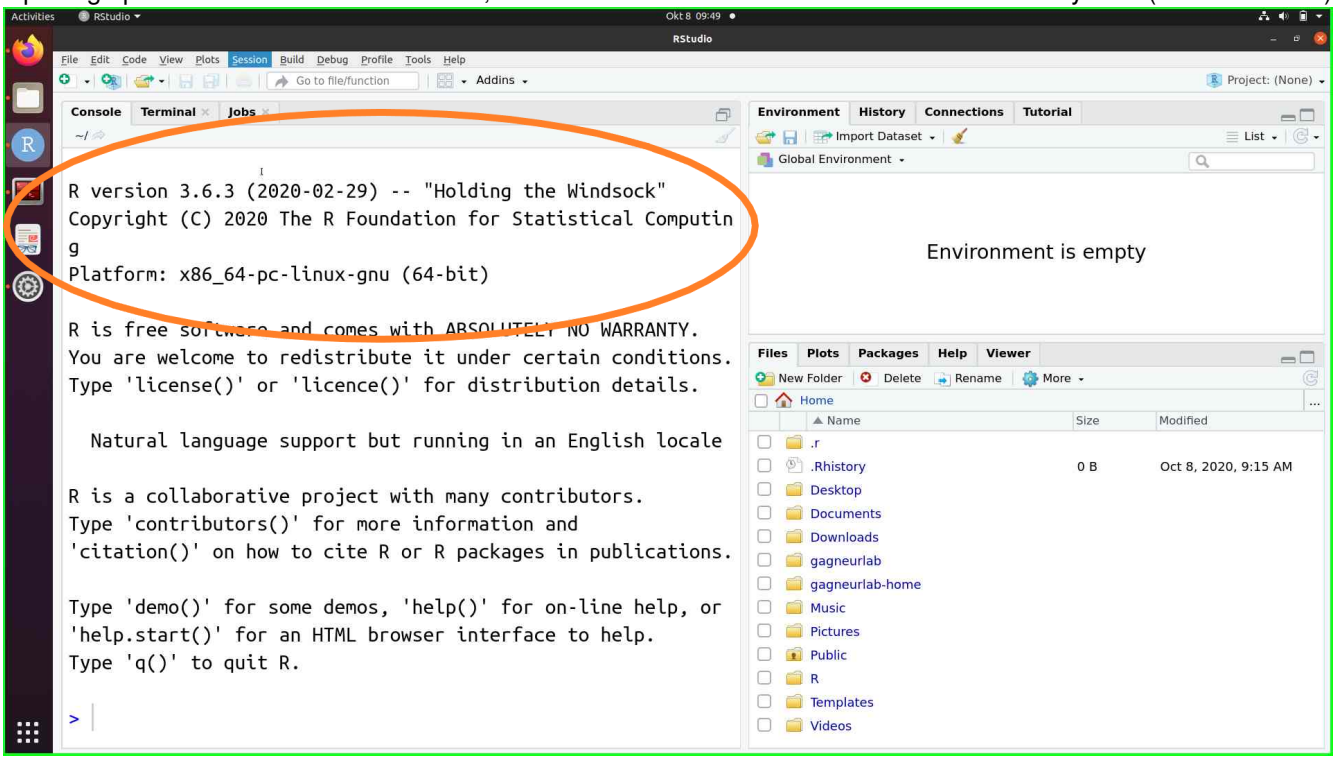
```
## [1] 0.1550577
```

- **?** operator
  `?mean` shows you information about the function you are calling. Look to the bottom right window.

We will introduce further details of the R language as they are needed throughout the course and refer you to the Appendix II for a more detailed introduction to R.

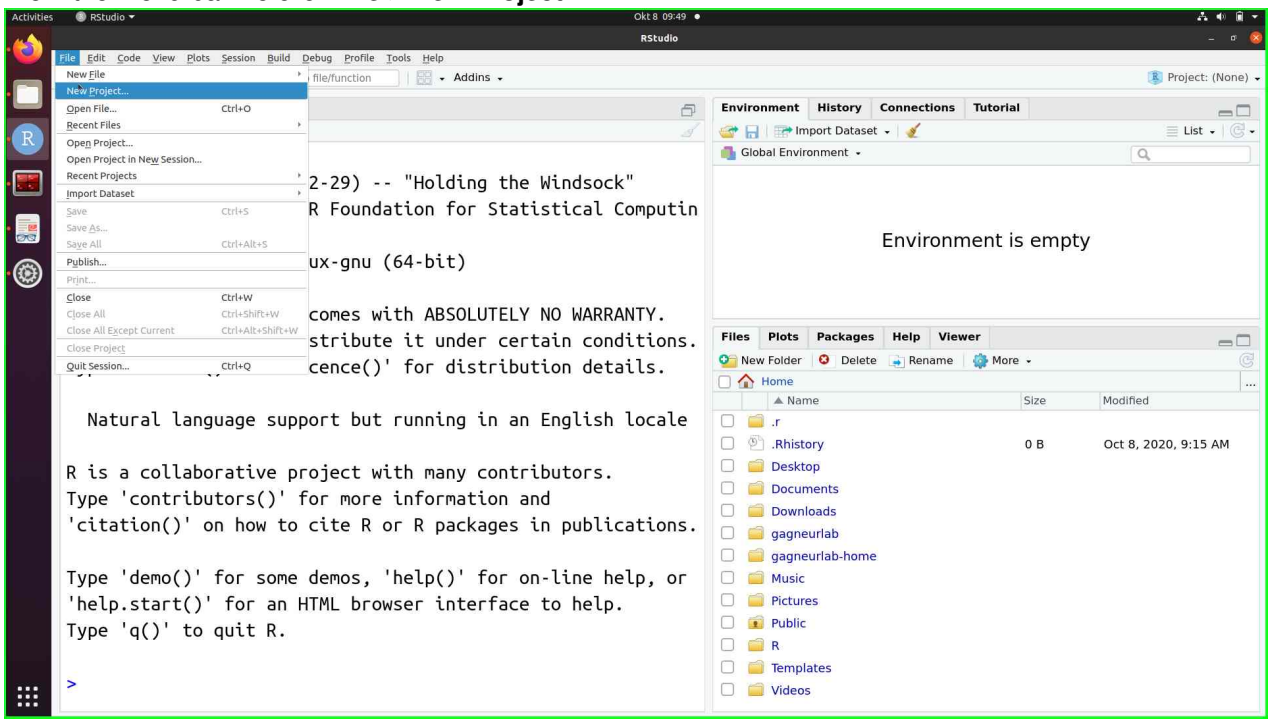
3.2 Introduction to basic RStudio usage

Opening up Rstudio in an fresh R session, we see the R version and the architecture of our system (32-bit or 64-bit)

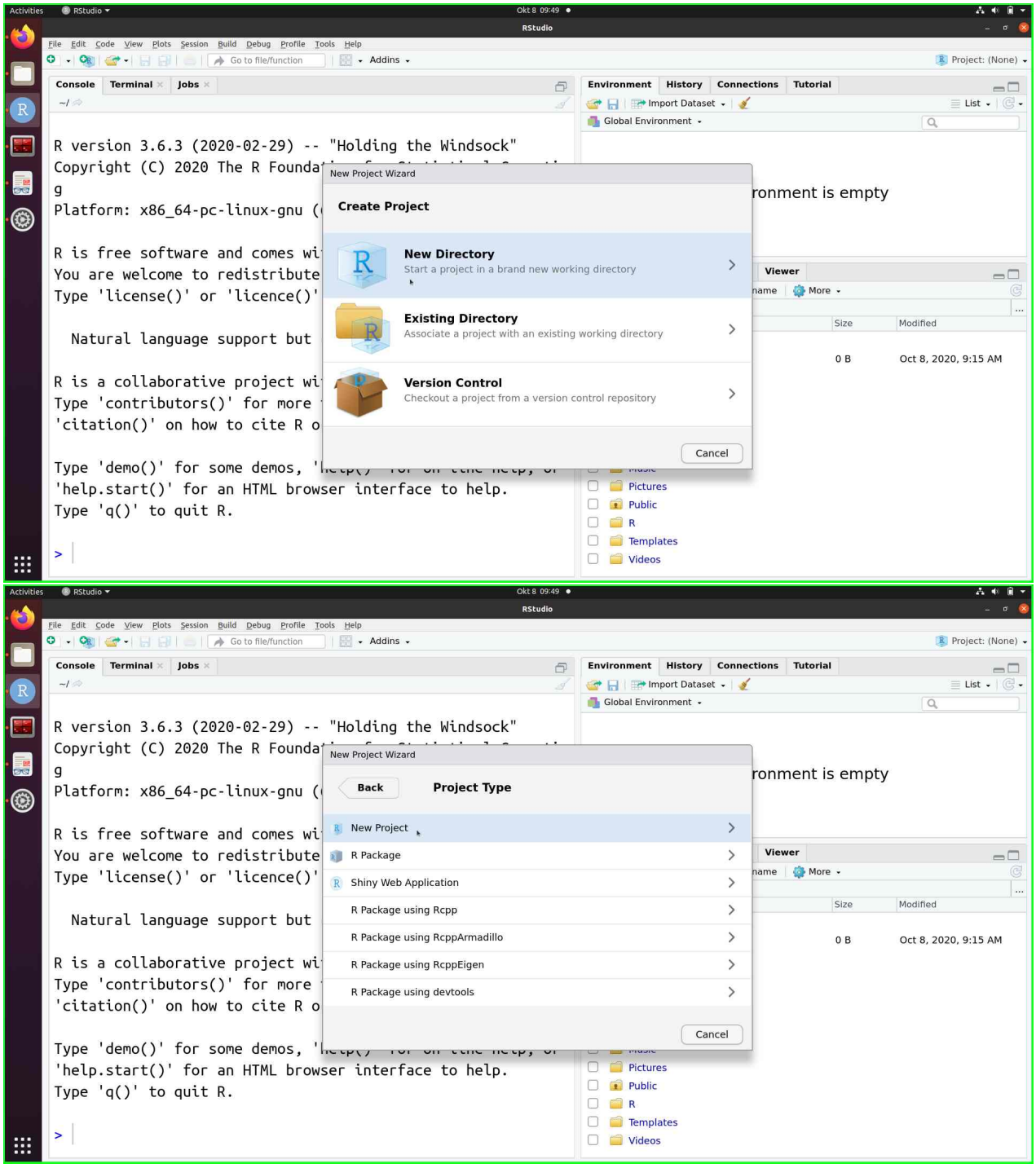


Now are going to show a super simple tutorial of a project setup.

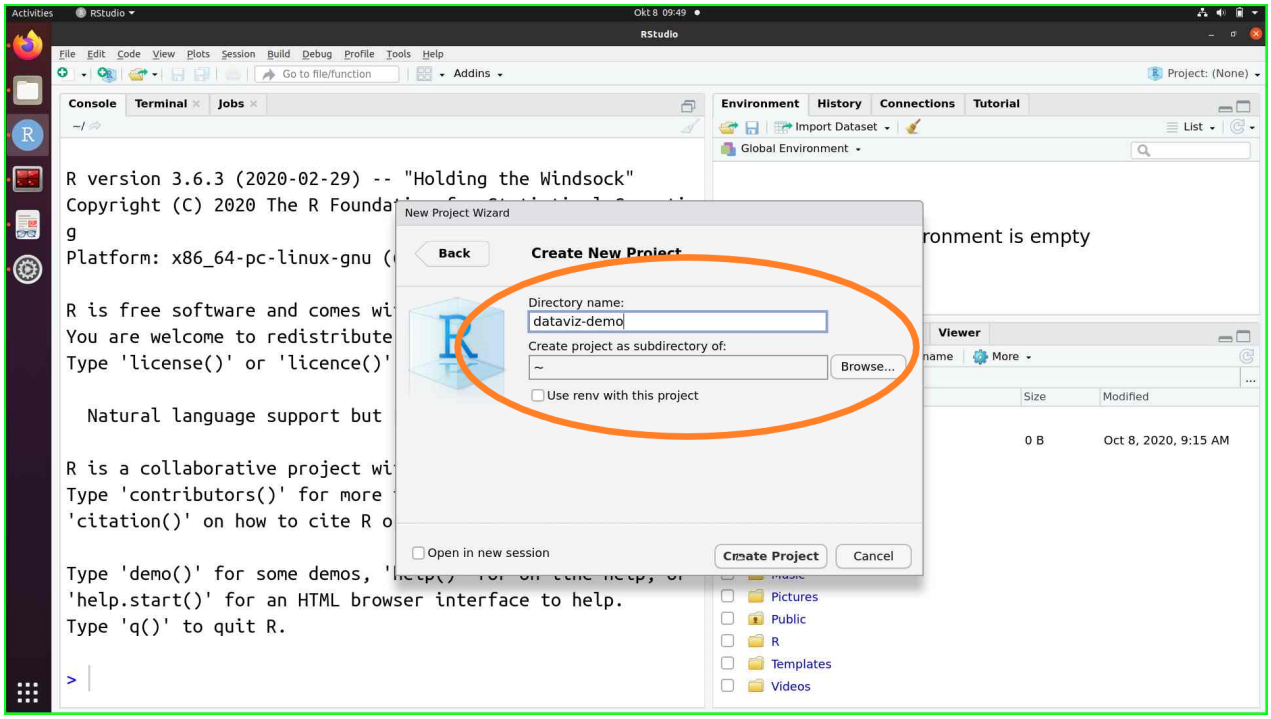
1. From the menu bar we click **File > New Project**



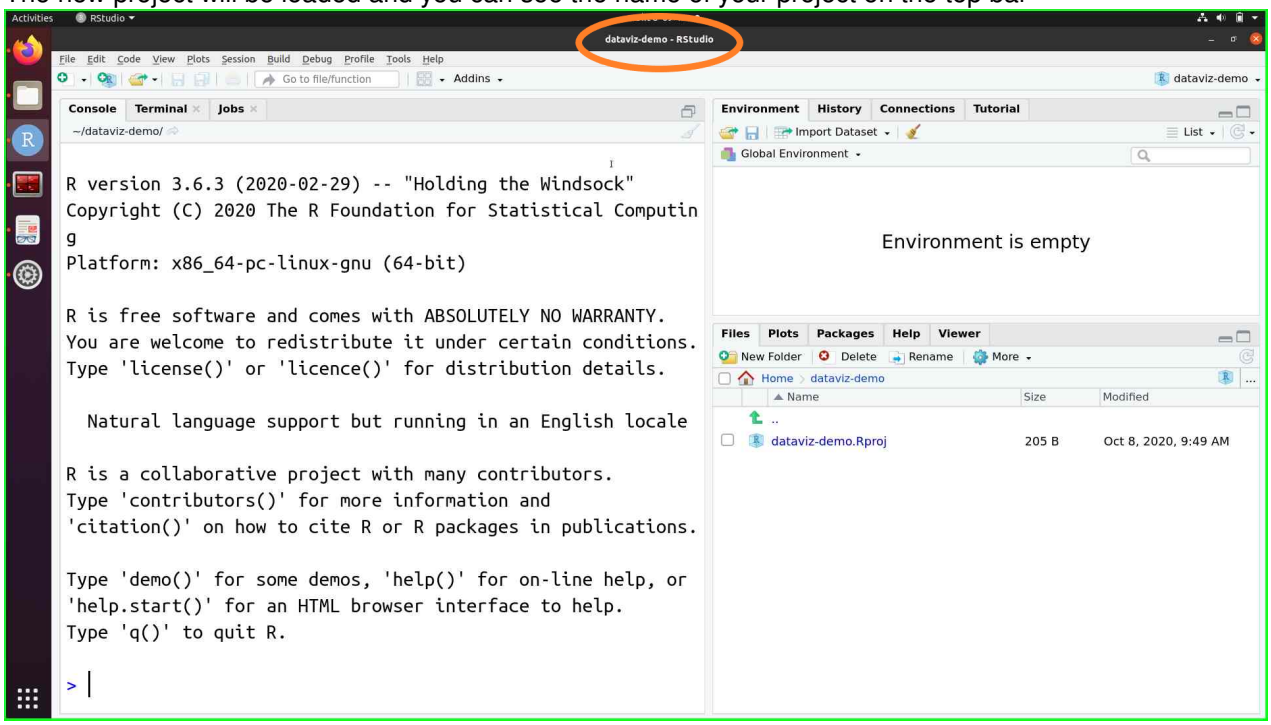
2. Then we choose **New Directory > New Project**



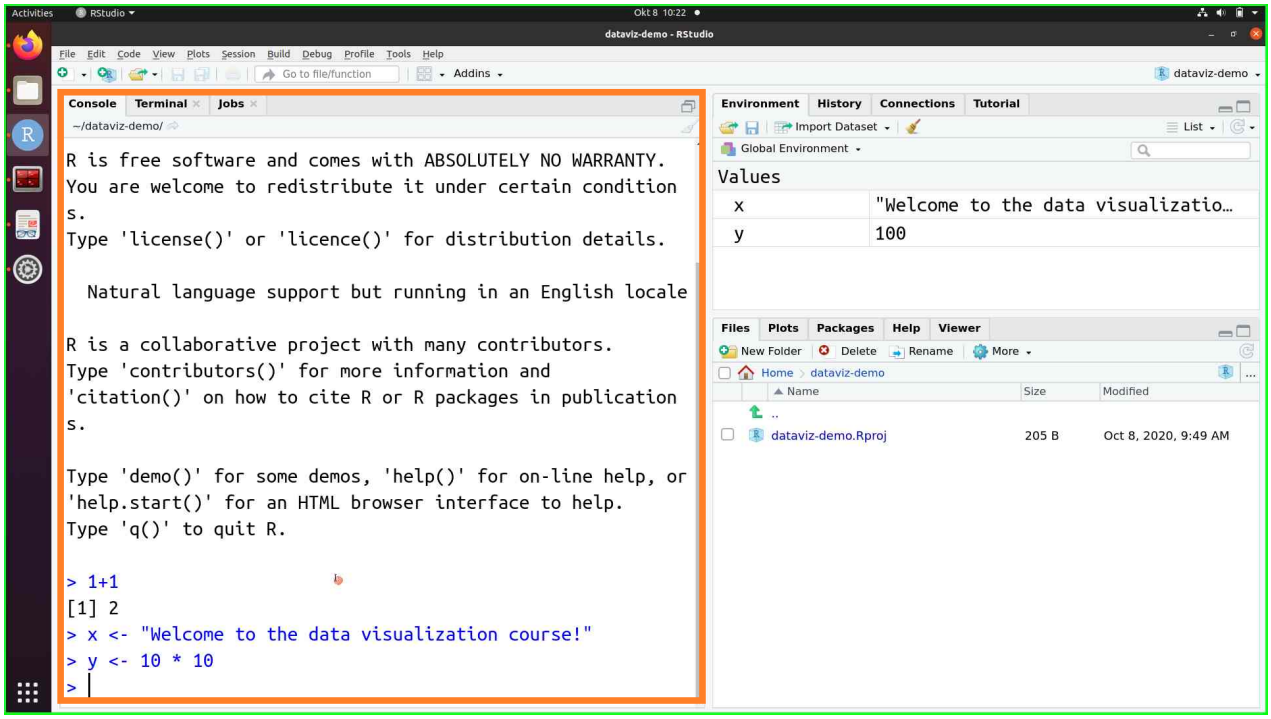
3. Give a name to your **New Project** eg. *dataviz-demo*, and set the path where the new project will be created and click **Create Project**



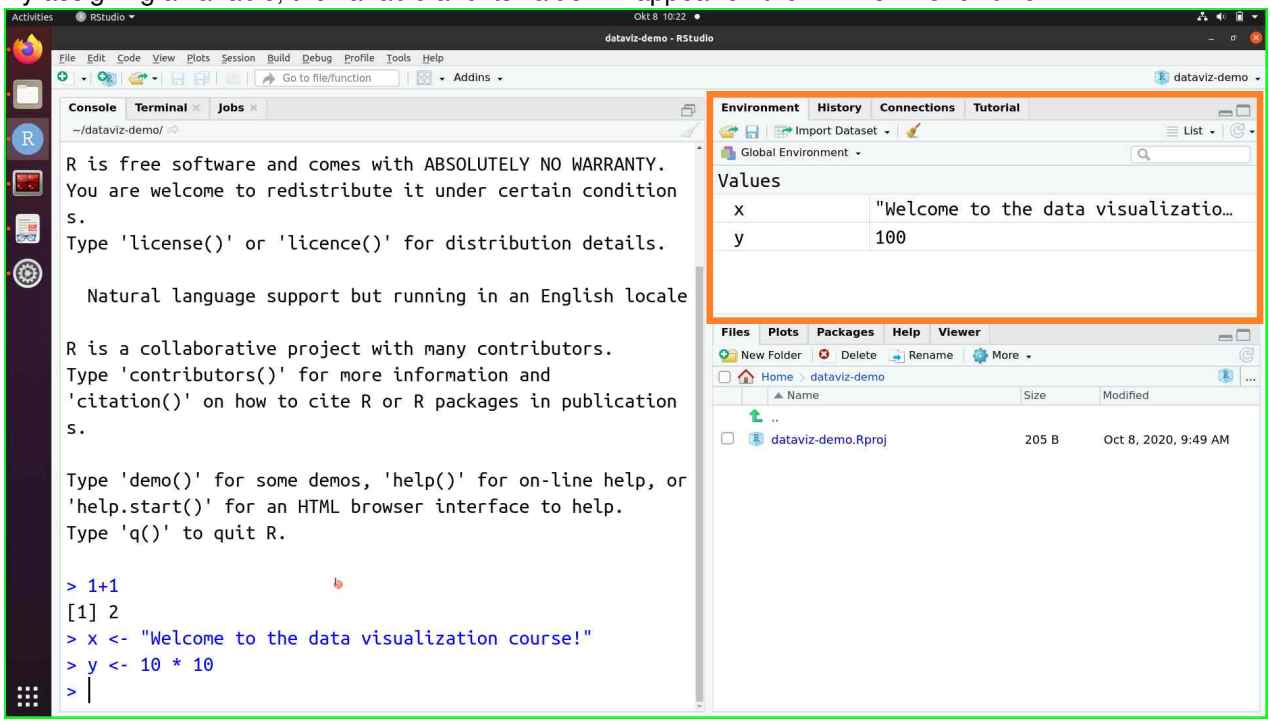
4. The new project will be loaded and you can see the name of your project on the top bar



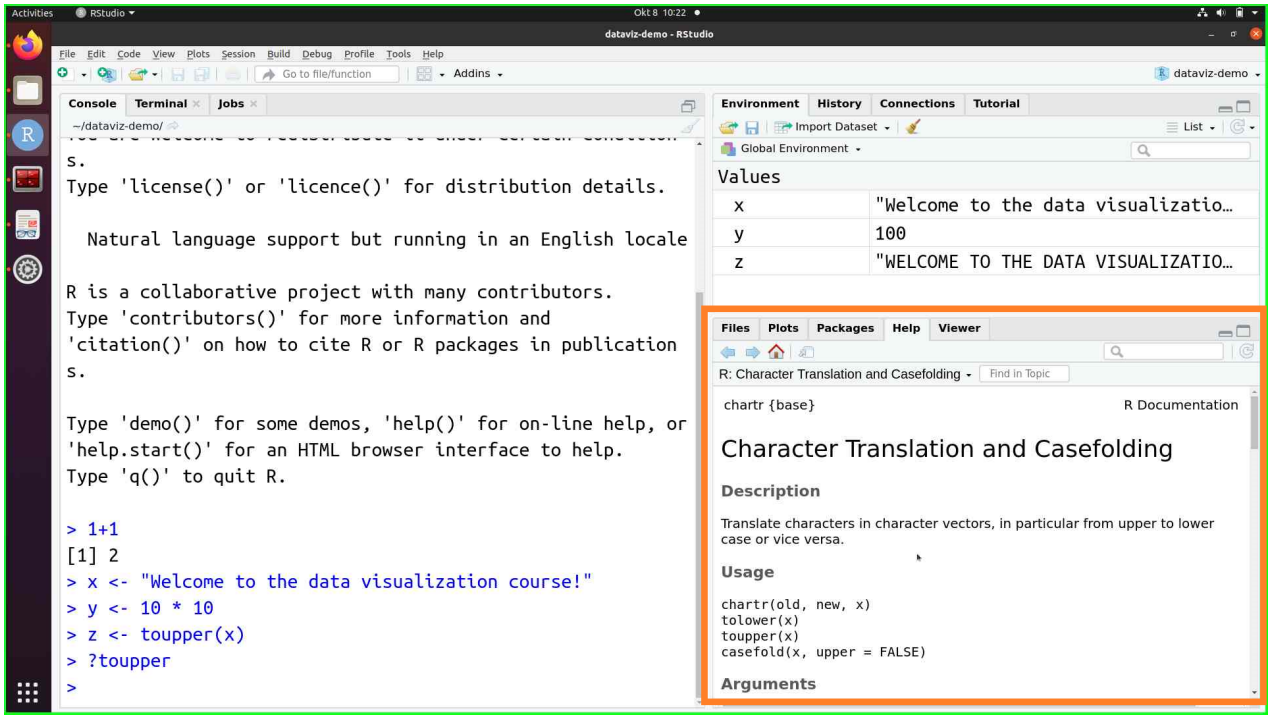
5. Now you can start typing command on the **Console Panel**



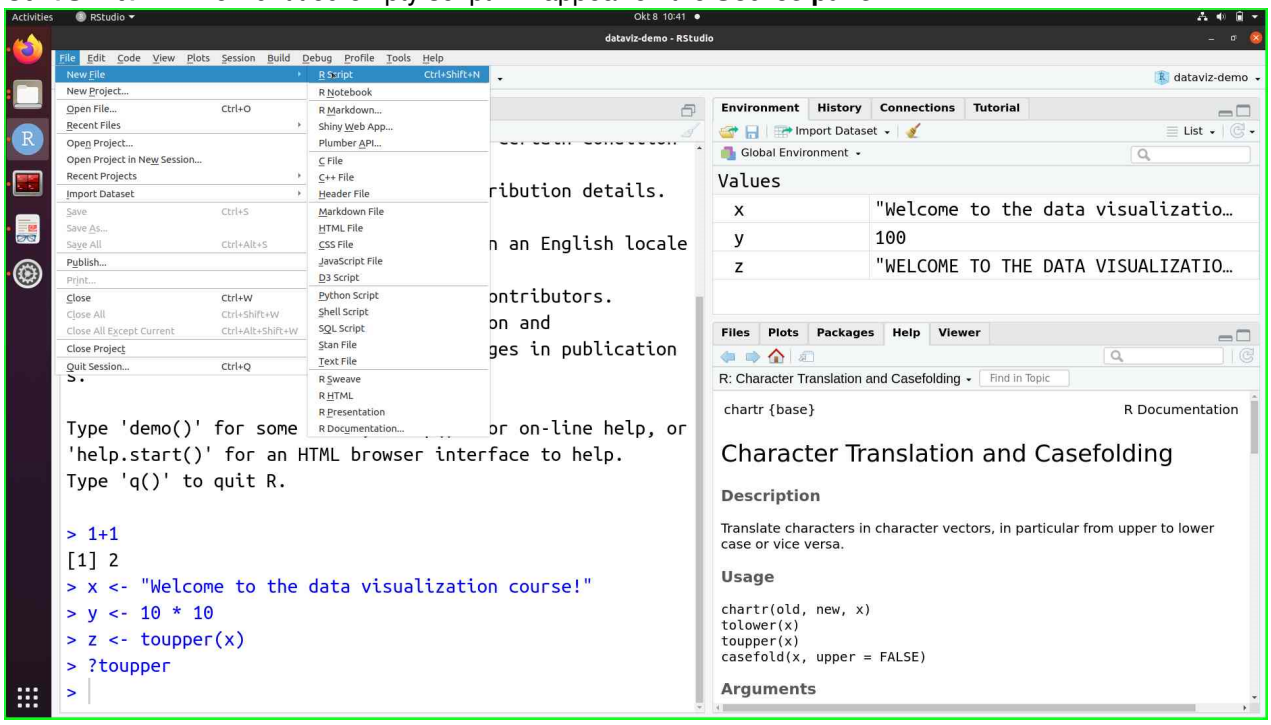
6. By assigning a variable, the variable and its value will appear on the **Environment Panel**

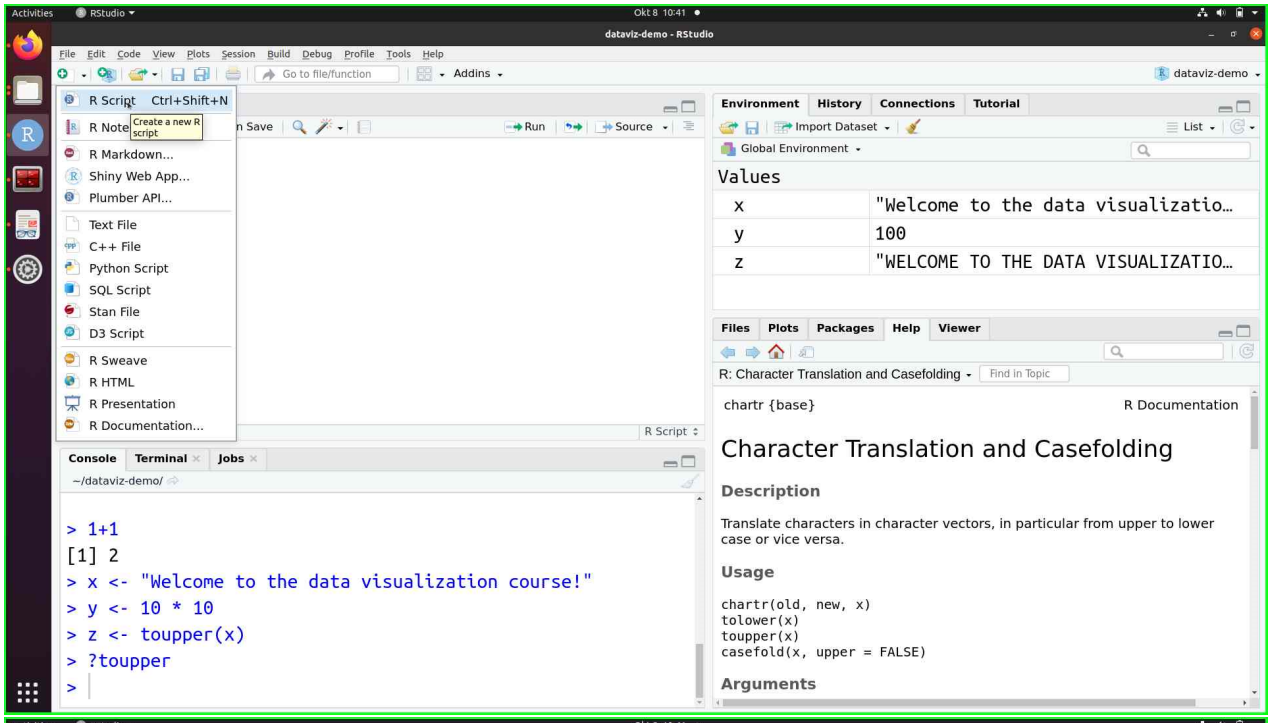


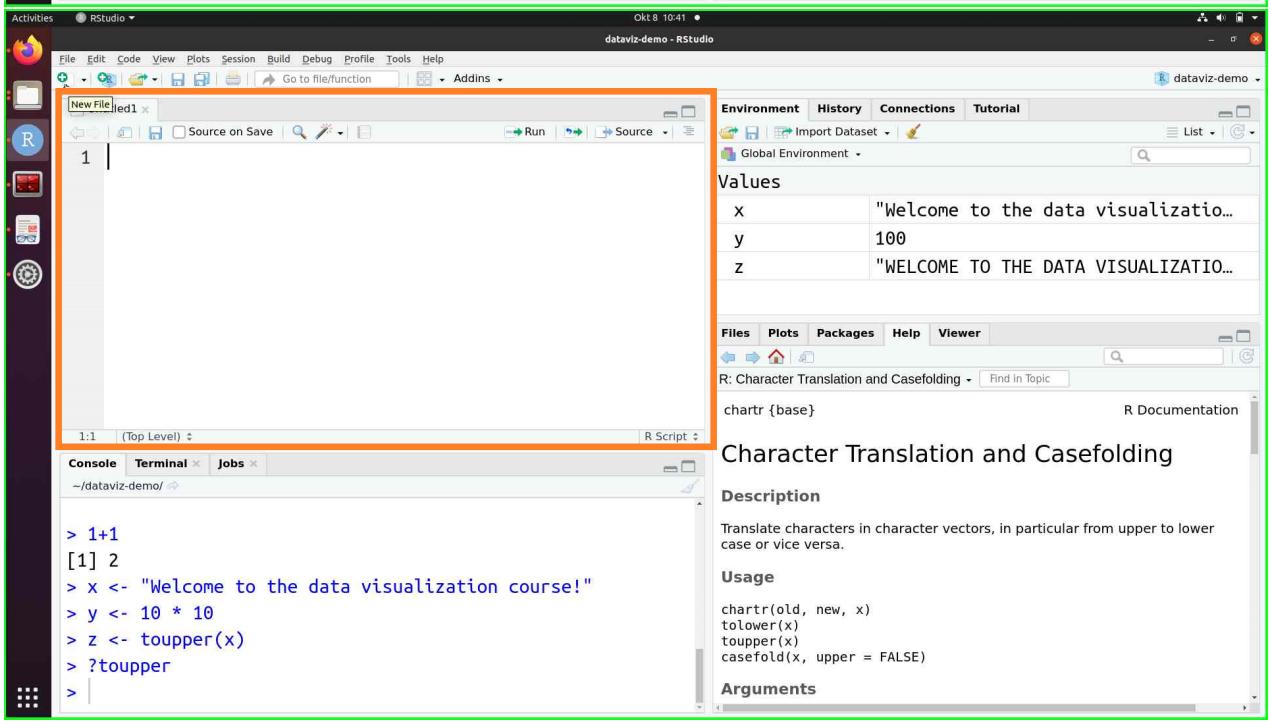
7. By typing the `?` before an function name e.g. `?toupper`, the toupper function will appear on the **Help Panel** with all the documentation of the function as well as usefull examples.



