Un'intRoduzione

Ottavia M. Epifania, Ph.D

Lezione di Dottorato @Università Cattolica del Sacro Cuore (MI)

8-9 giugno 2023

Table of contents

1. PeRché?
2. Come è fatto
3. Con cosa lavoriamo
4. Le basi
5. L'ambiente e l'ordine
6. Working directories
7. Structures in R
8. The king of data structure: data frames
9. Data input and output
10. Programming
11. Graphics
12. R for statistical computing
13. Classical hypothesis testing in R
14. Generalized Linear Models (GLMs)

Materiale del corso

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Le cose belle

- Open-source: `R` è gratuito, `RStudio` (la IDE di `R`) no, ma la versione free è più che sufficiente
- Permette la replicabilità dei risultati $\rightarrow$ è gratis
- *R community is the beast feature of R*
- Aiuto online e gratuito

Le cose un po' meno belle

- Difficile da imparare all'inizio
- Non è intuitivo (all'inizio) $\rightarrow$ se non si ha già una vaga idea di dove partire non si riesce a fare nulla

Le cose cose belle nelle cose brutte

- Più dimestichezza nell'analisi dei dati, più conoscenza del dato, modelli più complessi
- Permette di addentrarsi sempre di più nei linguaggi di programmazione
- Imparare un altro linguaggio di programmazione dopo aver imparato `R` è (quasi) una passeggiata

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

ToL - master - RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function | Addins

tolTempiModelli.R | tolTempi.qmd | dataPrep.R

Source on Save | Run | Source

```
1  # DATA PREPARATION TOL
2  # OTTAVIA, MAGGIO 2023
3  #
4  rm(list = ls())
5  library(lubridate)
6  library(dplyr)
7
8  setwd("H:/.shortcut-targets-by-id/174OKGGGXve6z7MtNnOOcT3W23DXIim
9
10 name.data.43 = paste0("AdapTol_43/", # do not change
11                      "tol43_2023_05_08.csv") # change according t
12 name.data.52 = paste0("AdapTol_52/", # do not change
13                      "tol52_2023_05_08.csv") # change according t
14 name.data.45 = paste0("AdapTol_45/", # do not change
15                      "tol45_2023_05_08.csv") # change according t
16 name.env = ls()
17 name.env = name.env[grepl("name.data", name.env)]
18
19 for (i in 1:length(name.env)) {
20   assign(gsub("name.", "", name.env[i]),
21          read.csv(get(name.env[i]), header = T, sep = ","))
22 }
23
```

198:4 | total_time | R Script

Console | Terminal | Background Jobs

R 4.2.3 · H:/.shortcut-targets-by-id/174OKGGGXve6z7MtNnOOcT3W23DXlimiV/PsycAssist/

```
> ggplot(all.data, aes(x = n_moves, y = as.numeric(preplanning))) + geom_
point()
Warning message:
Removed 8 rows containing missing values (geom_point).
>
```

Environment | History | Connections | Tutorial

Import Dataset | 242 MiB

R | Global Environment

total_time43	378 obs. of 29 variables
total_time45	166 obs. of 44 variables
total_time52	412 obs. of 36 variables
wide.acc	412 obs. of 28 variables
Values	
alleged.trials	num [1:3] 20 35 27
i	12L
name_desc	chr [1:3] "sbj_char43" "sbj_char45" "sbj_char52"
name.data.43	"AdapTol_43/tol43_2023_05_08.csv"
name.data.45	"AdapTol_45/tol45_2023_05_08.csv"
name.data.52	"AdapTol_52/tol52_2023_05_08.csv"
name.env	chr [1:3] "data.43" "data.45" "data.52"
names.print	chr [1:12] "accuracy43" "accuracy45" "accuracy52" "execution43" "executio

Files | Plots | Packages | Help | Git | Viewer | Presentation

Zoom | Export

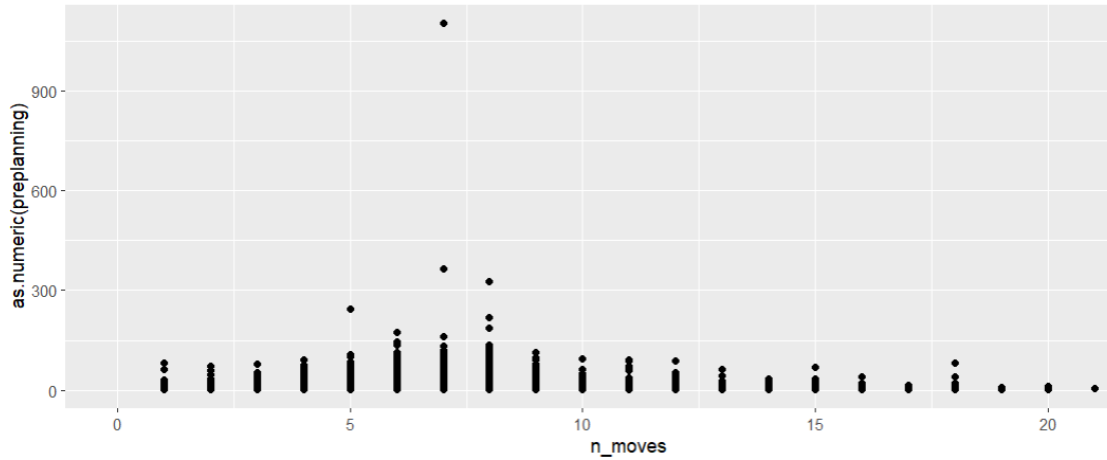


Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

La gestione delle working directory è una delle cose meno intuitive di R

Rischiate di perdere dei pezzi e di non avere tutto ordinato nelle cartelle

Per risolvere questo problema $\rightarrow$ si possono usare gli R project (progetti R)

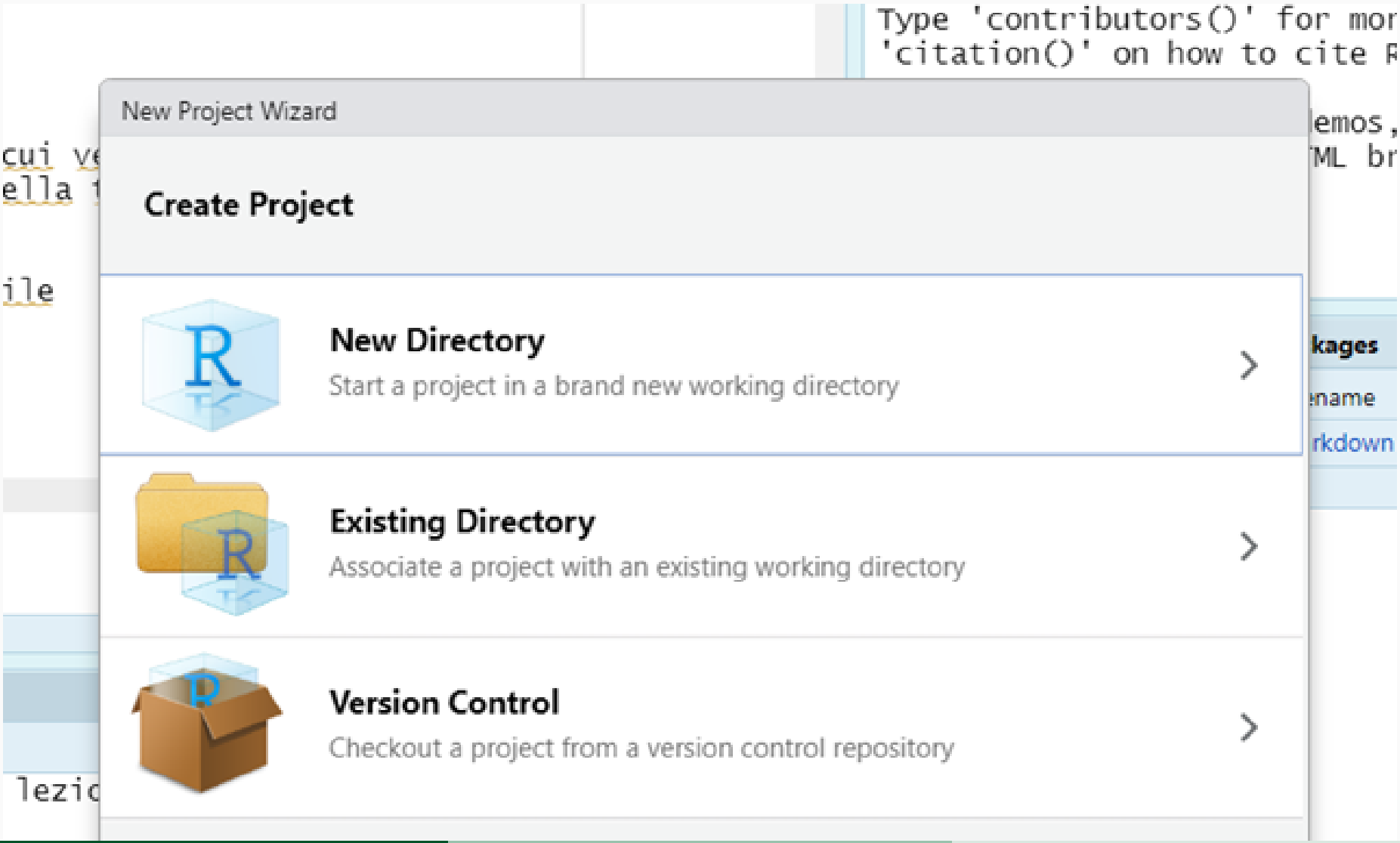
Un progetto R crea una sua directory, tutti i file che vengono salvati al suo interno sono sempre accessibile senza bisogno di settare *a mano* tutte le volte la directory

Sono molto comodi perché:

1. Permettono di avere più istanze R aperte contemporaneamente $\rightarrow$ è possibile lavorare su più progetti contemporaneamente
2. Tenendo tutti i file ordinati vi permettono di poter risalire a cosa avete fatto mesi prima (utile quando vi arrivano le revisioni di un paper)

Creare un progetto R

File → New project:



