Introduction to R programming for data science – day 3

Dr. Francesca Finotello
Medical University of Innsbruck, Austria

Loading
and saving files

Types of files

- R **functions** and **scripts** can be saved in .R files.
- R **objects** can be saved in .RData and .Rds files; .RData files can contain multiple objects.
- **Data.frames** can be saved into or load from plain-text files (.txt, .csv, .xslx, …).
- R **packages** containing additional functions can be installed from local archive files or repositories like CRAN, Bioconductor, and GitHub.

Absolute and relative paths

A **path** indicates the unique location of a file or directory.

It is expressed by a string of characters separated by a delimiting character (usually /).

Each component separated by the delimiting characters represents a directory.

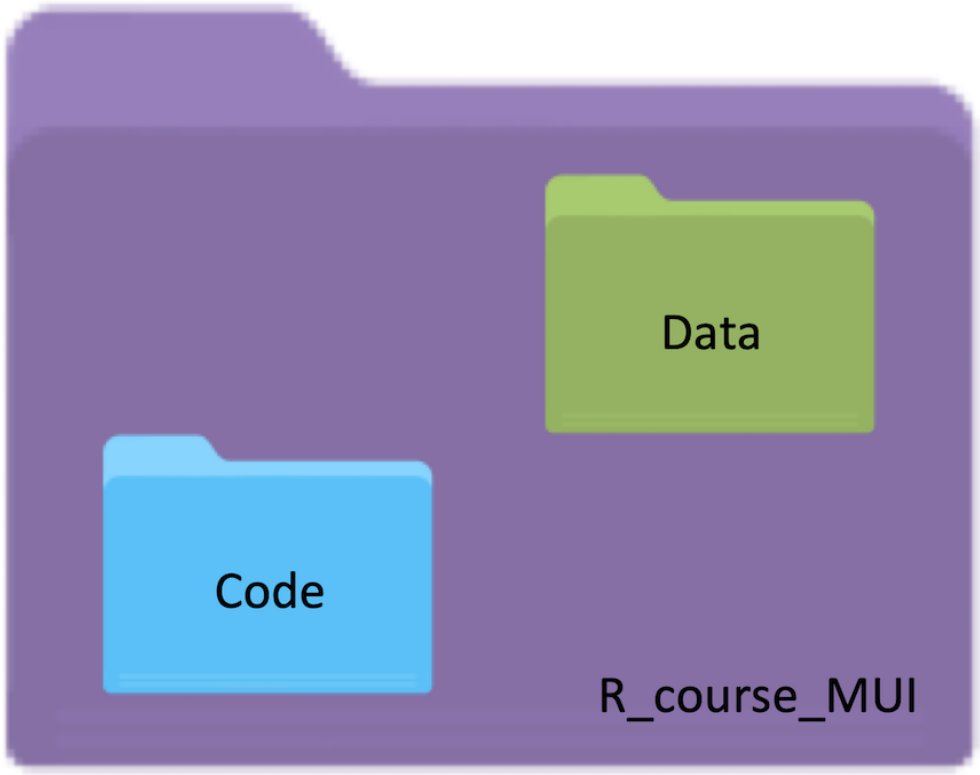
Absolute path: points to the same location in a file system, regardless of the current working directory. To do that, it must include the root directory.

Relative path: starts from some given working directory. A filename alone can be considered as a relative path based at the current working directory.

Absolute paths

/Users/francescafinotello/Desktop/R_course_MUI

/Users/francescafinotello/Desktop/R_course_MUI/Data



/Users/francescafinotello/Desktop/R_course_MUI/Code

File locations

Let's suppose that our working directory is:

/Users/francescafinotello/Desktop/R_course_MUI/Code

We can load/save whatever R script in the "Code" directory just by specifying the file name.

However, if we want to load/save a file in the "Data" directory we need to specify its location:

- Absolute path:
  /Users/francescafinotello/Desktop/R_course_MUI/Data/myfile.txt
- Relative path:
  ../Data/myfile.txt

Special characters for relative paths

.. goes one step up in the directory tree (i.e. out of the “Code” directory in the previous example)

. indicates the current directory

Exercises

According to slide #5 and supposing our working directory is /Users/francescafinotello/Desktop/R_course_MUI/Code

Ex. 3.1: What is the *relative path* of a file in the “Data” directory?

Ex. 3.2: What is the *relative path* of a file in the “Desktop” directory?

Working directory

The getwd function can be used to known the absolute path of the working directory:

```
getwd()
```

```
## [1] "/Users/francescafinotello/Dropbox/R_course_MUI/Course_slides"
```

The setwd function can be used to set the path (absolute or relative) of the working directory:

```
setwd("/Users/francescafinotello/Desktop/")
```

Handling R scripts and data files

Source R code

The function source can be used to

- Run a full script (e.g. the code implementing a full pipeline for differential gene expression analysis of RNA-seq data)

```
source("DGE_analysis.R")
```

- Import one or more R functions you want to use (e.g. your BMI function to compute the body mass index score)

```
source("BMI_function.R")
```

Save and load RData files

The function save can be used to save one ore more R objects into an .RData file. The **file** option must always be present to specify the path to the output file

```
age <- 40
height <- 180
save(age, height,                    # List of objects to be saved
  file = "Data/Day3/Example.RData")  # Path to output file
```

The function load can be used to import .RData files

```
load("Data/Day3/Example.RData")

weight <- 80
save(age, height, weight,
  file = "Data/Day3/Example_updated.RData")
```

Installing packages

CRAN or local archive files

R packages can be installed from CRAN or local archive files using the “Packages” panel of Rstudio

Packages > Install

By selecting one of the two following options

1. Package Archive File (.tgz; .tar.gz) > [Locate the archive file]
2. Install from Repository (CRAN) > [Specify the package name]

Or, for functions available on CRAN, by using directly the install.packages function

```
install.packages("xlsx")
```

Bioconductor

Bioconductor is an open source software project for the analysis of genomic data with a repository containing >1500 R packages

Bioconductor pages contain different info about a package, including:

- Installation instructions
- Package manual and vignette
- Source code as archive files

R packages can be installed from Bioconductor with the following code (example on edgeR)

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("edgeR", version = "3.8")
```

Read and write text files

Reading tables with read.table

Plain text files in table format can be loaded into R as data.frames using the function read.table

```
read.table(file,                # Input file
    header = FALSE,             # Use first row as column names?
    sep = "",                   # Column separator in the input file
    row.names = 1,              # Column containing the row names
    nrows = -1,                 # Number of rows to be read
    skip = 0,                   # Number of rows to be skipped
    check.names = TRUE,         # Check and fix column names?
                                # (e.g. "123-A" --> "X123.A")
    stringsAsFactors = TRUE, # Save strings as factors?
    ...)
```

Once you load a table into R, you can check how it looks like using head and tail, and check its dimensions with dim

Reading tables with read.delim and read.csv

Depending on the column separator in the input file, read.delim and read.csv can be also used to import files in table format

```
read.delim(file,
      header = TRUE,
      sep = "\t", ...)

read.csv(file,
      header = TRUE,
      sep = ",", ...)
```

The are both based on the read.table function, but they use different paremeter settings (e.g. “sep”)

Save data in plain text files

Data.frames and matrices can be saved into text files

```
( DF <- data.frame(name=c("Mary", "John", "Lisa"),
  age=c(19, 30, 20),
  city=c("New York", "Seattle", "New York")) )
```

```
##   name age     city
## 1 Mary  19 New York
## 2 John  30  Seattle
## 3 Lisa  20 New York
```

```
write.table(DF,
  quote = FALSE,
  sep = "\t",
  row.names = FALSE,
  col.names = TRUE,
  file =  "Data/Day3/Friends_table.txt")
```

Read and write Excel files

The xlsx package

The xlsx package provides R functions to handle Excel files (97/2000/XP/2003/2007 formats)

It is available on CRAN and can be installed from the “Packages” window or by executing