8. Typing endlessly commands in the console can cumbersome, so in order for us to be able to reproduce our code, what we do is that we write our commands in a script file. To make a new script file we can either click **File > New File > R** script or click or the **+** icon (below the File menu) and then click **R script** or simply press **Ctrl+Shift+N**. A new untitled empty script will appear on the **Source panel**




dataviz-demo - RStudio
R Script Ctrl+Shift+N
Create a new R script
R Markdown...
Shiny Web App...
Plumber API...
Text File
C++ File
Python Script
SQL Script
Stan File
D3 Script
R Sweave
R HTML
R Presentation
R Documentation...
> 1+1
[1] 2
> x <- "Welcome to the data visualization course!"
> y <- 10 * 10
> z <- toupper(x)
> ?toupper
Values
x "Welcome to the data visualizatio…
y 100
z "WELCOME TO THE DATA VISUALIZATIO…
chartr {base} R Documentation
Character Translation and Casefolding
Description
Translate characters in character vectors, in particular from upper to lower case or vice versa.
Usage
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
Arguments


dataviz-demo - RStudio
New File
> 1+1
[1] 2
> x <- "Welcome to the data visualization course!"
> y <- 10 * 10
> z <- toupper(x)
> ?toupper
Values
x "Welcome to the data visualizatio…
y 100
z "WELCOME TO THE DATA VISUALIZATIO…
chartr {base} R Documentation
Character Translation and Casefolding
Description
Translate characters in character vectors, in particular from upper to lower case or vice versa.
Usage
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
Arguments

Chapter 4

Data Wrangling using Data.table

Data Wrangling includes tasks of processing raw data into interesting formats. . . .

Data.tables are a modern implementation of tables in R. We will exclusively used data.tables in this course. Base R provides a similar structure called data.frames. However, those are a lot slower and often a little more complicated to use, so we highly recommend using data.table.

Similar to a data.frame, but because it modifies columns by *reference*, a data.table is a memory efficient and faster implementation of data.frame offering:

- sub-setting
- ordering
- merging

It accepts all data.frame functions, it has a shorter and more flexible syntax (not so straightforward in the beginning but pays off) and saves time on two fronts:

- programming (easier to code, read, debug and maintain)
- computing (fast and memory efficient)

The general form of data.table syntax is:

```
DT[ i,  j,  by ] # + extra arguments
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do?
     ---> on which rows?
```

The way to read this out loud is: "Take DT, subset rows by i, then compute j grouped by by. Here are some basic usage examples expanding on this definition.

Like data.frame, data.table is a list of vectors. It doesn't have row names; instead, it's a collection of attributes.

Each column can be a different type, including a list

It has enhanced functionality in `[`. We can make operations inside `[]`.

For a detailed documentation go check the help page for data.table!

4.1 Creating and loading data

To create a data.table, we just name its columns and populate them. All the columns have to have the same length.

```
# install.packages('data.table')
library(data.table)
DT <- data.table(x = rep(c("a", "b", "c"), each = 3), y = c(1, 3, 6), v = 1:9)
DT  # note how column y was recycled
```

```
##    x y v
## 1: a 1 1
## 2: a 3 2
## 3: a 6 3
## 4: b 1 4
## 5: b 3 5
## 6: b 6 6
## 7: c 1 7
## 8: c 3 8
## 9: c 6 9
```

If we want to convert any other R object to a data.table, all we have to do is to call the `as.data.table()` function.

```
# This way we can for example convert any inbuilt data set into a data.table:
titanic_dt <- as.data.table(Titanic)
class(titanic_dt)
```

```
## [1] "data.table" "data.frame"
```

Here you can see that the class function informs us that DT is both a data.table and a data.frame as data.tables inherit from data.frames, an older table class in R.

Alternativly we can read files from disk and process them using data.table. The easiest way to do so is to use the function `fread()`.

```
flights <- fread("path_to_file/flightsLAX.csv")
```

Calling `head()` on the table showes us the first few lines of the table and we observe that reading the file was succesfull.

```
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
## 6: 2015     1   1           4      UA          1545      N76517            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 CLT             44      228     2125          753
## 4:                 MSP             35      188     1535          605
## 5:                 MIA            103      255     2342          839
## 6:                 IAH            112      156     1379          607
```

4.2 Inspecting tables

A first step in any analysis should involve inspecting the data we just read in. This often starts by looking the head of the table as we did above.

The next information we are often interested in is the size of our data set. We can use the following commands to obtain it.

```
nrow(flights)  # ncol(flights)
```

```
## [1] 389369
```

```
dim(flights)
```

```
## [1] 389369     13
```

4.2.1 Technical detail

The structure of the data.table essentially is a list of list. To get the dimensions of our data.table we can either use the `dim()` function to get the dimensions of our data.table (M rows x N columns) or to use the `ncol()` `nrow()` functions.

```
class(DT)
```

```
## [1] "data.table" "data.frame"
```

```
is.list(DT)
```

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

```
nrow(DT)  # ncol(DT)
```

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

```
dim(DT)
```

```
## [1] 9 3
```

Next we are often interested in the range or the number of occurence of values in certain columns. To obtain this information we can use the summary command on the table.

```
summary(flights[, 1:6])
```

```
##       YEAR          MONTH             DAY        DAY_OF_WEEK
##  Min.   :2015   Min.   : 1.000   Min.   : 1.0   Min.   :1.000
##  1st Qu.:2015   1st Qu.: 3.000   1st Qu.: 8.0   1st Qu.:2.000
##  Median :2015   Median : 6.000   Median :16.0   Median :4.000
##  Mean   :2015   Mean   : 6.198   Mean   :15.7   Mean   :3.934
##  3rd Qu.:2015   3rd Qu.: 9.000   3rd Qu.:23.0   3rd Qu.:6.000
##  Max.   :2015   Max.   :12.000   Max.   :31.0   Max.   :7.000
##    AIRLINE          FLIGHT_NUMBER
##  Length:389369      Min.   :   1
##  Class :character   1st Qu.: 501
##  Mode  :character   Median :1296
##                     Mean   :1905
##                     3rd Qu.:2617
##                     Max.   :6896
```

This provides us already a lot of information about our data. We can for example see that all data is from 2015 as all values in the YEAR column are 2015. But for categorical data this is not very insight full, as we can see for the AIRLINE column.

To investigate categorical columns we can have a look at their unique elements using:

```
flights[, unique(AIRLINE)]
```

```
##  [1] "AA" "US" "DL" "UA" "OO" "AS" "B6" "NK" "VX" "WN" "HA" "F9" "MQ"
```

This command provided us the airline identifiers present in the dataset. Another valueable information for categoricals is how often each category occurs. This can be obtained using the following commands:

```
flights[, table(AIRLINE)]
```

```
## AIRLINE
##    AA    AS    B6    DL    F9    HA    MQ    NK    OO    UA    US    VX    WN
## 65483 16144  8216 50343  2770  3112   368  8688 73389 54862  7374 23598 75022
```

Or:

```
flights[, .N, by = "AIRLINE"]
```

```
##     AIRLINE     N
##  1:      AA 65483
##  2:      US  7374
##  3:      DL 50343
##  4:      UA 54862
##  5:      OO 73389
##  6:      AS 16144
##  7:      B6  8216
##  8:      NK  8688
##  9:      VX 23598
## 10:      WN 75022
## 11:      HA  3112
## 12:      F9  2770
## 13:      MQ   368
```

Especially the second command will be explained in detail later.

4.3 Accessing rows and columns of data.tables

Data.tables can be subsetted by indices, if we for example care about the second element of the table. We can do the following.

```
flights[2, ]  # Access the 2nd row (also flights[2] or flights[i = 2])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 MIA             15      258     2342          756
```

```
flights[1:2]  # Access multiple consecutive rows.
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
```

```
flights[c(3, 5)]  # Access multiple rows.
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      US          2013      N584UW            LAX
## 2: 2015     1   1           4      AA           115      N3CTAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 CLT             44      228     2125          753
## 2:                 MIA            103      255     2342          839
```

Additionally we can do the same for columns:

```
head(flights[, 2])
```

```
##    MONTH
## 1:     1
## 2:     1
## 3:     1
## 4:     1
## 5:     1
## 6:     1
```

```
head(flights[, c(3, 5)])
```

```
##    DAY AIRLINE
## 1:   1      AA
## 2:   1      AA
## 3:   1      US
## 4:   1      DL
## 5:   1      AA
## 6:   1      UA
```

Alternatively to subsetting by indices we can subset columns by their names.

```
head(flights[, .(MONTH)])
```

```
##    MONTH
## 1:     1
## 2:     1
## 3:     1
## 4:     1
## 5:     1
## 6:     1
```

```
head(flights[, .(DAY, AIRLINE)])
```

```
##    DAY AIRLINE
## 1:   1      AA
## 2:   1      AA
## 3:   1      US
## 4:   1      DL
## 5:   1      AA
## 6:   1      UA
```

4.4 Sub-setting rows according to some condition

A often more usefull way is to subset rows using conditions! We can subset a data table using the following operators:

- `==`
- `<`
- `>`
- `!=`
- `%in%`

For example if we are interested in analysing all the flights operated by “AA” (American Airlines) we can do that using the following command:

```
head(flights[AIRLINE == "AA"])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      AA           115      N3CTAA            LAX
## 4: 2015     1   1           4      AA          2410      N3BAAA            LAX
## 5: 2015     1   1           4      AA          1515      N3HMAA            LAX
## 6: 2015     1   1           4      AA          1686      N4XXAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 MIA            103      255     2342          839
## 4:                 DFW            600      150     1235         1052
## 5:                 ORD            557      202     1744         1139
## 6:                 STL            609      183     1592         1134
```

Alternatively if we are interested in all flights from any destination to the airports in NYC (JFK and LGA), we can subset the rows using the following command:

```
flights[DESTINATION_AIRPORT %in% c("LGA", "JFK")]
```

```
##         YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER
##      1: 2015     1   1           4      B6            24      N923JB
##      2: 2015     1   1           4      DL           476      N196DN
##      3: 2015     1   1           4      AA           118      N788AA
##      4: 2015     1   1           4      VX           404      N621VA
##      5: 2015     1   1           4      UA           275      N598UA
##     ---
## 12011: 2015    12  31           4      AA           180      N796AA
## 12012: 2015    12  31           4      B6           524      N934JB
## 12013: 2015    12  31           4      B6           624      N942JB
## 12014: 2015    12  31           4      DL          1262      N394DL
## 12015: 2015    12  31           4      B6          1124      N943JB
##         ORIGIN_AIRPORT DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE
##      1:            LAX                JFK            620      279     2475
##      2:            LAX                JFK            650      274     2475
##      3:            LAX                JFK            650      268     2475
##      4:            LAX                JFK            728      268     2475
##      5:            LAX                JFK            806      277     2475
##     ---
## 12011:            LAX                JFK           1640      259     2475
## 12012:            LAX                JFK           1645      261     2475
## 12013:            LAX                JFK           2107      280     2475
## 12014:            LAX                JFK           2244      256     2475
## 12015:            LAX                JFK           2349      274     2475
##         ARRIVAL_TIME
##      1:         1413
##      2:         1458
##      3:         1436
##      4:         1512
##      5:         1606
##     ---
## 12011:           18
## 12012:           18
## 12013:          513
## 12014:          625
## 12015:          748
```

Additionally we can concatenate multiple condition using the logical OR `|` or AND `&` operator. If we for example

want to inspect all flights departing between 6am and 7am operated by American Airlines we can use the following statement:

```
head(flights[AIRLINE == "AA" & DEPARTURE_TIME > 600 & DEPARTURE_TIME < 700])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          1686      N4XXAA            LAX
## 2: 2015     1   1           4      AA          1361      N3KYAA            BNA
## 3: 2015     1   1           4      AA          2420      N3ERAA            LAX
## 4: 2015     1   1           4      AA           338      N3DJAA            LAX
## 5: 2015     1   1           4      AA          2424      N3DLAA            LAX
## 6: 2015     1   1           4      AA           222      N3JSAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 STL            609      183     1592         1134
## 2:                 LAX            607      255     1797          847
## 3:                 DFW            619      149     1235         1119
## 4:                 SFO            644       55      337          803
## 5:                 DFW            641      146     1235         1149
## 6:                 BOS            650      271     2611         1442
```

4.5 Accessing a data.table by columns (DT[i, j, by])

Once you're inside the `[]`, you're in the data.table environment. Inside this scope, there's no need to put the column names in quotation marks, as columns are seen as variables already.

It is not advisable to access a column by its number. **Use the column name instead**.

The table format can change (new columns can be added), so using column names prevents bugs e.g. if you have a data set with 50 columns, how do you know which one is column 18?

This way the code is more readable: `flights[, TAIL_NUMBER]` instead of `flights[, 7]`.

```
flights[1:10, TAIL_NUMBER]  # Access column x (also DT$x or DT[j=x]).
```

```
##  [1] "N3KUAA" "N3HYAA" "N584UW" "N547US" "N3CTAA" "N76517" "N925SW" "N719SK"
##  [9] "N435SW" "N560SW"
```

```
flights[4, TAIL_NUMBER]  # Access a specific cell.
```

```
## [1] "N547US"
```

When accessing many columns, we probably want to return a data.table instead of a vector. For that, we need to provide R with a list, so we use `list(colA, colB)`, or its simplified version `.(colA, colB)`.

```
# Note that 1 and 3 were coerced into strings because a vector can have only 1
# type
flights[1:2, c(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
## [1] "N3KUAA" "N3HYAA" "LAX"    "LAX"
```

```
# Access a specific subset. Data.frame: DF[1:2, c('x','y')]
flights[1:2, list(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

```
# Same as before.
flights[1:2, .(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
```

```
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

4.5.1 Technical Detail Data.table environment

The following examples are to show how the `[]` bring us inside the data.table environment.

Using DT and DF, compute the product of columns *y* and *v*.

Use the `with(data, expr, ...)` function, which evaluates an R expression in an environment constructed from `data`.

```
# We enter the environment of DT, and simply compute the product
with(DT, y * v)

# Easier way, with [] we're inside the environment. Data.table runs a
# with(dt,...) on the j argument
DT[, y * v]

# In data.frame, the [] doesn't imply we're inside the environment
DF[, y * v]
```

4.6 Basic operations

We saw already that inside the `[]`, columns are seen as variables, so we can apply functions to them.

```
# Similar to mean(flights[, AIR_TIME])
flights[, mean(AIR_TIME, na.rm = TRUE)]
```

```
## [1] 162.1379
```

```
flights[AIRLINE == "OO", mean(AIR_TIME, na.rm = TRUE)]
```

```
## [1] 68.02261
```

To compute operations in multiple columns, we must provide a list (unless we want the result to be a vector).

```
# Same as flights[, .(mean(AIR_TIME), median(AIR_TIME))]
flights[, list(mean(AIR_TIME, na.rm = TRUE), median(AIR_TIME, na.rm = TRUE))]
```

```
##          V1  V2
## 1: 162.1379 150
```

```
# Give meaningful names
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm = TRUE), median_AIR_TIME = median(AIR_TIME,
    na.rm = TRUE))]
```

```
##    mean_AIR_TIME median_AIR_TIME
## 1:      162.1379             150
```

4.6.1 Technical detail We can also apply basic operations on multiple columns

`sapply(DT, FUN)`, applies function FUN column-wise to DT. Remember that `sapply` returns a vector, while `lapply` returns a list.

```
sapply(iris_dt, class)  # Try the same with lapply
```

```
## Error in lapply(X = X, FUN = FUN, ...): object 'iris_dt' not found
```

```
sapply(iris_dt, sum)
```

```
## Error in lapply(X = X, FUN = FUN, ...): object 'iris_dt' not found
```

```
# Note that we can access columns stored as variables by setting with=F.  In this
# case, `colnames(iris_dt)!='Species'` returns a logical vector and ` iris_dt` is
# subseted by the logical vector

# Same as sapply(iris_dt[, 1:4], sum) sapply(iris_dt[,
# colnames(iris_dt)!='Species', with = F], sum)
```

4.7 The 'by' option (DT[i, j, by])

The `by =` option allows us to apply a function to groups wtihin a data.table. For example, we can use the `by =` to compute the mean flight time per airline:

```
# Compute the mean air time of every airline
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME
##  1:      AA    219.48133
##  2:      US    210.39488
##  3:      DL    207.07201
##  4:      UA    211.62008
##  5:      OO     68.02261
##  6:      AS    141.01870
##  7:      B6    309.79568
##  8:      NK    179.55828
##  9:      VX    185.36374
## 10:      WN    105.19976
## 11:      HA    307.95961
## 12:      F9    159.94041
## 13:      MQ    102.15210
```

This way we can easily spot that one airline conducts on average shorter flights.

```
# Compute the mean and standard deviation of the air time of every airline
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm = TRUE), sd_AIR_TIME = sd(AIR_TIME,
    na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME sd_AIR_TIME
##  1:      AA    219.48133   92.889719
##  2:      US    210.39488  105.224833
##  3:      DL    207.07201   88.908566
##  4:      UA    211.62008   94.832456
##  5:      OO     68.02261   41.065036
##  6:      AS    141.01870   51.806424
##  7:      B6    309.79568   28.457740
##  8:      NK    179.55828   78.194706
##  9:      VX    185.36374  113.504572
## 10:      WN    105.19976   69.257334
## 11:      HA    307.95961   23.905491
## 12:      F9    159.94041   61.412379
## 13:      MQ    102.15210    8.531046
```

Although we could write `flights[i = 5, j = AIRLINE]`, we usually ommit the `i =` and `j =` from the syntax, and write `flights[5, ARILINE]` instead. However, for clarity we usually include the `by =` in the syntax.

4.8 Counting occurences (the .N command)

The .N is a special in-built variable that counts the number observations within a table.

Evaluating .N alone is equal to nrow() of a table.

```
flights[, .N]
```

```
## [1] 389369
```

```
nrow(flights)
```

```
## [1] 389369
```

But the .N command becomes a lot more powerful when used with grouping or conditioning. We already saw earlier how we can use it to count the number of occurrences of elements in categorical columns.

```
# Get the number of flights for each AIRLINE
flights[, .N, by = "AIRLINE"]
```

```
##     AIRLINE     N
##  1:      AA 65483
##  2:      US  7374
##  3:      DL 50343
##  4:      UA 54862
##  5:      OO 73389
##  6:      AS 16144
##  7:      B6  8216
##  8:      NK  8688
##  9:      VX 23598
## 10:      WN 75022
## 11:      HA  3112
## 12:      F9  2770
## 13:      MQ   368
```

Remembering the data.table definition: "Take **DT**, subset rows using **i**, then select or calculate **j**, grouped by **by**", we can build even more powerful statements using all three elements.

```
# For each airline, get the number of observations to NYC (JFK)
flights[DESTINATION_AIRPORT == "JFK", .N, by = "AIRLINE"]
```

```
##    AIRLINE    N
## 1:      B6 2488
## 2:      DL 2546
## 3:      AA 3804
## 4:      VX 1652
## 5:      UA 1525
```

4.9 Creating new columns (the := command)

The := operator updates the data.table you are working on, so writing DT <- DT[,... := ...] is redundant. This operator, plus all set functions (TODO clarify), change their input by *reference*. No copy of the object is made, that is why it is faster and uses less memory.

```
# Add a new column called SPEED whose value is the DISTANCE divided by AIR_TIME
flights[, `:=`(SPEED, DISTANCE/AIR_TIME * 60)]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
```

```
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
## 6: 2015     1   1           4      UA          1545      N76517            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Having computed a new column using the := operator we can use it for further analyses.

```
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm = TRUE), mean_SPEED = mean(SPEED,
    na.rm = TRUE), mean_DISTANCE = mean(DISTANCE, na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIR_TIME mean_SPEED mean_DISTANCE
##  1:      AA     219.48133   461.2839     1739.2331
##  2:      US     210.39488   452.1641     1658.2581
##  3:      DL     207.07201   466.0330     1656.2165
##  4:      UA     211.62008   464.2928     1693.5504
##  5:      OO      68.02261   349.5549      437.2337
##  6:      AS     141.01870   439.0120     1040.0340
##  7:      B6     309.79568   484.8242     2486.1489
##  8:      NK     179.55828   450.0221     1402.1591
##  9:      VX     185.36374   433.0870     1432.5384
## 10:      WN     105.19976   409.3803      760.2593
## 11:      HA     307.95961   497.3118     2537.8107
## 12:      F9     159.94041   461.0684     1235.6664
## 13:      MQ     102.15210   435.5580      737.0000
```

Now we can see that the flights by the carrier “OO” are not just shorter, but also slow. This could for example lead us to the hypothesis, that “OO” is a small regional carrier, which operates slower planes.

Additionally we can use the := operator to remove columns. If we for example observe that tail numbers are not important for our analysis we can remove them with the following statement:

```
flights[, `:=`(TAIL_NUMBER, NULL)]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336            LAX
## 2: 2015     1   1           4      AA           258            LAX
## 3: 2015     1   1           4      US          2013            LAX
## 4: 2015     1   1           4      DL          1434            LAX
## 5: 2015     1   1           4      AA           115            LAX
## 6: 2015     1   1           4      UA          1545            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Here we observe, that the tail numbers are gone from the data.table

4.9.1 Technical Detail Multiple assignments

With the following syntax we can assign multiple new columns at once.

```
# Add columns with sepal and petal area. Note the syntax of multiple assignment.
# iris_dt[, `:=` (Sepal.Area = Sepal.Length * Sepal.Width, Petal.Area =
# Petal.Length * Petal.Width)][1:3]
```

You can also delete columns by using the `:=` command.

```
# Let's assume setosa flowers are orange, versicolor purple and virginica pink.
# Add a column with these colors. iris_dt[Species == 'setosa', color := 'orange']
# iris_dt[Species == 'versicolor', color := 'purple'] iris_dt[Species ==
# 'virginica', color := 'pink'] unique(iris_dt[, .(Species, color)])

# We can delete this new column by setting it to NULL iris_dt[, color := NULL]
# colnames(iris_dt)
```

4.10 By reference

What do we mean when we say that data.table modifies columns *by reference*? It means that no new copy of the object is made in the memory, unless we actually create one using `copy()`.

```
or_dt <- data.table(a = 1:10, b = 11:20)
# No new object is created, both new_dt and or_dt point to the same memory chunk.
new_dt <- or_dt
new_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # or_dt was also affected by changes in new_dt
```

```
## [1] "a"  "b"  "ab"
```

```
or_dt <- data.table(a = 1:10, b = 11:20)
copy_dt <- copy(or_dt)  # By creating a copy, we have 2 objects in memory
copy_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # Changes in the copy don't affect the original
```

```
## [1] "a" "b"
```

4.11 Merging

Merge, join or combine 2 data tables into one by common column(s).

`merge(x, y, by, by.x, by.y, all, all.x, all.y, ...)`

In data.table, the default is to merge by shared *key* columns while in data.frame, the default is to merge by columns with the same name. We can also specify the columns to merge `by`.

There are different types of merging:

- **Inner (default)**: consider only rows with matching values in the key columns.
- **Outer or full**: return all rows and columns from x and y. If there are no matching values, return NAs.
- **Left (all.x)**: consider all rows from x, even if they have no matching row in y.
- **Right (all.y)**: consider all rows from y, even if they have no matching row in x.

4.11.1 Merging - example

If we for example want to know how the airports are called behind the IATA_CODE codes, we can load another tabe containing additional information about the airports.

```
airports <- fread("path_to_file/airports.csv")
```

```
head(airports)
```

```
##    IATA_CODE                              AIRPORT        CITY STATE COUNTRY
## 1:       ABE Lehigh Valley International Airport   Allentown    PA     USA
## 2:       ABI             Abilene Regional Airport     Abilene    TX     USA
## 3:       ABQ   Albuquerque International Sunport Albuquerque    NM     USA
## 4:       ABR            Aberdeen Regional Airport    Aberdeen    SD     USA
## 5:       ABY  Southwest Georgia Regional Airport      Albany    GA     USA
## 6:       ACK           Nantucket Memorial Airport   Nantucket    MA     USA
##    LATITUDE  LONGITUDE
## 1: 40.65236  -75.44040
## 2: 32.41132  -99.68190
## 3: 35.04022 -106.60919
## 4: 45.44906  -98.42183
## 5: 31.53552  -84.19447
## 6: 41.25305  -70.06018
```

```
head(merge(flights, airports, by.x = "DESTINATION_AIRPORT", by.y = "IATA_CODE"))
```

```
##    DESTINATION_AIRPORT YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER
## 1:                 ABQ 2015     1   1           4      OO          2575
## 2:                 ABQ 2015     1   1           4      WN          2077
## 3:                 ABQ 2015     1   1           4      OO          2623
## 4:                 ABQ 2015     1   1           4      WN          2520
## 5:                 ABQ 2015     1   1           4      OO          6474
## 6:                 ABQ 2015     1   1           4      WN          4721
##    ORIGIN_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:            LAX            921       87      677         1214 466.8966
## 2:            LAX            943       88      677         1232 461.5909
## 3:            LAX           1449       89      677         1750 456.4045
## 4:            LAX           1851       90      677         2134 451.3333
## 5:            LAX           1951      100      677         2256 406.2000
## 6:            LAX           2041       88      677         2328 461.5909
##                              AIRPORT        CITY STATE COUNTRY LATITUDE
## 1: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 2: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 3: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 4: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 5: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 6: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
##    LONGITUDE
## 1: -106.6092
## 2: -106.6092
## 3: -106.6092
## 4: -106.6092
## 5: -106.6092
## 6: -106.6092
```

4.11.2 Merging - inner and outer examples

Below we created some artificial tables to showcase the differences between the different types of merges.

```
dt1 <- data.table(table = "table1", id = c(1, 2, 3), a = rnorm(3))
dt2 <- data.table(table = "table2", id = c(1, 2, 4), b = rnorm(3))
```

```
# Inner merge: default one, all = FALSE
merge(dt1, dt2, by = "id", all = F)
```

```
##    id table.x           a table.y          b
## 1:  1  table1 -0.05219929  table2  0.7553791
## 2:  2  table1 -1.01967524  table2 -0.4714429
```

```
# Outer (full) merge: all = TRUE
merge(dt1, dt2, by = "id", all = T)
```

```
##    id table.x           a table.y            b
## 1:  1  table1 -0.05219929  table2  0.755379084
## 2:  2  table1 -1.01967524  table2 -0.471442931
## 3:  3  table1 -0.00330542    <NA>           NA
## 4:  4    <NA>          NA  table2  0.002540745
```

Note that the column order got changed after the merging.

```
# Left merge: all.x = TRUE
merge(dt1, dt2, by = "id", all.x = T)[]
```

```
##    id table.x           a table.y          b
## 1:  1  table1 -0.05219929  table2  0.7553791
## 2:  2  table1 -1.01967524  table2 -0.4714429
## 3:  3  table1 -0.00330542    <NA>         NA
```

```
# Right merge: all.y = TRUE
merge(dt1, dt2, by = "id", all.y = T)[]
```

```
##    id table.x           a table.y            b
## 1:  1  table1 -0.05219929  table2  0.755379084
## 2:  2  table1 -1.01967524  table2 -0.471442931
## 3:  4    <NA>          NA  table2  0.002540745
```

4.12 Summary

By now, you should be able to answer the following questions:

- Why do we say that data.table is an enhanced data.frame?
- How to subset by rows or columns? Remember: DT[i, j, by].
- How to add columns?
- How to make operations with different columns?
- Which are the different types of merging?