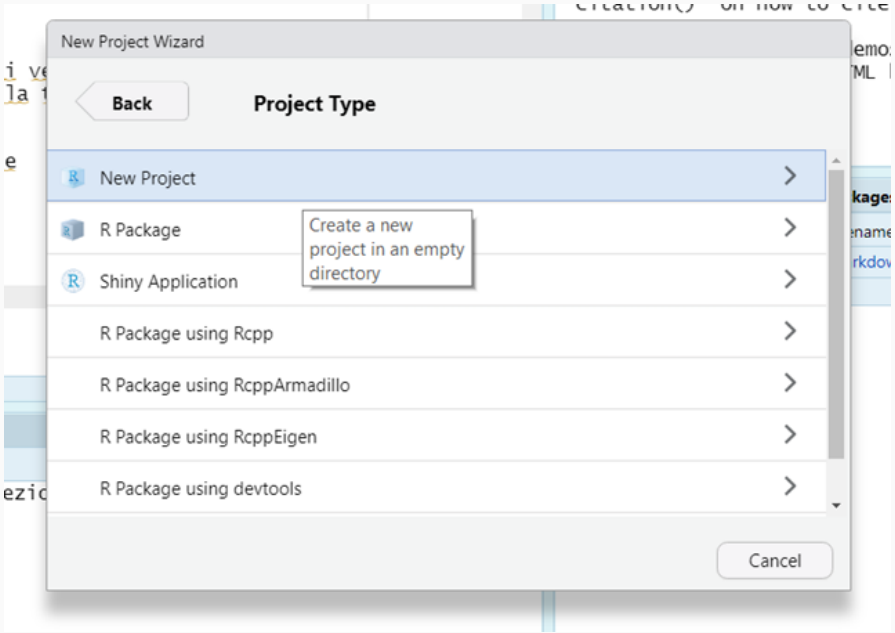

New Project Wizard
Back
Project Type
New Project
R Package
Shiny Application
R Package using Rcpp
R Package using RcppArmadillo
R Package using RcppEigen
R Package using devtools
Create a new project in an empty directory
Cancel

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Simboli, variabili, funzioni

Simboli variabili funzioni

Simboli

Operatori:

```
3 + 2   # più
3 - 2   # meno
3 * 2   # per
3 / 2   # diviso
5 > 2   # maggiore di
2 < 5   # minore di
5 == 3 + 2 # uguale
5 != 3  # diverso
```

Parentesi e altri simboli

() [] {} "" : ; ,

Operatori e parentesi si possono combinare insieme per risolvere equazioni:

```
(15 + 22)/(13 * 4)
```

```
[1] 0.7115385
```

Variabili

La varaibile (in senso statistico) viene registrata in una variabile (in senso informatico)

Variabile (Informatica)

è un oggetto che contiene informazione
Si distinguono in base al tipo di informazione che contengono, che in R può essere:

- ‘int‘: numeriche (discrete, 1, 2, 3 ....)
- ‘num‘: numeriche (continue, 0.01, 0.02, 0.03, ...)
- ‘logi‘: logiche (TRUE, FALSE)
- ‘chr‘: caratteri ("a", "B", "c", ...)
- ‘factor‘: fattori distinti da diversi livelli

Nomi delle variabili

Alle variabili si può dare il nome che si vuole → non può iniziare con un numero

`modello_1` → Sì , `1Modello` → No

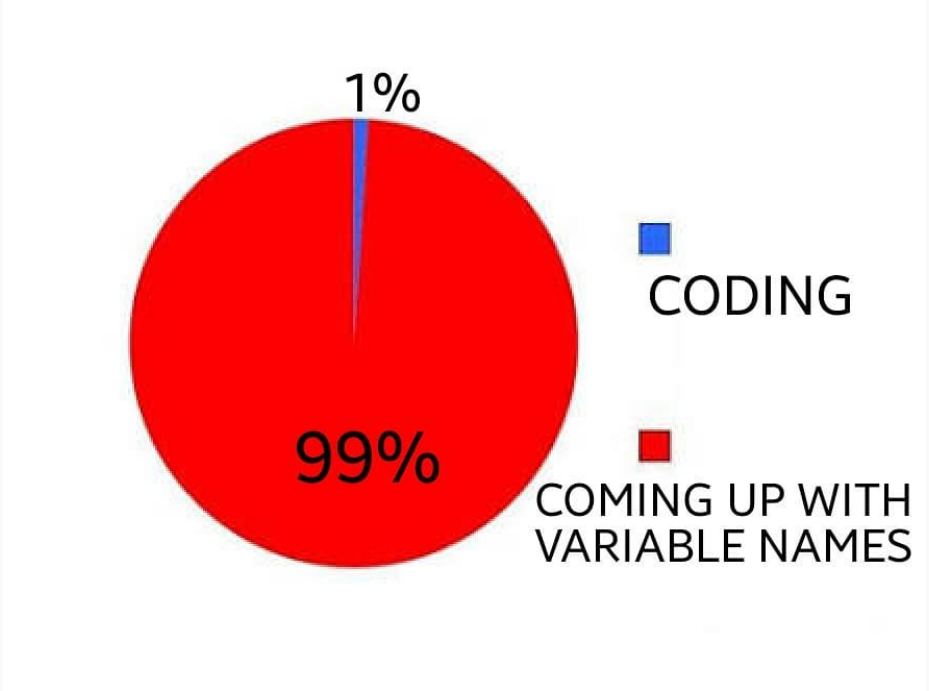
Warning!

Attenzione perché R è case sensitive:

`A` ≠ `a`

Altro vincolo: Non si possono usare nomi che possono essere confusi con funzioni/oggetti interni di `R`:

'FALSE' è un oggetto logico di R, non possiamo usarlo come nome di una nostra variabile.



Molto bene

`media_maschi`, `modello1`, `statistiche_descrittive`

Molto male

`x3`, `x2`, `x1`

Assign

Le variabili vengono create "assegnando" loro i risultati delle operazioni

Esistono fondamentalmente due comandi di assegnazione

1. `x = exp(2^2)` =(non piace molto ai puristi di R ma è ugaule a Python
2. `X <- log(2^2)` =) Fate contenti i puristi di R

La logica è che l'oggetto a destro (l'operazione e quindi il risultato che se ne ottiene) viene assegnato (= oppure <-) viene assegnato alla variabile (o oggetto) a destra

Attenzione! Siccome `R` è case sensitive: `x` e `X` sono due oggetti differenti!

Le funzioni

```
my.function(arg1, arg2, arg3 = default)
```

Sono definite dal loro nome (che solitamente riflette cosa fanno – la loro funzione), un paio di parentesi tonde

AL loro interno si possono settare diversi argomenti:

- Alcuni hanno dei valori di default $\rightarrow$ vengono applicati a meno che l'utente non li cambi
- Altri argomenti sono obbligatori e vanno settati dall'utente

Un esempio

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

- `mean`: è il nome della funzione per calcolare la media
- `x`: è il primo argomento della funzione, indica che va passato alla funzione un oggetto `x` (una variabile) su cui calcolare la media
- `trim`: secondo argomento della funzione con un default (`trim = 0`), specifica se applicare il trimming sui dati prima del calcolo della media
- `na.rm`: terzo argomento della funzione con un default (`na.rm = FALSE`), determina il trattamento dei dati mancanti
- `...` indica che si possono passare altri argomenti alla funzione

concatenate

è la funzione più usata su R

c()

Serve per concatenare diversi oggetti (variabili) per combinarli in un unico oggetto →

```
x = c(1, 2, 3) # crea un vettore con tre numeri
x
```

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

```
X = 1:3 # crea lo  stesso identico vettore
X
```

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

```
x == X
```

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

Aiuto

'R' community is the best feature of 'R'

Copia & Incolla l'errore su google

Se non si sa fare qualcosa "how to **[something]** in r"

Chiedere a `R`: Nella consolle si può scrivere `?nome.funzione()` per accedere alla documentazione sulla funzione:

```
?mean()
```

Apre la documentazione sulla funzione `mean()`

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Take out the trash

L'ambiente di R dovrebbe essere sempre ordinato

Gli oggetti che non servono più andrebbero eliminati

MA si può anche togliere tutto insieme

```
ls() # lista gli oggetti che sono nell'environment
rm(A) # rimuove l'oggetto A all'environment
rm(list=ls()) # rimuove tutto dall'environment
```

Save the environment

It might be useful to save all the computations you have done:

```
save.image("my-computations.RData")
```

Then you can upload the environment back:

```
load("my-Computations.RData")
```

When to save the environment

The computations are slow and you need them to be always and easily accessible

The best practice is to save the script and document it in an `RMarkdown` file $\rightarrow$ Reproducibilty!

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

If you choose not to use the R projects (what a bad, bad, bad idea), you need to know your directories:

```
getwd() # the working directory in which you are right now

dir() # list of what's inside the current working directory
```

Change your working directory:

```
setwd("C:/Users/huawei/OneDrive/Documenti/GitHub/RcouRse")
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Functions and arguments (pt. I)

Almost everything in R is done with functions, consisting of:

- a name: `mean`
- a pair of brackets: `()`
- some arguments: `na.rm = TRUE`

```
mean(1:5, trim = 0, na.rm = TRUE)
```

```
[1] 3
```

Arguments may be set to default values; what they are is documented in `?mean()`

Functions and arguments (pt. II)

Arguments can be passed

- without name (in the defined order)
- with name (in arbitrary order) $\rightarrow$ *keyword matching*

```
mean(x, trim = 0.3, na.rm = TRUE)
```

No arguments? No problems, just brackets:

```
ls(), dir(), getwd()
```

Want to see the code of a function? Just type its name in the console without brackets:

```
chisq.test
```

Vectors

Vectors are created by **c**ombining together different objects

Vectors are created by using the `c()` function.

All elements inside the `c()` function **must** be separated by a comma

Different types of objects $\rightarrow$ types of vectors:

- `int`: numeric integers
- `num`: numbers
- `logi`: logical
- `chr`: characters
- `factor`: factor with different levels

int and num

int: refers to integer: -3, -2, -1, 0, 1, 2, 3

```
months = c(5, 6, 8, 10, 12, 16)
```

```
[1]  5  6  8 10 12 16
```

num: refers to all numbers from $-\infty$ to ∞: 1.0840991, 0.8431089, 0.494389, -0.7730161, 2.9038161, 0.9088839

```
weight = seq(3, 11, by = 1.5)
```

```
[1]  3.0  4.5  6.0  7.5  9.0 10.5
```

logi

Logical values can be TRUE (T) or FALSE (F)

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

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

logical vectors are often obtained from a comparison:

```
months > 12
```

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

chr and factor

chr: characters: a, b, c, D, E, F

```
v_chr = c(letters[1:3], LETTERS[4:6])
```

```
[1] "a" "b" "c" "D" "E" "F"
```

factor: use numbers or characters to identify the variable levels

```
ses = factor(c(rep(c("low", "medium", "high"), each = 2)))
```

```
[1] low    low    medium medium high   high
Levels: high low medium
```

Change order of the levels:

```
ses1 = factor(ses, levels = c("medium", "high", "low"))
```

```
[1] low    low    medium medium high   high
Levels: medium high low
```

Create vectors

Concatenate elements with `c()`: `vec = c(1, 2, 3, 4, 5)`

Sequences:

```
-5:5 # vector of 11 numbers from -5 to 5
```

```
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
seq(-2.5, 2.5, by = 0.5) # sequence in steps of 0.5
```

```
 [1] -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5
```

Repeating elements:

```
rep(1:3, 4)
```

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

Create vectors II

```
rep(c("condA", "condB"), each = 3)
```

```
[1] "condA" "condA" "condA" "condB" "condB" "condB"
```

```
rep(c("on", "off"), c(3, 2))
```

```
[1] "on"  "on"  "on"  "off" "off"
```

```
paste0("item", 1:4)
```

```
[1] "item1" "item2" "item3" "item4"
```

Don't mix them up unless you truly want to

`int` + `num` → `num`

`int/num` + `logi` → `int/num`

`int/num` + `factor` → `int/num`

`int/num` + `chr` → `chr`

`chr` + `logi` → `chr`

Vectors and operations

Vectors can be summed/subtracted/divided and multiplied with one another

```
a = c(1:8)
a
```

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

```
b = c(4:1)
b
```

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

```
a - b
```

```
[1] -3 -1  1  3  1  3  5  7
```

If the vectors do not have the same length, you get a warning

Vectors and operations PT. II

The function is applied to each value of the vector:

```
sqrt(a)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.6
```

The same operation can be applied to each element of the vector:

```
(a - mean(a))^2 # squared deviation
```

```
[1] 12.25  6.25  2.25  0.25  0.25  2.25  6.25 12.25
```

Matrices and arrays

Create a 3×4 matrix:

```
A = matrix(1:12, nrow=3, ncol = 4, byrow = TRUE)
```

Label and transpose:

```
rownames(A) = c(paste("a", 1:3))  # colnames()
t(A)  # transpose matrix
```

```
     a 1 a 2 a 3
[1,]   1   5   9
[2,]   2   6  10
[3,]   3   7  11
[4,]   4   8  12
```

Matrices and arrays

Matrix can be created by concatenating columns or rows:

```
cbind(a1 = 1:4, a2 = 5:8, a3 = 9:12) # column bind
rbind(a1 = 1:4, a2 = 5.8, a3 = 9:12) # row bind
```

Matrices and arrays

```
array(data, c(nrow, ncol, ntab))
```

```
my_array = array(1:30, c(2, 5, 3)) # 2 x 5 x 3 array
```

```
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20

, , 3

....
```

Work with vectors, matrices, arrays

Index elements in vectors: `vector_name[position]`

```
weight[2]          # second element in vector weight
weight[6] = 15.2 # replace sixth element of weight
weight[seq(1, 6, by = 2)] # elements 1, 3, 5
weight[2:6]        # elements 2 to 6
weight[-2]       # without element 2
```

Logic applies as well:

```
weight[weight > 7] # values greater than 7
weight[weight >= 4.5 & weight < 8] # values between 4.5
                                   # and 8
```

String processing

```
substr(x, start, stop)          # extract substring
grep(pattern, x)                # match pattern (poistion)
grep(pattern, x)                # match pattern (TRUE/FALSE)
gsub(pattern, replacement, x) # replace pattern
```

pattern = regular expression (?regex):

```
foo       # match pattern foo
.*        # match arbitrary character zero or more times
[a-z0-9] # match alphanumeric character
```

Example

Match string that starts with a or b and replace it by its starting letter.

```
gsub("(^[ab]).*", "\\1", c("aaa", "bbc", "cba"))
```

```
[1] "a"   "b"   "cba"
```

Work with vectors, matrices, arrays II

Index elements in matrices: `matrix_name[row, column]`

```
A[2, 3] # cell in row 2 column 3

A[2, ] # second row

A[, 3] # third column
```

Work with vectors, matrices, arrays III

Index elements in arrays `array_name[row, col, tab]`

```
my_array[2, 1, 3] # cell in 2nd row 1st col of 3rd tab
```

```
my_array[, , 3] # 3rd tab
```

```
my_array[1, ,2] # 1st row in tab 2
```

Lists

Can store different objects (e.g., vectors, data frames, other lists):

```
my_list = list(w = weight, m = months, s = ses1, a = A)
```

The components of the list can be indexed with `$` or `[[]]` and the name (or position) of the component:

Extract `months`:

```
my_list[["m"]] # my_list$m
```

```
[1]  5  6  8 10 12 16
```

Extract `weight`:

```
my_list[[1]] # my_list$weight or my_list[["w"]]
```

```
[1]  3.0  4.5  6.0  7.5  9.0 10.5
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Data frames are lists that consist of vectors and factors of equal length. The rows in a data frame refer to one unit:

```
id = paste0("sbj", 1:6)
babies = data.frame(id, months, weight)
```

```
babies
```

```
    id months weight
1 sbj1      5    3.0
2 sbj2      6    4.5
3 sbj3      8    6.0
4 sbj4     10    7.5
5 sbj5     12    9.0
6 sbj6     16   10.5
```

Working with data frames

Index elements in a data frame:

```
babies$months # column months of babies

babies$months[2] # second element of column months

babies[, "id"] # column id

babies[2, ] # second row of babies (obs on baby 2)
```

Logic applies:

```
babies[babies$weight > 7, ] # all obs above 7 kg
babies[babies$id %in% c("sbj1", "sbj6"), ] # obs of sbj1
                                           # and sbj7
```

Working with data frames II

```
dim(babies) # show the dimensions of the data frame
```

```
[1] 6 3
```

```
names(babies) # variable names (= colnames(babies))
```

```
[1] "id"     "months" "weight"
```

```
View(babies) # open data viewer

plot(babies) # pariwise plot
```

You can use these commands also on other R objects

Working with data frames III

```
str(babies) # show details on babies
```

```
'data.frame':   6 obs. of  3 variables:
 $ id    : chr  "sbj1" "sbj2" "sbj3" "sbj4" ...
 $ months: num  5 6 8 10 12 16
 $ weight: num  3 4.5 6 7.5 9 10.5
```

```
summary(babies) # descriptive statistics
```

```
      id                  months          weight
 Length:6           Min.   : 5.0   Min.   : 3.000
 Class :character   1st Qu.: 6.5   1st Qu.: 4.875
 Mode  :character   Median : 9.0   Median : 6.750
                    Mean   : 9.5   Mean   : 6.750
                    3rd Qu.:11.5   3rd Qu.: 8.625
                    Max.   :16.0   Max.   :10.500
```

Sorting

`order()`:

```
babies[order(babies$weight), ] # sort by increasing weight
```

```
    id months weight
1 sbj1      5    3.0
2 sbj2      6    4.5
3 sbj3      8    6.0
4 sbj4     10    7.5
5 sbj5     12    9.0
6 sbj6     16   10.5
```

```
babies[order(babies$weight,    # sort by decreasing weight
             decreasing = T), ]
```

Multiple arguments in `order`:

```
babies[order(babies$weight, babies$months, decreasing = TRUE), ]
```

Aggregating

Aggregate a response variable according to grouping variable(s) (e.g., averaging per experimental conditions):

```
# Single response variable, single grouping variable
aggregate(y ~ x, data = data, FUN, ...)

# Multiple response variables, multiple grouping variables
aggregate(cbind(y1, y2) ~ x1 + x2, data = data, FUN, ...)
```

Aggregating: Example

```
ToothGrowth # Vitamin C and tooth growth (Guinea Pigs)
```

```
    len supp dose
1   4.2   VC  0.5
2  11.5   VC  0.5
3   7.3   VC  0.5
....
```

```
aggregate(len ~ supp + dose, data = ToothGrowth, mean)
```

```
  supp dose   len
1   OJ  0.5 13.23
2   VC  0.5  7.98
3   OJ  1.0 22.70
4   VC  1.0 16.77
5   OJ  2.0 26.06
6   VC  2.0 26.14
```

Reshaping: Long to wide

Data can be organized in wide format (i.e., one line for each statistical unit) or in long format (i.e., one line for each observation).

```
Indometh # Long format
```

```
   Subject time conc
1        1 0.25 1.50
2        1 0.50 0.94
3        1 0.75 0.78
4        1 1.00 0.48
5        1 1.25 0.37
6        1 2.00 0.19
....
```

Long to wide

```
# From long to wide
df.w <- reshape(Indometh, v.names = "conc", timevar = "time",
    idvar = "Subject", direction = "wide")
```

```
   Subject conc.0.25 conc.0.5 conc.0.75 conc.1 conc.1.25 conc.2 con
1        1      1.50     0.94      0.78   0.48      0.37   0.19   0
12       2      2.03     1.63      0.71   0.70      0.64   0.36   0
23       3      2.72     1.49      1.16   0.80      0.80   0.39   0
34       4      1.85     1.39      1.02   0.89      0.59   0.40   0
45       5      2.05     1.04      0.81   0.39      0.30   0.23   0
56       6      2.31     1.44      1.03   0.84      0.64   0.42   0
   conc.5 conc.6 conc.8
....
```

Reshaping: Wide to long

```
# From wide to long
df.l <- reshape(df.w, varying = list(2:12), v.names = "conc",
    idvar = "Subject", direction = "long", times = c(0.25, 0.5,
        0.75, 1, 1.25, 2, 3, 4, 5, 6, 8))
```

```
        Subject time conc
1.0.25        1 0.25 1.50
2.0.25        2 0.25 2.03
3.0.25        3 0.25 2.72
....
```

```
df.l[order(df.l$Subject), ] # reorder by subject
```

```
        Subject time conc
1.0.25        1 0.25 1.50
1.0.5         1 0.50 0.94
1.0.75        1 0.75 0.78
....
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Reading tabular txt files:

`ASCII` text files in tabular or spread sheet form (one line per observation, one column per variable) are read using `read.table()`

```
data = read.table("C:/RcouRse/file.txt", header = TRUE)
```

`data` is a data frame where the original numerical variables are converted in numeric vectors and character variables are converted in factors (not always).

Arguments:

- `header`: variable names in the first line? `TRUE/FALSE`
- `sep`: which separator between the columns (e.g., comma, `\t`)
- `dec`: `1.2` or `1,2`?

Reading other files

```
data = read.csv("C:/RcouRse/file.csv",
                header = TRUE, sep = ";",
                dec = ",")
```

From SPSS (file .sav):

```
install.packages("foreign")
library(foreign)
data = read.spss("data.sav", to.data.frame = TRUE)
```

Combine data frames

If they have the same number of columns/rows