```
install.packages("xlsx")
```

Once installed, it can be loaded with library, which is the function to load (installed) R packages into R

```
library("xlsx"")
```

Read Excel files

The read.xlsx function from the xlsx package can be used to read Excel files

```
read.xlsx(file,           # File to be read
  sheetIndex,             # Number of the sheet to be read
  sheetName = NULL,       # Character indicating the sheet name
  startRow = NULL,        # First row to be read
  endRow = NULL,          # Last row to be read
  header = TRUE,          # Does the first row contain column names?
  keepFormulas = FALSE,   # Display formulae (instead of results)?
  ...)
```

Write Excel files

The write.xlsx function from the xlsx package can be used to write Excel files

```
write.xlsx(x,              # Table to be written
  file,                    # Path to the output file
  sheetName = "Sheet1",    # Character indicating the sheet name
  col.names = TRUE,        # Write column names?
  row.names = TRUE,        # Write row names?
  ...)
```

Excel and gene names

Gene name errors are widespread in the scientific literature

Mark Ziemann, Yotam Eren and Assam El-Osta

Genome Biology 2016 **17**:177
https://doi.org/10.1186/s13059-016-1044-7
Published: 23 August 2016

Abstract

The spreadsheet software Microsoft Excel, when used with default settings, is known to convert gene names to dates and floating-point numbers. A programmatic scan of leading genomics journals reveals that approximately one-fifth of papers with supplementary Excel gene lists contain erroneous gene name conversions.

Loops and control structures

Loops

Loops are used to repeat the same code multiple times. They are constructed using *reserved* words.

In R, there are three loop structures (we will use only for):

```
for (variable in sequence) {
  # Chunck of code to be repeated...
}
```

```
while (condition) {
  # Chunck of code to be repeated...
}
```

```
repeat {
  # Chunck of code to be repeated...
}
```

For loop

```
mysum <- 0
for (i in seq(1:10)) {
  mysum <- mysum + i
}
mysum
```

```
## [1] 55
```

```
days <- c("Monday", "Tuesday", "Wednesday")
for (i in 1:length(days)) {
  cat("Day ", i, " of the week: ", days[i], "\n", sep="")
}
```

```
## Day 1 of the week: Monday
## Day 2 of the week: Tuesday
## Day 3 of the week: Wednesday
```

The if/else statement (1)

The if/else statement executes a block of code if a specified condition is TRUE. If the condition is FALSE, another block of code is executed.

```
temperature <- 39
fever_thresh <- 37.2

if (temperature > fever_thresh) {
  cat("Ouch, you have a fever. Stay in bed!\n")

}  else {
  cat("Don't panic! Your body temperature is normal.\n")

}
```

```
## Ouch, you have a fever. Stay in bed!
```

The if/else statement (2)

The if/else statement can be used to consider more than two conditions.

```
glycemia <- 100
hypo_thresh <- 70 # mg/dL
hyper_thresh <- 130 # mg/dL

if (glycemia < hypo_thresh) {
  cat("Hypoglycemia\n")
}  else  if (glycemia > hyper_thresh) {
  cat("Hyperglycemia\n")
} else {
  cat("Normoglycemia\n")
}
```

```
## Normoglycemia
```

Other control statements: break

break can be used to interrupt a loop

```
x <- c(0, 1, 22, 100, 5, 8, 90, 6, 100)
wantedNum <- 100 # Number we are looking for
for (i in 1:length(x)) {
  if (x[i] == wantedNum) {
    cat(wantedNum, " found after ", i, " iterations.\n", sep="")
    break
  }
}
```

```
## 100 found after 4 iterations.
```

```
i
```

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

Other control statements: next

next can be used to skip a loop iteration

```
x <- c(0, 1, 22, -9, 5, 8)
sumPosX <- 0 # Sum of all positive numbers in x
for (i in 1:length(x)) {

  if (x[i] > 0) {
    sumPosX <- sumPosX + x[i]

  } else {
    next
  }
}
sumPosX
```

```
## [1] 36
```

Exercises

Ex. 3.3 (part 1)

We can use R to downlaod the preprocessed RNA-seq data from a study entitled “Recurrent Tumor Cell-Intrinsic and -Extrinsic Alterations during MAPKi-Induced Melanoma Regression and Early Adaptation” (GEO accession: GSE75299).

Create a directory called “RNAseqStudies” and, whithin this, two subdirectories called “Data” and “Scripts”.

Download and unzip the data from this link and save it into the “Data” folder.

This text file contains the normalized RNA-seq data (as FPKM) across genes (rows) and patients’ samples (columns).

Continues…

Ex. 3.3 (part 2)

From R, savec an R script called “GSE75299_Prepare_Data.R” into the “Scripts” folder and set the latter as working directory.

In this script, write and save the code to load the downloaded text file into a data.frame called *geneExpr*. When loading the data, set the right parameters to use the first line as header and the first column (i.e. the gene names) as row names.

Save the final data.frame into a file called “Data/GSE75299_RNAseq.RData”.

Then, clean your workspace with rm(ls=list()) and load this .RData file to check that the full procedure succeeded.

Ex. 3.4 (part 1)

Create a script to load and analyze the “Data/GSE75299_RNAseq.RData” prepared in Ex. 3.3.

In the script, initialize two empty vectors, *CD8B_pre* and *CD8B_on*, that will store the expression levels of the CD8B gene in pre- and on-treatment samples, respectively.

Create a *for* loop that iterates across all samples (i.e. data.frame columns) and, for each samples, save the expression value of the CD8B gene in:

- CD8B_pre, if the corresponding sample name contains the string “baseline” (e.g. “Pt1-baseline”)
- CD8B_on, if the corresponding sample name does NOT contain the string “baseline” (e.g. “Pt1-D85”).

Continues…

Ex. 3.4 (part 2)

Hint: you can use the grep function to check whether a string is contained in an array of strings

```
as.logical(length(grep("baseline", "Pt1-baseline")))
```

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

```
as.logical(length(grep("baseline", "Pt1-D85")))
```

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

Finally, compute the median expression of CD8B in the pre- and on-treatment samples.

Do you notice an increased or decreased expression after treatment?

Ex. 3.5, optional (part 1)

We can use the xlsx R package to handle the Supplemetary Material of a recent study entitled “Genomic and Transcriptomic Features of Response to Anti-PD-1 Therapy in Metastatic Melanoma”.

First, install the xlsx R package on your laptop following the instructions of the previous slides.

Second, download the supplementary data from this link, save it into the “Data” folder, and have a look at its format using Excel.

The “S2A” sheet contains the statistics regarding the differentially expressed genes in responding vs. non-responding patients, but the actual tabular data are not starting on the first row.

Continues…

Ex. 3.5, optional (part 2)

Create a script “Scripts/AntiPD1_Prepare_Data.R” to:

- Load the results contained in the S2A sheet using the read.xlsx function (Hint: specify the appropriate **sheetName** and **startRow** parameters) and save it into a data.frame called *DGEres*
- Rename the column called “diffAvg” as “logFC” (this are the gene log-fold-changes in responders vs. non-responders)
- Select only the genes (i.e. rows) with Pval < 0.05.
- Save the final data.frame into an Excel file called “Data/AntiPD1_DGE_results.xlsx” using the write.xlsx function

Ex. 3.6, optional

Create a script to load and analyze the “Data/AntiPD1_DGE_results.xlsx” prepared in Ex. 3.5.

In the script, initialize to zero two variables, *nUP* and *nDN*, that will count the number of up- and down-regulated genes, respectively, in responders vs. non responders.

Create a *for* loop that iterates across all genes (i.e. the data.frame rows) and add one to:

- nUP, if logFC > 0
- nDN, if logFC < 0

Print to screen the results using the cat function.

Note: in real analyses, p-values adjusted for multiple testing (FDR) should be used.

Ex. 3.7

```
y <- c(-1, 100, 2, 4)

for (i in 1:length(y)) {
  if (y[i] == 100) {
    break
  }
}
```

What is the value of *i*?

1. 100
2. 4
3. 2
4. None of the above

Ex. 3.8