Even if you were able to answer them, practice:

- Check all vignettes and reference manual from https://cran.r-project.org/web/packages/data.table/
- A really concise cheat sheet:
- https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

4.13 Data.table resources

https://cran.r-project.org/web/packages/data.table/

https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

http://r4ds.had.co.nz/relational-data.html

http://adv-r.had.co.nz/Environments.html

Chapter 5

Tidy data

This chapter is partially adopted from "Introduction to Data Science" by Rafael A. Irizarry (https://rafalab.github.io/dsbook/).

We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations. The `murders` dataset is an example of a tidy dataset.

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

To see how the same information can be provided in different formats, consider the following example:

This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the **dslabs** package. Originally, the data was in the following format:

The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header.

Although not immediately obvious, as you go through the script you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

5.1 Tidy data definition

One can briefly summarize the tidy definition in the three following statements:

1. Each **variable** must have its own **column**.
2. Each **observation** must have its own **row**.
3. Each **value** must have its own **cell**.

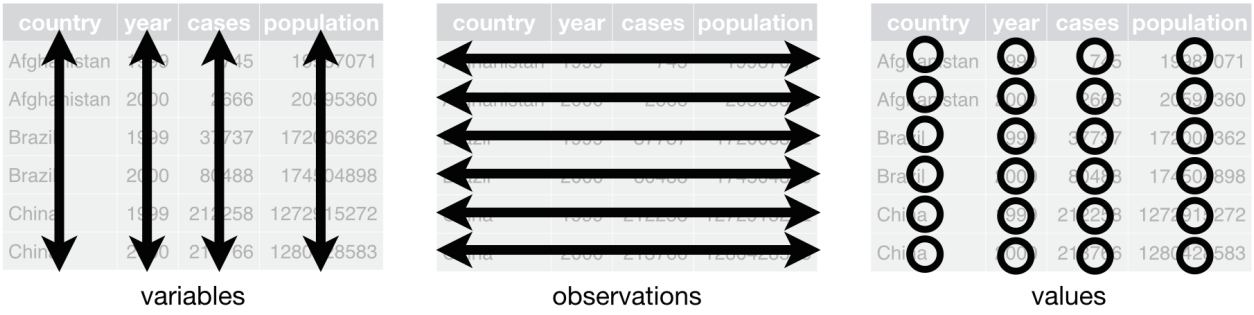


Figure 5.1: **Tidy data table layout.** Each variable has a column, each observation a row and each value a cell.

5.2 Common signs of untidy datasets

On the other hand one can often quickly identify untidy datasets by one or more of the following statements:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

5.3 Advantages of tidy data

- Easier manipulation using data.table commands
- sub-setting by rows
- sub-setting by columns
- `by` operations
- Many other tools work better with tidy data - consistent way of storing data
- example: ggplot2
- Vectorized operations become easier to use

Tidy data can be easily manipulated

```
head(dt)
```

```
##        country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

For example in the table above we can easily compute the rate of cases within the population or the number of cases per year using the following commands:

```
# Compute rate per 10,000
dt[, `:=`(rate, cases/population * 10000)]  # vectorized operations; dt is modified
head(dt)
```

```
##        country year  cases population     rate
```

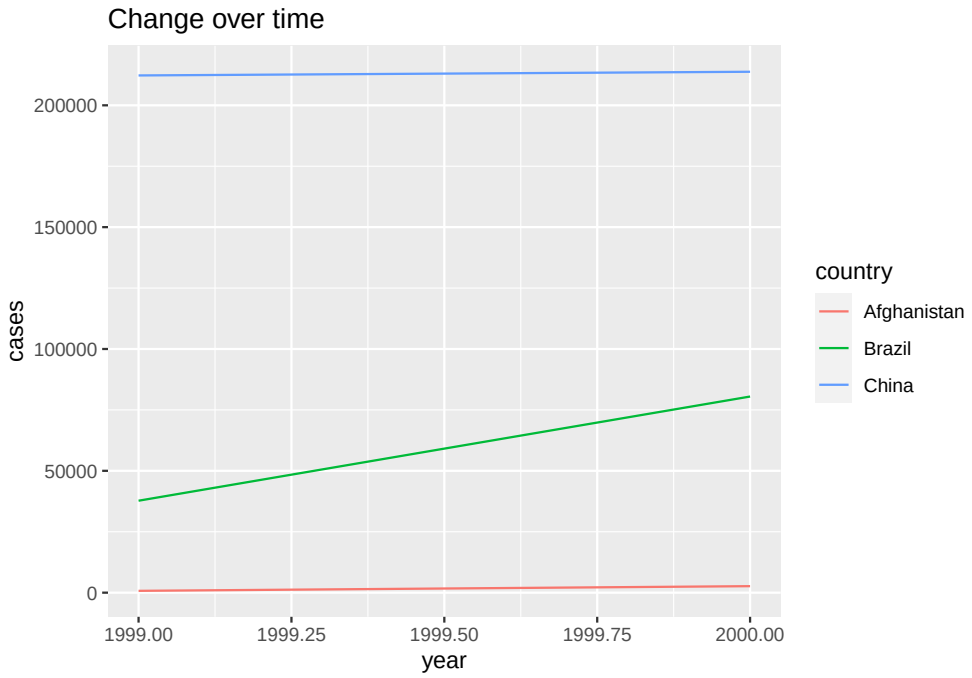
```
## 1: Afghanistan 1999    745   19987071 0.372741
## 2: Afghanistan 2000   2666   20595360 1.294466
## 3:      Brazil 1999  37737  172006362 2.193930
## 4:      Brazil 2000  80488  174504898 4.612363
## 5:       China 1999 212258 1272915272 1.667495
## 6:       China 2000 213766 1280428583 1.669488
```

```
# Compute cases per year
dt[, .(cases = sum(cases)), by = year]   # note that this does not modify dt
```

```
##    year  cases
## 1: 1999 250740
## 2: 2000 296920
```

Additionally, tidy data works better with many packages like ggplot2 which we are going to use in this course.

```
ggplot(dt, aes(year, cases, color = country)) + ggtitle("Change over time") + geom_line()
```



In the remaining part of this chapter we will learn how to transform untidy datasets into tidy ones using data.table functions.

5.3.1 Column headers are values, not variable names

Untidy: 1999 and 2000 are values of the variable *year*.

```
table4a
```

```
##       country   1999   2000
## 1: Afghanistan    745   2666
## 2:      Brazil  37737  80488
## 3:       China 212258 213766
```

This can be solved by using the **data.table** function `melt()`. See the code below. When melting all values in all columns specified in the `measure.vars` argument are gathered into one column whose name can be specified using the `value.name` argument. Additionally a new column named using the argument `variable.name` is created containing all the values which where previously stored in the column names.

Figure 5.2: Melting the country dataset

```
melt(table4a, id.vars = "country", measure.vars = c("1999", "2000"), variable.name = "year",
    value.name = "cases")
```

```
##        country year  cases
## 1: Afghanistan 1999    745
## 2:      Brazil 1999  37737
## 3:       China 1999 212258
## 4: Afghanistan 2000   2666
## 5:      Brazil 2000  80488
## 6:       China 2000 213766
```

```
# would work also without specifying *either* measure.vars OR id.vars
```

5.3.2 Multiple variables are stored in one column

Untidy: multiple variables are stored in the *count* column.

```
table2
```

```
##         country year       type      count
##  1: Afghanistan 1999      cases        745
##  2: Afghanistan 1999 population   19987071
##  3: Afghanistan 2000      cases       2666
##  4: Afghanistan 2000 population   20595360
##  5:      Brazil 1999      cases      37737
##  6:      Brazil 1999 population  172006362
##  7:      Brazil 2000      cases      80488
##  8:      Brazil 2000 population  174504898
##  9:       China 1999      cases     212258
## 10:       China 1999 population 1272915272
## 11:       China 2000      cases     213766
## 12:       China 2000 population 1280428583
```

This problem can be solved using the dcast function. Here is the help of that function:

```
## Help
dcast(data, formula, fun.aggregate = NULL, sep = "_", ..., margins = NULL, subset = NULL,
    fill = NULL, drop = TRUE, value.var = guess(data), verbose = getOption("datatable.verbose"))
```

To use it for our case we only need to specify which column is the key in the "formula" and which value should be spread.

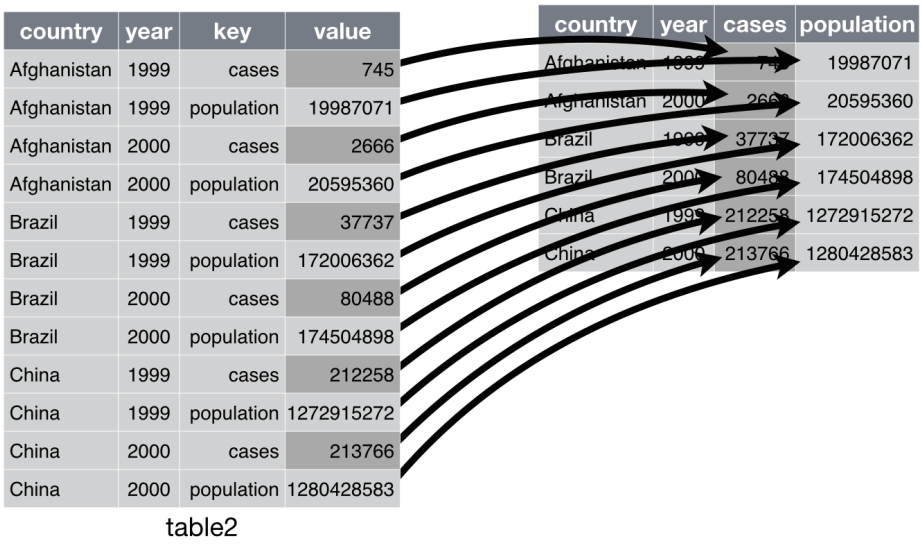


Figure 5.3: Decasting the country dataset

```
dcast(table2, ... ~ type, value.var = "count")
```

```
##        country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

5.4 Separating and Uniting (1 <-> more variables)

Typical problem:

1. One column contains multiple variables
2. Multiple columns contain one variable

Untidy datasets

```
## One column contains multiple variables
print(table3)
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
## Multiple columns contain one variable
print(table5)
```

```
##        country century year             rate
```

```
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```

Solution in R:

1) variable -> multiple variables
   - `tidyr::separate()`
2) multiple variables -> 1 variables
   - `tidyr::unite()`

other useful functions:

- `data.table::tstrsplit`, `strsplit`, `paste`, `substr`

To separate 1 variable to multiple variables we use `tidyr::separate()`.

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE,
    extra = "warn", fill = "warn", ...)
```

```
table3
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
separate(table3, col = rate, into = c("cases", "population"))
```

```
##        country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

```
separate(table3, col = rate, into = c("cases", "population")) %>% class
```

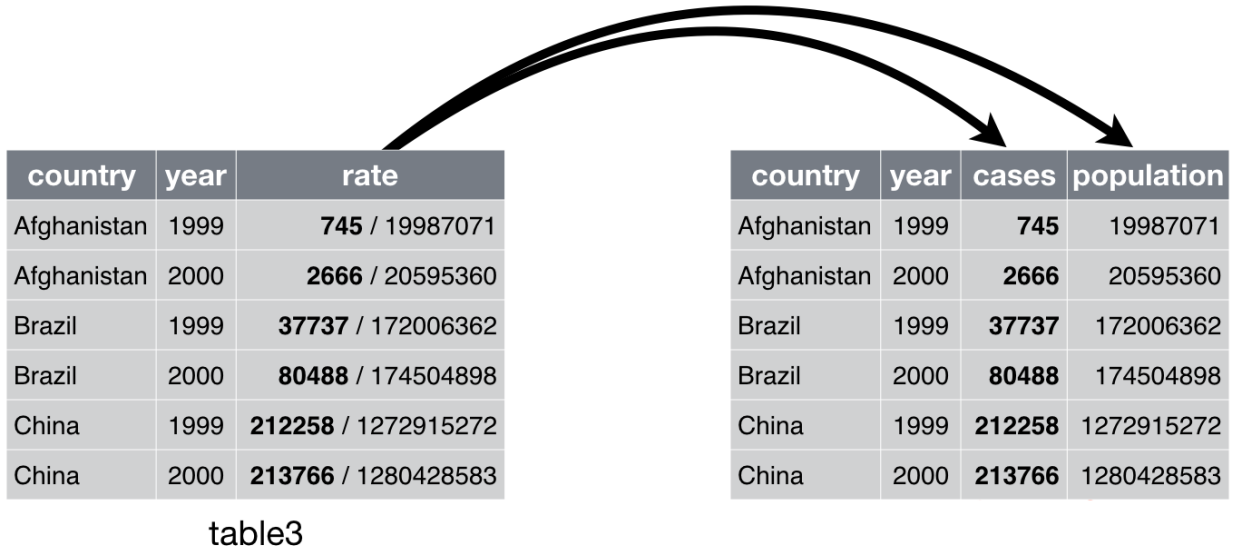
```
## [1] "data.table" "data.frame"
```

To unite multiple variables to 1 variable we use `tidyr::unite()`.

```
unite(data, col, ..., sep = "_", remove = TRUE)
```

```
table5
```

```
##        country century year              rate
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```



country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

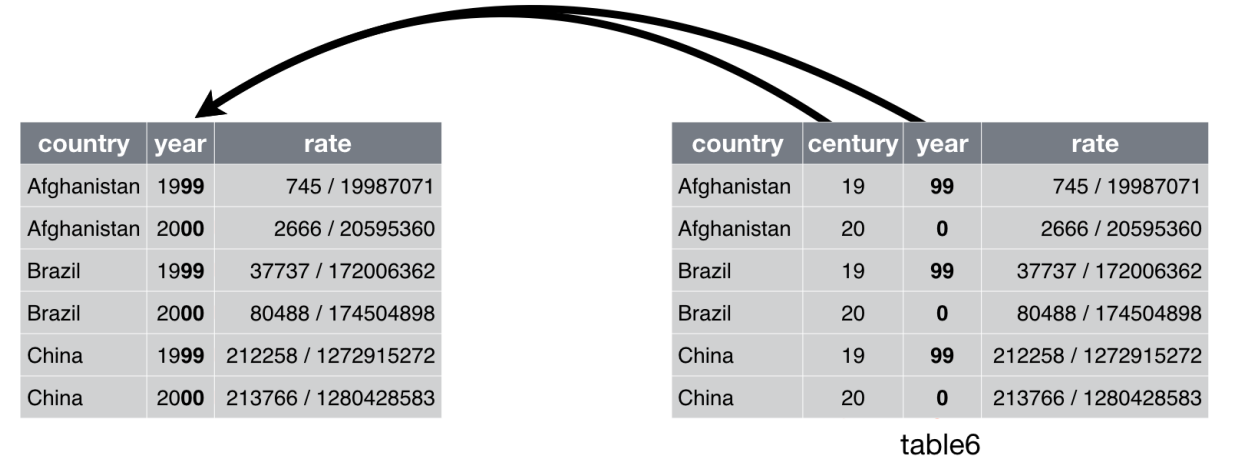
table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 5.4: Seperated country dataset

```
unite(table5, col = new, century, year, sep = "")
```

```
##        country  new              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```



country	year	rate
Afghanistan	19**99**	745 / 19987071
Afghanistan	20**00**	2666 / 20595360
Brazil	19**99**	37737 / 172006362
Brazil	20**00**	80488 / 174504898
China	19**99**	212258 / 1272915272
China	20**00**	213766 / 1280428583

country	century	year	rate
Afghanistan	19	**99**	745 / 19987071
Afghanistan	20	**0**	2666 / 20595360
Brazil	19	**99**	37737 / 172006362
Brazil	20	**0**	80488 / 174504898
China	19	**99**	212258 / 1272915272
China	20	**0**	213766 / 1280428583

table6

Figure 5.5: United country dataset

5.4.0.0.1 List of data.tables

```
split(table1, table1$country) %>% lapply(. %>% subset(select = -country) %>% as.data.table)
```

```
## $Afghanistan
##    year cases population     rate
## 1: 1999   745   19987071 0.372741
## 2: 2000  2666   20595360 1.294466
##
## $Brazil
##    year cases population     rate
## 1: 1999 37737  172006362 2.193930
## 2: 2000 80488  174504898 4.612363
##
## $China
##    year  cases population     rate
## 1: 1999 212258 1272915272 1.667495
## 2: 2000 213766 1280428583 1.669488
```

To sum up:

- In a tidy dataset, each variable must have its own column
- Each row corresponds to one unique observation
- Each cell contains a single value
- Tidy datasets are easier to work with
- Data.table library has functions to transform untidy datasets to tidy

Words to live by

Happy families are all alike; every unhappy family is unhappy in its own way.

- Leo Tolstoy

Tidy datasets are all alike, but every messy dataset is messy in its own way.

- Hadley Wickham

5.5 Non-tidy data

- Performance advantage using certain functions
  - `colSums()` or `heatmap()` on matrices
- Field convention
- Memory efficiency
  - don't worry, you should be fine with tidy-data in `data.table`

Interesting blog post:

- http://simplystatistics.org/2016/02/17/non-tidy-data/

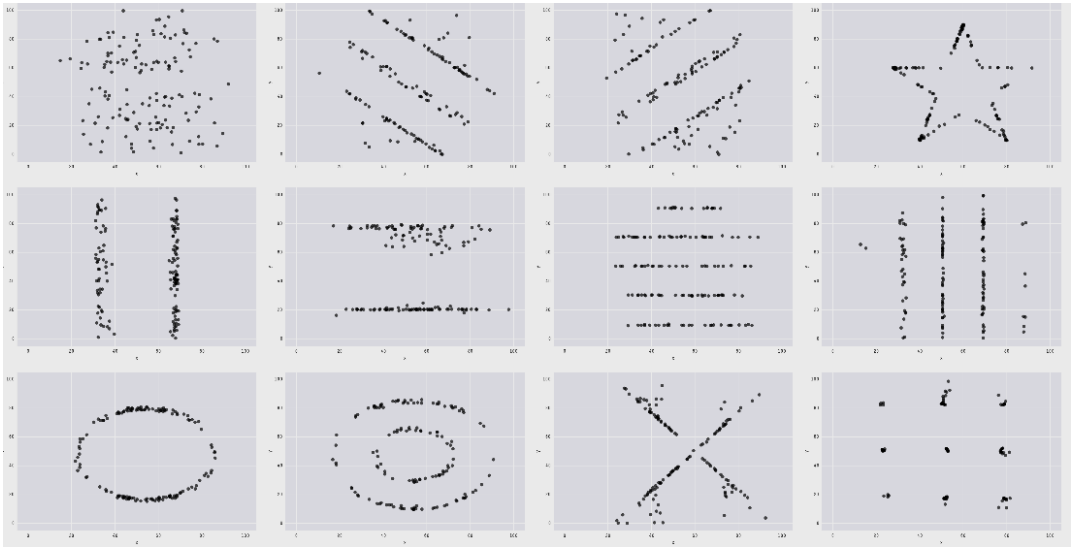
Chapter 6

Low dimensional visualizations

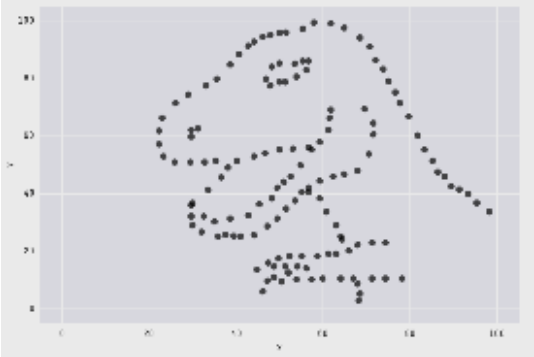
6.1 Why plotting?

With the increasing amounts of data, it's become increasingly difficult to manage and make sense of it all. It becomes nearly impossible for a single person to go through data line-by-line and see distinct patterns and make observations. In this context, data visualization becomes crucial for gaining insight into data that traditional descriptive statistics cannot. For instance, plotting can give hints about bugs in our code (or even in the data!) and can help us to develop and improve methods and models.

Do you see a pattern in these plots?



Maybe now?



All those plots, including the infamous datasaurus, actually share the same statistics!

```
X Mean: 54.26
Y Mean: 47.83
X SD  : 16.76
Y SD  : 26.93
Corr. : -0.06
```

When only looking at the statistics, we would have probably wrongly assumed that the datasets were identical. This illustration highlights why it is important to visualize data and not just rely on descriptive statistics.

We can consider another example given by a vector `height` containing (hypothetical) height measurements for 500 adults in Germany:

```
length(height)
```

```
## [1] 500
```

```
head(height, n = 20)
```

```
##  [1] 1.786833 1.715319 1.789841 1.787259 1.748659 1.660702 1.688214 1.716738
##  [9] 1.729740 1.838209 1.764362 1.694045 1.678355 1.716974 1.716535 1.711482
## [17] 1.741350 1.689864 1.749606 1.674311
```

Calculating the mean height returns the following output:

```
mean(height)
```

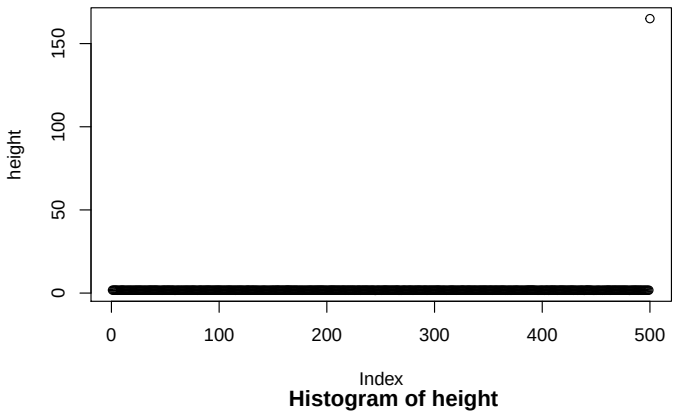
```
## [1] 2.056583
```

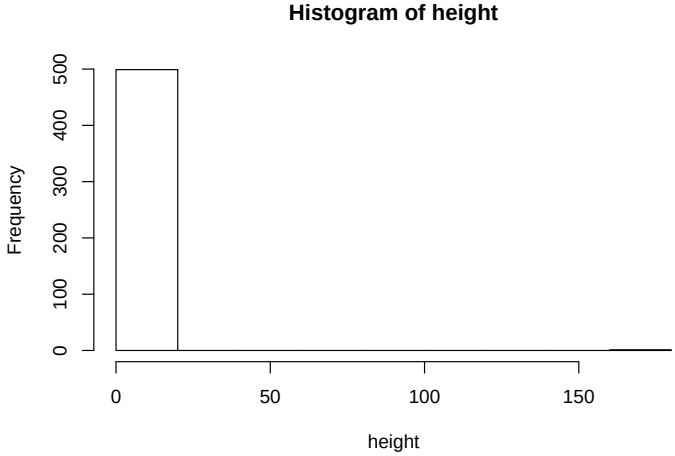
Wait... what?

What happened?

Assuming that adults in Germany are actually not exceptionally tall, we can plot the data for investigating this situation.

```
plot(height)
hist(height)
```





Interestingly, there is an outlier in our data! One particular person seems to have a height above 150 meters that is obviously wrong. As a result, it inflates our mean giving us the false impression that Germans are exceptionally tall.

A quick way to fix our dataset is to remove our outlier.

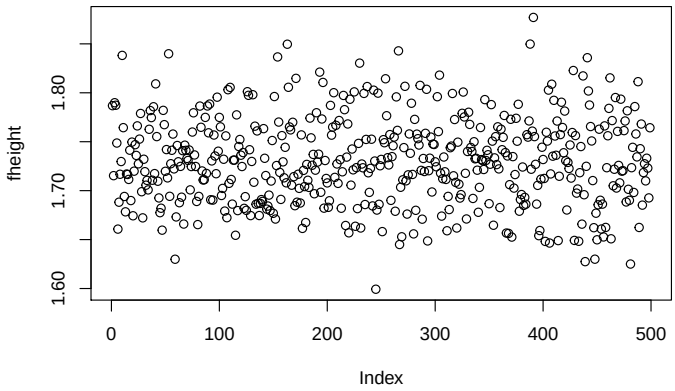
```
fheight <- height[height < 3]
```

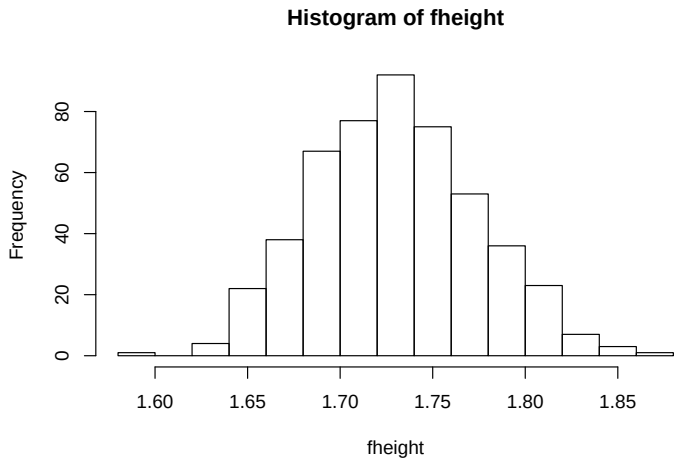
Now our plotted data seems more realistic and the mean height makes sense.