```
all_data = rbind(data, data1, data2) # same columns
all_data = cbind(data, data1, data2) # same rows
```

If they have different rows/columns but they share at least one characteristic (e.g., `ID`):

```
all_data = merge(data1, data2,
                 by = "ID")
```

If there are different IDs in the two datasets $\rightarrow$ added in new rows

`all_data` contains all columns in `data1` and `data2`. The columns of the IDs in `data1` but not in `data2` (and vice versa) will be filled with `NA`s accordingly

Export data

Writing text (or csv) file:

```
write.table(data, # what you want to write
            file = "mydata.txt", # its name + extension
            header = TRUE, # first row with col names?
            sep = "\t",  # column separator
            ....) # other arguments
```

R environment (again):

```
save(dat, file = "exp1_data.rda") # save something specific
save(file = "the_earth.rda")      # save the environment
load("the_earth.rda")             # load it back
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Be ready to make mistakes (a lot of mistakes)

Coding is ~~hard~~ art

Eyes on the prize, but take your time (and the necessary steps) to get there

Remember: You're not alone $\rightarrow$ `stackoverflow` (or Google in general) is your best friend

ifelse()

Conditional execution:

Easy: `ifelse(test, if true, if false)`

```
ifelse(weight > 7, "big boy", "small boy")
```

```
[1] "small boy" "small boy" "small boy" "big boy"   "big boy"
```

Pros

- Super easy to use
- Can embed multiple `ifelse()` cycles

Cons

- It works fine until you have simple tests

if () {} else {}

If you have only one condition:

```
if (test_1) {
  command_1
} else {
  command_2
}
```

if () {} else {}

Multiple conditions:

```
if (test_1) {
  command_1
} else if (test_2) {
  command_2
} else {
  command_3
}
```

`test_1` (and `test_2`, if you have it) **must** evaluate in either `TRUE` or `FALSE`

```
if(!is.na(x)) y <- x^2 else stop("x is missing")
```

Loops

`for()` and `while()`

Repeat a command over and over again:

```
# Don't do this at home
x <- rnorm(10)
y <- numeric(10)   # create an empty container
for(i in seq_along(x)) {
y[i] <- x[i] - mean(x)
}
```

The best solution would have been:

```
y = x - mean(x)
```

Avoiding loops

Don't loop, `apply()`!

`apply()`

```
X <- matrix(rnorm(20),
            nrow = 5, ncol = 4)
apply(X, 2, max) # maximum for each column
```

```
[1] 1.4203744 1.0716663 2.3329026 0.9084482
```

Avoiding loops

Don't loop, apply()!

apply()

```
X <- matrix(rnorm(20),
            nrow = 5, ncol = 4)
apply(X, 2, max) # maximum for each column
```

```
[1] 1.4203744 1.0716663 2.3329026 0.9084482
```

for()

```
y = NULL
for (i in 1:ncol(X)) {
  y[i] = max(X[, i])
}
```

Avoiding loops

Group-wise calculations: `tapply()`

`tapply()` (`t` for table) may be used to do group-wise calculations on vectors. Frequently it is employed to calculate group-wise means.

```
with(ToothGrowth,
     tapply(len, list(supp, dose), mean))
```

```
     0.5     1     2
OJ 13.23 22.70 26.06
VC  7.98 16.77 26.14
```

(You could have done it with `aggregate()`)

Writing functions

Compute Cohen's *d*:

```
dcohen = function(group1, group2) { # Arguments
  mean_1 = mean(group1) ; mean_2 = mean(group2)
  var_1 = var(group1) ; var_2 = var(group2) # body
  d = (mean_1 - mean_2)/sqrt(((var_1 + var_2)/2) )
  return(d) # results
}
```

Use it:

```
dcohen(data$placebo, data$drug)
```

Named arguments

Take this function:

```
fun1 <- function(data, data.frame, graph, limit) { ... }
```

It can be called as:

```
fun1(d, df, TRUE, 20)
fun1(d, df, graph=TRUE, limit=20)
fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

Positional matching and keyword matching (as in built-in functions)

Defaults

Arguments can be given default values → the arguments can be omitted!

```
fun1 <- function(data, data.frame, graph=TRUE,
                 limit=20) { ... }
```

It can be called as

```
ans <- fun1(d, df)
```

which is now equivalent to the three cases above, but:

```
ans <- fun1(d, df, limit=10)
```

which changes one of the defaults.

Methods and classes

The return value of a function may have a specified `class` → determines how it will be treated by other functions.

For example, many classes have tailored `print` methods.

```
methods(print)
```

```
  [1] print.acf*
  [2] print.AES*
  [3] print.all_vars*
  [4] print.anova*
  [5] print.any_vars*
  [6] print.aov*
  [7] print.aovlist*
....
```

Define a print method!

. . . as another function:

```
print.cohen <- function(obj){
  cat("\nMy Cohen's d\n\n")
  cat("Effect size: ", obj$d, "\n")
  invisible(obj) # return the object
}
```

We have to change our dcohen function a bit:

```
dcohen = function(group1, group2) { # Arguments
  ...
  dvalue = list(d = d)
  class(dvalue) = "cohend"
  return(dvalue) # results
}
```

Example

Compute the Cohen's *d* between a test group and a control group:

```
set.seed(082022) # results equal for everyone
data <- data.frame(drug = rnorm(6, 10),
                   placebo = rnorm(6, 2))
my_d = dcohen(data$drug, data$placebo)
print.cohen(my_d)
```

```
My Cohen's d

Effect size:  6.900794
```

Debugging

Use the `traceback()` function:

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }
```

Call `foo()` and...

```
foo() #
[1] 1
Error: object 'a.variable.which.does.not.exist' not found
```

Use `traceback()`:

```
traceback() # find out where the error occurred
2: bar(2)
1: foo()
```

Note: `traceback()` appears as default

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

- Traditional graphics
- Grid graphics & `ggplot2`

For both:

- High level functions $\rightarrow$ actually produce the plot
- Low level functions $\rightarrow$ make it looks better $=)$

Export graphics file

```
postscript()  # vector graphics
pdf()

png()           # bitmap graphics
tiff()
jpeg()
bmp()
```

Remember to run off the graphic device once you've saved the graph:

```
dev.off()
```

(You can do it also manually)

Traditional graphics I

High level functions

```
plot()        # scatter plot, specialized plot methods
boxplot()
hist()        # histogram
qqnorm()      # quantile-quantile plot
barplot()
pie()         # pie chart
pairs()       # scatter plot matrix
persp()       # 3d plot
contour()     # contour plot
coplot()      # conditional plot
interaction.plot()
```

`demo(graphics)` for a guided tour of base graphics!

Traditional graphics II

Low level functions

```
points()        # add points
lines()         # add lines
rect()
polygon()
abline()        # add line with intercept a, slope b
arrows()
text()          # add text in plotting region
mtext()         # add text in margins region
axis()          # customize axes
box()           # box around plot
legend()
```

Plot layout

Each plot is composed of two regions:

- The plotting regions (contains the actual plot)
- The margins region (contain axes and labels)

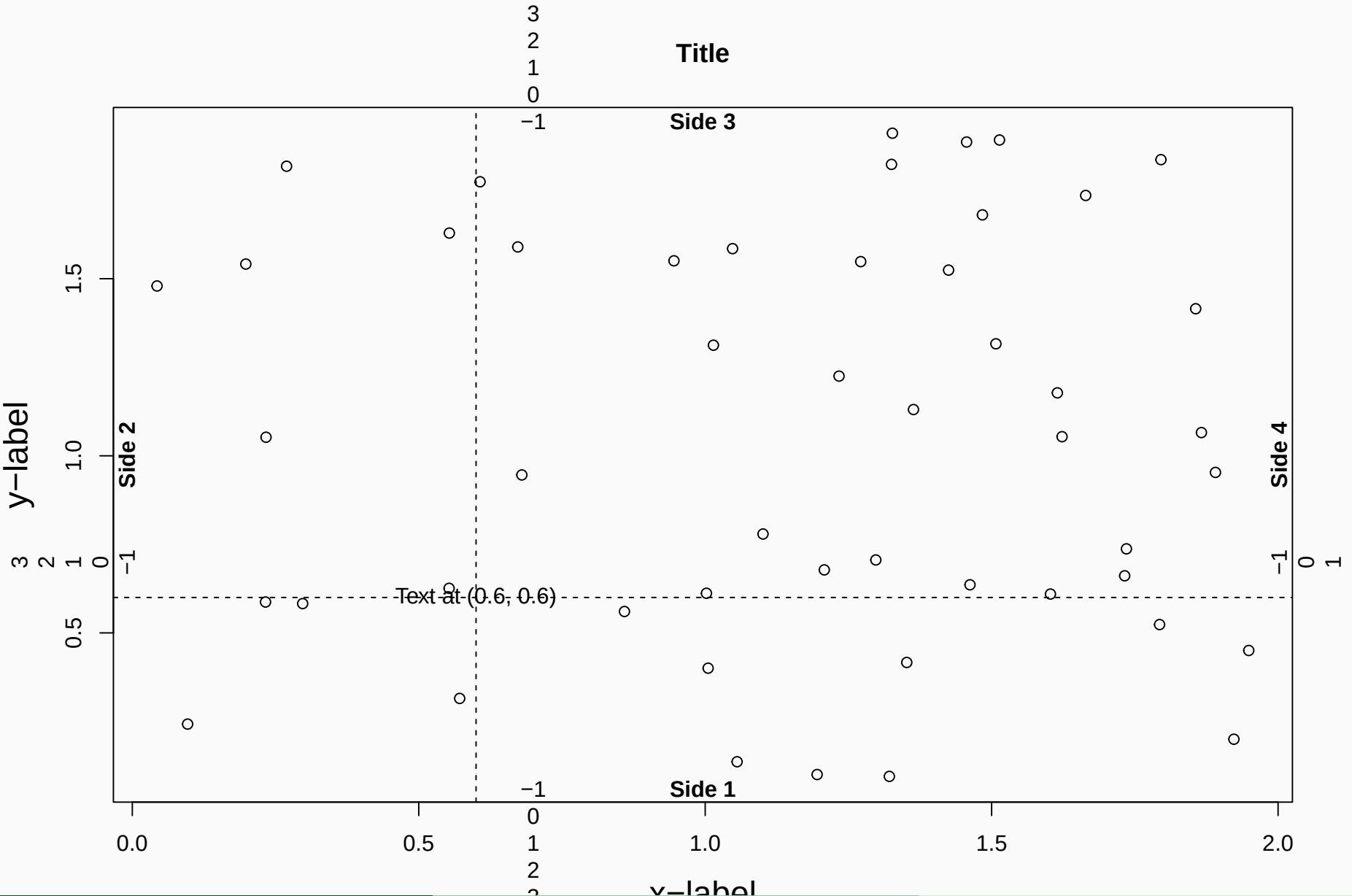
A scatter plot:

```
x <- runif(50, 0, 2) # 50 uniform random numbers
y <- runif(50, 0, 2)
plot(x, y, main="Title",
     sub="Subtitle", xlab="x-label",
     ylab="y-label") # produce plotting window
```