```
j <- 0
x <- c(0, -1, 3, -4, 4, 2)

for (i in 1:length(x)) {
  if (x[i] > 0) {
    j <- j + x[i]
  }
}
```

What is the value of *i*?

1. 9
2. 6
3. 4
4. None of the above

Ex. 3.9

```
glycemia <- 120
hypo_thresh <- 70; hyper_thresh <- 130 # mg/dL
warn <- NA
if (glycemia < hypo_thresh) {
  warn <- "Hypoglycemia"
}  else  if (glycemia > hyper_thresh) {
  warn <- "Hyperglycemia"
} else {
  warn <- "Normoglycemia"
}
```

What is the value of *warn*?

1. Hypoglycemia
2. Hyperglycemia
3. Normoglycemia
4. NA

Plots

Plot

The plot function can be used to plot R objects. Its default usage produces a scatterplot of two variables *x* and *y*.

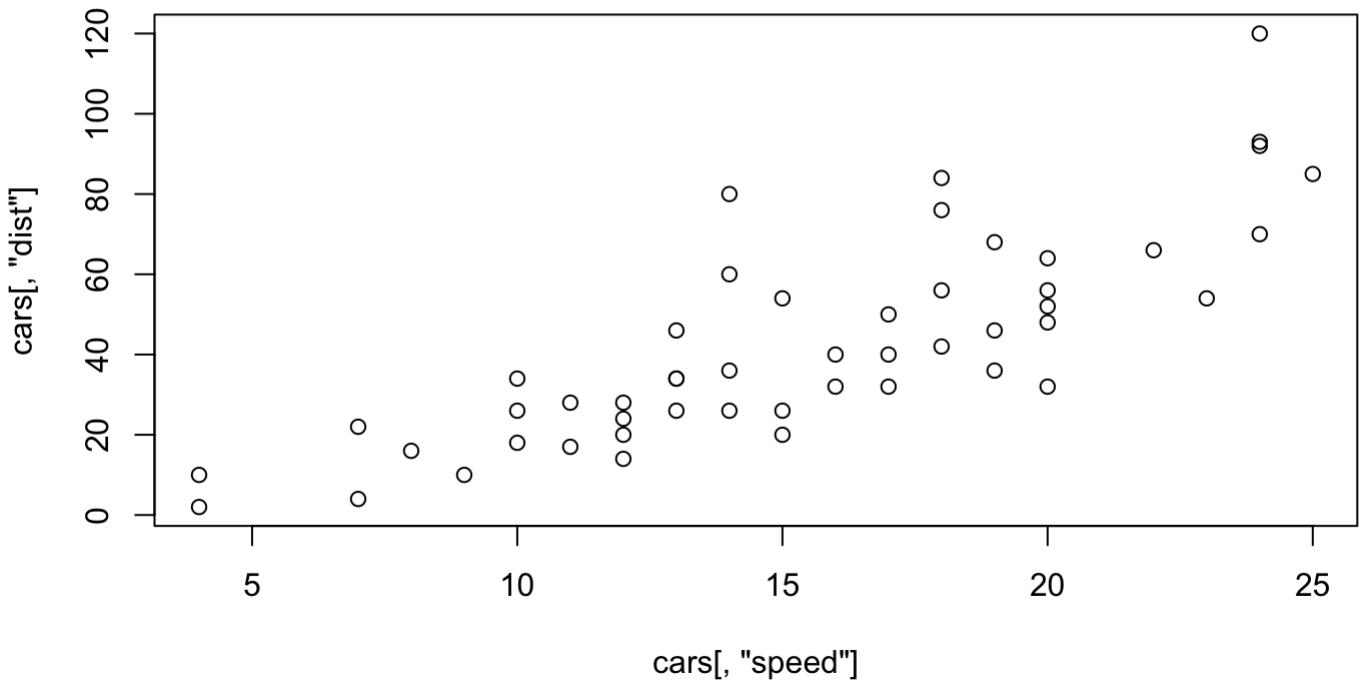
To show how plot works, we will use the dataset “cars”" (already available as part of the “stats”" package). It contains car speed and distance taken to stop recorded in the 1920s.

```
data(cars)
head(cars)
```

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

Plot the cars dataset