```
mean(fheight)
```

```
## [1] 1.730043
```

```
plot(fheight)
hist(fheight)
```





6.2 Grammar of graphics

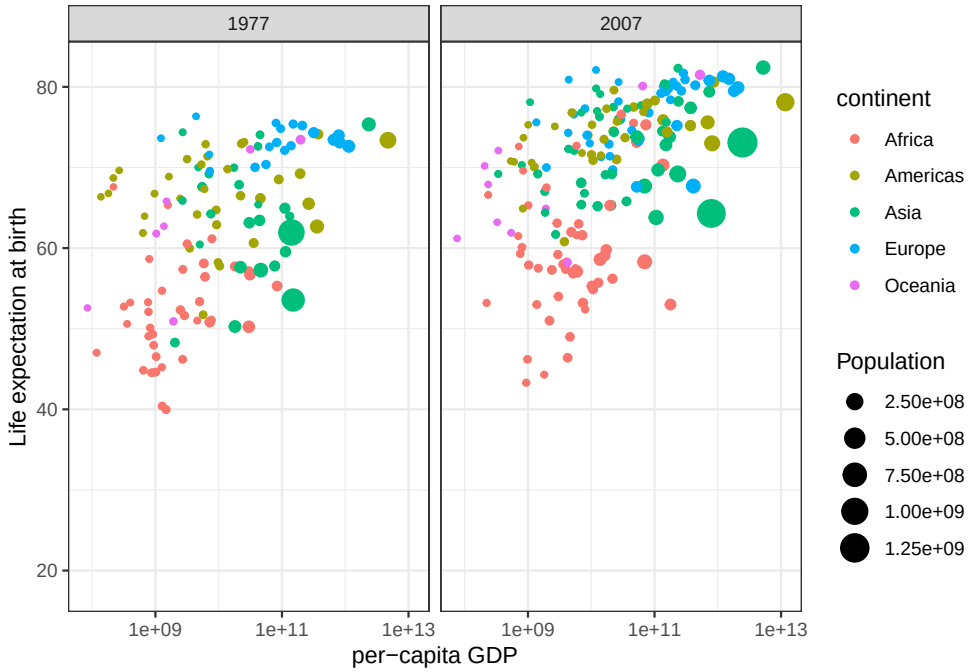
Plotting with the help of the package `ggplot2` has become widely used by R programmers. The plotting logic of `ggplot2` is known as the grammar of graphics. The grammar of graphics is a visualization theory developed by Leland Wilkinson in 1999. It has influenced the development of graphics and visualization libraries alike and is based on 3 key principles:

- Separation of data from aesthetics (e.g. x and y-axis, color-coding)
- Definition of common plot/chart elements (e.g. scatter plots, box-plots, etc.)
- Composition of these common elements (one can combine elements as layers)

Now let's try to create a sophisticated example based on the R package `ggplot2` for visualizing the relationship of between the per-capita GDP `gdp` and the life expectancy at birth `life_expectancy` for every country from the dataset `gapminder`. We want to compare this relationship for the years 1977 and 2007:

```
# install.packages('gapminder')
library(gapminder)
gm_dt <- as.data.table(gapminder)[year %in% c(1977, 2007)]

ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population)) + facet_grid(~year) + scale_x_log10() + labs(y = "Life expectation at birth",
    x = "per-capita GDP", size = "Population") + mytheme
```



We may, for instance, use such visualization to find differences in the life expectancy of each country and each continent.

Now... how do we create such a sophisticated plot step by step? First, we will introduce the major components of grammar of graphics that will allow us to create and understand the previous example.

6.2.1 Major components of the grammar of graphics

The following components are considered in the context of the grammar of graphics:

Data: `data.table` (or `data.frame`) object where columns correspond to variables

Aesthetics: visual characteristics that represent data (`aes`) - e.g. position, size, color, shape, transparency, fill

Layers: geometric objects that represent data (`geom_`) - e.g. points, lines, polygons, ...

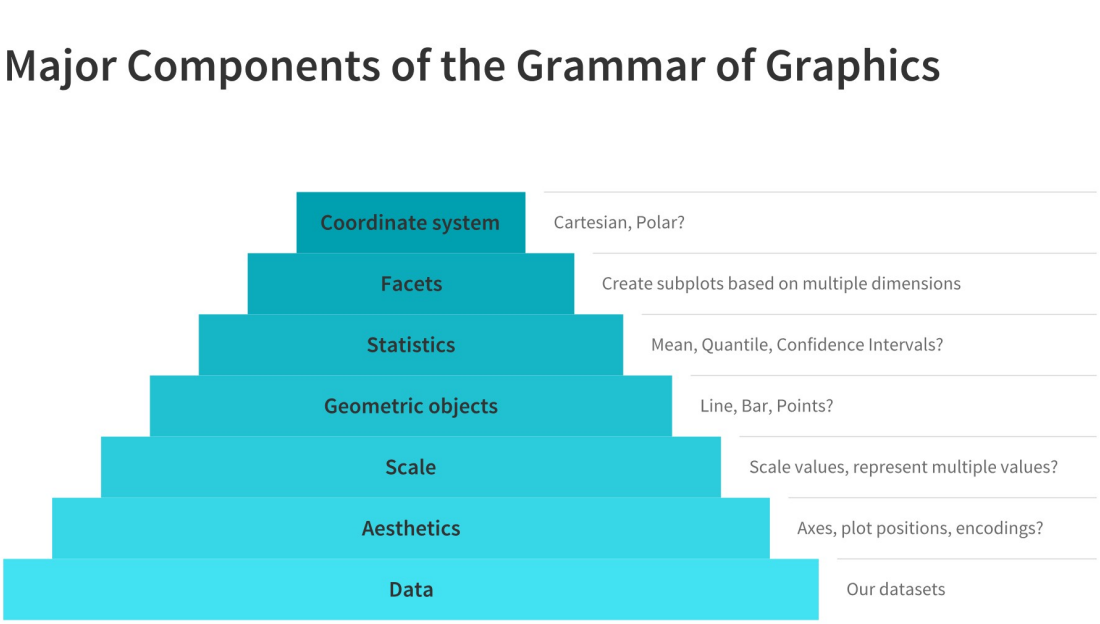
Scales: for each aesthetic, describes how visual characteristic is converted to display values (`scale_`) - e.g. log scales, color scales, size scales, shape scales, ...

Facets: describes how data is split into subsets and displayed as multiple subgraphs (`facet_`)

Stats: statistical transformations that typically summarize data (`stat`) - e.g. counts, means, medians, regression lines, ...

Coordinate system: describes 2D space that data is projected onto (`coord_`) - e.g. Cartesian coordinates, polar coordinates, map projections, ...

The following illustration represents the abstraction of grammar of graphics:



6.2.2 Defining the data and layers

In our example, we consider the `gapminder` dataset, which serves as the data component of our visualization. First, we have a quick look at the data. We want to plot the variable `life_expectancy` against the variable `gdp` so we better have a look at the first lines of the data:

```
head(gm_dt[, .(country, continent, gdp, life_expectancy, year)])
```

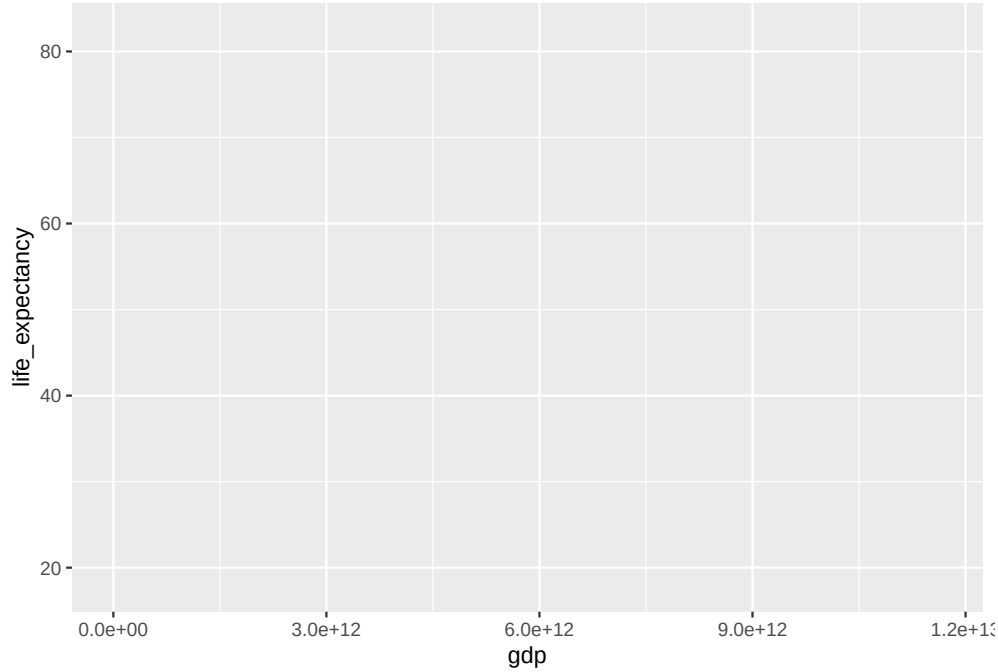
```
##                country continent          gdp life_expectancy year
## 1:              Albania    Europe           NA           70.54 1977
## 2:              Algeria    Africa  29829518900           57.13 1977
## 3:               Angola    Africa           NA           45.12 1977
## 4: Antigua and Barbuda  Americas    268265361           69.64 1977
## 5:            Argentina  Americas 193500979043           69.24 1977
## 6:              Armenia      Asia           NA           72.44 1977
```

For starting with the visualization we initiate a `ggplot` object which generates a plot with background:

```
ggplot()
```

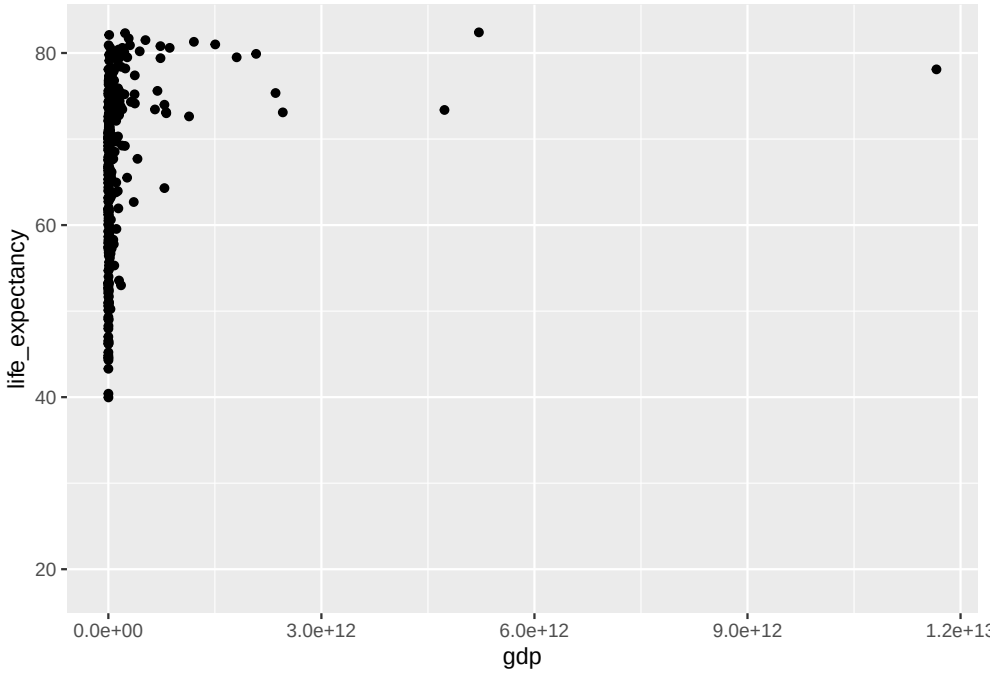
Next, we can define the data to be plotted, which needs to be a `data.table` (or `data.frame`) object and the `aes()` function. This `aes()` function defines which columns in the `data.table` object map to `x` and `y` coordinates and if they should be colored or have different shapes and sizes based on the values in a different column. These elements are called "aesthetic" elements, which we observe in the plot.

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy))
```



As we can see, we obtain a plot with labeled axis and ranges. We want to visualize the data with a simple **scatter plot**. In a scatter plot, the values of two variables are plotted along two axes. Each pair of values is represented as a point. In R, a scatter plot can be plotted with `ggplot2` using the function `geom_point`. We want to construct a scatter plot containing the `gdp` on the x-axis and the `life_expectancy` on the y-axis. For this we combine the function `geom_point()` to the previous line of code with the operator `+`:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point()
```



One of the advantages of plotting with `ggplot` is that it returns an object which can be stored (e.g. in a variable called `p`). The stored object can be further edited.

```
p <- ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point()
```

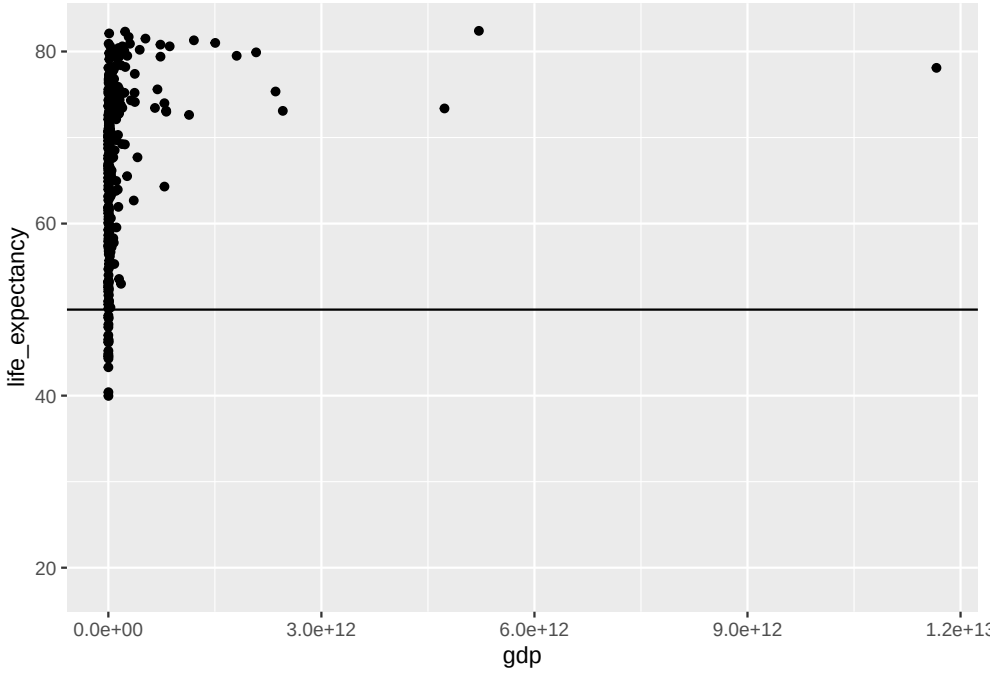
We can inspect the names of elements of the stored object with `names()`:

```
names(p)
```

```
## [1] "data"        "layers"      "scales"      "mapping"     "theme"
## [6] "coordinates" "facet"       "plot_env"    "labels"
```

We can also save the `ggplot` object with the help of the function `saveRDS()`. Then, we can read the saved object again with the help of the function `readRDS()` and add a horizontal line at `y=50` to the plot:

```
saveRDS(p, "../extdata/my_first_plot.rds")
p <- readRDS("../extdata/my_first_plot.rds")
p + geom_hline(yintercept = 50)
```

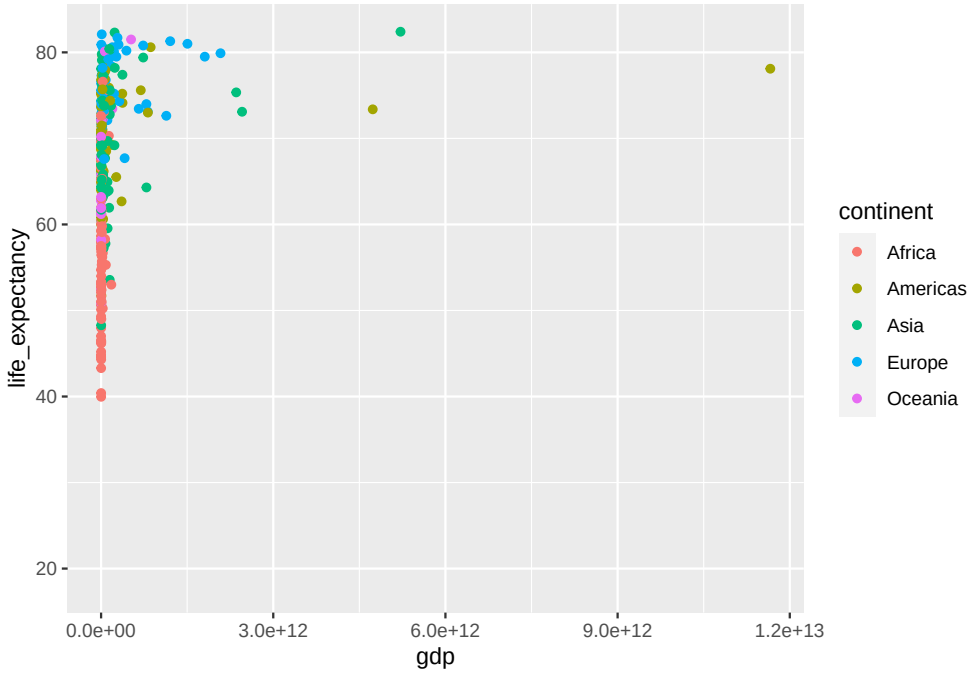


6.2.3 Mapping of aesthetics

6.2.3.1 Mapping of color, shape and size

We can easily map variables to different colors, sizes or shapes depending on the value of the specified variable. To assign each point to its corresponding continent, we can define the variable `continent` as the `color` attribute in `aes()` as follows:

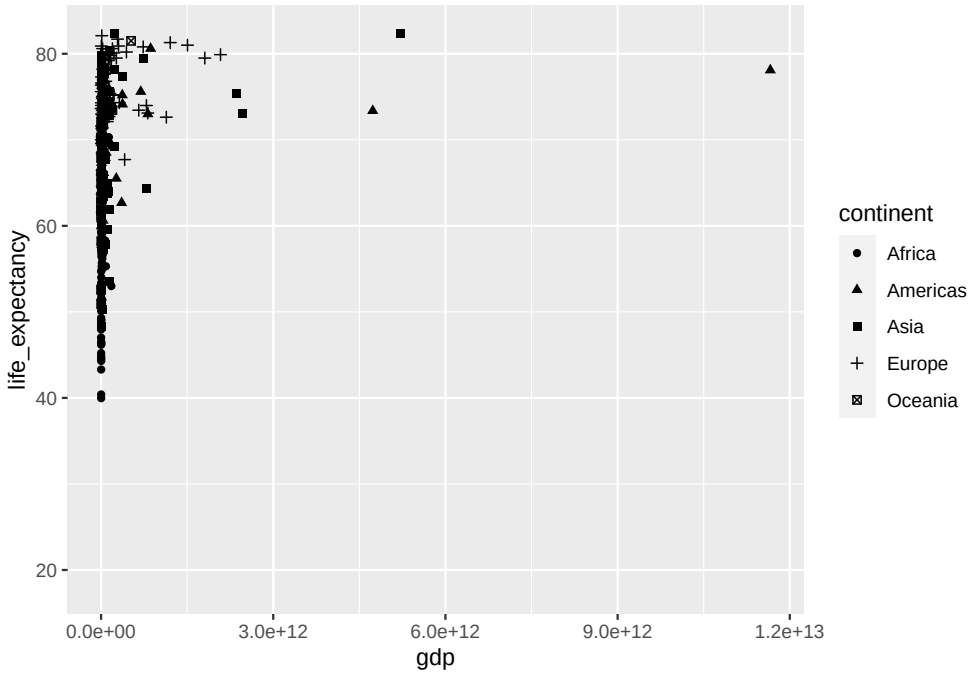
```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent)) + geom_point()
```



American `color` or British `colour` are both acceptable as the argument specification.

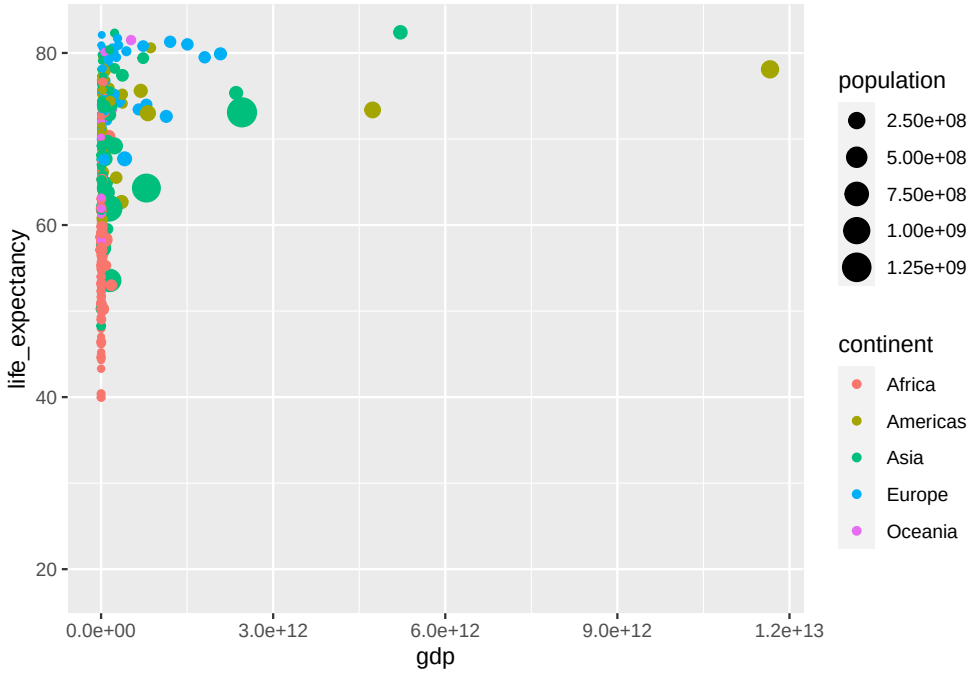
Instead of color, we can also use different shapes for characterizing the different continents in the scatter plot. For this we specify the `shape` argument in `aes()` as follows:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, shape = continent)) + geom_point()
```



Additionally, we distinguish the population of each country by giving a size to the points in the scatter plot:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent, size = population)) +
    geom_point()
```

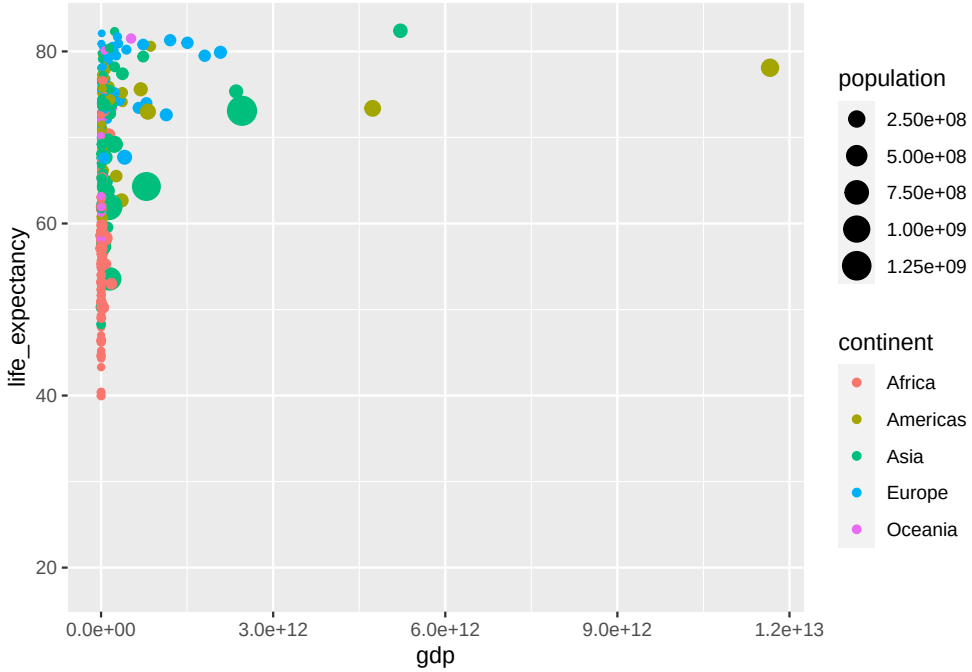


6.2.3.2 Global versus individual mapping

Mapping of aesthetics in `aes()` can be done globally or at individual layers.

For instance, in the previous plot we define the variables `gdp` and `life_expectancy` in the `aes()` function inside `ggplot()` for a **global** definition. Global mapping is inherited by default to all geom layers (`geom_point` in the previous example), while mapping at **individual** layers is only recognized at that layer. For example, we define the `aes(x=gdp, y=life_expectancy)` globally, but the color attributes only locally for the layer `geom_point`:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population))
```

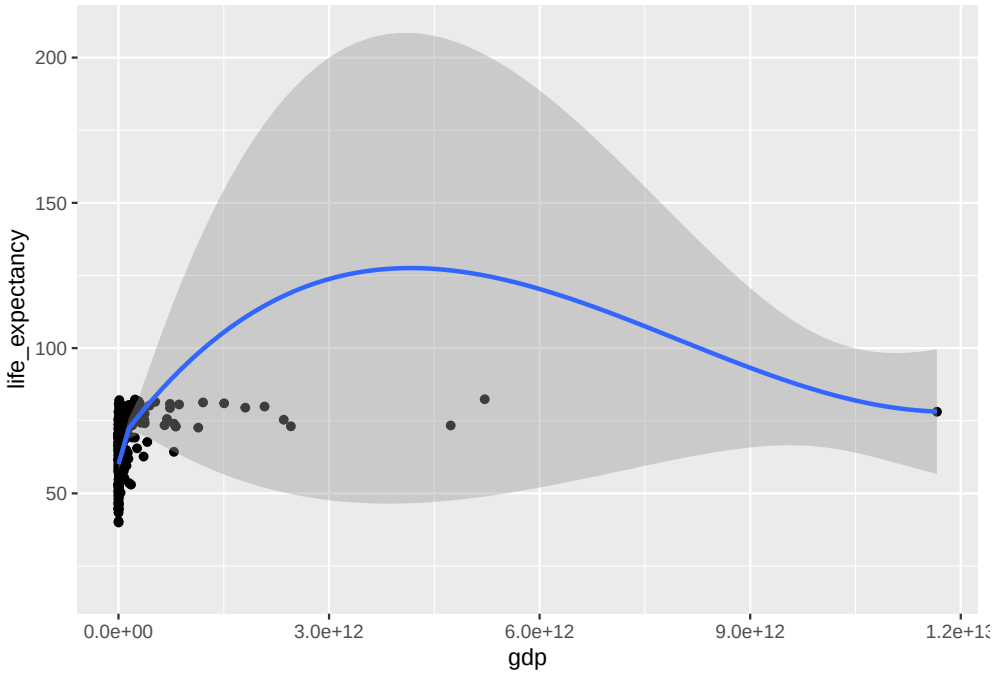


Note that individual layer mapping cannot be recognized by other layers. For instance, we can add another layer for smoothing with `stat_smooth()`.

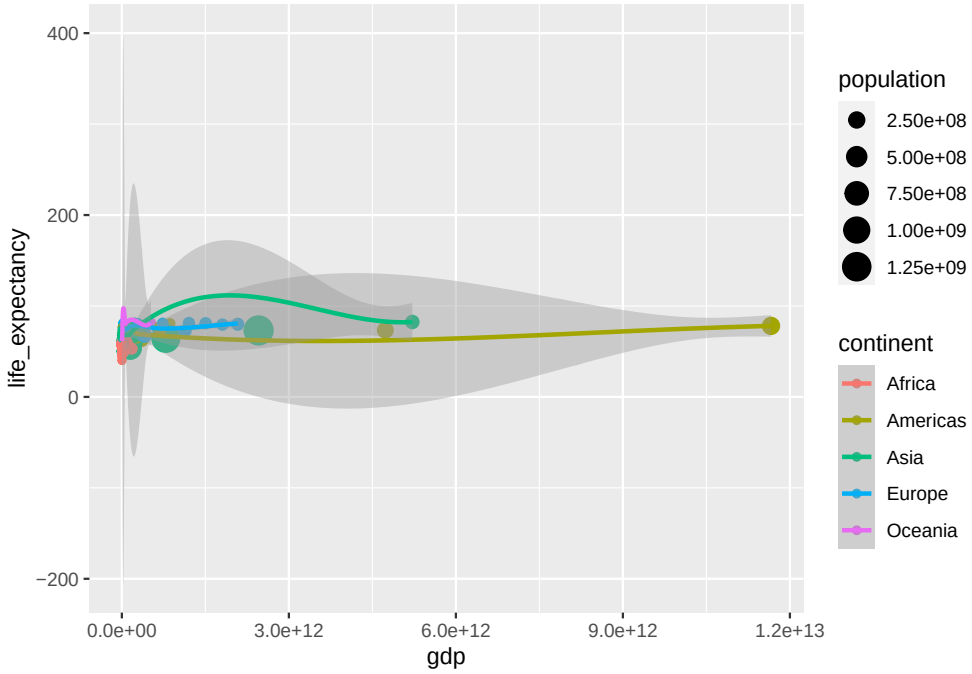
```
# this doesn't work as stat_smooth didn't know aes(x , y)
ggplot(data = gm_dt) + geom_point(aes(x = gdp, y = life_expectancy)) + stat_smooth()
```

```
## Error: stat_smooth requires the following missing aesthetics: x and y
```

```
# this would work but too redundant
ggplot(data = gm_dt) + geom_point(aes(x = gdp, y = life_expectancy)) + stat_smooth(aes(x = gdp,
    y = life_expectancy))
```



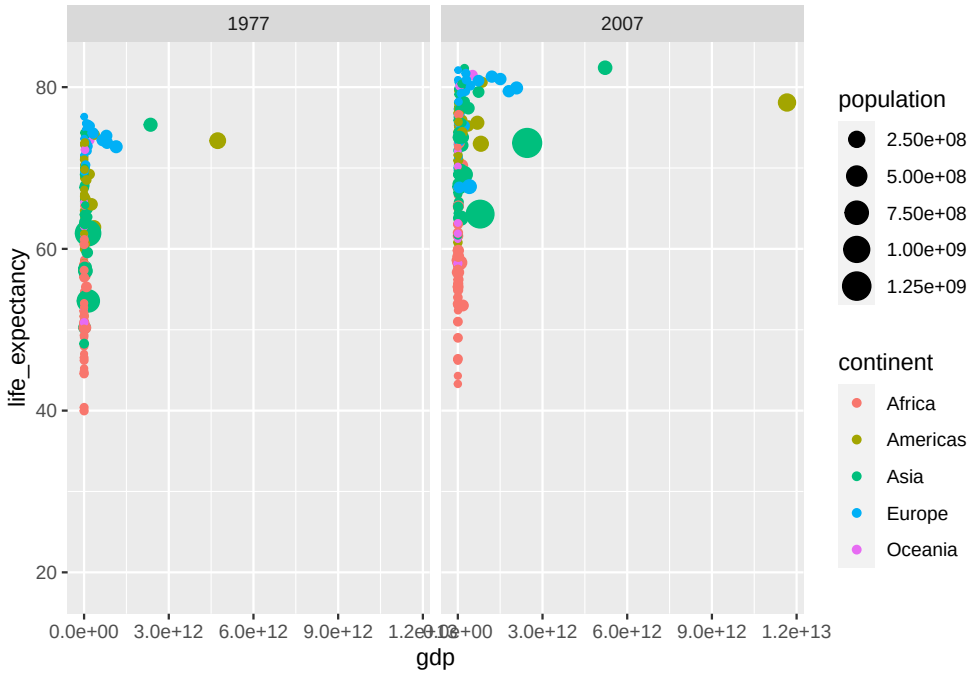

```
# the common aes(x, y) shared by all the layers can be put in the ggplot()
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent)) + geom_point(aes(size = populati
    stat_smooth()
```



6.2.4 Facets, axes and labels

For comparing the data from the year 1977 with the data from 2007, we can add a facet with `facet_wrap()`:

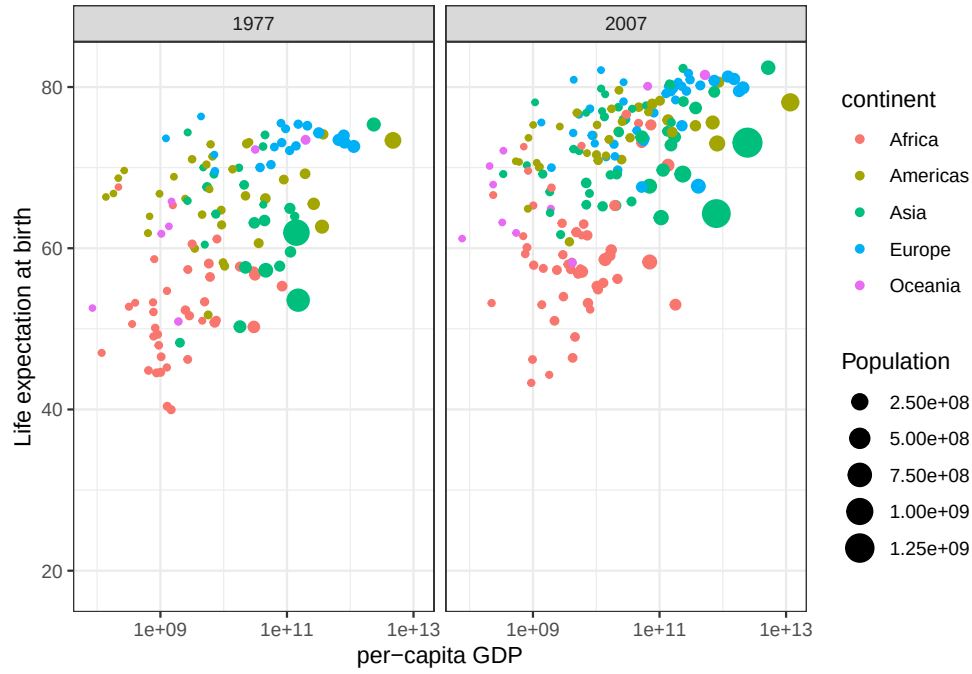
```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent, size = population)) +
    geom_point() + facet_wrap(~year)
```



For a better visualization of the data points, we can consider log scaling, which we will describe more in detail later. Finally, we can adapt the axes labels of the plot with `labs()` and define a theme of our plot:

```
mysize <- 15
mytheme <- theme(axis.title = element_text(size = mysize), axis.text = element_text(size = mysize),
    legend.title = element_text(size = mysize), legend.text = element_text(size = mysize)) +
    theme_bw()

ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population)) + facet_grid(~year) + scale_x_log10() + labs(x = "per-capita GDP",
    y = "Life expectation at birth", size = "Population") + mytheme
```



We remark here that `ggplot2` allows many further adaptions to plots, such as specifying axis breaks and limits. Some of these are covered in the appendix at the end of this script.