Now add some text:

```
text(0.6, 0.6, "Text at (0.6, 0.6)")
abline(h=.6, v=.6, lty=2) # horizont. and vertic.
                          # lines
```

Margins region



Rome wasn't built in a day and neither your graph

Display the interaction between the two factors of a two-factorial experiment:

```
dat <- read.table(header=TRUE, text="
A B rt
a1 b1 825
a1 b2 792
a1 b3 840
a2 b1 997
a2 b2 902
a2 b3 786
")
```

Force `A` and `B` to be `factor`:

```
dat[,1:2] = lapply(dat[,1:2], as.factor)
```

First: The plot

```
plot(rt ~ as.numeric(B), dat, type="n", axes=FALSE,
     xlim=c(.8, 3.2), ylim=c(750, 1000),
     xlab="Difficulty", ylab="Mean reaction time (ms)")
```

Populate the content

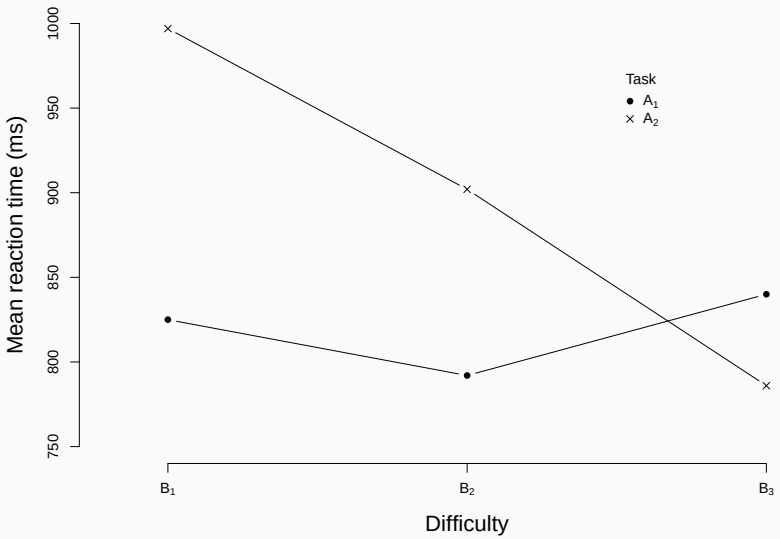
Plot the data points separately for each level of factor `A`.

```
points(rt ~ as.numeric(B), dat[dat$A=="a1",],
       type="b", pch=16)
points(rt ~ as.numeric(B), dat[dat$A=="a2",],
       type="b", pch=4)
```

Add axes and a legend.

```
axis(side=1, at=1:3, expression(B[1], B[2], B[3]))
axis(side=2)
legend(2.5, 975, expression(A[1], A[2]), pch=c(16, 4),
       bty="n", title="Task")
```

Final result



- Error bars may be added using the `arrows()` function.
- Via `par()` many graphical parameters may be set (see `?par`), for example `par(mgp=c(2, .7, 0))` reduces the distance between labels and axes

Graphical parameters I

```
adj # justification of text
bty # box type for legend
cex # size of text or data symbols (multiplier)
col # color, see colors()
las # rotation of text in margins
lty # line type (solid, dashed, dotted, ...)
lwd # line width
mpg # placement of axis ticks and tick labels
pch # data symbol type
tck # length of axis ticks
type # type of plot (points, lines, both, none)
```

Graphical parameters II

```
par()
```

```
mai # size of figure margins (inches)
mar # size of figure margins (lines of text)
mfrow # number of sub-figures on a page:
      # par(mfrow=c(1, 2)) creates two sub-figures
oma # size of outer margins (lines of text)
omi # size of outer margins (inches)
pty # aspect ratio of plot region (square, maximal)
```

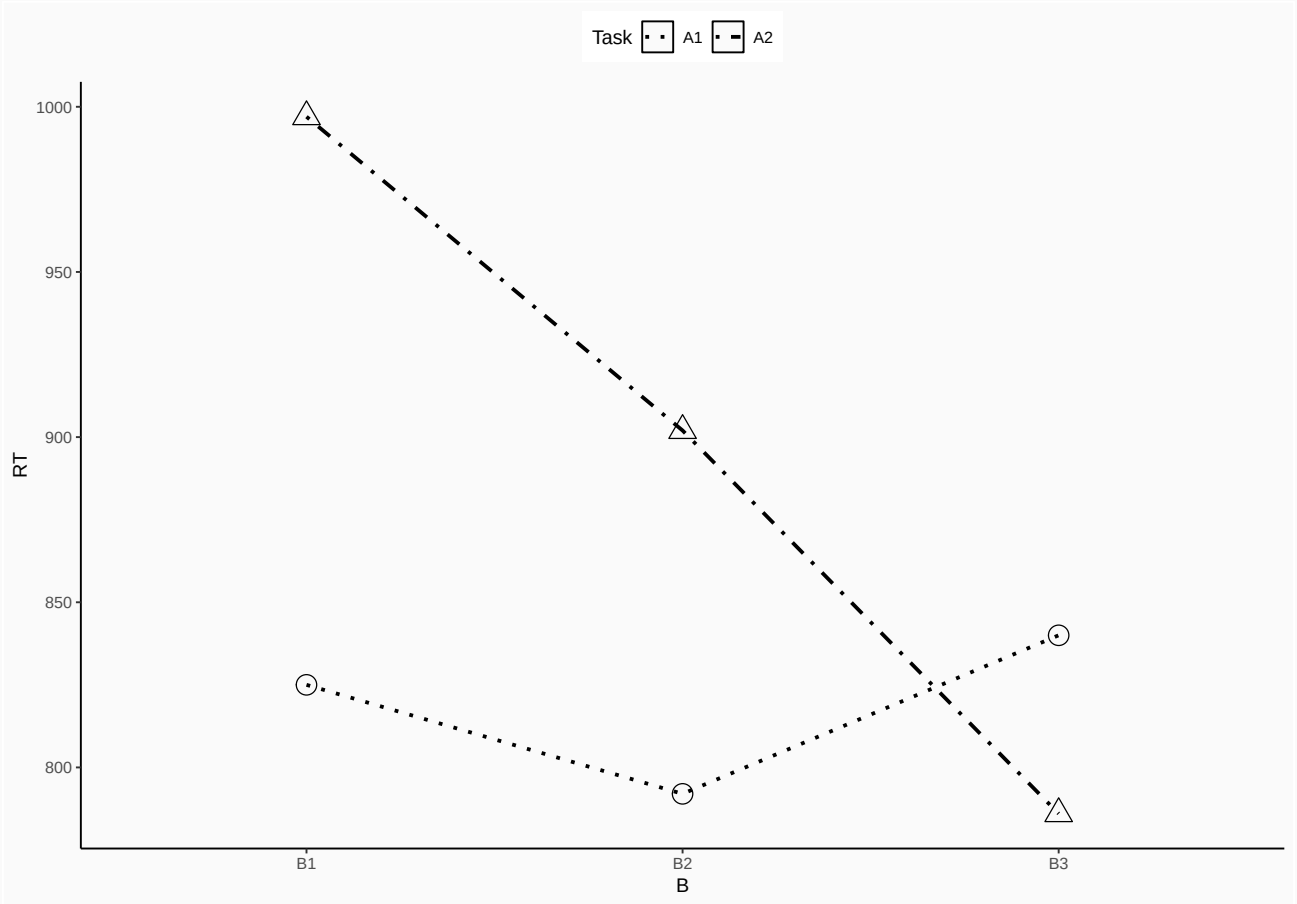
ggplot2

ggplot2 (Grammar of Graphics plot, Wickman, 2016) is one of the best packages for plotting raw data and results:

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

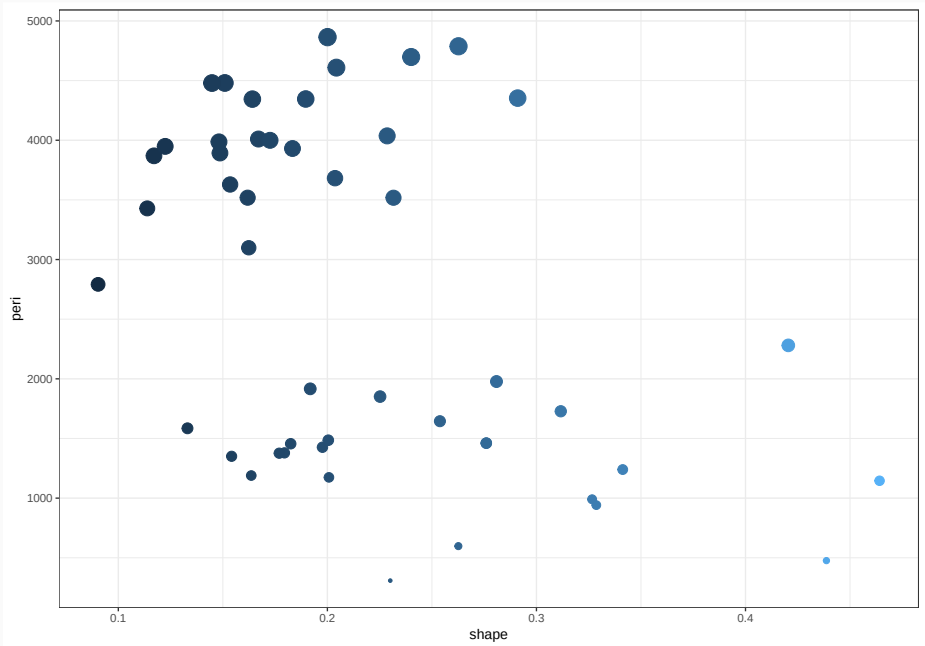
The code for the previous plot:

```
ggplot(dat, aes(x = B, y = rt, group = A)) +
  geom_point(pch=dat$A, size = 5) +
  geom_line(aes(linetype=A), size=1)  + theme_classic() +
  ylab("RT") +  scale_linetype_manual("Task", values =c(3,4),
                                      labels = c("A1", "A2")) +
  scale_x_discrete(labels = c("B1", "B2", "B3")) +
  theme(legend.position="top",
        panel.background =  element_rect(fill = "#FAFAFA",
                                         colour = "#FAFAFA"),
        plot.background = element_rect(fill = "#FAFAFA"),
        legend.key = element_rect(fill = "#FAFAFA"))
```