```
plot(cars[,"speed"], cars[,"dist"])
```

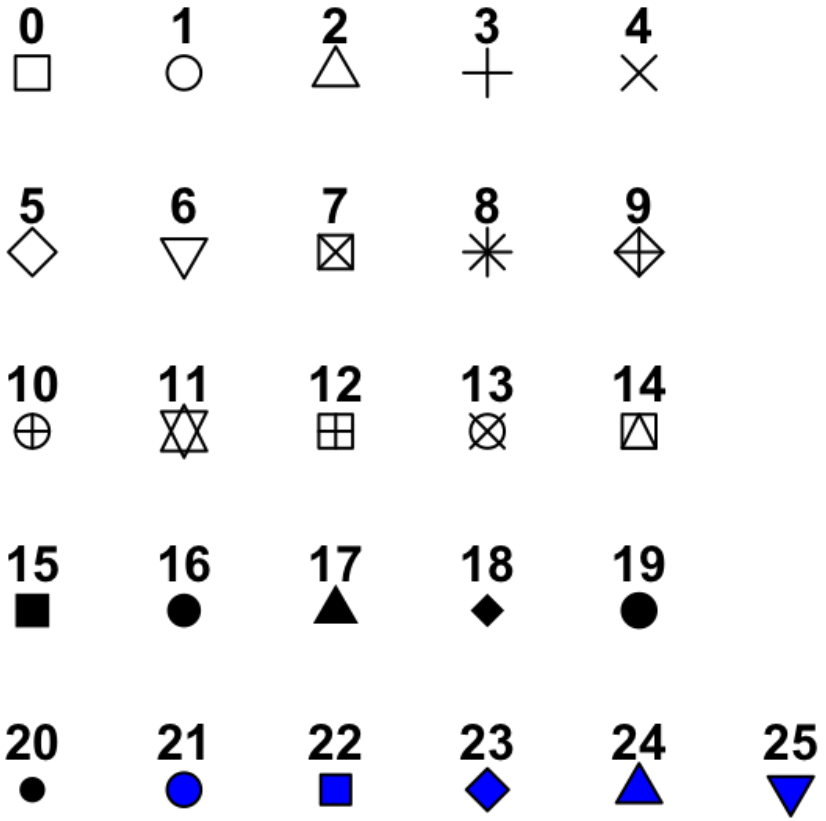


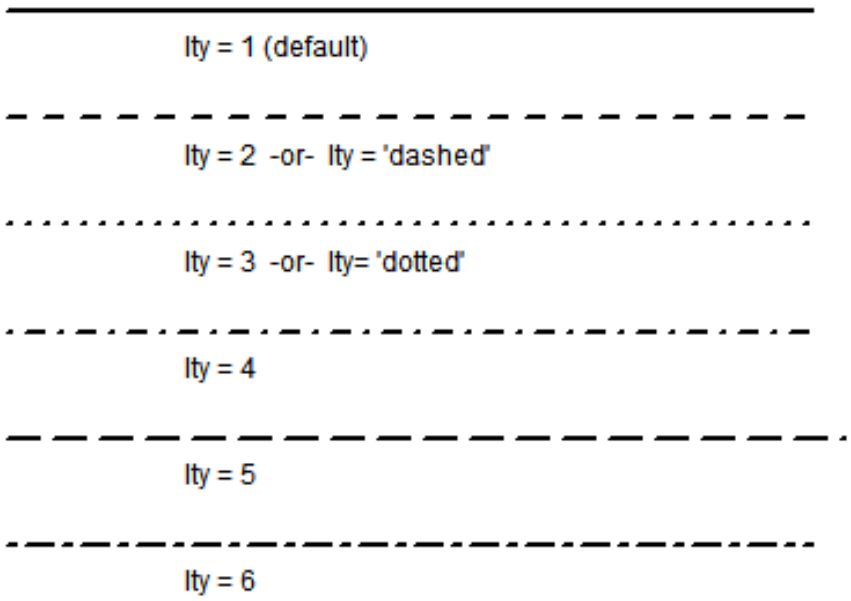
Plot arguments

The plot function has many arguments (see help(plot) and the web), among which:

- **type**: type of plot ("p" for points, "l" for lines, "b" for both…)
- **main**: title for the plot
- **xlab/ylab**: x/y-axis label
- **col**: colors for lines and points (single value or vector)
- **pch**: plotting characters or symbols (single value or vector)
- **lty**: line types (single value or vector)
- **lwd**: line width (default 1)
- **xlim/ylim**: numeric vectors giving the x/y-axis range
- **cex**: magnification (default 1; see also **cex.axis**, **cex.main**, **cex.lab**)

The pch, lty, and col arguments

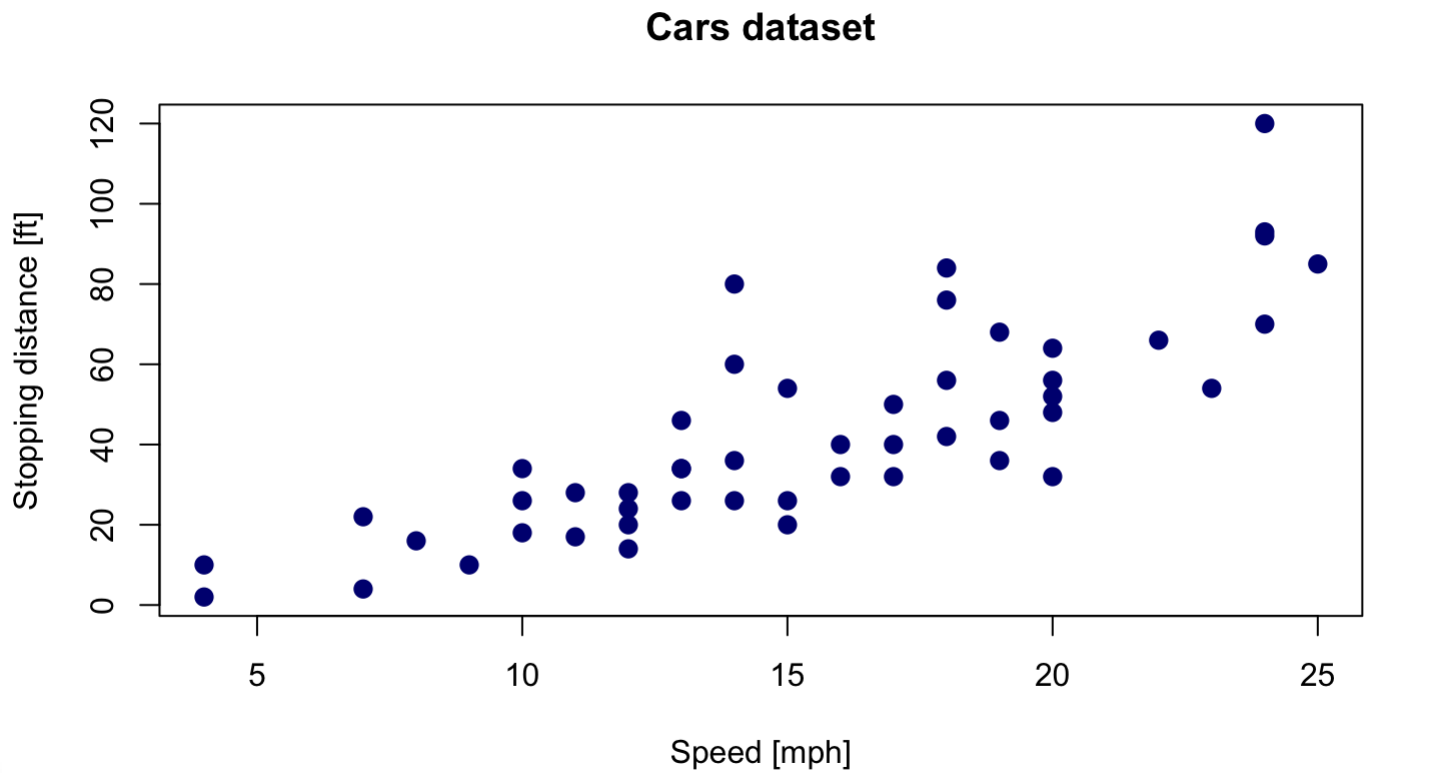




Colors can be specified as numbers, characters of color names or HEX codes, or using the rgb function (see next examples).

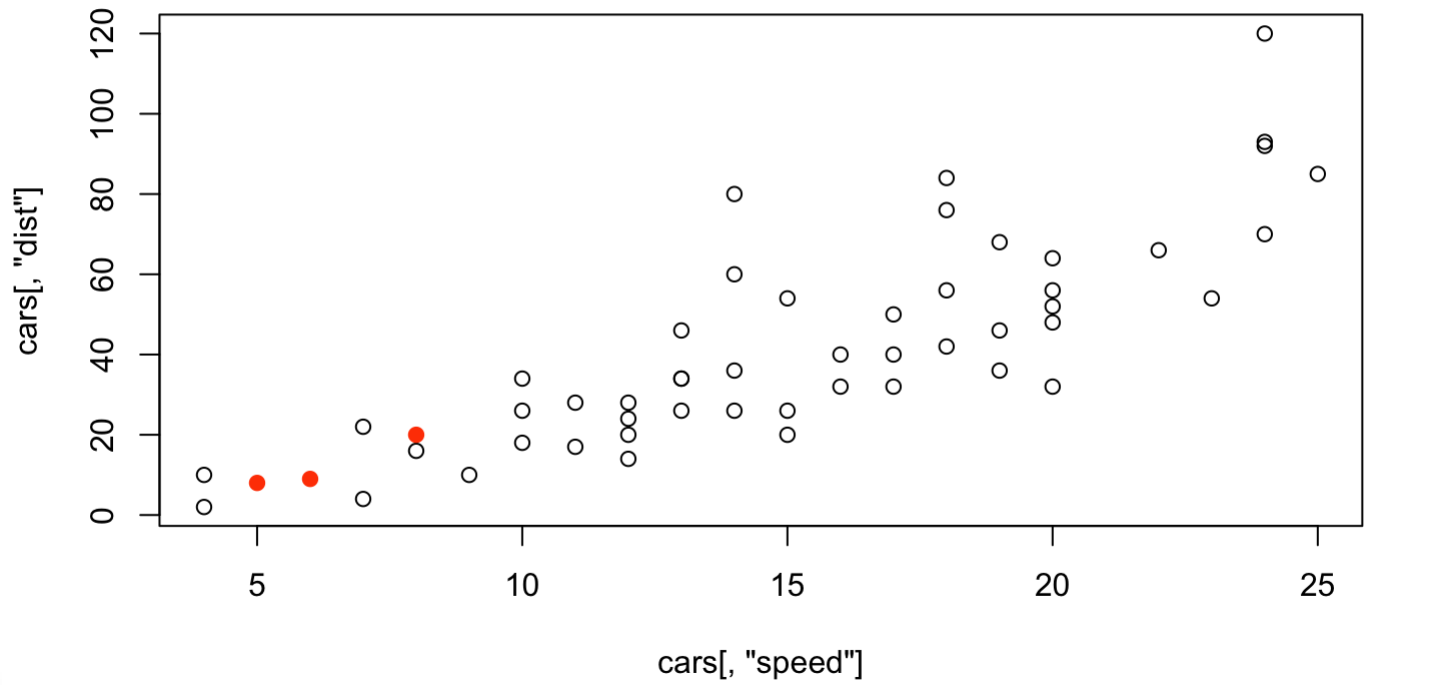
Re-plot the cars dataset