6.3 Different types of one- and two-dimensional plots

In the previous examples, we had a look at scatter plots which are suitable for plotting the relationship between two continuous variables. However, there are many more types of plots (e.g. histograms, boxplots) which can be used for plotting in different scenarios. Mainly, we distinguish between plotting one or two variables and whether the variables are continuous or discrete.

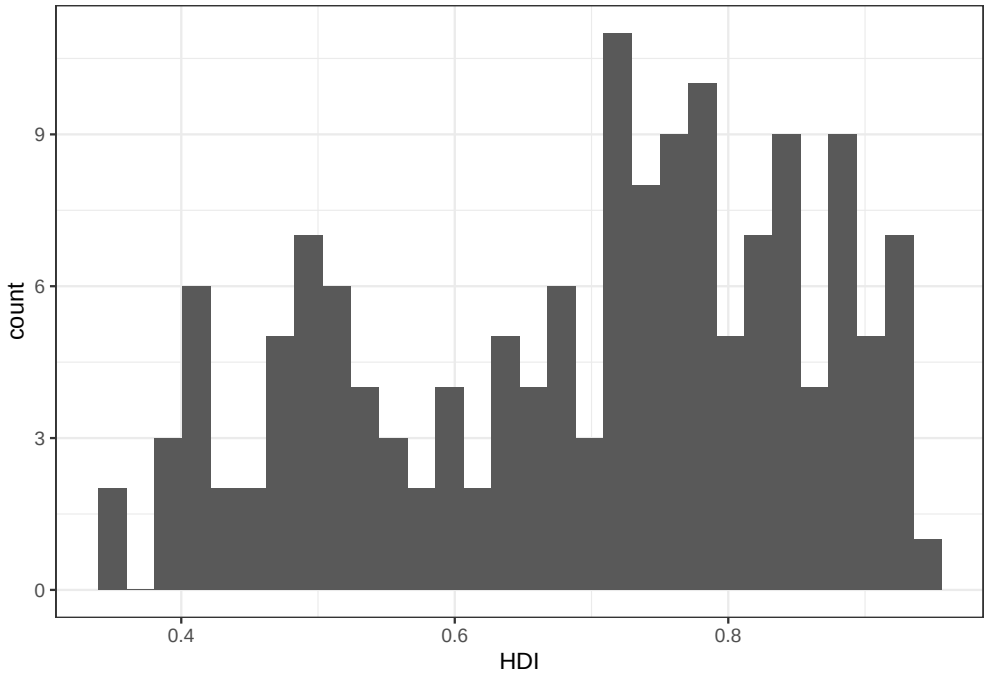
6.3.1 Plots for one single continuous variable

6.3.1.1 Histograms

A histogram represents the frequencies of values of a variable bucketed into ranges. It takes as input numeric variables only. A histogram is similar to a bar plot but the difference is that it groups the values into continuous ranges. Each bar in a histogram represents the height of the number of values present in that range. Each bar of the histogram is called a bin.

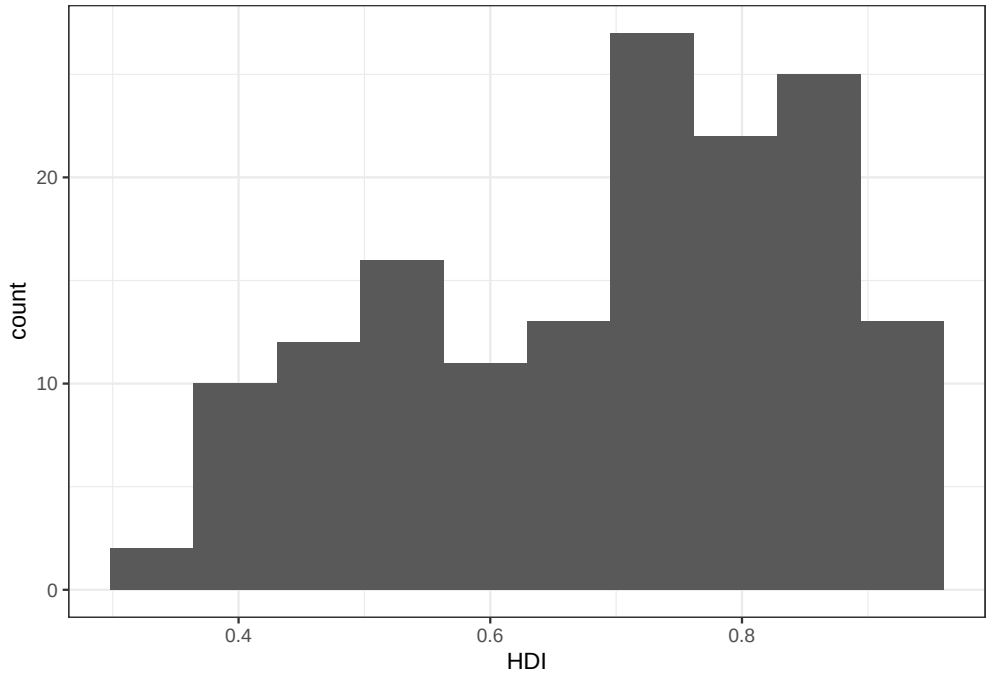
We can construct a histogram of the Human Development Index (HDI) in the `ind` dataset with the function `geom_histogram()`:

```
ggplot(ind, aes(HDI)) + geom_histogram() + mytheme
```



By default, the number of bins in `ggplot2` is 30. We can simply change this by defining the number of desired bins in the `bins` argument of the `geom_histogram()` function:

```
ggplot(ind, aes(HDI)) + geom_histogram(bins = 10) + mytheme
```

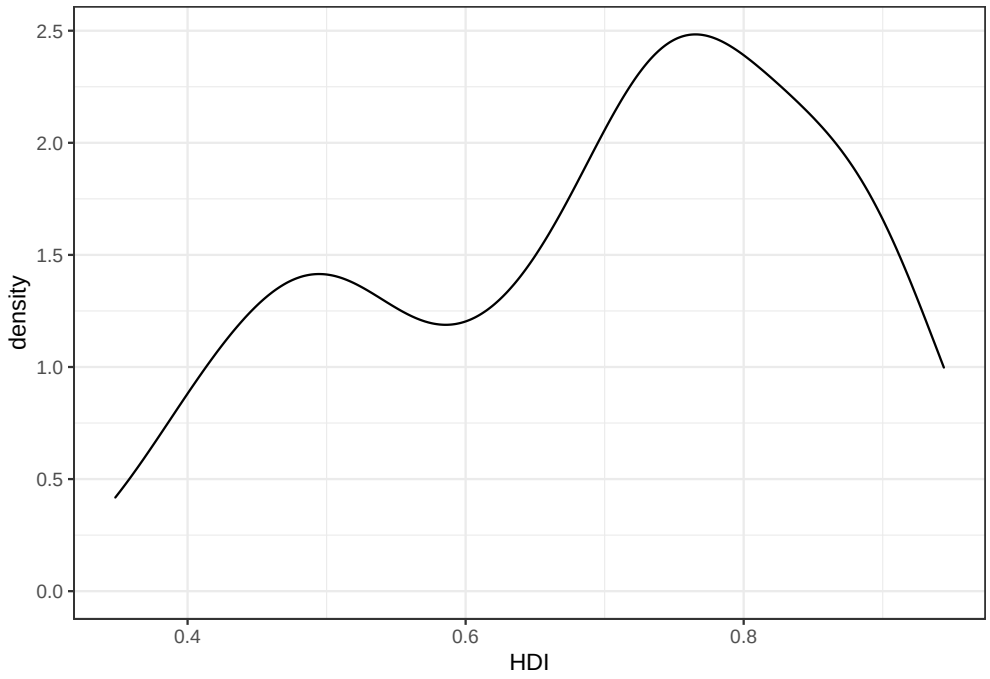


6.3.1.2 Density plots

Histograms are sometimes not optimal to investigate the distribution of a variable due to discretization effects during the binning process. A variation of histograms is given by density plots. They are used to represent the distribution of a numeric variable. These distribution plots are typically obtained by kernel density estimation to smoothen out the noise. Thus, the plots are smooth across bins and are not affected by the number of bins, which helps create a more defined distribution shape.

As an example, we can visualize the distribution of the Human Development Index (HDI) in the `ind` dataset by means of a density plot with `geom_density()`:

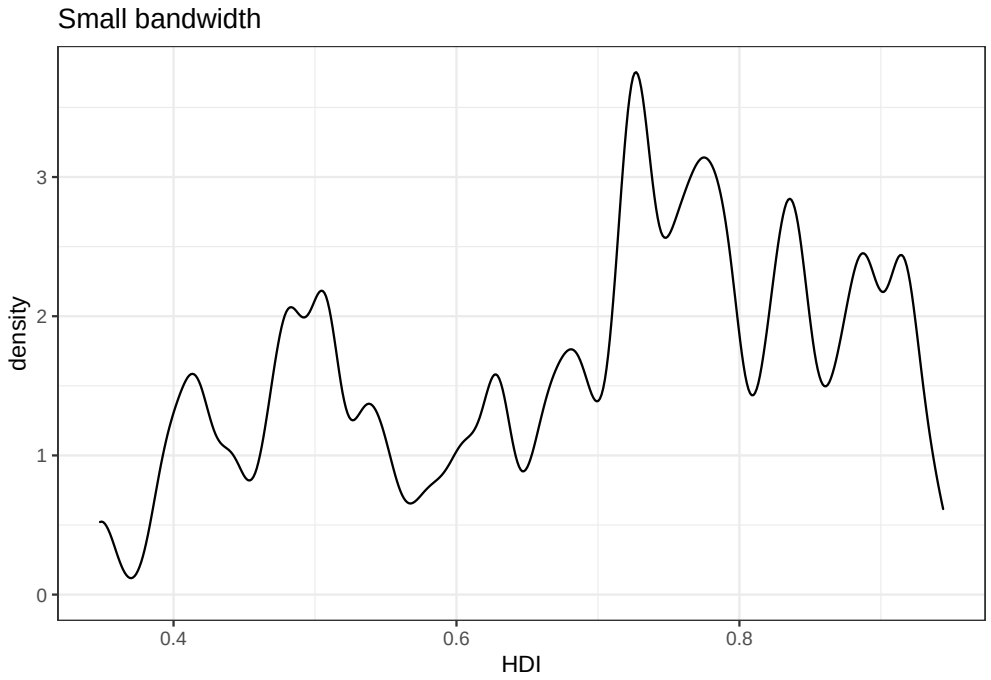
```
ggplot(ind, aes(HDI)) + geom_density() + mytheme
```



The `bw` argument of the `geom_density()` function allows to tweak the bandwidth of a density plot manually. The default option is a bandwidth rule, which is usually a good choice.

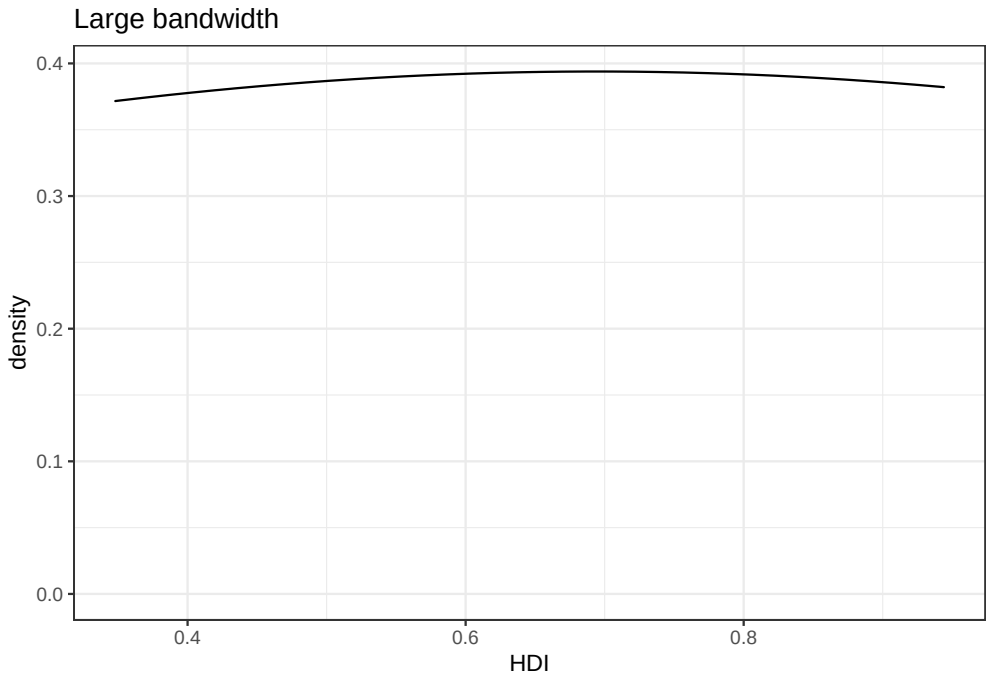
Setting a small bandwidth on the previous plot has a huge impact on the plot:

```
ggplot(ind, aes(HDI)) + geom_density(bw = 0.01) + ggtitle("Small bandwidth") + mytheme
```



Setting a large bandwidth has also a huge impact on the plot:

```
ggplot(ind, aes(HDI)) + geom_density(bw = 1) + ggtitle("Large bandwidth") + mytheme
```



Thus, we should be careful when changing the bandwidth, since we can get a wrong impression from the distribution of a continuous variable.

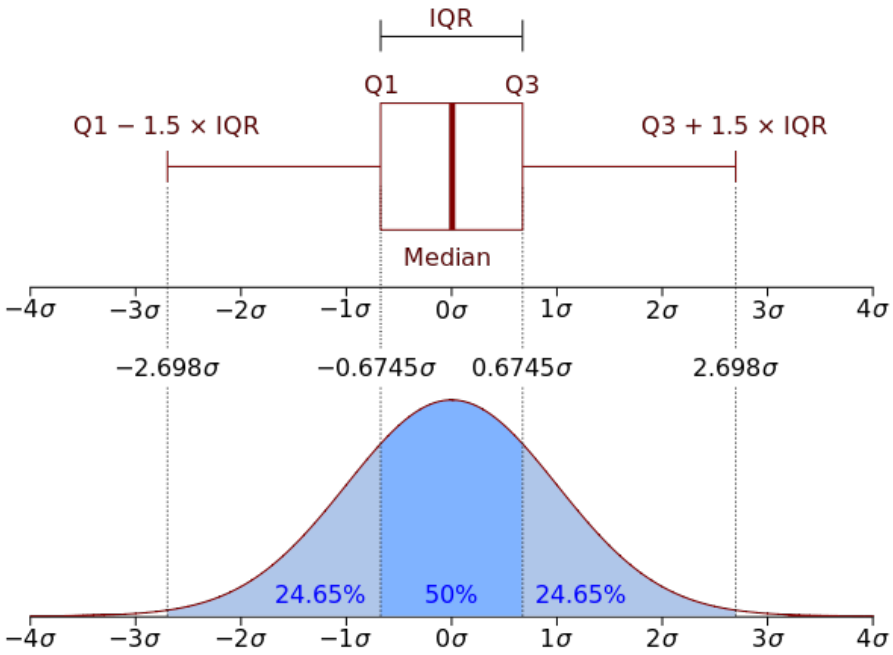
6.3.1.3 Boxplots

Box plots can give a good graphical insight into the distribution of the data. They show the median, quantiles, and how far the extreme values are from most of the data.

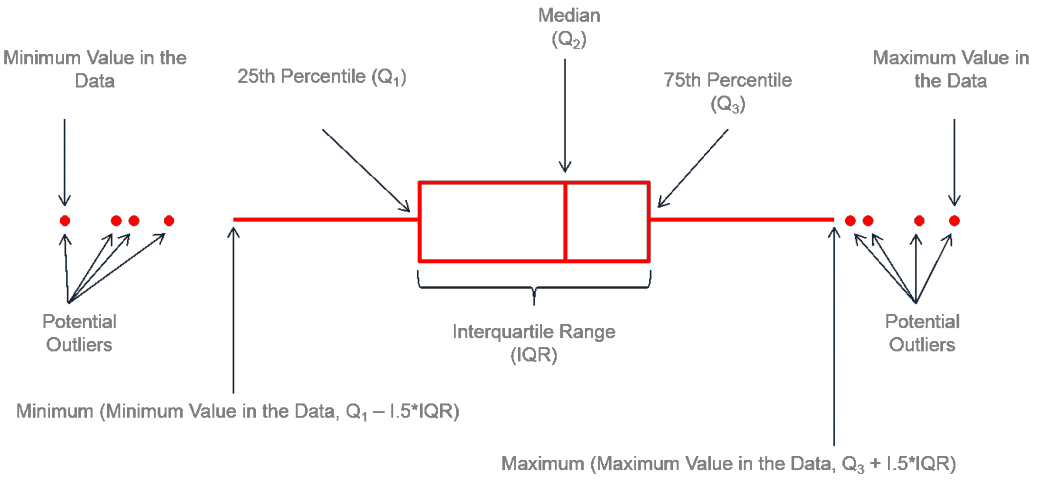
Five values are essential for constructing a boxplot:

- the median: the center of the data, middle value of a sorted list, 50% quantile of the data
- the first quantile (Q1): 25% quantile of the data
- the third quantile (Q3): 75% quantile of the data
- the interquantile range (IQR): the distance between Q1 and Q3

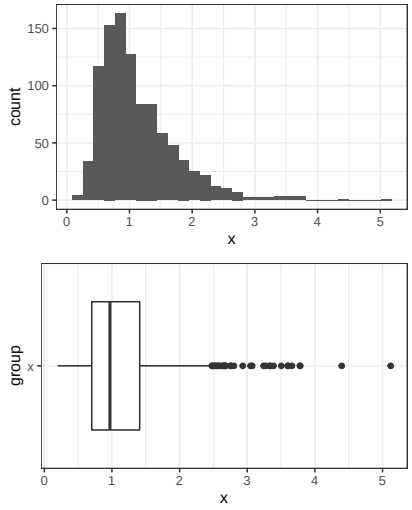
Every boxplot has lines at Q1, the median, and Q3, which together build the box of the boxplot. The other major feature of a boxplot is its whiskers. The whiskers are determined with the help of the IQR. Here, we compute 1.5 × IQR below Q1 and 1.5 × IQR above Q3. Anything outside of this range is called an outlier. We then draw lines at the smallest and largest point within this subset (Q1 - 1.5 × IQR to Q3 + 1.5 × IQR) from the dataset. These lines define our whiskers which reach the most extreme data point within $\pm 1.5 \times IQR$.



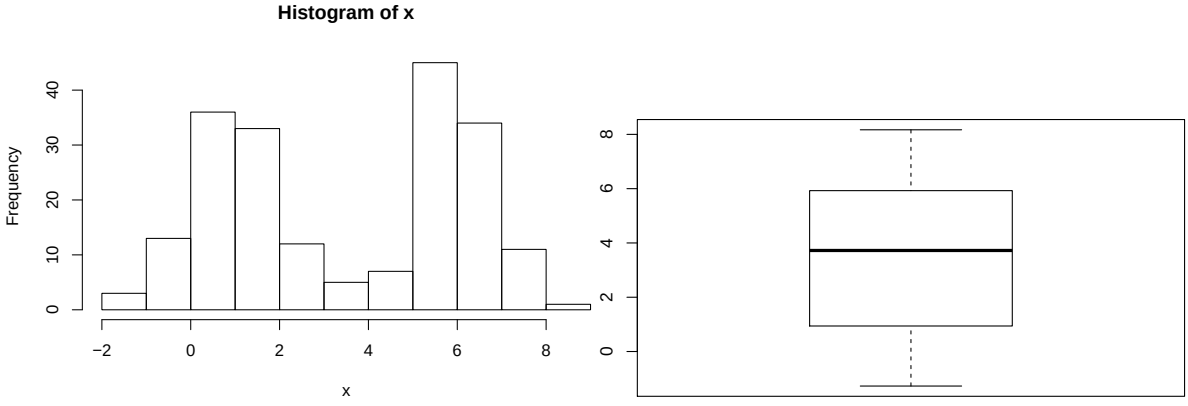
It is possible to not show the outliers in boxplots, as seen in the visualization before. However, we strongly recommend keeping them. Outliers can reveal interesting data points (discoveries “out of the box”) or bugs in data preprocessing.



For instance, we can plot the distribution of a variable `x` with a histogram and visualize the corresponding boxplot:



Boxplots are particularly suited for plotting non-gaussian symmetric and non-symmetric data and for plotting exponentially distributed data. However, boxplots are not well suited for bimodal data, since they only show one mode (the median). In the following example, we see a bimodal distribution in the histogram and the corresponding boxplot, which does not properly represent the distribution of the data.



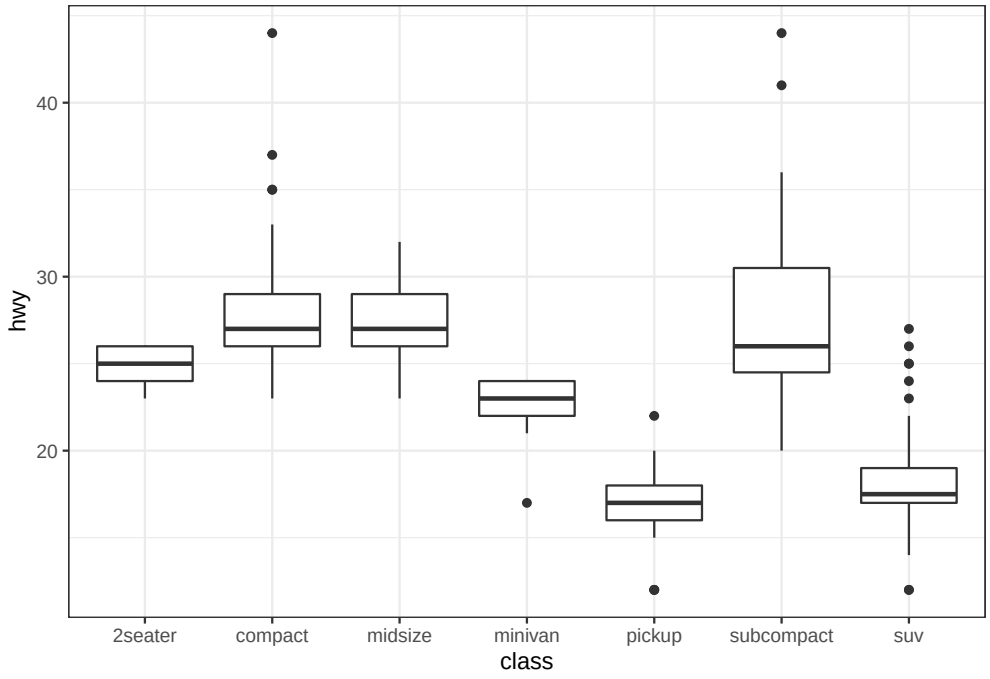
6.3.2 Plots for two variables: one continuous, one discrete

6.3.2.1 Boxplots by category

As illustrated before, boxplots are well suited for plotting one continuous variable. However, we can also use boxplots to show distributions of continuous variables with respect to some categories. This can be particularly interesting for comparing the different distributions of each category.

For instance, we want to visualize the highway miles per gallon `hwy` for every one of the 7 vehicle classes (compact, SUV, minivan, etc.). For this, we define the categorical `class` variable on the `x` axis and the continuous variable `hwy` on the `y` axis.

```
ggplot(mpg, aes(class, hwy)) + geom_boxplot() + mytheme
```

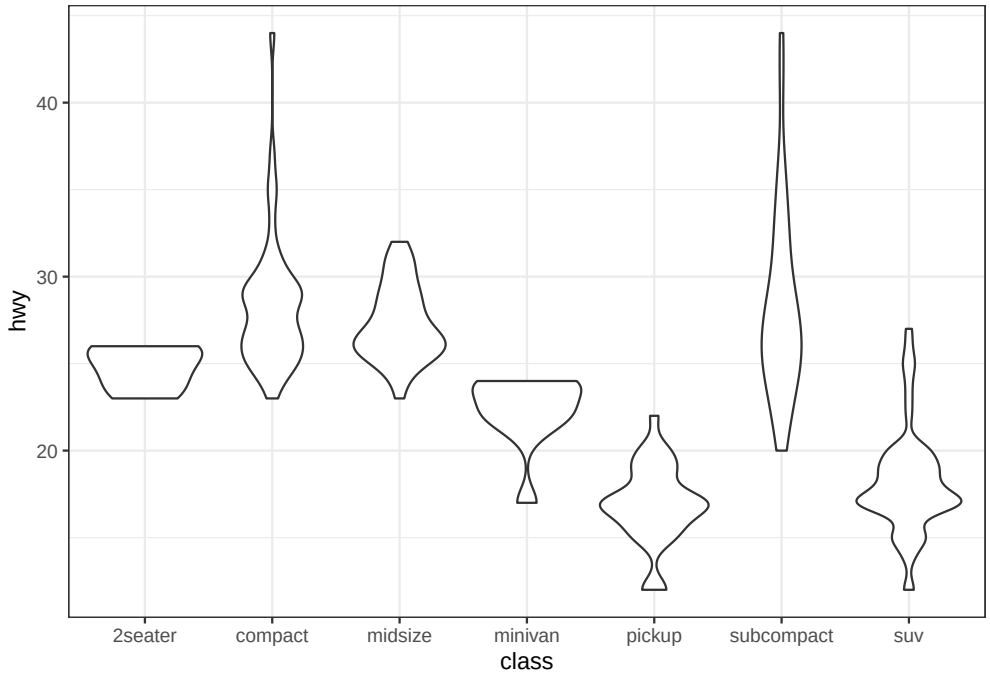


6.3.2.2 Violin plots

A violin plot is an alternative to the boxplot for visualizing either one continuous variable (grouped by categories). An advantage of the violin plot over the boxplot is that it also shows the entire distribution of the data. This can be particularly interesting when dealing with multimodal data.

For a direct comparison, we show a violin plot for the `hwy` grouped by `class` as before with the help of the function `geom_violin()`:

```
ggplot(mpg, aes(class, hwy)) + geom_violin() + mytheme
```

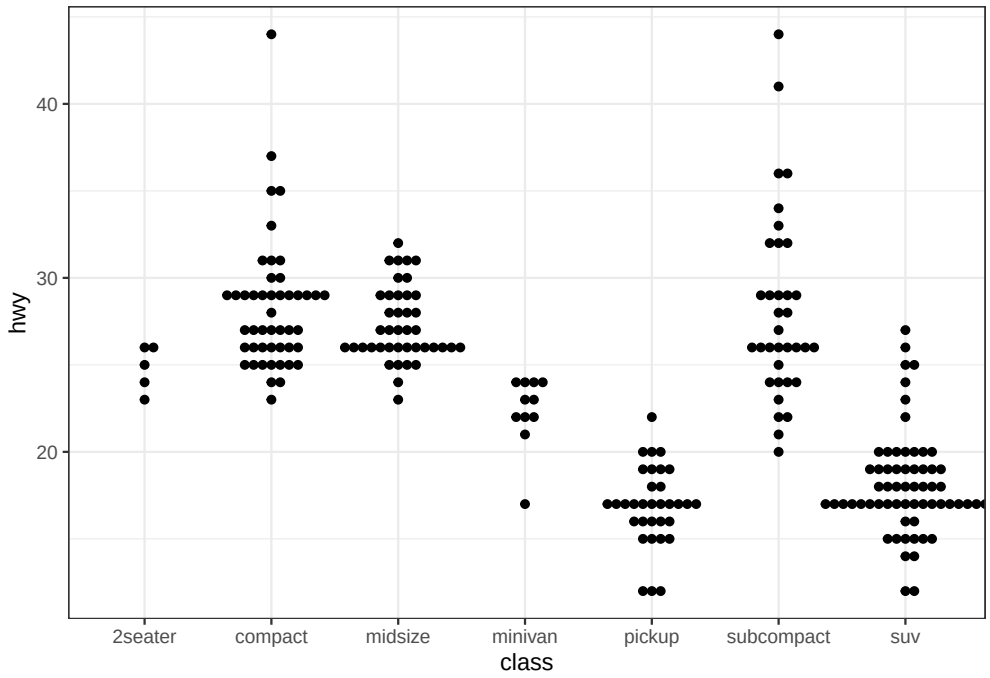


6.3.2.3 Beanplots

Another alternative to the popular boxplot is the beanplot. In a beanplot, the individual observations are shown as small lines in a one-dimensional scatter plot. Moreover, the estimated density of the distributions is visible in a beanplot. It is easy to compare different groups of data in a beanplot and to see if a group contains enough observations to make the group interesting from a statistical point of view.