Task
A1
A2
1000
950
900
850
800
RT
B1
B2
B3
B

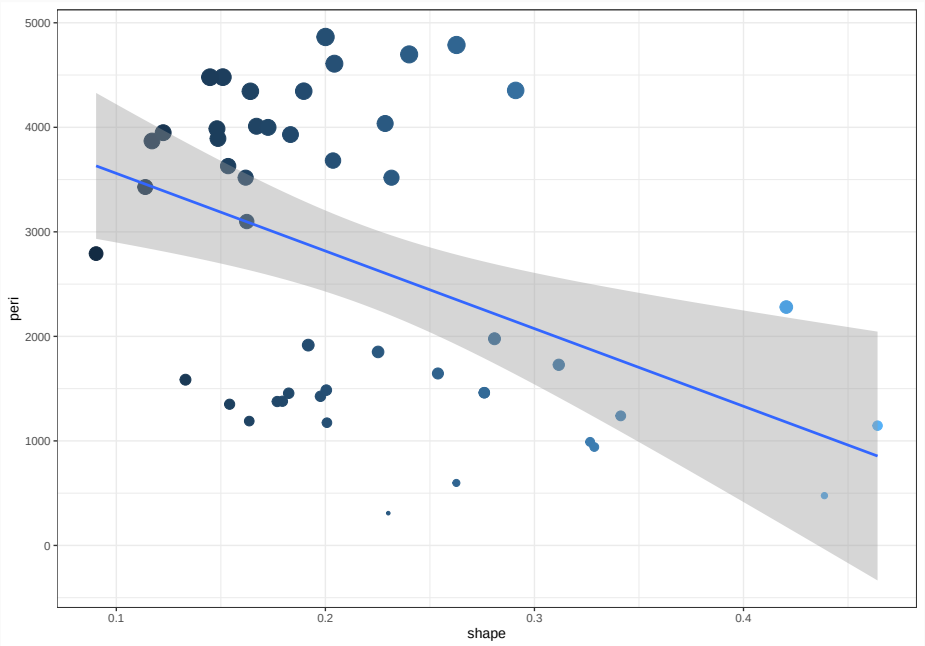
Raw data

```
ggplot(rock,
       aes(y=peri,x=shape, color =shape,
           size = peri)) + geom_point() +
  theme_bw() + theme(legend.position = "none")
```

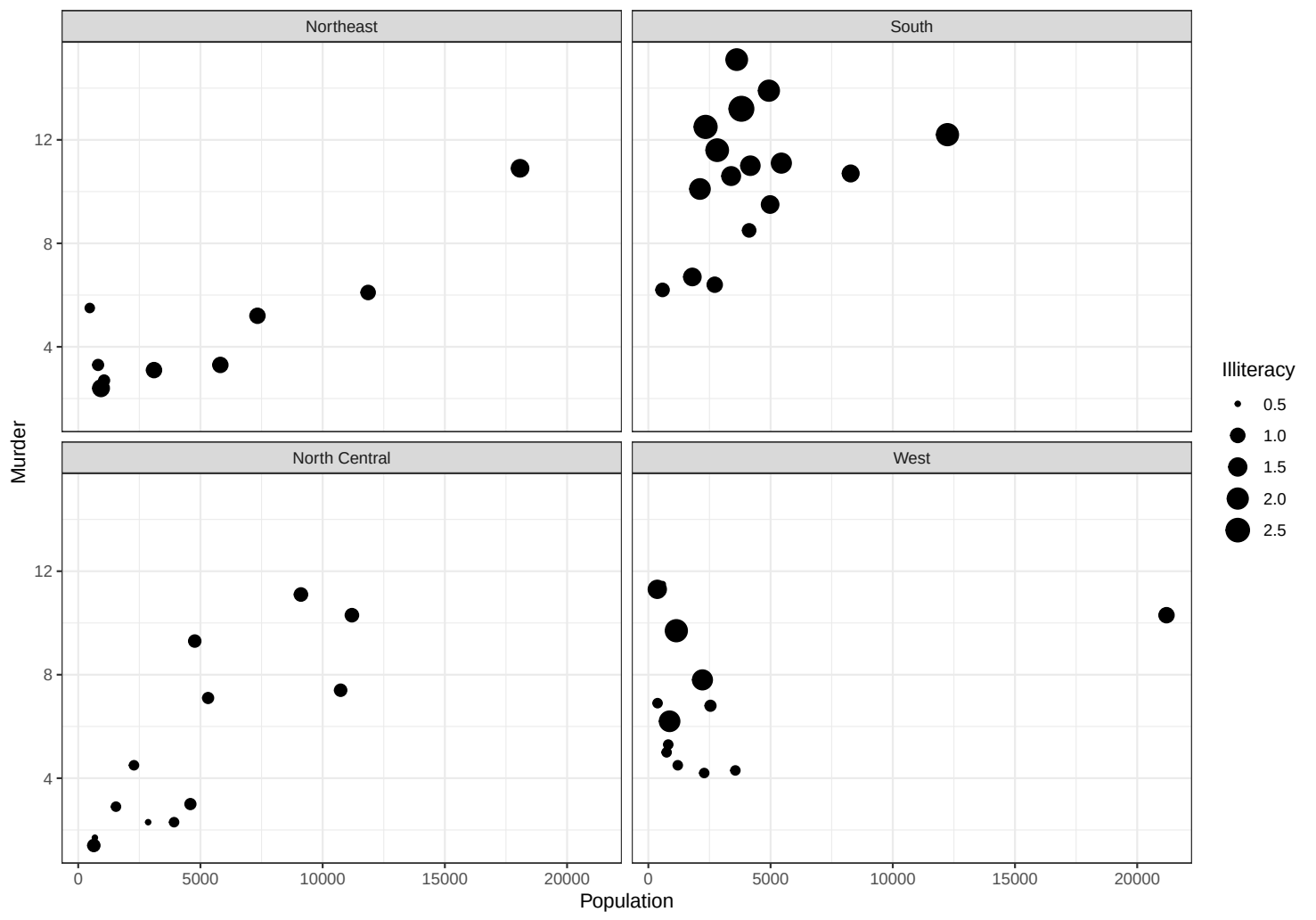


Linear model

```
ggplot(rock,
       aes(y=peri,x=shape, color =shape,
           size = peri)) + geom_point() +
  theme_bw() + theme(legend.position = "none") +
  geom_smooth(method="lm")
```



Multi Panel


Northeast
South
North Central
West
Murder
Population
Illiteracy
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
4
8
12

Multi panel code

```
states = data.frame(state.x77, state.name = state.name,
                    state.region = state.region)

ggplot(states,
       aes(x = Population, y = Murder,
           size = Illiteracy)) + geom_point() +
  facet_wrap(~state.region) + theme_bw()
```

Different plots in the same panel

use `grid.arrange()` function from the `gridExtra` package:

```
install.packages("grideExtra") ; library(gridExtra)
```

```
murder_raw = ggplot(states,  # raw data
                    aes(x = Illiteracy, y = Murder)) +
                .....

murder_lm = ggplot(states,  # lm
                   aes(x = Illiteracy, y = Murder)) +
                .....
```

Combine the plots together:

```
grid.arrange(murder_raw, murder_lm,
                nrow=1) # plots forced to be the same row
```

Combine the plots together

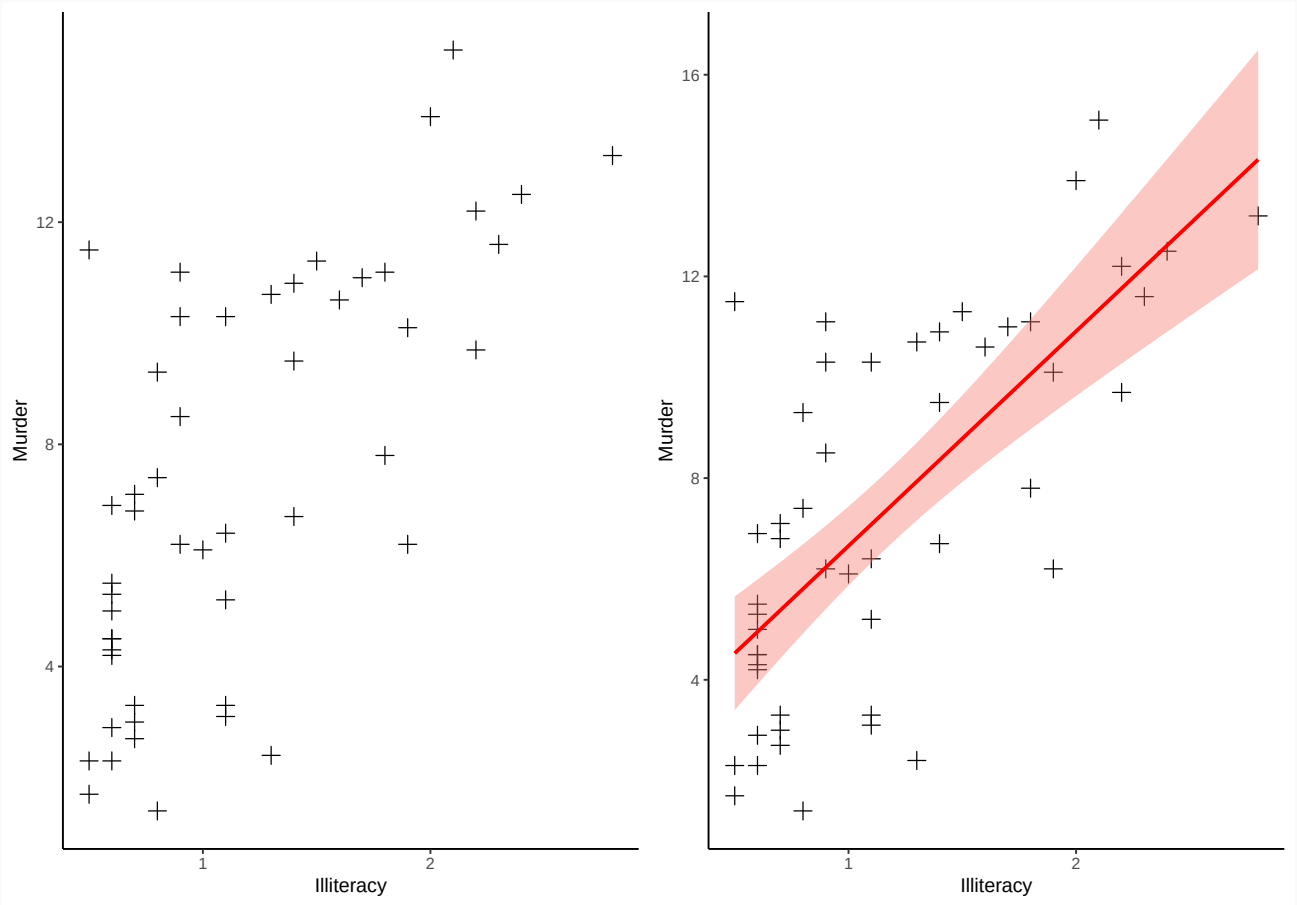


Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

The `stats` package (built-in package in R) contains function for statistical calculations and random number generator

see `library(help=stats)`

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Nominal data:

- `binom.test()`: exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment
- `chisq.test()`: contingency table χ^2 tests

Metric response variable:

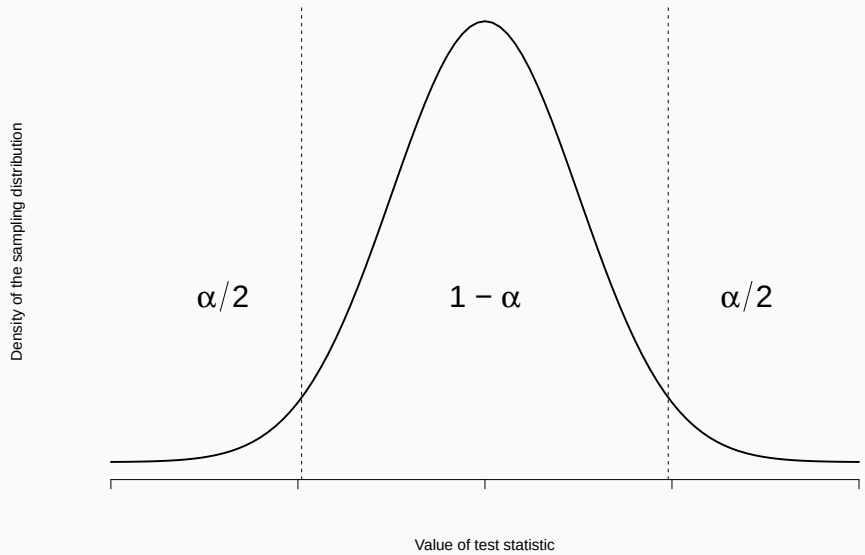
- `cor.test()`: association between paired samples
- `t.test()`: one- and two-sample t tests (also for paired data)
- `var.test()`: F for testing the homogeneity of variances

What is the *p*-value?

p-value:

> *conditional probability of obtaining a test stastic that is at least as extreme as the one observed, given that the null hyphothesis is true*

If $p < \alpha$ (i.e., the probability of rejecting the null hypothesis when it is true) $\rightarrow$ the null hypothesis is rejected



Binomial test

Observations X_i **must** be independent

Hypotheses:

1. H_0: $p = p_0$ H_1: $p \neq p_0$
2. H_0: $p = p_0$ H_1: $p < p_0$
3. H_0: $p = p_0$ H_1: $p > p_0$

Test statistic:

$$T = \sum_{i=1}^{n} X_i, \quad T \sim \mathcal{B}(n, p_0)$$

In R:

```
binom.test(5, 10, p = 0.25)
```

χ^2 test

Independence of observations

Hypothesis:

- H_0: $P(X_{ij} = k) = p_k$ for all $i = 1, \ldots, r$ and $j = 1, \ldots, c$
- H_0: $P(X_{ij} = k) \neq P(X_{i'j} = k)$ for *at least* one $i \in \{1, \ldots, r\}$ and $j \in \{1, \ldots, c\}$

Test statistic:

$$\chi^2 = \sum_{i=1}^{n} \frac{(x_{ij} - \hat{x}_{ij})^2}{\hat{x}_{ij}}, \ \chi^2 \sim \chi^2(r-1)(c-1)$$

In R:

```
tab <- xtabs(~ education + induced, infert)
chisq.test(tab)
```

Correlation test

Hypothesis:

- H_0: $\rho_{XY} = 0$, H_1: $\rho_{xy} \neq 0$
- H_0: $\rho_{XY} = 0$, H_1: $\rho_{xy} < 0$
- H_0: $\rho_{XY} = 0$, H_1: $\rho_{xy} > 0$

Test statistic:

$$T = \frac{r_{xy}}{\sqrt{1 - r_{xy}^2}} \sqrt{n-2}, \;\; T \sim t(n-2)$$

In R:

```
cor.test(~ speed + dist, cars,
         alternative = "two.sided")
```

Two (indepdent) sample t test

Independent samples from normally distributions where σ^2 are unknown but homogeneous

- H_0: $\mu_{x_1-x_2} = 0$, H_1: $\mu_{x_1-x_2} \neq 0$
- H_0: $\mu_{x_1-x_2} = 0$, H_1: $\mu_{x_1-x_2} < 0$
- H_0: $\mu_{x_1-x_2} = 0$, H_1: $\mu_{x_1-x_2} > 0$

Test statistic:

$$T = \frac{\bar{x_1} - \bar{x_2}}{\sigma_{\bar{x_1}-\bar{y_2}}}, \; T \sim t(n_1 + n_2 - 2)$$

R function:

```
t.test(len ~ supp, data = ToothGrowth,
       var.equal = TRUE)
```

Two (depedent) sample t test

Observations on the same sample

Hypothesis:

- H_0: $\mu_D = 0$, H_1: $\mu_D \neq 0$
- H_0: $\mu_D = 0$, H_1: $\mu_D < 0$
- H_0: $\mu_D = 0$, H_1: $\mu_D > 0$

Test statistic:

$$T = \frac{d}{\sigma_d}, \quad T \sim t(m-1)$$

R function:

```
with(sleep,
     t.test(extra[group == 1],
            extra[group == 2], paired = TRUE))
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Formulae

Statistical models are represented by formulae which are extremely close to the actual statistical notation:

in `R`	Model
`y ~ 1 + x`	$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$
`y ~ x`	(same but short)
`y ~ 0 + x`	$y_i = \beta_1 x_i + \varepsilon_i$
`y ~ x_A * x_B`	$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_j + (\beta_1 \beta_2) x_{ij} + \varepsilon_{ij}$

Linear models

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \varepsilon$$

In R:

```
lm(y ~ x1 + x2 + ... + xk, data)
```

Extractor functions I

```
coef()     # Extract the regression coefficients
summary()  # Print a comprehensive summary of the results of
           # the regression analysis
anova()    # Compare nested models and produce an analysis
resid()    # Extract the (matrix of) residuals
plot()     # Produce four plots, showing residuals, fitted
           # values and some diagnostics
model.matrix()
           # Return the design matrix
```

Extractor functions II

```
vcov()      # Return the variance-covariance matrix of the
            # main parameters of a fitted model object
predict()   # A new data frame must be supplied having the
            # same variables specified with the same labels
            # as the original. The value is a vector or
            # matrix of predicted values corresponding to
            # the determining variable values in data frame
step()      # Select a suitable model by adding or dropping
            # terms and preserving hierarchies. The model
            # with the smallest value of AIC (Akaike's
            # Information Criterion) discovered in the
            # stepwise search is returned
```

Generalized linear models

$$g(\mu) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \varepsilon$$

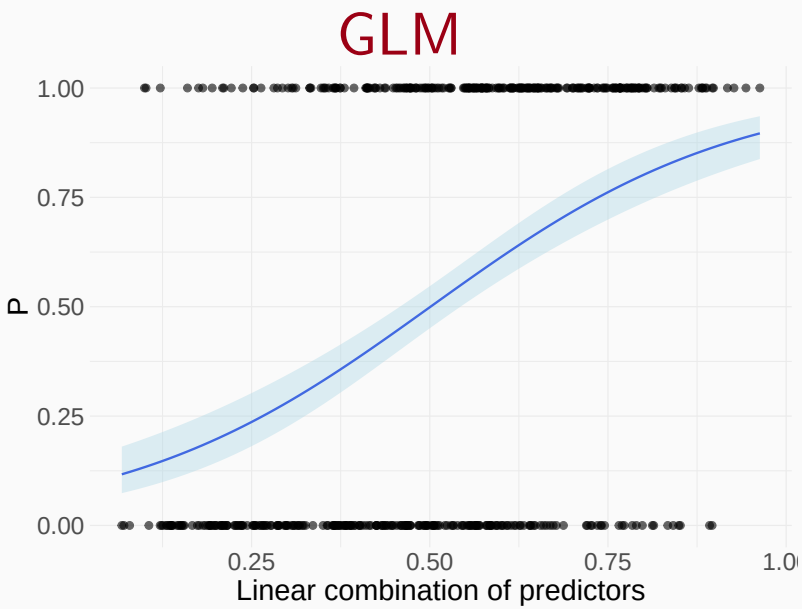
$g()$ is the link functions that connects the mean to the linear combination of predictors.

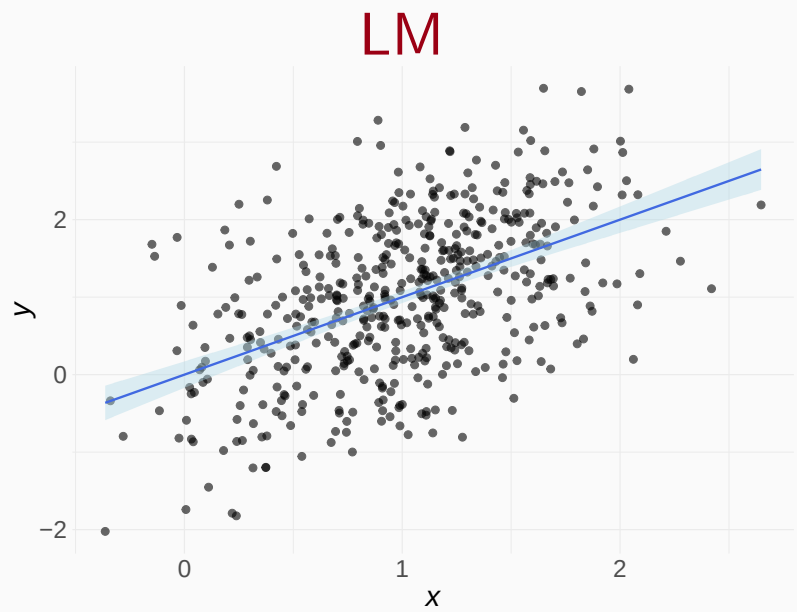
A GLM is composed of three elements: The response distribution, the link function, and the linear combination of predictors

In R:

```
glm(y ~ x1 + x2 + ... + xk, family(link), data)
```

LM vs GLM


GLM
1.00
0.75
0.50
0.25
0.00
p
0.25
0.50
0.75
Linear combination of predictors


LM
2
0
−2
y
0
1
2
x

Families

A special link function to each response variable. In R some different link functions are available by default:

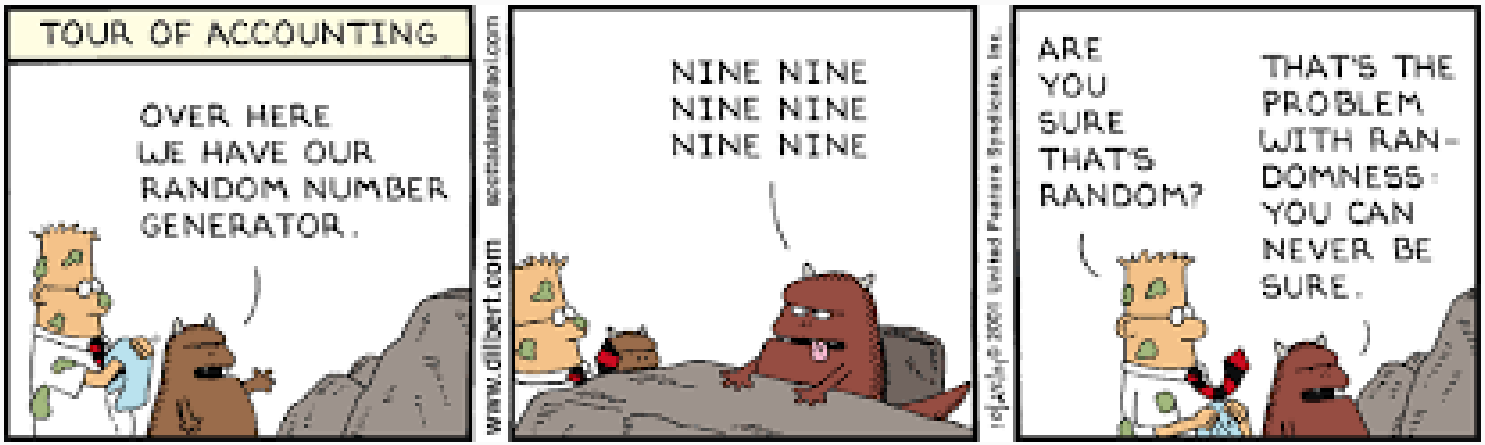
```
## Family name       Link functions
binomial             logit, probit, log, cloglog
gaussian             identity, log, inverse
Gamma                identity, inverse, log
inverse.gaussian     1/mu^2, identity, inverse, log
poisson              log, identity, sqrt
quasi                logit, probit, cloglog, identity, inverse,
                     log, 1/mu^2, sqrt
```

Table of Contents

1 PeRché?
2 Come è fatto
3 Con cosa lavoriamo
4 Le basi
5 L'ambiente e l'ordine
6 Working directories
7 Structures in R
8 The king of data structure: data frames
9 Data input and output
10 Programming
11 Graphics
12 R for statistical computing
13 Classical hypothesis testing in R
14 Generalized Linear Models (GLMs)

Random numbers generation

Use a random monster:



but its better with R

Random numbers drawn from a statistical distribution → the distribution name (see `??Distributions` for an exhaustive list of distribution) prefixed by `r` (random)

```
rnorm(10, mean = 0, sd = 1) # draw 10 numbers from a
                            # normal distr.
rt(10, df = 20)             # draw 10 numbers from a
                            # t distr. with 20df
```

Sampling (with or without replacement) from a vector:

```
sample(1:5, size = 10, replace = T)
```

```
 [1] 5 1 1 2 5 4 4 1 2 5
```

Make the simulations replicable by *seeding* them:

```
set.seed(999)
rpois(4, 5)
```

Bootstrap by resampling

- Compute the sample statistics on multiple bootstrap samples Bs drawn with replacement from the original data
- Assess the variability of the statistics via the distribution of the bootstrap replicates (i.e., the statics computed on the bootstrap samples)

Bootstrap confidence intervals

Percentile intervals are the $1 - \alpha$ confidence intervals for the sample statistics with limits given by the quantiles of the bootstrap distribution

In R

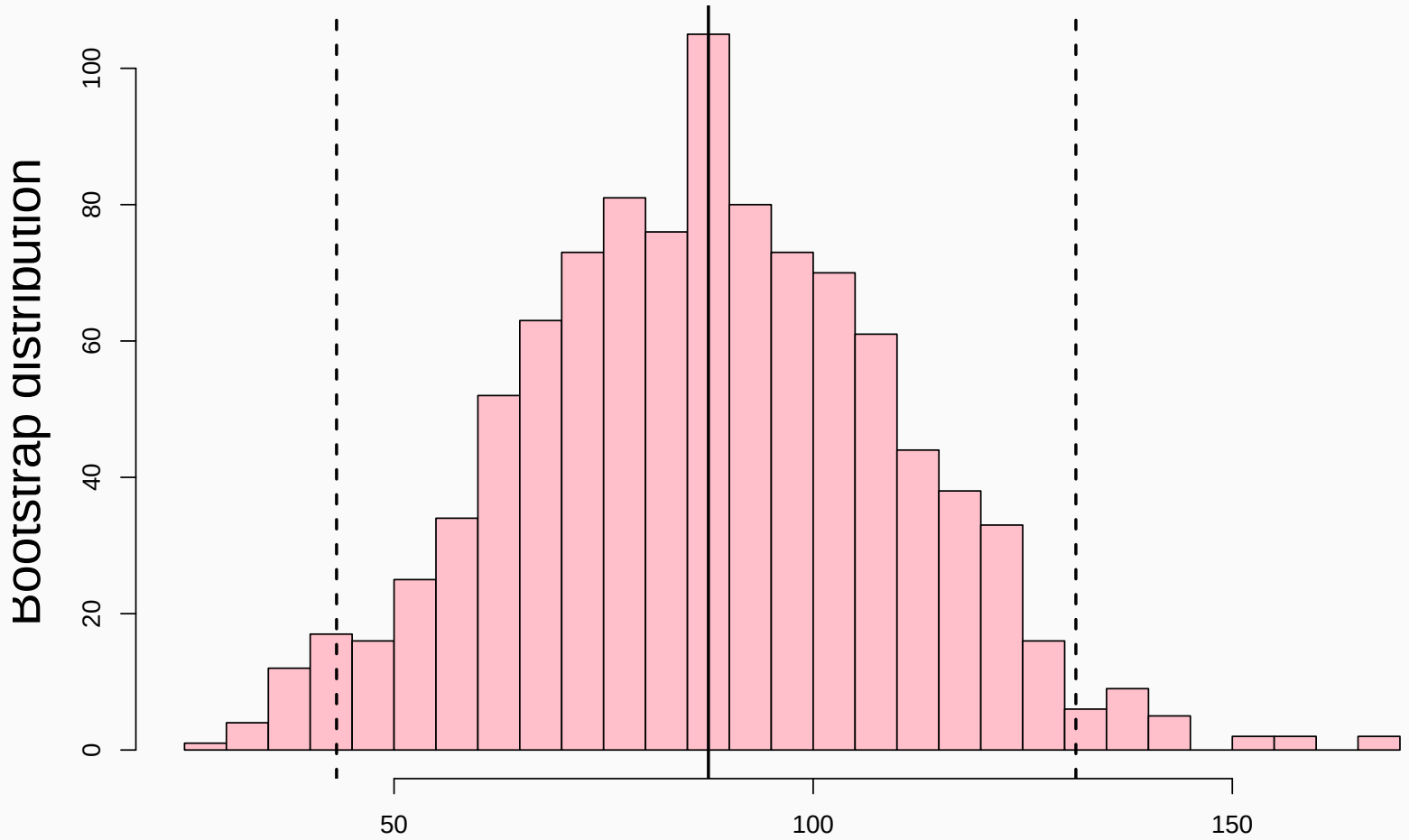
```
# example taken from Prof. Wickelmaier
mouse <- data.frame(
    grp = rep(c("trt", "ctl"), c(7, 9)),
    surv = c(94, 197, 16, 38, 99, 141, 23, # trt
              52, 104, 146, 10, 50, 31, 40, 27, 46) # ctl
    )
mean(mouse$surv[mouse$grp == "trt"]) #
```

```
[1] 86.85714
```

```
## Resampling
sam1 <- numeric(1000) # 1000 bootstrap replicates
for(i in seq_along(sam1)){
 trt <- sample(mouse$surv[mouse$grp == "trt"], 7, replace=T)
 sam1[i] <- mean(trt)
}
```

```
quantile(sam1, c(.025, .975))
```

```
    2.5%     97.5%
43.14286 131.33929
```



Parametric bootstrap

For the likelihood ratio test:

- Fit a general (M_1) and a restricted model (M_0) to the original data x. Compute the original likelihood ratio s(x) between M_1 and M_0
- Simulate B bootstrap samples based on the stochastic part of the restricted model: These are observations for which H_0 is true
- For each sample, fit M_1 and M_0 and compute the bootstrap replicate of the likelihood ratio between them
- Assess the significance of the original likelihood ratio via the sampling distribution of bootstrap replicates

```r
## Model fit to original data
lm0 <- lm(surv ~ 1, mouse)  # H0: no difference between gr
lm1 <- lm(surv ~ grp, mouse) # H1: group effect
anova(lm0, lm1)              # original likelihood ratio
```

```
[1] 1.257516
```

```r
 ## Parametric bootstrap
sim1 <- numeric(1000)
for(i in seq_along(sim1)){
  surv0 <- simulate(lm0)$sim_1  # simulate from null model
  m0 <- lm(surv0 ~ 1, mouse)    # fit null model
  m1 <- lm(surv0 ~ grp, mouse)  # fit alternative model
  sim1[i] <- anova(m0, m1)$F[2] # bootstrap likeli. ratio
  }
```

The bootstrap $p - value$ is the proportion of bootstrap replicates that exceed the original likelihood ratio:

```
mean(sim1 >
anova(lm0, lm1)$F[2])
```

```
[1] 0.304
```