```
plot(cars[,"speed"], cars[,"dist"],
  xlab="Speed [mph]", ylab="Stopping distance [ft]",
  main="Cars dataset", pch=19, col="navyblue", cex=1.2)
```



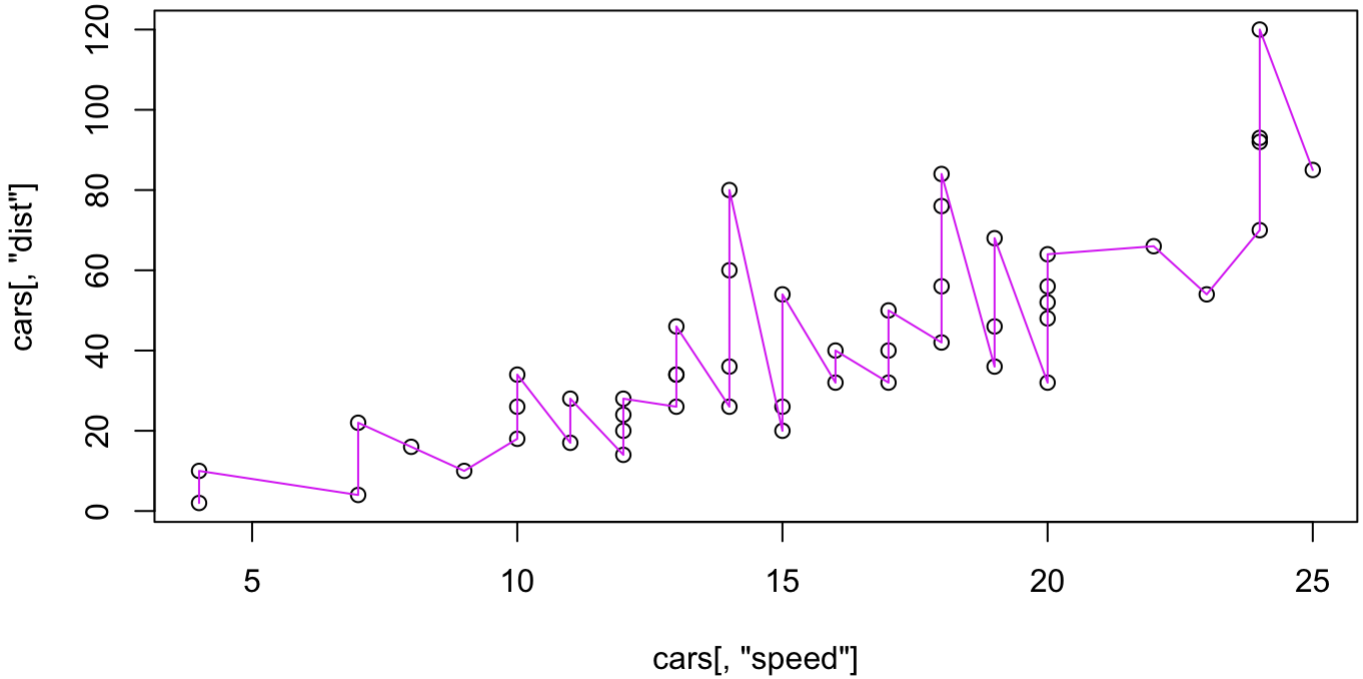
The points function

```
plot(cars[,"speed"], cars[,"dist"])
x <- c(5, 6, 8); y <- c(8, 9, 20)
points(x, y, col="orangered", pch=19) # Add points to a previous plot
```



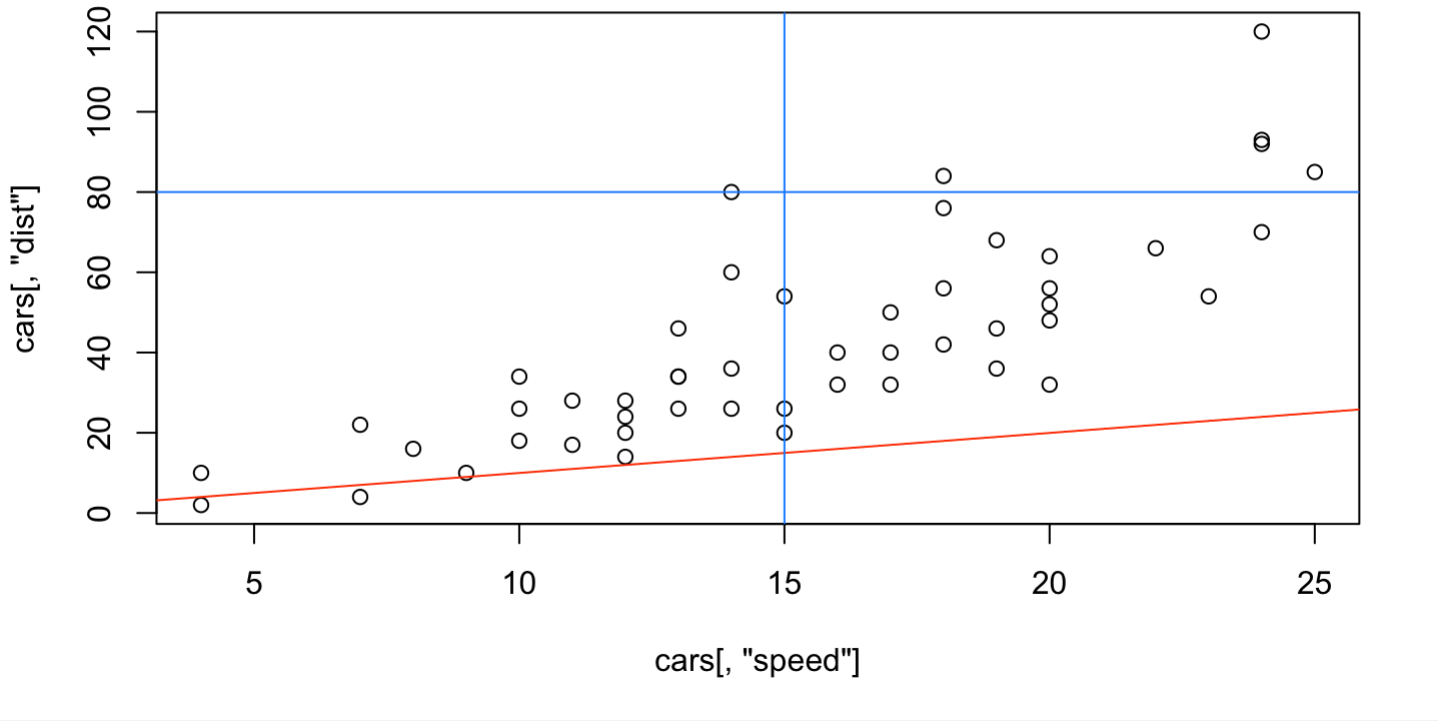
The lines function