For creating beanplots in R, we can use the package `ggbeeswarm`. We use the function `geom_besswarm()` to create a beanplot to visualize once again the `hwy` grouped by `class`:

```
# install.packages('ggbeeswarm')
library(ggbeeswarm)
ggplot(mpg, aes(class, hwy)) + geom_beeswarm() + mytheme
```



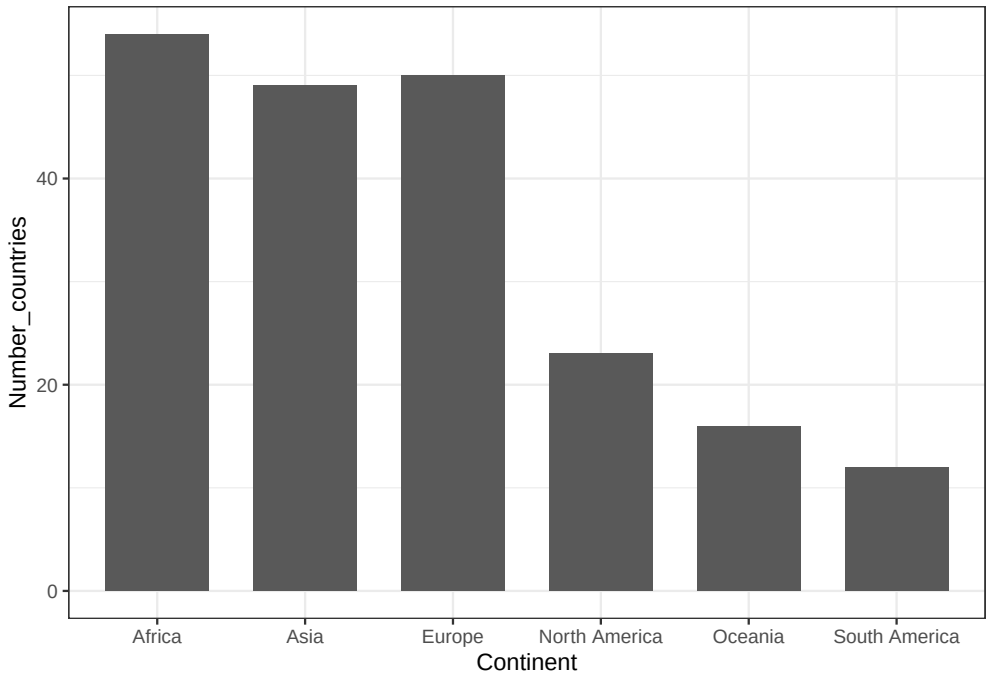
We remark that beanplots are useful only up to a certain number of data points. The creation of beanplots for larger datasets may become too expensive.

6.3.2.4 Barplots

Barplots are often used to highlight individual quantitative values per category. Bars are visual heavyweights compared to dots and lines. In a barplot, we can combine two attributes of 2-D location and line length to encode quantitative values. In this manner, we can focus the attention primarily on individual values and support the comparison of one to another.

For creating a barplot with `ggplot2` we can use the function `geom_bar()`. In the next example, we visualize the number of countries (defined in the `y` axis) per continent (defined in the `x` axis).

```
ggplot(countries_dt, aes(Continent, Number_countries)) + geom_bar(stat = "identity",
    width = 0.7) + mytheme
```



6.3.2.5 Barplots with errorbars

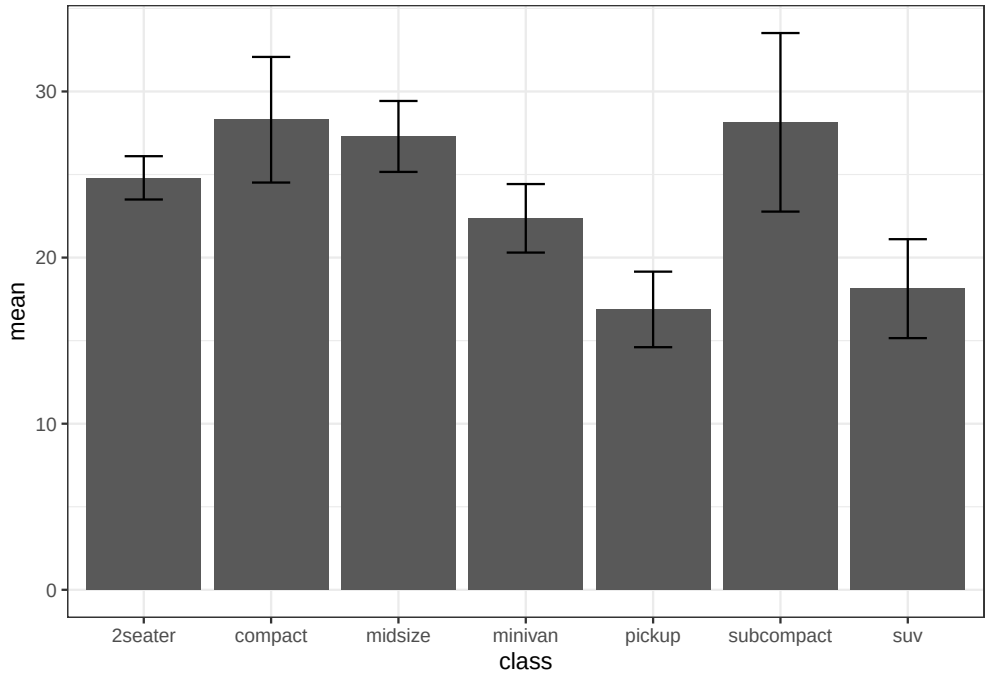
Visualizing uncertainty is important, otherwise, barplots with bars as a result of an aggregation can be misleading. One way to visualize uncertainty is with error bars.

As error bars, we can consider the standard deviation (SD) and the standard error of the mean (SEM). We remark that SD and SEM are completely different concepts. On the one hand, SD indicates the variation of quantity in the sample. On the other hand, SEM represents how well the mean is estimated.

The central limit theorem implies that: $SEM = SD/\sqrt{(n)}$, where n is the sample size (number of observations). With large n, SEM tends to 0.

In the following example, we plot me average highway miles per gallon `hwy` per vehicle class `class` including error bars computed as the average plus/minus standard devation of `hwy`:

```
as.data.table(mpg) %>% .[, .(mean = mean(hwy), sd = sd(hwy)), by = class] %>% ggplot(aes(class,
    mean, ymax = mean + sd, ymin = mean - sd)) + geom_bar(stat = "identity") + geom_errorbar(width = 0.3) +
    mytheme
```



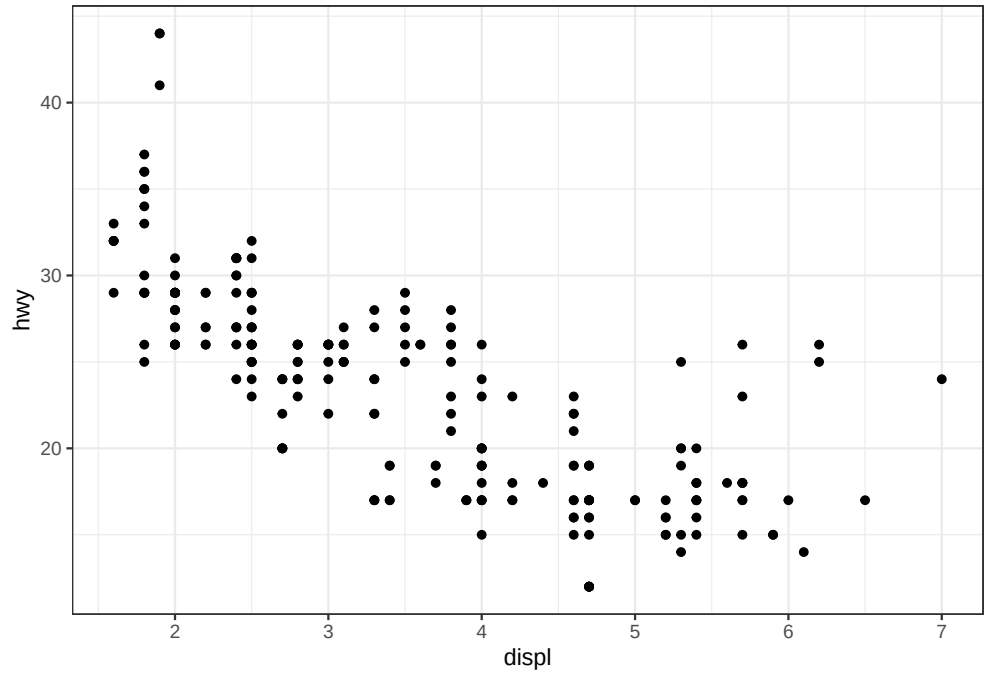
6.3.3 Plots for two continuos variables

6.3.3.1 Scatter plots

Scatter plots are a useful plot type for easily visualizing the relationship between two continuous variables. Here, dots are used to represent pairs of values corresponding to the two considered variables. The position of each dot on the horizontal (`x`) and vertical (`y`) axis indicates values for an individual data point.

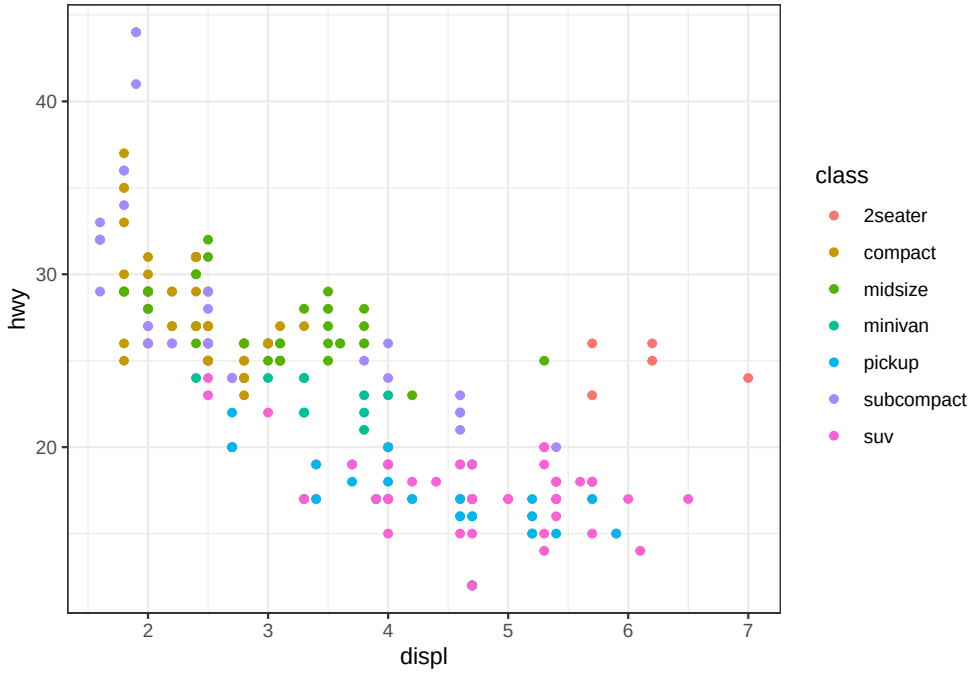
In the next example, we analyze the relationship between the engine displacement in liters `displ` and the highway miles per gallon `hwy` from the `mpg` dataset:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + mytheme
```



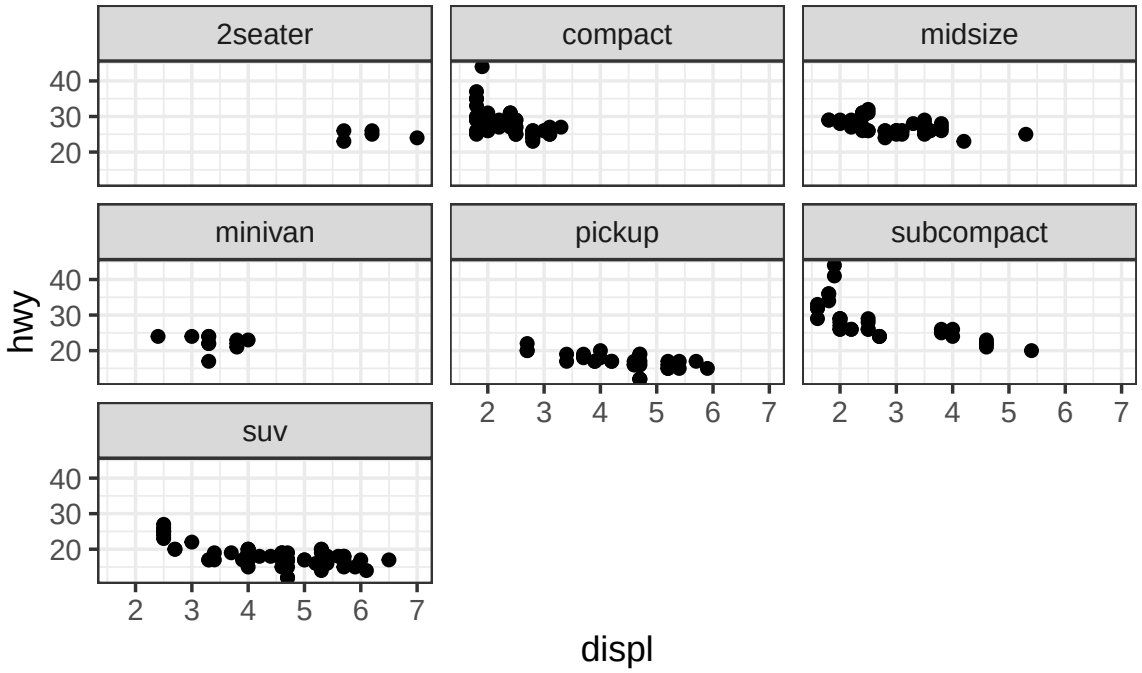
We can modify the previous plot by coloring the points depending on the vehicle class:

```
ggplot(mpg, aes(displ, hwy, color = class)) + geom_point() + mytheme
```



Sometimes, too many colors can be hard to distinguish. In such cases, we can use `facet` to separate them into different plots:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_wrap(~class) + mytheme
```

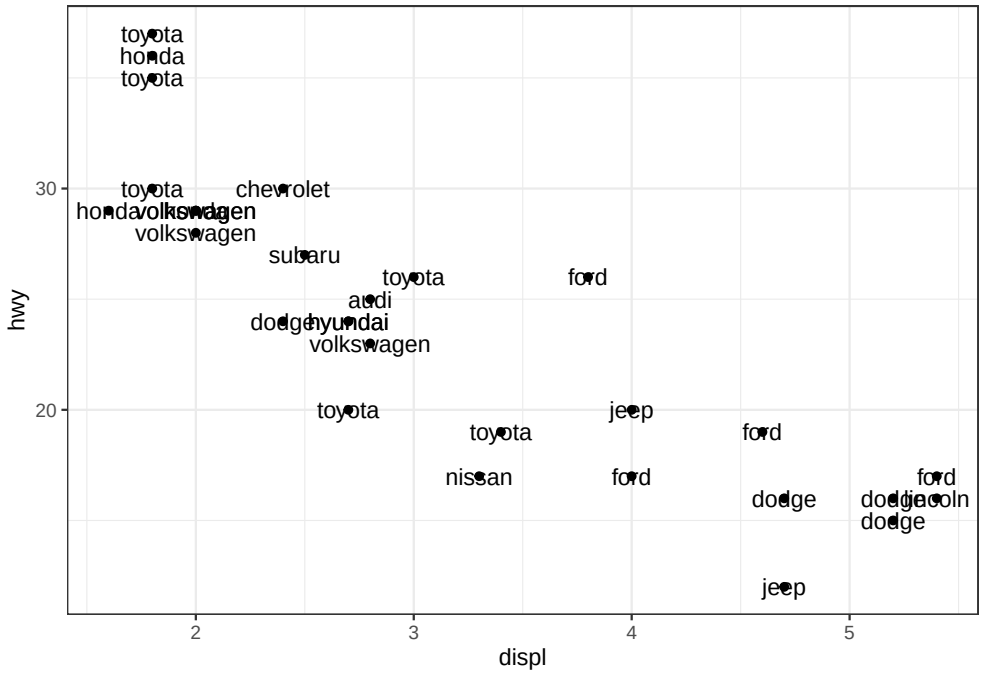


6.3.3.1.1 Text labeling

For labeling the individual points in a scatter plot, `ggplot2` offers the function `geom_text()`. However, these labels tend to overlap. To avoid this, we can use the library `ggrepel` which offers a better text labeling through the function `geom_text_repel()`.

We first show the output of the classic text labeling with `geom_text()` for a random subset of 40 observations of the dataset `mpg`. Here we plot the engine displacement in liters `displ` vs. the highway miles per gallon `hwy` and label by `manufacturer`:

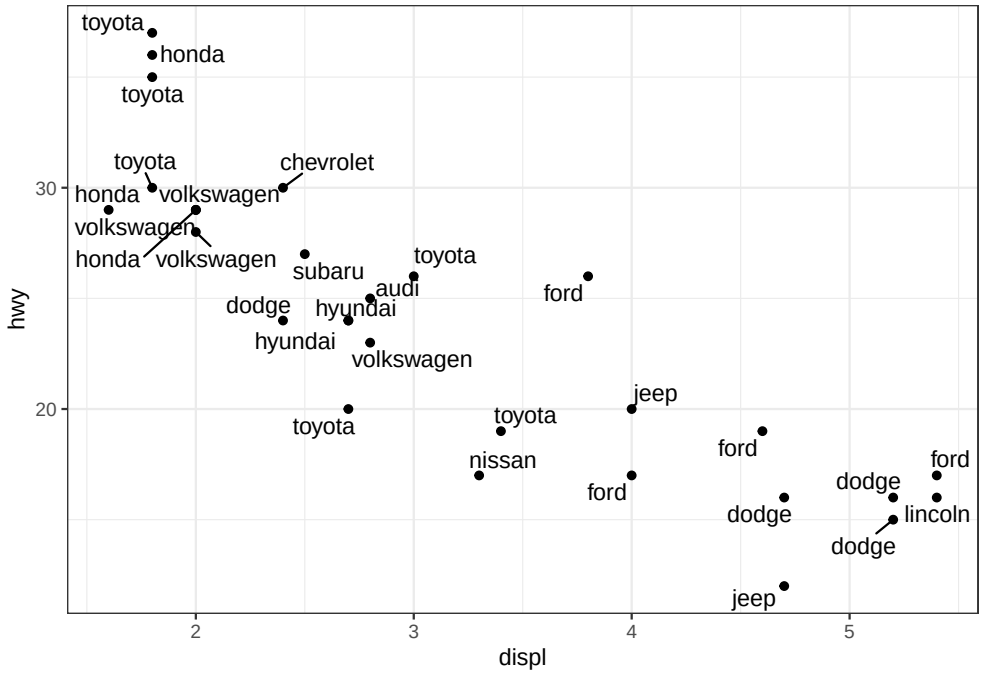
```
set.seed(12)
mpg_subset <- mpg[sample(1:nrow(mpg), 30, replace = FALSE), ]
ggplot(mpg_subset, aes(displ, hwy, label = manufacturer)) + geom_point() + geom_text() +
    mytheme
```



As seen in the previous illustration, the text labels overlap. This complicates understanding the plot. Therefore, we exchange the function `geom_text()` by `geom_text_repel()` from the library `ggrepel`:

```
library(ggrepel)

ggplot(mpg_subset, aes(displ, hwy, label = manufacturer)) + geom_point() + geom_text_repel() +
    mytheme
```

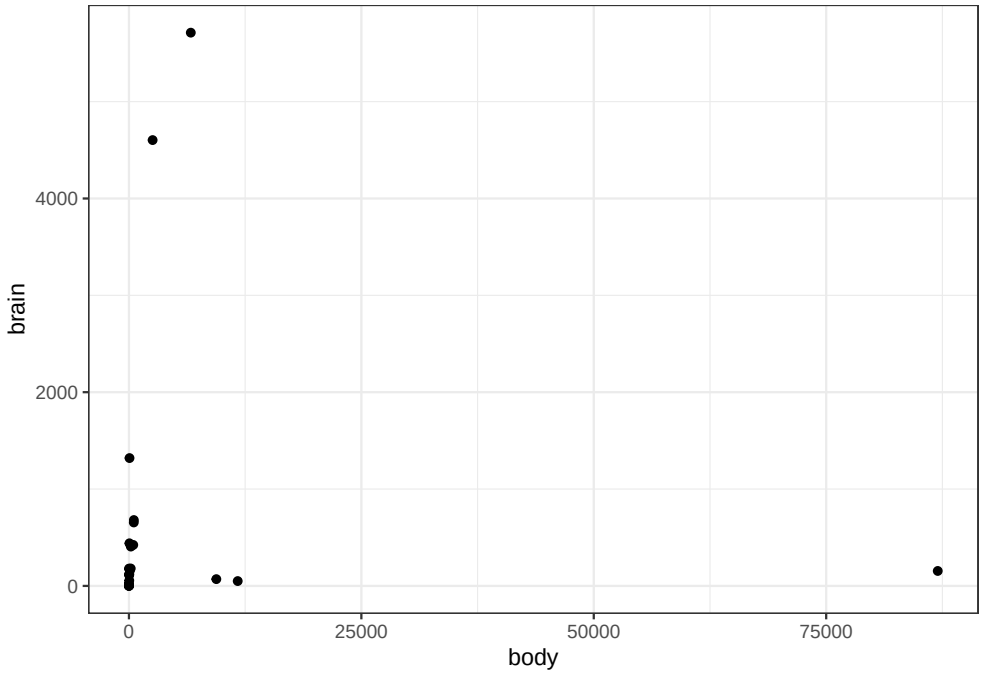


6.3.3.1.2 Log scaling

We consider another example where we want to plot the weights of the brain and body of different animals using the dataset `Animals`. This is what we obtain after creating a scatterplot.

```
library(MASS)  # to access Animals data sets
animals_dt <- as.data.table(Animals)

ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() + mytheme
```

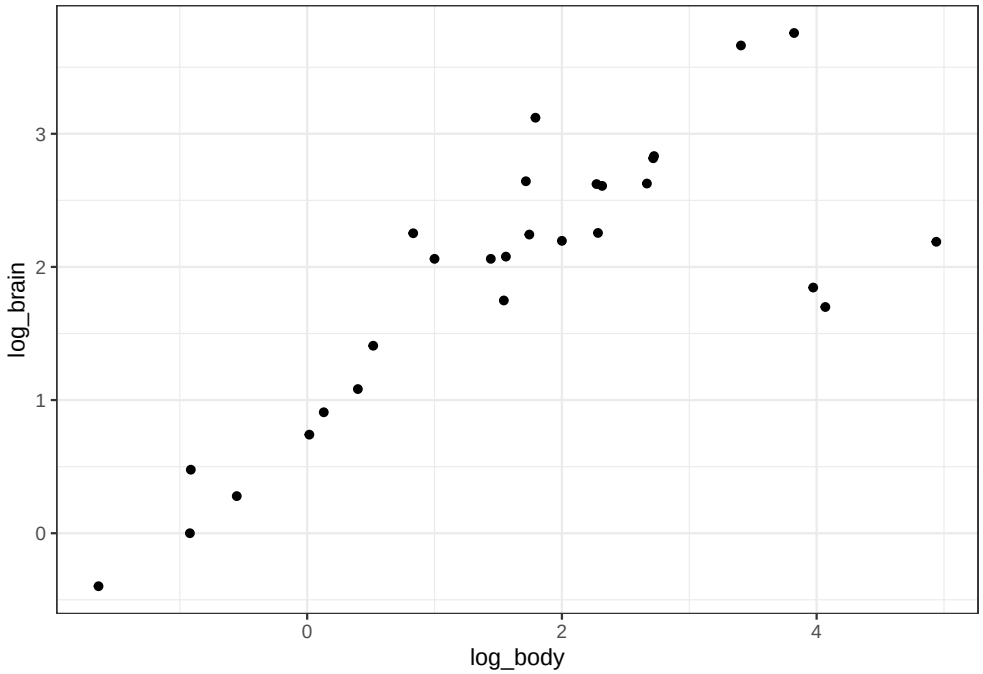


We can clearly see that there are a few points which are notably larger than most of the points. This makes it harder to interpret the relationships between most of these points. In such cases, we can consider **logarithmic** transformations and/or scaling. More precisely, a first idea would be to manually transform the values into a

logarithmic space and plot the transformed values instead of the original values:

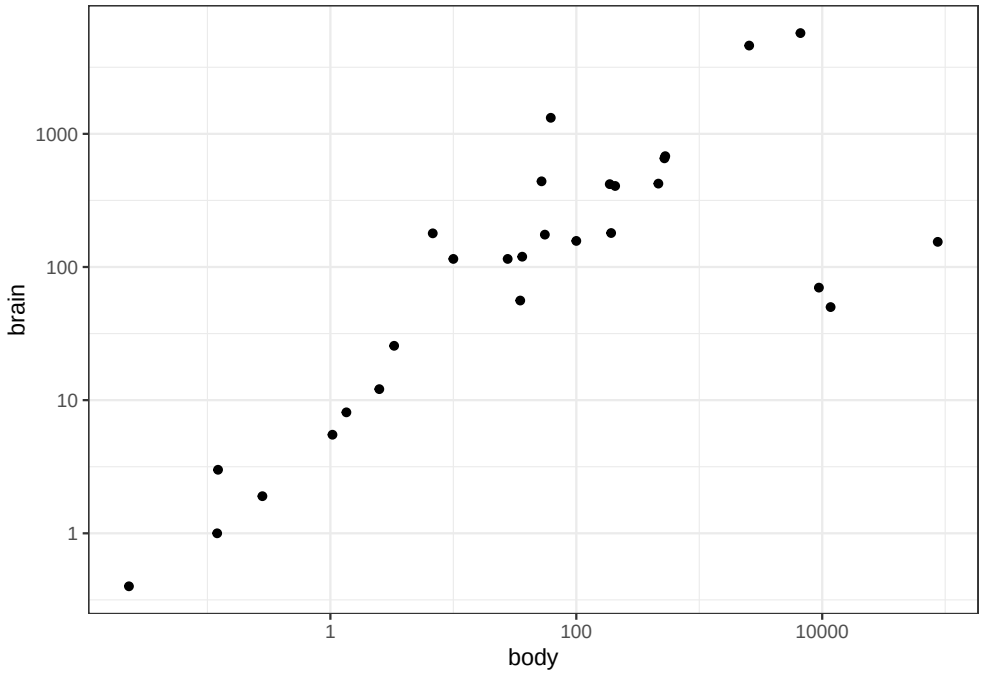
```
animals_dt[, `:=`(c("log_body", "log_brain"), list(log10(body), log10(brain)))]

ggplot(animals_dt, aes(x = log_body, y = log_brain)) + geom_point() + mytheme
```



Alternativelt, `ggplot2` offers to simply scale the data without the need to transform. This can be done with the help of the functions `scale_x_log10()` and `scale_y_log10()` which allow appropriate scaling and labeling of the axes:

```
ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() + scale_x_log10() + scale_y_log10() +
    mytheme
```

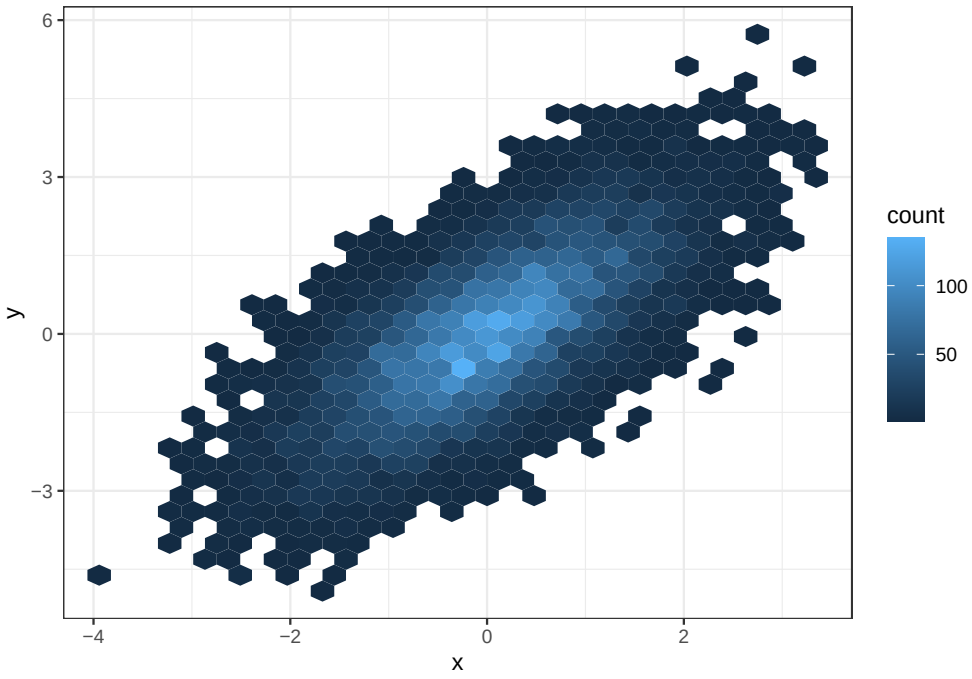


6.3.3.2 Density plots in 2D

Using scatterplots can become problematic when dealing with a huge number of points. This is due to the fact that points may overlap and we cannot clearly see how many points are at a certain position. In such cases, a 2D density plot is particularly well suited. This plot counts the number of observations within a particular area of the 2D space.

The function `geom_hex()` is particularly useful for creating 2D density plots in R:

```
x <- rnorm(10000)
y = x + rnorm(10000)
data.table(x, y) %>% ggplot(aes(x, y)) + geom_hex() + mytheme
```

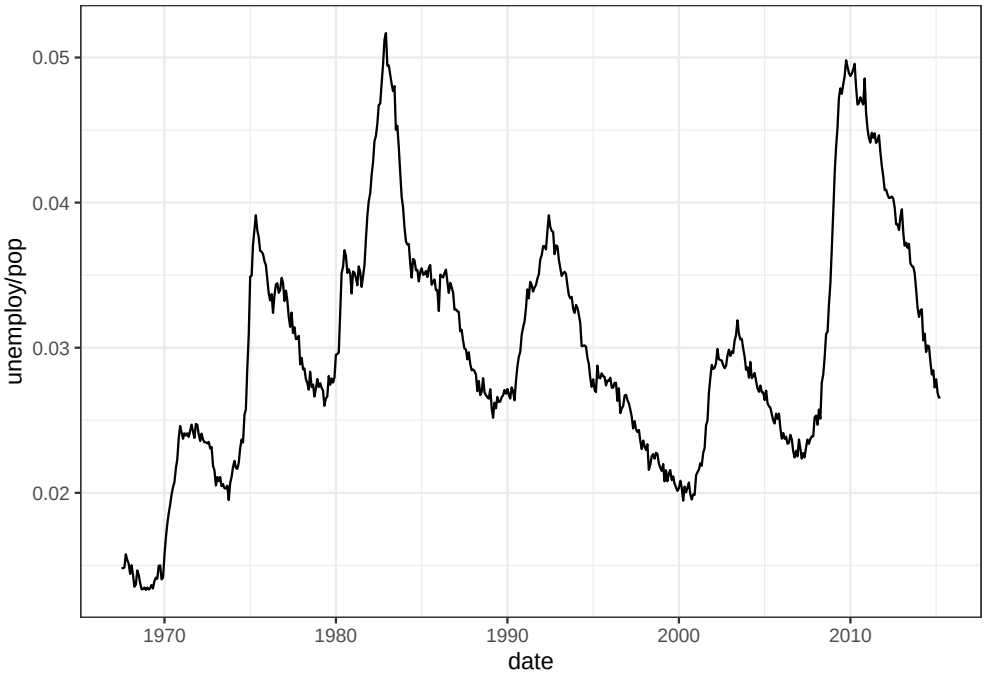


6.3.3.3 Line plots

A line plot can be considered for connecting a series of individual data points or to display the trend of a series of data points. This can be particularly useful to show the shape of data as it flows and changes from point to point. We can also show the strength of the movement of values up and down through time.