```
plot(cars[,"speed"], cars[,"dist"])
lines(x=cars[,"speed"], y=cars[,"dist"],
  col="#dc42f4") # Add lines to a previous plot
```



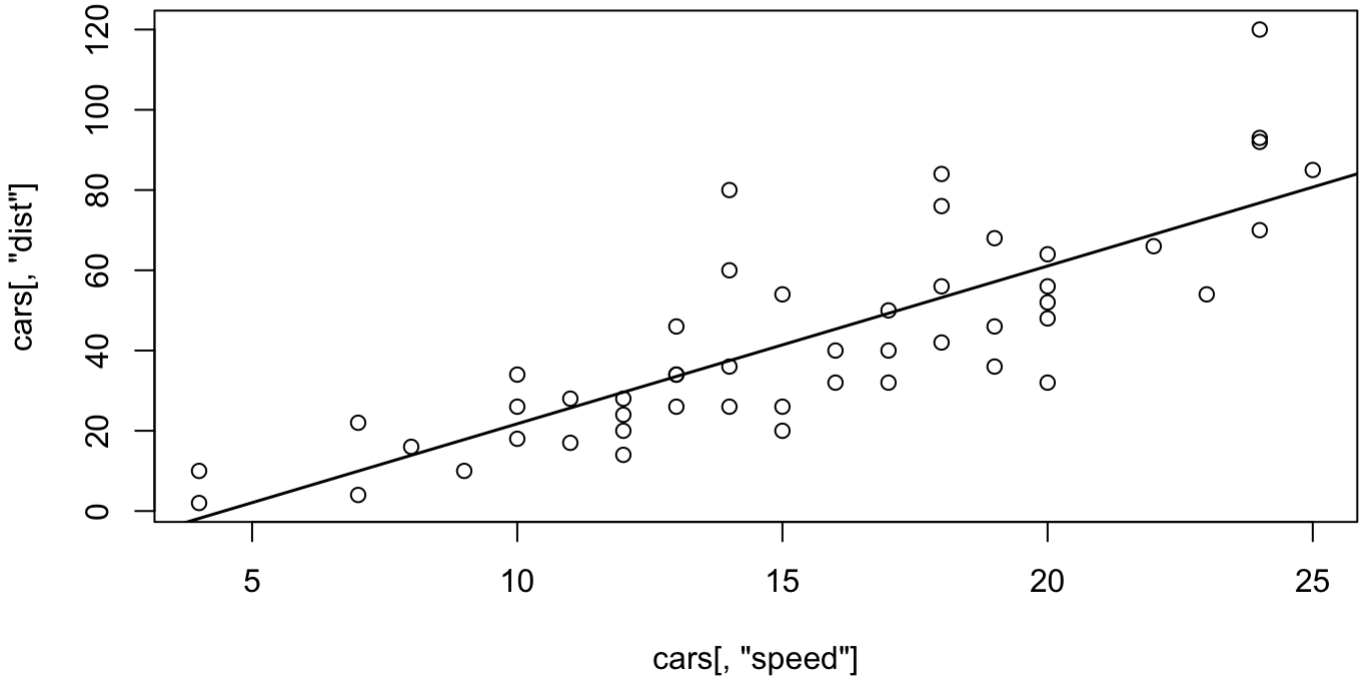
The abline function

```
plot(cars[,"speed"], cars[,"dist"])
abline(a=0, b=1, col="orangered") # Intercept and slope (here x=y)
abline(v=15, col="dodgerblue") # Vertical line
abline(h=80, col="dodgerblue") # Horizontal line
```



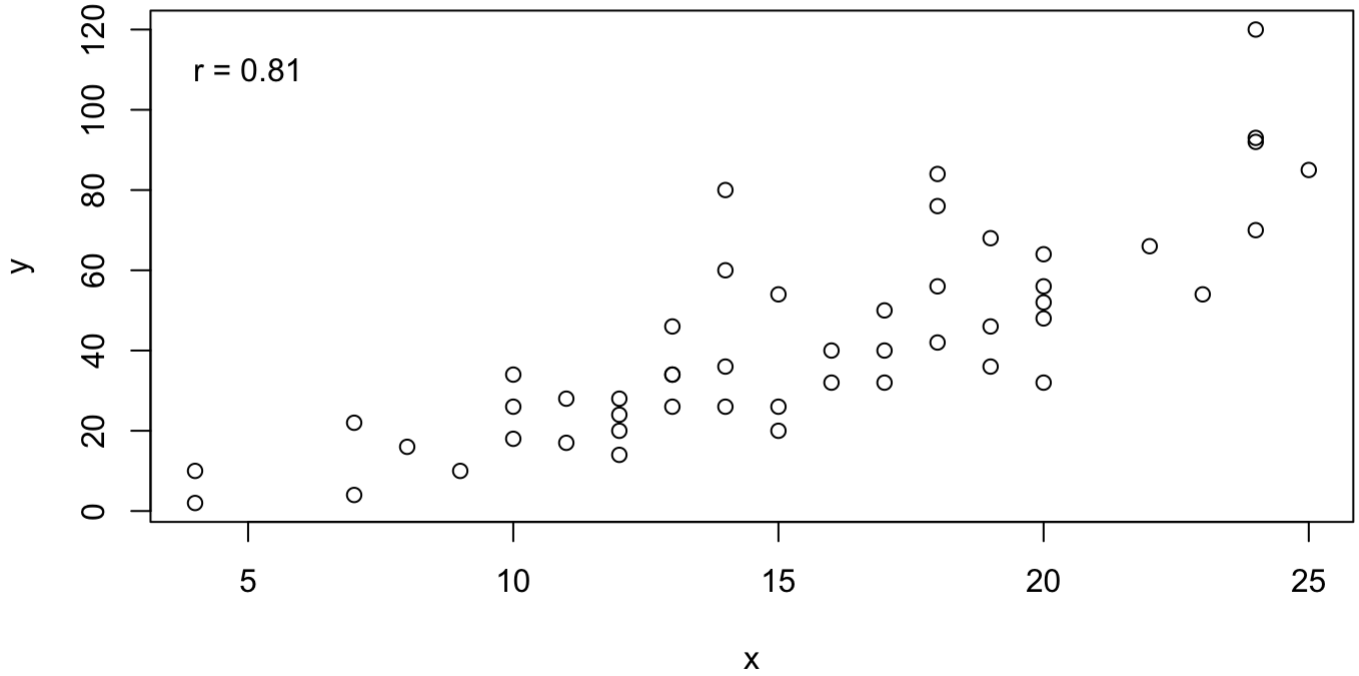
Adding a linear fit

```
plot(cars[,"speed"], cars[,"dist"])
lfit <- lm(dist~speed, data=cars) # Linear fit of dist (y) on speed (x)
abline(lfit, col=1, lwd=1.5)
```



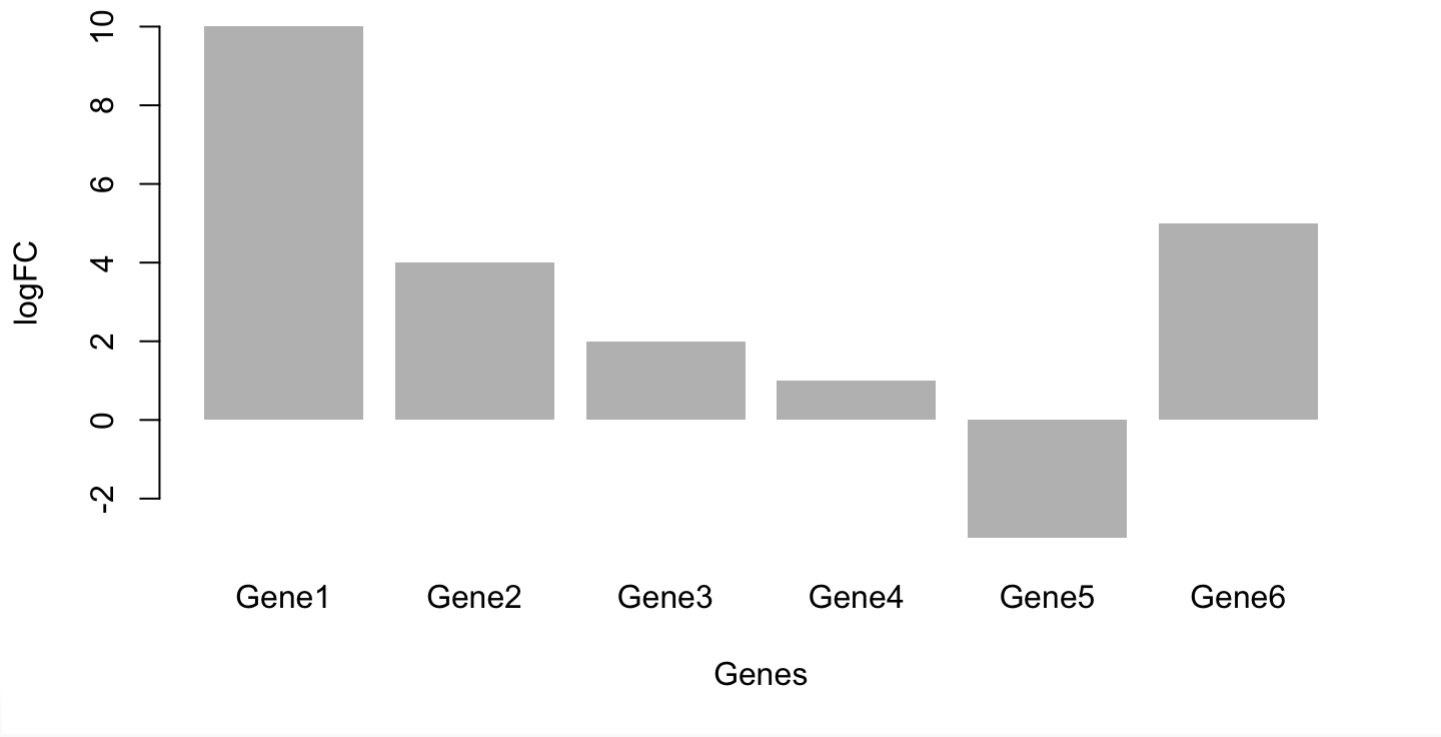
The text function: adding text