As an example we show the connection between the individual datapoints of unemployment rate over the years:

```
ggplot(economics, aes(date, unemploy/pop)) + geom_line() + mytheme
```



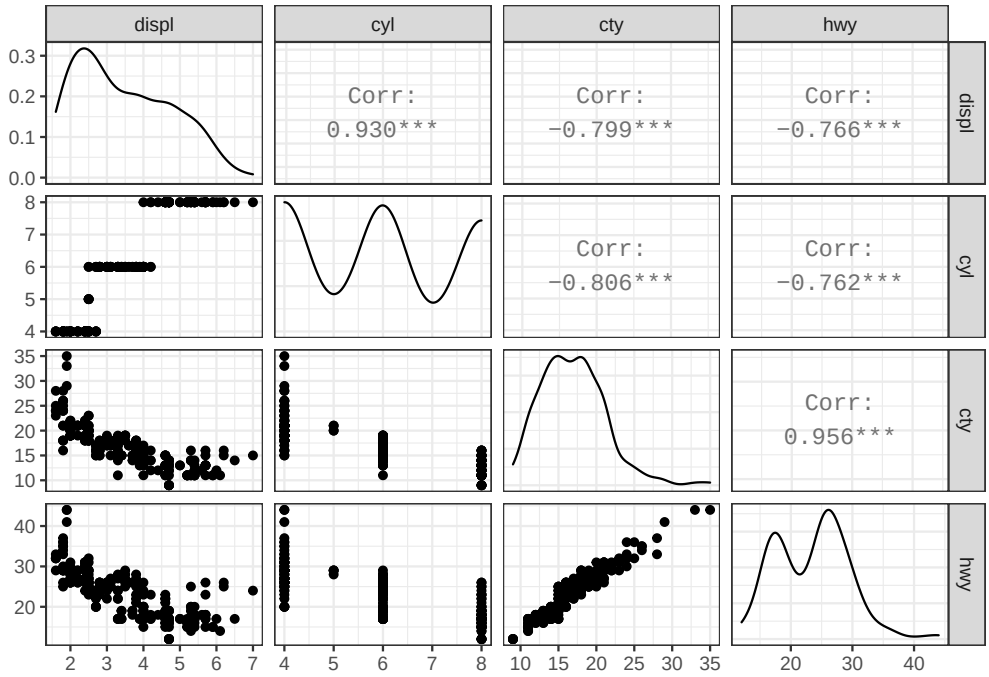
6.4 Further plots for low dimensional data

6.4.1 Plot matrix

A so-termed plot matrix is useful for exploring the distributions and correlations of a few variables in a matrix-like representation. Here, for each pair of considered variables, a scatterplot is created and the correlation coefficient is computed. For every single variable, a histogram is created for showing the respective distribution.

We can use the function `ggpairs()` from the library `GGally` for constructing plot matrices. As an example, we analyze the variables `displ`, `cyl`, `cty` and `hwy`from the dataset `mpg`:

```
library(GGally)
ggpairs(mpg, columns = c("displ", "cyl", "cty", "hwy")) + mytheme
```



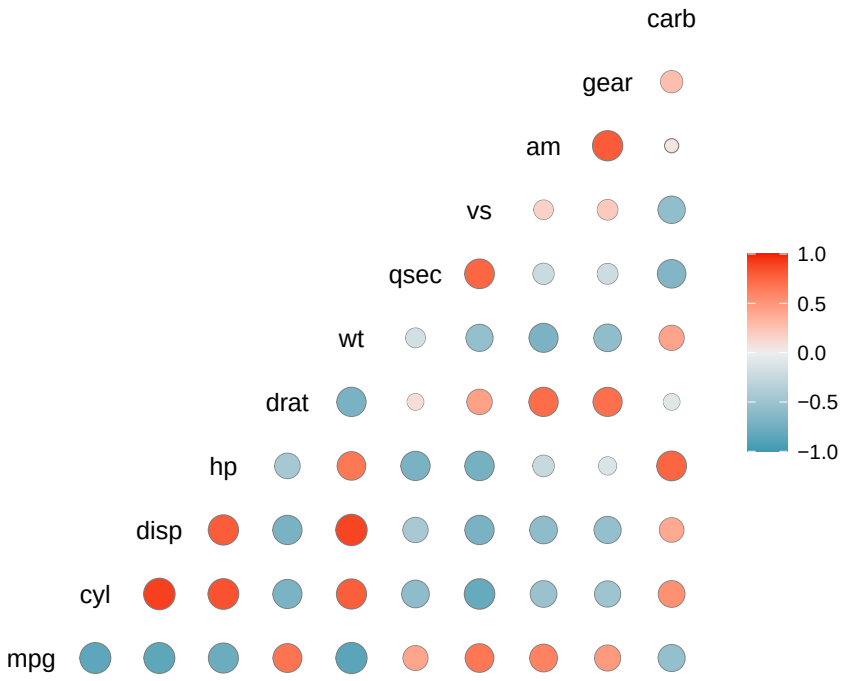
We remark that this plot is not well suited for comparing more than a few variables.

6.4.2 Correlation plot

A correlation plot is a graphical representation of a correlation matrix. It is useful to highlight the most correlated variables in a dataset. In this plot, correlation coefficients are colored according to the value. A correlation matrix can be also reordered according to the degree of association between variables.

Correlation plots are also called “corrgrams” or “correlograms”. As an example, we visualize the correlation between the variables of the dataset `mtcars` with the help of the function `ggcorr()` from the library `GGally`:

```
ggcorr(mtcars, geom = "circle")
```



6.5 Summary

This first chapter of data visualization covered the basics of the grammar of graphics and `ggplot2` to plot low dimensional data. We introduced the different types of plots such as histograms, boxplots and barplots and discussed when to use which plot.

6.6 Resources

- Plotting libraries:
  - http://www.r-graph-gallery.com/portfolio/ggplot2-package/
  - http://ggplot2.tidyverse.org/reference/
  - https://plot.ly/r/
  - https://plot.ly/ggplot2/
- Udacity's Data Visualization and D3.js
  - https://www.udacity.com/courses/all
- Graphics principles
  - https://onlinelibrary.wiley.com/doi/full/10.1002/pst.1912
  - https://graphicsprinciples.github.io/

Chapter 7

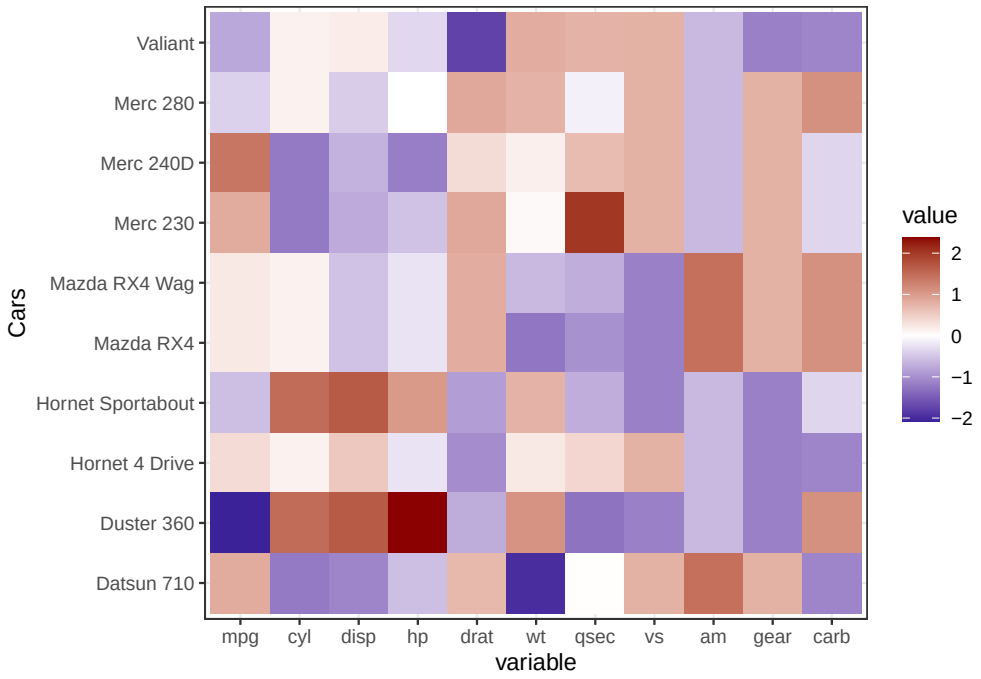
High dimensional visualizations

7.1 Heatmaps

For the representation of high dimensional datasets with several variables, we can use heatmaps. A heatmap is a graphical representation of data that uses a system of color-coding to represent different values.

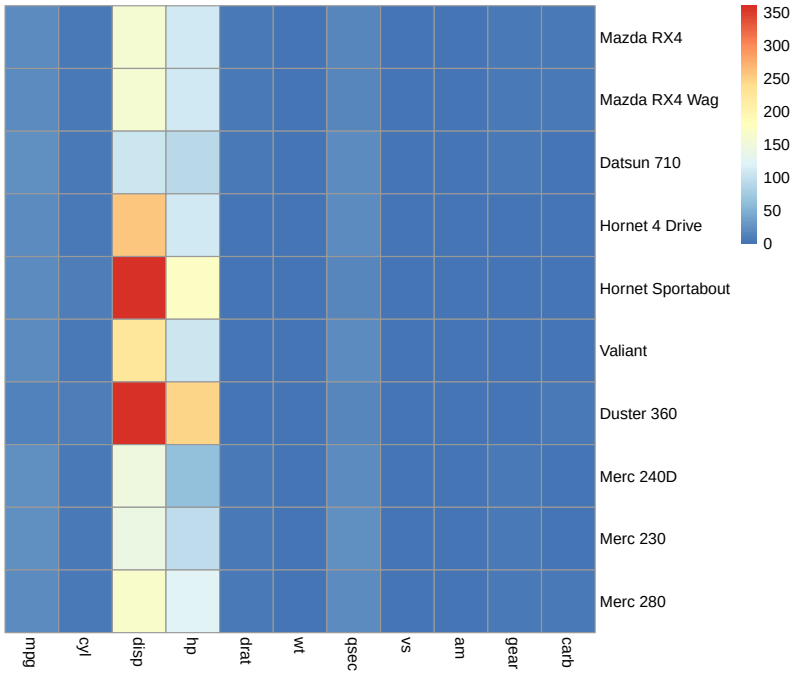
In the next example, we present a heatmap constructed from the melted version of the `mtcars` dataset. Here, we display the values corresponding to the variables of the different vehicles.

```
mtcars_melted <- as.data.table(scale(mtcars[1:10, ]), keep.rownames = "Cars") %>%
    melt(id.var = "Cars")
ggplot(mtcars_melted, aes(variable, Cars)) + geom_tile(aes(fill = value)) + scale_fill_gradient2(low = "dar
    mid = "white", high = "darkred") + mytheme
```



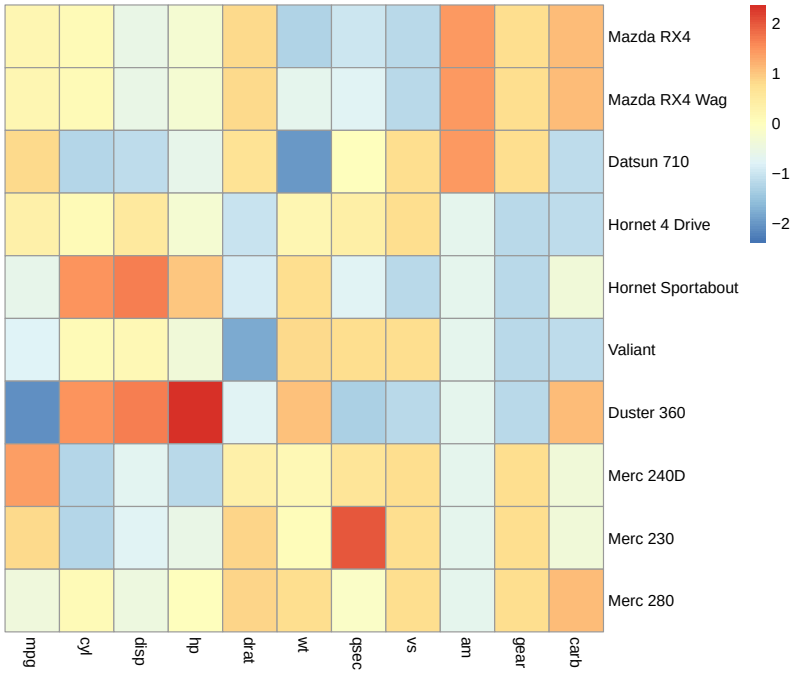
There is an alternative way to plot heatmaps in R with the library `pheatmap` for pretty heatmaps. We can construct the same heatmap for the `mtcars` dataset as before in an easier manner without the need for previously melting the data.

```
library(pheatmap)  ## pretty heatmap
pheatmap(mtcars[1:10, ], cluster_rows = F, cluster_cols = F, show_rownames = T)
```



With `pheatmap` we can also scale the data by rows or columns by setting the argument `scale` accordingly:

```
library(pheatmap)  ## pretty heatmap
pheatmap(mtcars[1:10, ], cluster_rows = F, cluster_cols = F, scale = "column", show_rownames = T)
```



7.2 Clustering

Clustering is the task of dividing a dataset of observations into a number of groups such that observations in the same group are more similar to the observations in the same group than to those in other groups. Groups are often referred to as clusters.

There are several algorithms for solving clustering tasks such as spectral clustering, Gaussian mixture models,

Hierarchical clustering and k-Means clustering. In the following sections, we will focus exclusively on Hierarchical clustering and k-Means clustering.

7.2.1 Hierarchical clustering

Hierarchical clustering builds a hierarchy of clusters. The algorithm starts with all the observations assigned to a cluster of their own. Then two nearest clusters are merged into the same cluster. In a bottom-up approach, clusters are successively merged until there is only a single cluster left.

The two clusters selected for merging are defined according to a distance metric, which plays a fundamental role in Hierarchical clustering. One possibility for the distance metric is to consider the mean or average linkage or UPGMA. Here, the distance between each pair of observations in each cluster is added up and divided by the number of pairs to get an average inter-cluster distance. This can be computed as:

$$\frac{1}{|A||B|}\sum_{a\in A}\sum_{b\in B} d(a,b),$$

where A and B are two different clusters and d is a function (e.g. Euclidean distance or correlation metric) which measures the distance between two elements a and b.

7.2.1.1 Hierarchical clustering in R

In R, Hierarchical clustering can be performed in two simple steps. First, we can compute the distance between observations (rows of a `data.table`) across variables (columns of a `data.table`) with the help of the function `dist()`. Here, the Euclidean distance between rows is computed by default. Alternatives include the Manhattan or Minkowski distance. As an example, we provide the distance matrix of the `mpg` and `cyl` columns of the dataset `mtcars`:

```
d <- dist(mtcars[, c("mpg", "cyl")])  # find distance matrix
```

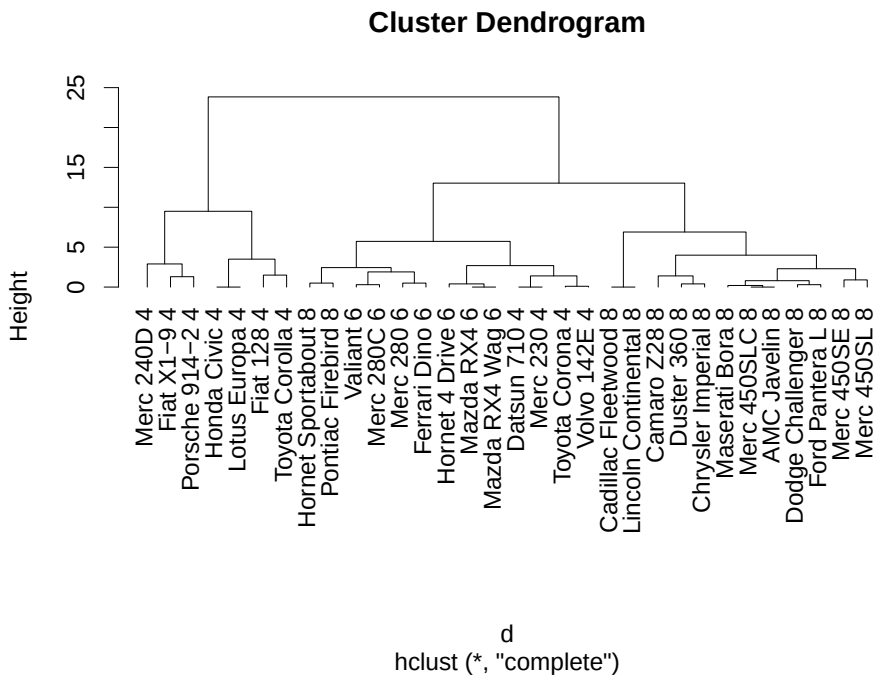
Then, we can use the resulting distance matrix to perform Hierarchical clustering with the help of the function `hclust()`:

```
hc <- hclust(d)  # apply hierarchical clustering
names(hc)
```

```
## [1] "merge"       "height"      "order"       "labels"      "method"
## [6] "call"        "dist.method"
```

The results of Hierarchical clustering can be shown using a dendrogram. Here, observations that are determined to be similar by the clustering algorithm are displayed close to each other in the `x`-axis. The height in the dendrogram at which two clusters are merged represents the distance between those two clusters.

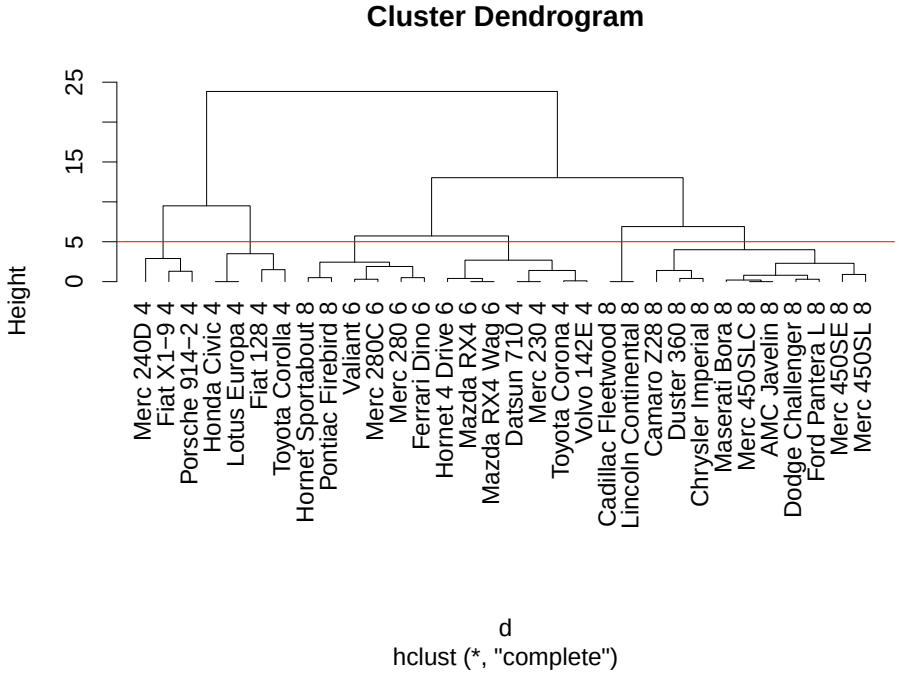
```
par(cex = 1.8)
plot(hc, hang = -1, labels = paste(rownames(mtcars), mtcars[, "cyl"]))
```



Now we can ask ourselves the question of how to actually obtain a partition from Hierarchical clustering. For this, we can apply a very ad-hoc method: we can simply cut the dendrogram at a certain height.

For example, we decide to cut the dendrogram at a height of 5, which results in 6 clusters:

```
par(cex = 1.8)
plot(hc, hang = -1, labels = paste(rownames(mtcars), mtcars[, "cyl"]))
abline(h = 5, col = "red")
```



We can often decide on the height threshold based on visual inspection or on the number of groups we wish to obtain.

For instance, we can set the height threshold to 10. This results in 3 clusters:

```
cl1 = cutree(hc, h = 10)
table(cl1)
```

```
## cl1
##  1  2  3
## 13 12  7
```

Alternatively, we can set the number of desired clusters to three:

```
cl2 = cutree(hc, k = 3)
table(cl2)
```

```
## cl2
##  1  2  3
## 13 12  7
```

We can compare both alternatives with the help of the function `table()` which returns a contingency table of the counts of each cluster:

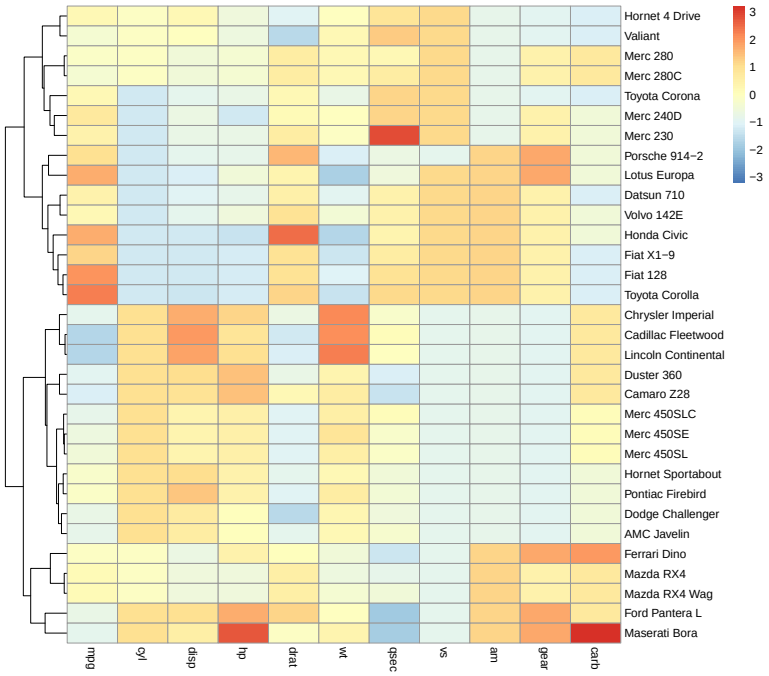
```
table(cl1, cl2)
```

```
##     cl2
## cl1  1  2  3
##   1 13  0  0
##   2  0 12  0
##   3  0  0  7
```

7.2.1.2 Pretty heatmaps including Hierarchical clustering

As illustrated before, the library `pheatmap` enables the easy creation of heatmaps. In the previous example, we set the parameters `cluster_rows` and `cluster_cols` to `FALSE` to avoid the default computation of Hierarchical clustering. If we want to include Hierarchical clustering, we can simply set these parameters to `TRUE` or let R consider its default values.

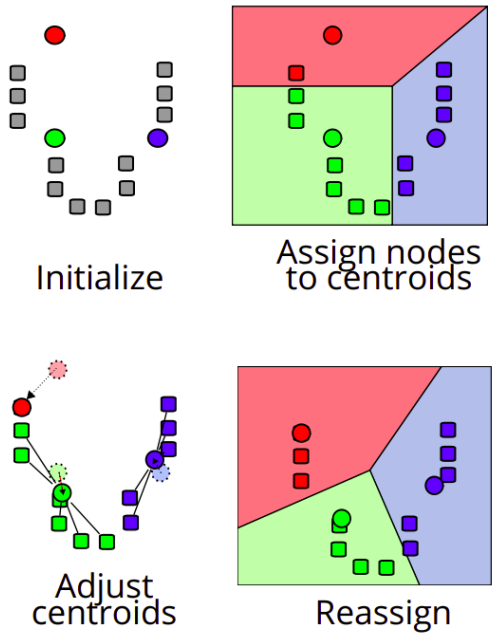
```
pheatmap(mtcars, cluster_rows = T, cluster_cols = F, scale = "column")
```



7.2.2 k-Means clustering

The most popular partitioning method is probably k-Means clustering. This clustering algorithm aims to partition n observations into k clusters. It is an iterative algorithm that aims to find local maxima in each iteration. Each observation is assigned to the cluster with the nearest mean, which serves as a prototype of the cluster. The Euclidean distance is typically used as a metric to define the distance between observations.

7.2.2.1 Steps for performing k-Means clustering



1. First, we have to specify the number of clusters k, e.g. $k = 3$. Then, we have to choose the k initial centroids (one for each cluster). Different methods such as random sampling are available for this task.
2. Next, we assign each sample to its nearest centroid by computing the Euclidean distance between each observation to each centroid. Here, we want to minimize the distance between the samples and its assigned centroids. Formally, it can be expressed as:

$$\underset{\mathbf{S}=\{S_1,\ldots,S_k\}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x}\in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 ,$$

   where $\mathbf{S}$ a the set of clusters, which together contain the n observations and μ_i is the centroid of cluster S_i.
3. Afterwards, we create new centroids by taking the mean value of all of the observations assigned to each previous centroid.
4. Then, we compute the difference between the old and the new centroids. We repeat steps 2 and 3 until the difference between centroids is less than a previously defined threshold.

7.2.2.2 Considerations and drawbacks of k-Means clustering

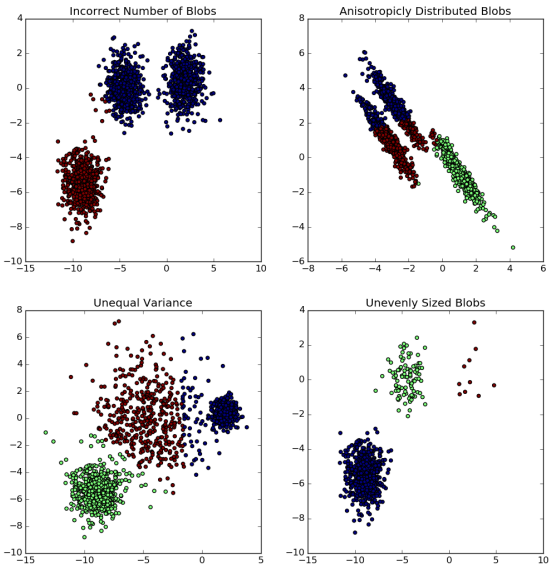
Even though k-Means clustering works properly in many applications, it requires the analyst to specify the number of clusters k to extract, which is not always a trivial task. The algorithm starts and ends with a fixed number of clusters.

Furthermore, the performance of k-Means clustering strongly depends on the initialization of the centroids for each cluster.

We also remark that convergence to a local minimum may produce counterintuitive (“wrong”) results.

7.2.2.3 Assumptions in k-Means clustering

In these examples, k-Means will not work properly:



We have to make sure that the following assumptions are met when performing k-Means clustering:

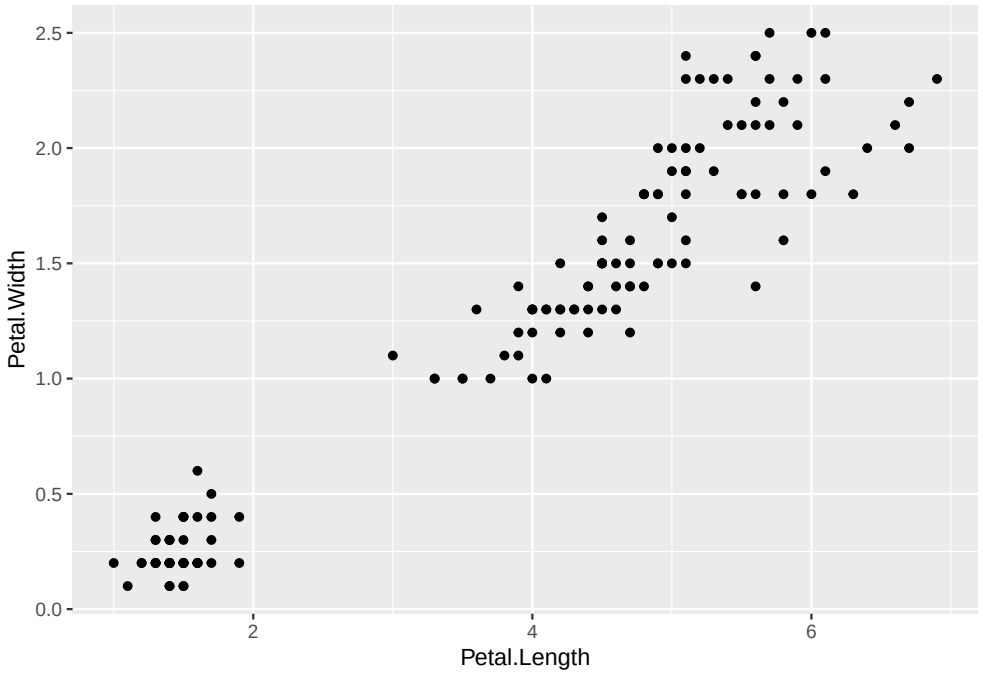
- The number of clusters k is properly selected
- The clusters are isotropically distributed
- The cluster have equal (or similar) variance
- The clusters are of similar size

7.2.2.4 k-Means clustering in R

As an example, we want to cluster the observations in the `iris` dataset according to their petal lengths and their petal widths. Similar lengths and widths should be found in the same cluster. Intuitively, observations from the same species (e.g. Setosa) should be found in the same cluster.