```
x <- cars[,"speed"]; y <- cars[,"dist"]; plot(x,y)
r <- round(cor(x,y), 2)
text(x=5, y=110, labels=paste("r =", r))
```



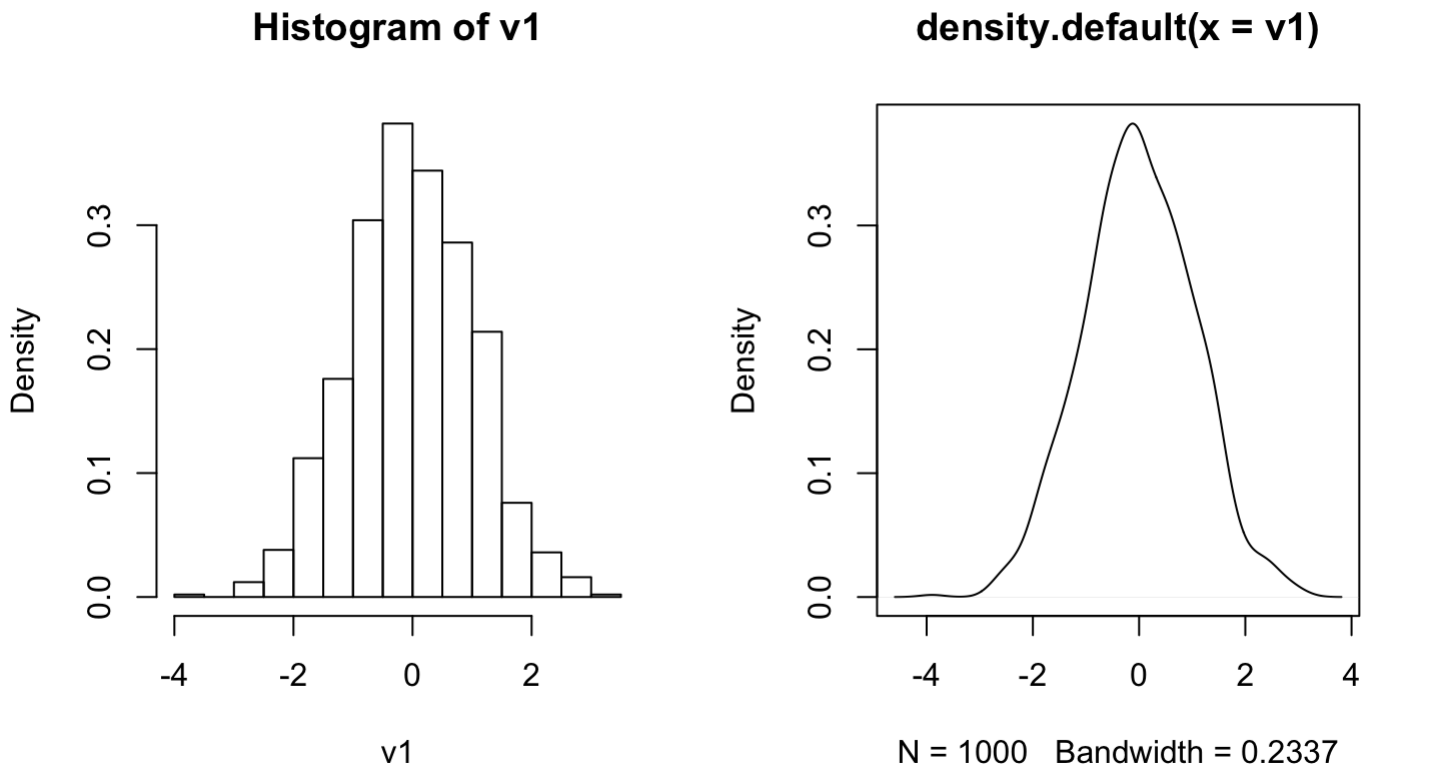
The barplot function

```
x <- c(10, 4, 2, 1, -3, 5)
names(x) <- c("Gene1", "Gene2", "Gene3", "Gene4", "Gene5", "Gene6")
barplot(x, xlab="Genes", ylab="logFC", border=NA)
```



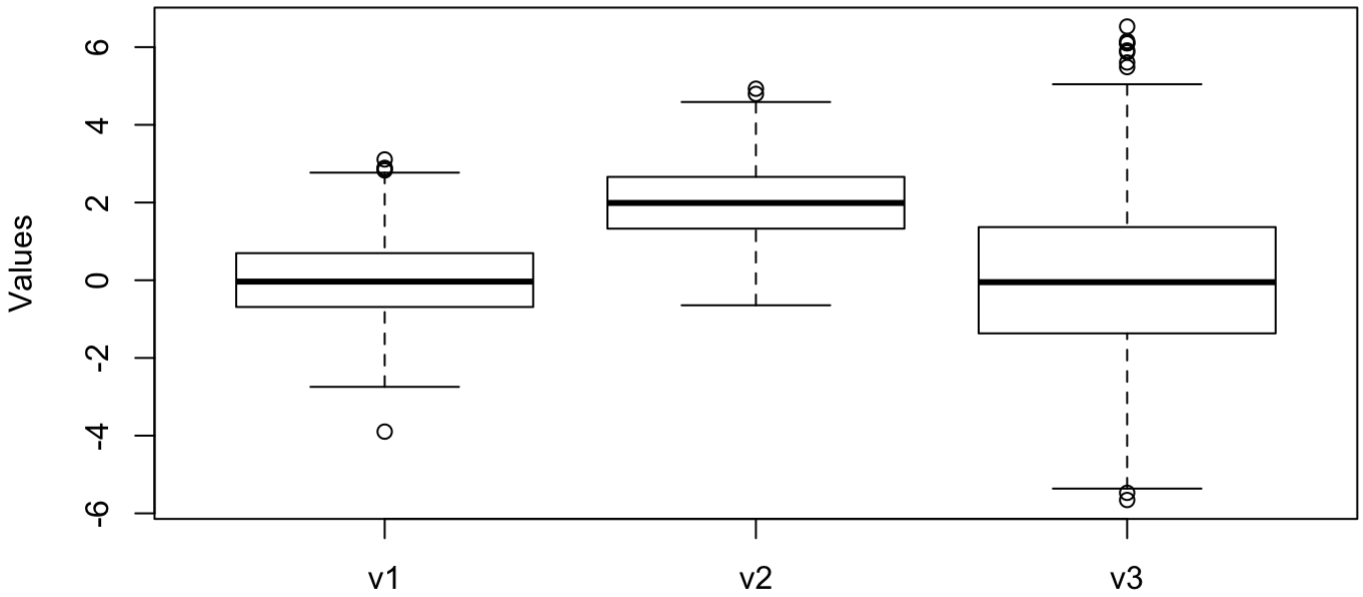
The hist and density functions