Since it is not clear at first, how many clusters we should build, we first try to plot the data:

```
library(datasets)
data(iris)
qplot(Petal.Length, Petal.Width, data = iris)
```



After observing the previous plot, we aim to group the observations into 3 clusters. In R, this can be easily achieved with the function `kmeans()`:

```
k <- 3
clust_km <- kmeans(iris[, c("Petal.Length", "Petal.Width")], k)
names(clust_km)
```

```
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

```
table(clust_km$cluster)
```

```
##
##  1  2  3
## 52 48 50
```

We can then assign the result of the clustering algorithm as a new column of the iris dataset:

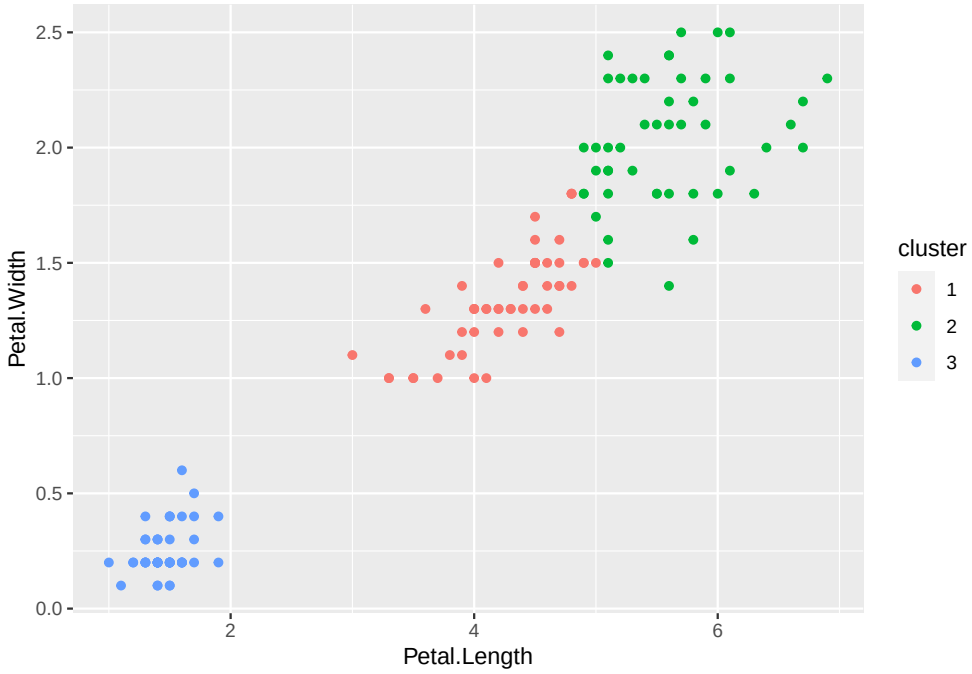
```
iris <- cbind(iris, cluster = factor(clust_km$cluster))
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species cluster
## 1          5.1         3.5          1.4         0.2  setosa       3
## 2          4.9         3.0          1.4         0.2  setosa       3
## 3          4.7         3.2          1.3         0.2  setosa       3
## 4          4.6         3.1          1.5         0.2  setosa       3
## 5          5.0         3.6          1.4         0.2  setosa       3
## 6          5.4         3.9          1.7         0.4  setosa       3
```

We can compare the actual `Species` column to the resulting clustering results with the two following plots:

```
qplot(Petal.Length, Petal.Width, color = Species, data = iris)
```

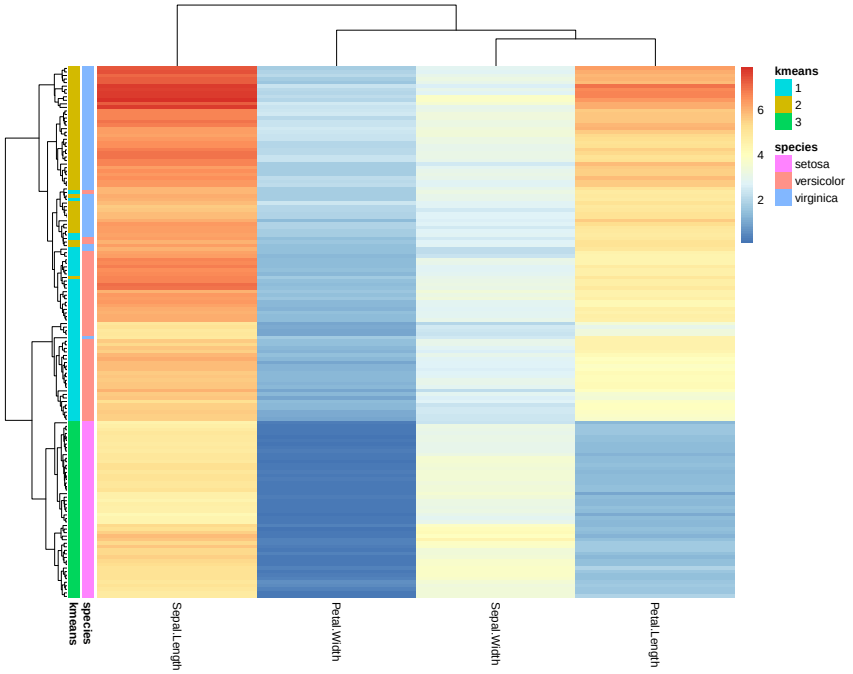

```
qplot(Petal.Length, Petal.Width, color = cluster, data = iris)
```



7.2.2.5 Annotating heatmaps with clustering results

We can conveniently include annotations to the heatmaps created by the library `pheatmap`. In this manner, we can compare the results of the k-Means clustering to the Hierarchical clustering and to the actual species of every observation:

```
plot.data <- iris[, 1:4]
rownames(plot.data) <- paste("iris", 1:nrow(plot.data), sep = ".")
row.ann <- data.frame(species = iris[, 5], kmeans = iris[, 6])
rownames(row.ann) <- rownames(plot.data)
pheatmap(plot.data, annotation_row = row.ann, show_rownames = F)
```



In this example, we see that both clustering methods seem to coincide with the actual species of most observations.

7.2.3 Difference between k-Means and Hierarchical clustering

Both Hierarchical clustering and k-Means clustering are established clustering algorithms. Here, we briefly state a few differences that may be considered, when deciding which algorithm to apply in practice.

The time complexity of k-Means clustering is linear, while that of Hierarchical clustering is quadratic. This implies that Hierarchical clustering can not handle extremely large datasets as efficiently as k-Means clustering.

In k-Means clustering, we start with a random choice of centroids for each cluster. Hence, the results produced by the algorithm are strongly dependent on the initialization method. Therefore, the results might differ when running the algorithm multiple times. Hierarchical clustering produces reproducible results.

Another difference is that k-Means clustering requires prior knowledge of the number of clusters we want to divide our observations into. However, we flexibly choose the number of clusters we find appropriate in Hierarchical clustering by interpreting the dendrogram and setting a height threshold.

7.2.4 Rand Index for cluster comparison

The proper evaluation and comparison of clustering results is certainly a challenging task. When clustering in only two groups, we could use evaluation measures from classification, which we will introduce later. Moving from two partitions of the data into arbitrarily many groups requires new ideas.

We remark that a partition is in our context the result from a clustering algorithm and, therefore, the divided dataset into clusters. Generally, two partitions (from different clustering algorithms) are considered to be similar when many pairs of points are grouped together in both partitions.

The Rand index is a measure of the similarity between two partitions. Formally, we introduce the following definitions:

- $S = \{o_1, \ldots, o_n\}$ a set of n elements (or observations)
- First partition $X = \{X_1, \ldots, X_k\}$ of S into k sets
- Second partition $Y = \{Y_1, \ldots, Y_l\}$ of S into l sets

- a number of **pairs** of elements of S that are **in the same set** in X and in Y
- b number of **pairs** of elements of S that are **in different sets** in X and in Y
- $\binom{n}{2}$ total number of pairs of elements of S

Then, the Rand index can be computed as

$$R = \frac{a+b}{\binom{N}{2}}$$

7.2.4.1 Properties of the Rand index

By definition, the Rand index has values between 0 and 1. A Rand index of 1 means that all pairs that are in the same cluster in the partition X are also in the same cluster in the partition Y **and** all pairs that are not in the same cluster in X are also not in the same cluster in Y. Hence, the two partitions are identical with a Rand index of 1. In general, the higher the Rand index, the more similar are both partitions.

7.2.4.2 Application of the Rand index in Hierarchical clustering

We can compute the Rand index for Hierarchical clustering if we fix the number of clusters. For instance: we perform a Hierarchical clustering and cut the tree in order to get 10 clusters. On the other hand, someone else had run k-Means clustering with $k = 9$ on the same data. We wonder how your two clusterings agree. The Rand index gives some similarity measure for that.

7.3 Dimensionality reduction with PCA

As described in the previous chapters, visualizing data is a crucial task of data analysis. However, this becomes challenging with high dimensional data. This contributes to the motivation for developing and using dimensionality reduction techniques. Moreover, dimensionality reduction allows for easier exploration and analysis of data.

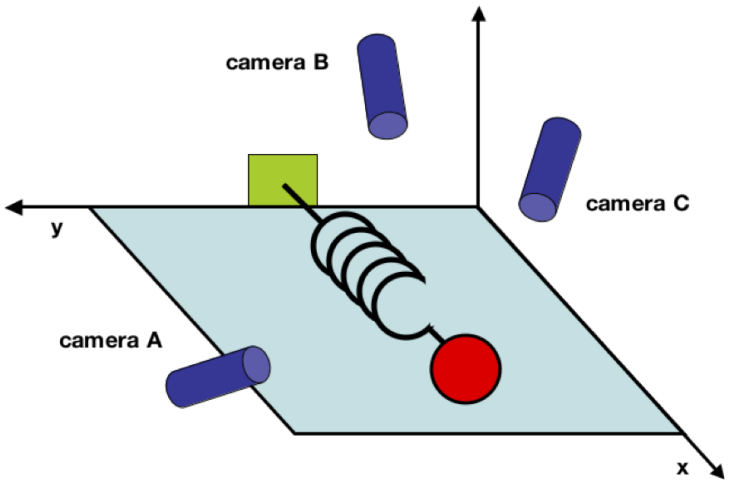
Principal component analysis, or PCA, is widely used for dimensionality reduction. PCA is a statistical procedure that uses an orthogonal transformation to convert a (large) set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, which are denoted as principal components. In this manner, PCA aims to reduce the number of variables, while preserving as much information from the original dataset as possible.

Intuitively, PCA can be thought of as fitting an n-dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipsoid is small, then the variance along that axis is also small, and, by omitting that axis and its corresponding principal component from our representation of the dataset, we lose only a commensurately small amount of information.

Principal components are the underlying structure in the data. Each of these explains a percentage of the total variation in the dataset. The first principal component has the largest possible variance. Respectively, the second principal component has the second-largest possible variance. High-dimensional data is often visualized by plotting the first two principal components after performing PCA.

7.4 An example for introducing PCA

As an example, we can study the motion of the physicist's ideal spring. We know a priori that the dynamics can be expressed as a function of a single variable x. Since we are rather ignorant experimenters in this field, we do not know how many axes we have and which of these axes are important.



We set up an experiment by first placing three cameras around the system. Each camera records a two dimensional image at 200 Hz. We do not know the correct x, y, z so we set up the cameras with axes a, b, c at arbitrary angles.

We now want to answer the question of how to get from our experiment to a simple equation for x. We know that we often have more dimensions measured than actually needed and that the data is sometimes noisy. In this scenario, the goal of PCA is to identify the most meaningful basis to re-express the data to reveal hidden dynamics and to determine that a unit basis in the direction of x is the most important dimension. More precisely we want to investigate, whether there is another basis that is a linear combination of the original basis, that best represents our data.

7.4.1 PCA terminology

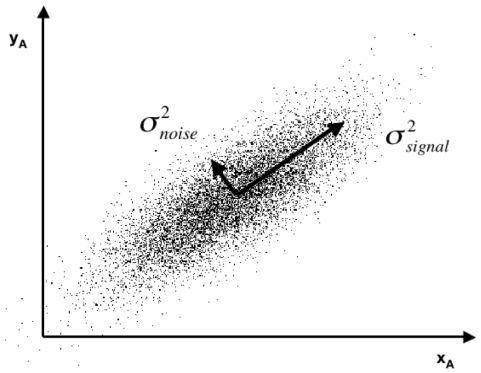
Formally, we consider a dataset represented by a $\mathbf{X}$ with n samples and p measured variables. We assume that the data is centered, i.e. all column means are equal to zero. We can possibly also assumed that the data is scaled, i.e. the standard deviation of all columns is equal to one.

In PCA, the matrix X is linearly transformed by a matrix W, which is called the loading or rotation matrix. The columns of W are the new basis vectors, which we refer to as the principal components. The column vectors of $\mathbf{T} = XW$ represent the projection of the data on the principal components.

We can now formulate the following questions we want to address:

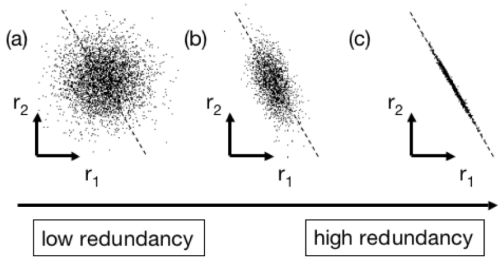
- What is the “best” way to re-express the data?
- What is a good choice for the new basis?
- Which properties would we like the transformed data to exhibit?

Mainly, we want to guarantee a **high signal to noise ratio** (SNR)



$$SNR = \frac{\sigma^2_{signal}}{\sigma^2_{noise}}$$

and **low redundancy** after PCA.



For formalizing these criteria, we denote the covariance matrix of X as

$$S(X) = X^T X,$$

where S_{ii} is variance of variable i ($= \sigma_i^2$) and S_{ij} is covariance between variables i and j as measure of redundancy. Then, the covariance matrix of the transformed data is defined as

$$S(Y) = Y^T Y = (XW)^T XW$$

In PCA, we aim to select first the direction with largest variance S_{ii} (best SNR) and we should have no redundancy, so $S_{ij} = 0$ for $i \neq j$. This last aspect corresponds to an orthonormal basis W.

7.4.2 Algorithm for PCA

Intuitively, we can describe the PCA algorithm in three main steps:

1. Find the direction w_1 that maximizes the variance in X
2. Look for the next direction w_2 perpendicular to w_1 that maximizes the variance
3. continue until all p directions have been identified

These steps can be translated into computation steps in linear algebra:

1. Compute the covariance matrix $S(X) = X^T X$
2. Set W to the matrix of eigenvectors of the covariance matrix $S(X)$
3. The eigenvalues of $X^T X$ equal to the variance of the projection of X

We remark that the principal components of a matrix X are the orthonormal directions maximizing the variance when X is projected onto. The principal components are the new basis of the data.

7.4.3 PCA in R

PCA can be easily performed in R by using the built-in function `prcomp()`. In the following examples, we perform PCA on the first four variables of the built-in dataset `mtcars`. We set the two arguments `scale` and `center` to `TRUE` so that the data is first centered and scaled before performing PCA.

```
pca_res <- prcomp(mtcars[, 1:4], center = TRUE, scale. = TRUE)
names(pca_res)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

The output can be stored in `pca_res`, which contains information about the center point (`center`), scaling (`scale`), standard deviation (`sdev`) of each principal component, as well as the values of each sample in terms of the principal components (`x`) and the relationship between the initial variables and the principal components (`rotation`).

An overview of the PCA result can be obtained with the function `summary()`, which describes the standard deviation, portion of variance and cumulative proportion of variance of each of the resulting principal components (`PC1`, `PC2`, ...). We remark that the cumulative proportion is always equal to one for the last principal component.

```
pca_sum <- summary(pca_res)
pca_sum
```

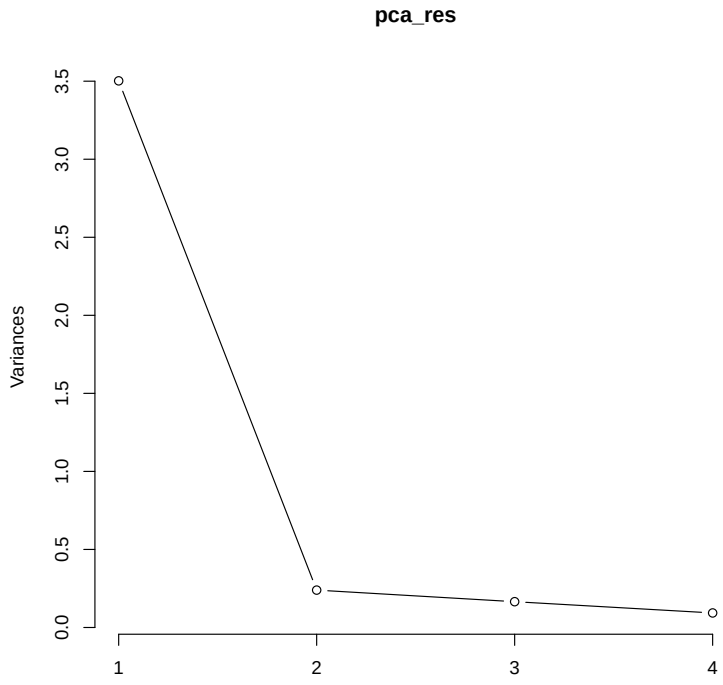
```
## Importance of components:
##                           PC1     PC2     PC3     PC4
## Standard deviation     1.8714 0.48934 0.40652 0.30517
## Proportion of Variance 0.8755 0.05986 0.04132 0.02328
## Cumulative Proportion  0.8755 0.93540 0.97672 1.00000
```

In this example, the first principal component explains 87.55% of the total variance and the second one 5.986%. So, just `PC1` and `PC2` can explain approximately 93.5% of the total variance.

7.4.3.1 Plotting PCA results in R

Plotting the results of PCA is particularly important. The so-called **scree plot** is a good first step for visualizing the PCA output, since it may be used as a diagnostic tool to check whether PCA works well on the selected dataset or not.

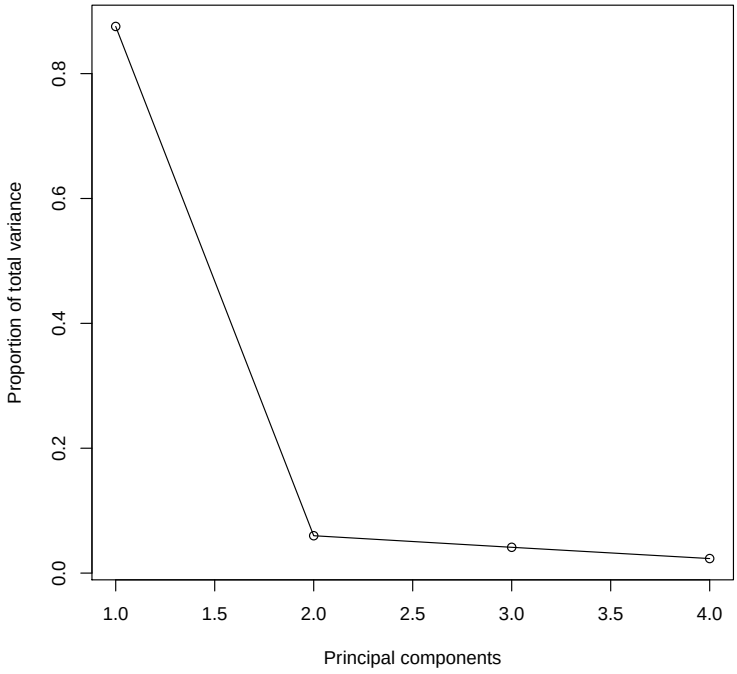
```
plot(pca_res, type = "l")
```



The scree plot shows the variance in each projected direction. The y-axis is eigenvalues, which essentially stand for the amount of variation. We can use a scree plot to select the principal components to keep. If the scree plot has an 'elbow' shape, it can be used to decide how many principal components to use for further analysis. For example, we may achieve dimensionality reduction by transforming the original four dimensional data (first four variables of `mtcars`) to a two-dimensional space by using the first two principal components.

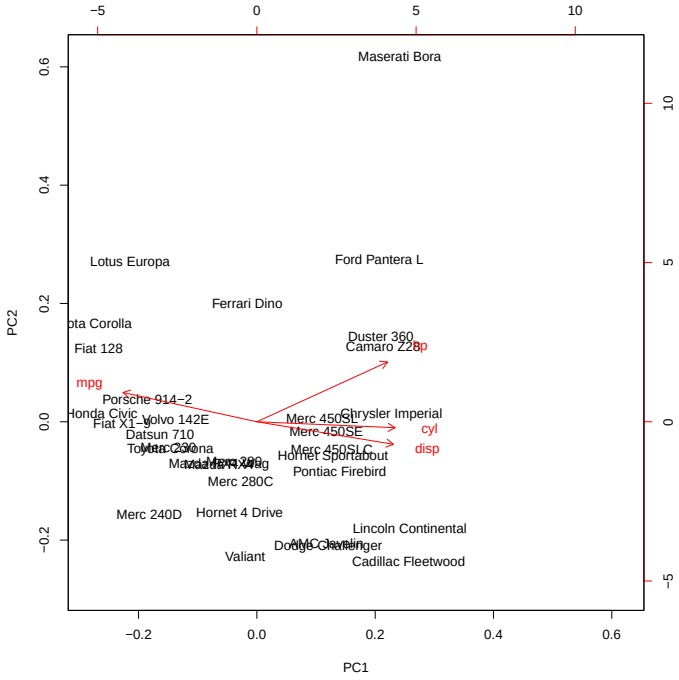
A variant of the scree plot can be considered by plotting the proportion of the total variance for every principal component.

```
plot(pca_sum$importance[2, ], type = "l", xlab = "Principal components", ylab = "Proportion of total varian
points(pca_sum$importance[2, ])
```



The **biplot** shows the projection of the data on the first two principal components. It includes both the position of each sample in terms of `PC1` and `PC2` and also shows how the initial variables map onto this. The correlation between variables can be derived from the angle between the vectors.

```
biplot(pca_res)
```



We can access the projection of the original data on the principal components by using the function `predict` as follows:

```
predict(pca_res)
```

```
##                     PC1        PC2        PC3        PC4
```

```
## Mazda RX4           -0.6767382 -0.199976205  0.138261099 -0.35866245
## Mazda RX4 Wag       -0.6767382 -0.199976205  0.138261099 -0.35866245
## Datsun 710          -1.7315422 -0.057999871  0.503432795  0.20231633
## Hornet 4 Drive      -0.3095112 -0.424895223 -0.316386321  0.12866096
## Hornet Sportabout    1.3627600 -0.164154097 -0.672960127 -0.06947829
## Valiant             -0.2078417 -0.628965554  0.226625858 -0.04213202
## Duster 360           2.2197422  0.398362269  0.030790698  0.10149522
## Merc 240D           -1.9243334 -0.430817504  0.076310853  0.31984579
## Merc 230            -1.5834761 -0.117769171  0.372404944  0.37031839
## Merc 280            -0.4056138 -0.182774161  0.361959531 -0.28750994
## Merc 280C           -0.2903254 -0.279187791  0.539066036 -0.28528975
## Merc 450SE           1.2436778 -0.043809833 -0.024340777 -0.47336217
## Merc 450SL           1.1695638  0.018170357 -0.138194959 -0.47478944
## Merc 450SLC          1.3424965 -0.126450087  0.127464799 -0.47145915
## Cadillac Fleetwood   2.7155808 -0.653339795 -0.021682040  0.57163417
## Lincoln Continental  2.7372446 -0.498778386  0.048811553  0.53531221
## Chrysler Imperial    2.4074735  0.034343665 -0.381172211  0.46405178
## Fiat 128            -2.8325258  0.344756743 -0.646960724 -0.02205276
## Honda Civic         -2.7790074  0.041645405 -0.411008765 -0.06661769
## Toyota Corolla      -2.9941488  0.455271315 -0.807485411 -0.06461595
## Toyota Corona       -1.5468147 -0.129239713  0.626409027  0.27399697
## Dodge Challenger     1.2781168 -0.585446574 -0.147529439 -0.33209823
## AMC Javelin          1.2456549 -0.570040098 -0.051857250 -0.40133072
## Camaro Z28           2.2612578  0.355257204  0.198524483  0.05328946
## Pontiac Firebird     1.4849188 -0.232767911 -0.901129006  0.12889521
## Fiat X1-9           -2.4113213 -0.007237189 -0.003023901 -0.01247117
## Porsche 914-2       -1.9589665  0.105957663  0.043702360  0.25379969
## Lotus Europa        -2.2687411  0.745915608 -0.362779678  0.17288862
## Ford Pantera L       2.1937302  0.759777960 -0.081924658  0.09881713
## Ferrari Dino        -0.1716195  0.552846586  0.501181915 -0.27900647
## Maserati Bora        2.5571561  1.711388877  0.374656997  0.01746634
## Volvo 142E          -1.4501082  0.009931715  0.660571220  0.30675036
```

7.5 Summary

In this chapter, we introduced heatmaps as a powerful and useful visualization for high dimensional data. Heatmaps can be generated in R with both libraries `ggplot` and `pheatmap`.

We also described two clustering algorithms: Hierarchical clustering and k-Means. Comparing clustering results can be achieved by computing the Rand index.

Furthermore, we introduced dimensionality reduction with PCA, which can be used for easier visualization of high dimensional data in a two dimensional space.

7.6 Resources

- This chapter is partly based on Hadley Wickham's book ggplot2: elegant graphics for data analysis
- Graphic Principles cheat sheet: https://graphicsprinciples.github.io/
- In-depth documentations:
  - ggplot2 cheetsheet
  - Introduction to ggplot2
  - plotly for R

- Data Analysis and Prediction Algorithms with R

Chapter 8

Reproducible data analysis with Snakemake

Basic idea:

- Decompose workflow into “rules” (steps).
- Rules define how to obtain output files from input files.
- Snakemake infers dependencies and execution order with a Directed Acyclic Graph.
- Any change in the input, only affected downstream rules will be executed.
- Disjoint paths in the DAG of jobs can be executed in parallel.
- Re-run all workflows with single `snakemake` command.

Example

```
rule sort:
    input:
        "path/to/dataset.txt"
    output:
        "dataset.sorted.txt"
    shell:
        "sort {input} > {output}"
```

```
rule sort:
    input:
        a="path/to/{dataset}.txt"
    output:
        b="{dataset}.sorted.txt"
    run:
        with open(output.b, "w") as out:
            for l in sorted(open(input.a)):
                print(l, file=out)
```

```
DATASETS = ["D1", "D2", "D3"] # use native python code

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=DATASETS)

rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
```

```
    shell:
        "sort {input} > {output}"
```

Chapter 9

Quizes

9.1 data.table

1. From DT, display a named vector with the means of *y* and *v*. The names of the elements are mean_y and mean_v. Hint: remember which command returns a vector and which one a data.table.

   A. DT[, c(mean_y = mean(y), mean_v = mean(v))]

   B. DT[, list(mean_y = mean(y), mean_v = mean(v))]

   C. DT[, .(mean_y = mean(y), mean_v = mean(v))]

2. As part of a new project, you got the results of a set of experiments from a lab in a data.table (x) with 3 columns: experiment id, sample id and value. On each experiment, more than one sample was measured. On another data.table (y) you got the dates of all the experiments the lab has made after a certain date. Some experiments are not part of your project, but the lab did not subset the table. Before a certain date, the lab could not find the experiments date, but you don't want to discard those results. You want to merge both data tables in order to have only one with 4 columns: experiment id, sample id, value and experiment date. Which merge would you use? Hint: Sketch both data tables if necessary.

   A. Inner, all = FALSE

   B. Full, all = TRUE

   C. Left, all.x = TRUE

   D. Right, all.y = TRUE

9.2 tidy data

1. What transformation are required to tidy following data?

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

A. Cast

B. Melt

C. Cast and Melt

D. Data are tidy already

2. What transformation are required to tidy following data?

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

A. Gather

B. Unite

C. Melt and cast

D. Melt, unite and cast

9.3 ggplot2

1. What's the result of the following command? `ggplot(data = mpg)`. Hint: ggplot builds plot layer by layer.

A. Nothing happens

B. A blank figure will be produced

C. A blank figure with axis will be produced

D. All data in `mpg` will be visualized

2. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty))`. Hint: ggplot builds plot layer by layer

   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

3. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty)) + geom_point()`. Hint: ggplot builds plot layer by layer.

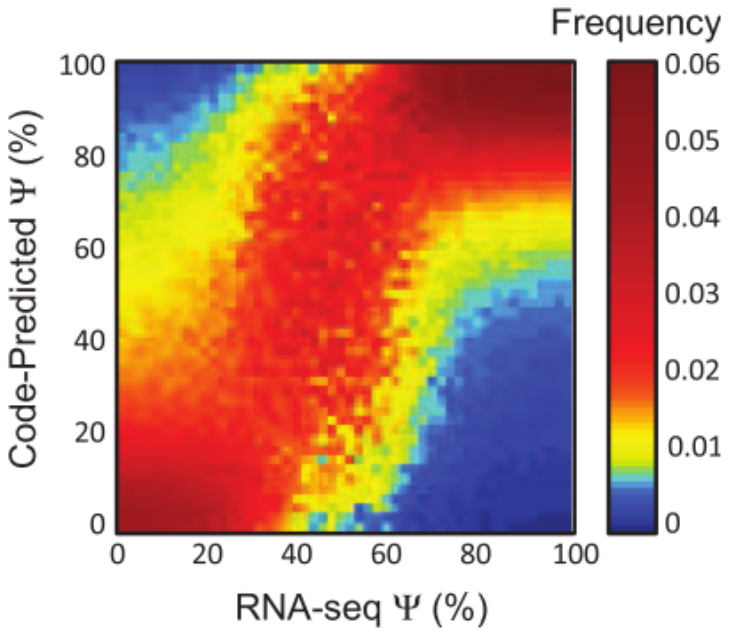
   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

4. What's wrong with the following plot?



Xiong et al., Science 2014

5. Which item is *not* chart junk?

   A. a bright red plot border

   B. light grey major grid lines

   C. bold labels and yellow grid lines

   D. data labels in Batik Gangster font

Chapter 10

Quiz solutions

10.1 data.table

1. A, including the column names inside c() will return a vector.

2. C, inner join will omit the experiments whose dates are not provided. Full join will add rows of the experiments we are not interested in. Left join will leave the rows we are interested in and simply add the dates when available and NAs when not. Right join will leave us with experiments we are not interested and omit ones that we are interested but don't have the date.

10.2 tidy data

1. B, melt

   Tidy form:

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Figure 10.1: Tidy religion dataset

2. D Melt, unite and cast

   Tidy form:

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

Figure 10.2: Tidy weather dataset

10.3 ggplot2

1. B, because neither variables were mapped nor geometry specified.
2. C, because while axis x and y are mapped, no geometry is specified.
3. D Axis x and y are mapped. But no geometry specified.
4. 1. Rainbow color where there are no breakpoints. Color boundaries are created based on author's interest.
   2. Manually twisted color scale.
5. B, light grey does not draw attention and grid lines are less important than data.