```
par(mfrow=c(1,2)) # Organize subplots (1x2)
v1 <- rnorm(1000, mean=0, sd=1)
hist(v1, freq=FALSE) # TRUE for frequencies, FALSE for densities
plot(density(v1)) # density itself does not plot!
```

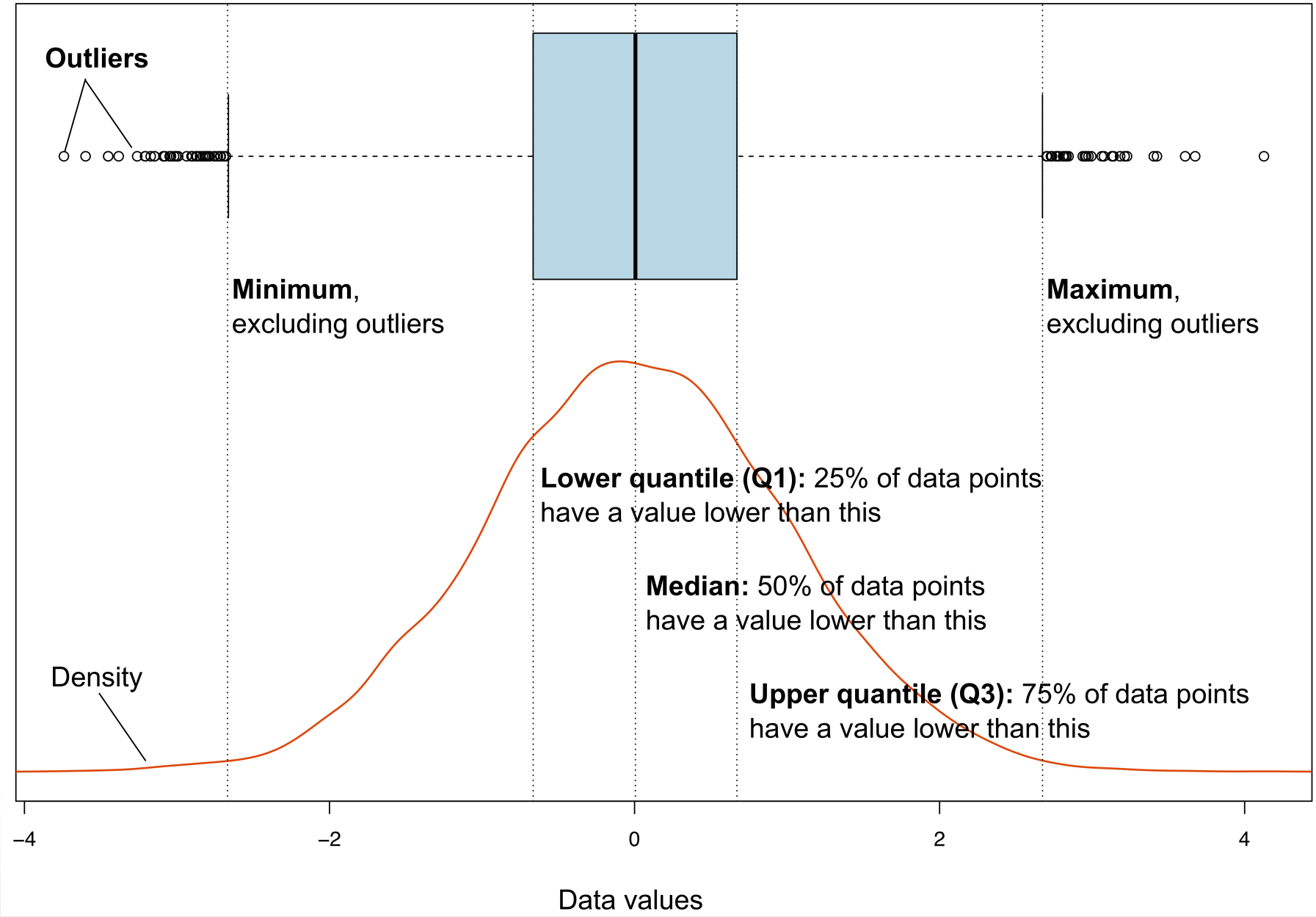


The boxplot function

```
v2 <- rnorm(1000, mean=2, sd=1)
v3 <- rnorm(1000, mean=0, sd=2)
boxplot(list(v1=v1, v2=v2, v3=v3), ylab="Values")
```



Boxplots


Outliers
Minimum, excluding outliers
Maximum, excluding outliers
Lower quantile (Q1): 25% of data points have a value lower than this
Median: 50% of data points have a value lower than this
Upper quantile (Q3): 75% of data points have a value lower than this
Density
−4
−2
0
2
4
Data values

Save plots in a file

Functions like pdf and png can be used to save R plots into images and take as main input the figure name and path (see help for additional parameters)

```
pdf("Figures/boxplot.pdf")
boxplot(list(v1=v1, v2=v2, v3=v3), ylab="Values")
dev.off()
```

```
png("Figures/boxplot.png")
boxplot(list(v1=v1, v2=v2, v3=v3), ylab="Values")
dev.off()
```

dev.off() must be used to close the file after plotting

Exercises

Ex. 3.10 (part 1)

We can use R to download the preprocessed data from a study entitled “High-resolution transcriptome of human macrophages” (GEO accession: GSE36952) where classical (M1) and alternative (M2) macrophages were subjected to RNA-seq.

Download and unzip the data from this link and save it into the “Data” folder.

This text file contains the normalized RNA-seq data (as RPKM) for three replicates of M1 and M2 macrophages

Continues…

Ex. 3.10 (part 2)

After loading the data into R, we can remove duplicate gene names and set them as row names using the following code:

```
ind <- which(!duplicated(RPKM[,1]))
RPKM <- RPKM[ind,]
rownames(RPKM) <- RPKM[,1]
RPKM <- RPKM[,-1]
```

Continues…

Ex. 3.10 (part 3)

We can visualize a special type of scatterplot used for RNA-seq data called MA-plot. MA-plots are used to show the difference in gene expression levels between pairs of samples or replicates, plotted against their average values. Expression data are considered on log-scale.

The following code generate an MA-plot to compare the two replicates called "M1_1" and "M1_2"

```
A1 <- ( log2(RPKM[,"M1_1"]+1) + log2(RPKM[,"M1_2"]+1) )/2 # average
M1 <- log2(RPKM[,"M1_1"]+1) - log2(RPKM[,"M1_2"]+1)       # difference
plot(A1, M1)
```

Continues…

Ex. 3.10 (part 4)

Generate two MA-plots to compare:

1. The two replicates “M1_1” and “M1_2” (both from classical macrophages)
2. The two types of macrophages “M1_1” (classical) and “M2_1” (alternative)

Save both figures in a png file, one above the other using par(mfrow=c(…))

Continues…

Ex. 3.10 (part 5)

In the MA-plots

- Keep the same **ylim** for both (-15, 15) to make the plots comparable
- Add three coloured, horizontal lines corresponding to y=0 (no differences), 4, and -4
- Color differently the genes surpassing the -4 and 4 lines (below or above, respectively)
- Add titles and axis labels, and experiment with colors and cex parameters

How many genes surpass the -4 and 4 lines in the two plots?

Are the differences bigger for the M1 replicates or between M1 and M2 macrophages?

Useful resources

CRAN: https://cran.r-project.org/

Bioconductor: https://www.bioconductor.org/

GitHub: https://github.com/

Ziemann M, Eren Y, El-Osta. Gene name errors are widespread in the scientific literature. Genome Biol 2016;17(1):177.

Color picker: https://www.google.com/search?q=color+picker

Plots: https://www.statmethods.net/graphs/index.html

Solutions

Ex. 3.1: ../Data/

Ex. 3.2: ../../

Ex. 3.4: Increased expression after treatment (pre=1.12; on=7.32)

Ex. 3.6: nUP=92; nDN=331

Ex. 3.7: 3) 2

Ex. 3.8: 2) 6

Ex. 3.9: 3) Normoglycemia

Ex. 3.10: Replicates: 7; M2 vs. M1: 101