Introduzione al Linguaggio R

Analisi Dati e Statistica, 2025–26

Paolo Bosetti
Università di Trento, Dipartimento di Ingegneria Industriale

Ultimo aggiornamento: 17/06/2026

Indice

```
options(width = 60)
set.seed(0)
```

1 Introduzione a R

L'analisi statistica richiede l'uso di software specifico

Oggi i due software/linguaggi più utilizzati in questo campo sono Python e R, seguiti da Matlab

Noi utilizzeremo R perché specifico per la statistica, orientato alla grafica e open source

1.1 Link utili

- GNU-R: https://cran.mirror.garr.it/CRAN/
- RStudio: https://posit.co/downloads/
- Cheat sheet: https://posit.co/resources/cheatsheets/
- Tidyverse: https://tidyverse.org
- Materiale corso: https://github.com/pbosetti/ADAS-25

1.2 Ambiente RStudio

- Installazione: prima R, poi RStudio
- **Solo su windows**, installare anche Rtools
- RStudio lavora su cartelle o (meglio) **progetti** (`.Rproj`)
- Un progetto contiene anche impostazioni specifiche e comuni ai file nella cartella
- Una **sessione** di RStudio può operare su un unico progetto
- Si possono aprire più sessioni contemporaneamente
- RStudio è un ambiente molto potente e complesso, adatto anche alla compilazione di *report* tecnici, articoli, libri e presentazioni (come questa)

Nota: Queste slide contengono una versione ridotta e integrata di R, utile per provare in diretta il codice qui illustrato: cliccare sull'icona del prompt in basso a sinistra, o premere il tasto §.

1.3 Il linguaggio R

- R è un linguaggio ad alto livello, declarativo, interpretato, a sintassi C-like
- R è sia un linguaggio, sia un interprete
- R è un *dynamically typed language*
- R è utilizzato sia in **modalità script** che in **modalità interattiva**
- R è nato come versione GNU open source di S, un linguaggio proprietario per analisi statistiche
- RStudio è una IDE proprietaria (ma free) per R

1.4 Assegnazioni

Ogni linguaggio usa delle **variabili** per memorizzare valori ed oggetti mediante un'operazione di **assegnazione**:

```
a <- 1
# ma anche
b = 2
# tuttavia si preferisce la notazione a freccia,
# perché funziona anche così:
3 -> c
# per visualizzare il valore di una variabile:
c
```

```
[1] 3
```

```
# in un colpo solo, assegnazione e visualizzazione:
(d <- "stringa")
```

```
[1] "stringa"
```

L'esecuzione di un comando fornisce direttamente un risultato:

```
12*12
```

```
[1] 144
```

Si noti il testo `[1]` all'inizio della riga di output: sarà chiaro più avanti

1.5 Tipi, o classi native

- R ha 6(+1) tipi o *classi* native
  - character: `"a"`, `"string"`, `'my text'`
  - numeric: `1`, `3.1416`
  - integer: `1L`
  - logical: `TRUE`, `FALSE` (oppure `T` e `F`)
  - complex: `1+4i`
  - function: una *funzione*
  - (raw: sequenza di bit)
- Ogni istanza è intrinsecamente un vettore
- Uno scalare è semplicemente un vettore di lunghezza 1

1.6 Valori speciali

- Sono definiti i seguenti valori speciali:
  - `NA`: valore mancante
  - `NULL`: niente
  - `Inf`: Infinito
  - `NaN`: Not a Number (esempio `0/0`)

1.7 Coercizione

- Quando si mescolano tipi differenti, ad es. in un vettore, R li trasforma in un tipo comune:

```
c(1L, 7, "2")
```

```
[1] "1" "7" "2"
```

```
c(T, 0)
```

```
[1] 1 0
```

```
as.numeric(c("a", "1"))
```

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

```
[1] NA  1
```

```
as.character(c(1, 1.7))
```

```
[1] "1"   "1.7"
```

1.8 Vettori

```
# Si costruiscono con l'operatore/funzione c():
v1 <- c(10, 2, 7.5, 3)
# oppure con una sequenza:
v2 <- 1:10
# anche con passo specificato:
v3 <- seq(1, 10, 0.5)
# Le funzioni si chiamano con le parentesi tonde,
# separando argomenti con ,
```

```
v1 <- c(10, 2, 7.5, 3)
v2 <- 1:10
v3 <- seq(1, 10, 0.5)
```

Si noti in questo caso l'output per `v3`:

```
(v3 <- seq(1, 10, 0.5))
```

```
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
5.5  6.0
[12]  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

Il primo elemento della prima riga è l'elemento `[1]` del vettore, mentre il primo elemento della seconda riga è l'elemento `[12]`. In tutto, il vettore `v3` ha 19 elementi

1.9 Vettori

Le variabili sono nativamente dei vettori. Gli scalari sono solo vettori di dimensione 1:

```
a <- 10
length(a)
```

```
[1] 1
```

```
length(v3)
```

```
[1] 19
```

Le funzioni e gli operatori agiscono quindi sempre su vettori (sono *vettorializzati*):

```
a * 2
```

```
[1] 20
```

```
v3 + 2
```

```
 [1]  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
7.5  8.0
[12]  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0
```

1.10 Facciamo qualche prova

```
(v1 <- c(7, 2.5, 3, 1:10))
v1[c(3,7,2)]
```

1.11 Introspezione

- Funzioni utili per ispezionare gli oggetti:
  - `mode()`: *storage mode*
  - `class()`: classe (alto livello, uguale a `mode()` per tipi base)
  - `typeof()`: tipo (basso livello)
  - `length()`: lunghezza vettore
  - `attributes()`: metadati
  - `str()`: struttura di un oggetto
  - `summary()`: riassunto statistico

1.12 Matrici

- Si costruiscono con la funzione `matrix()`

```
(m1 <- matrix(1:10, 2, 5))
```

- la funzione `array()` costruisce matrici *n*-dimensionali
- Una matrice è un vettore con attributo `dim`:

```
attr(m1, "dim")
v <- 1:4
attr(v, "dim") <- c(2,2) # equivale a dim(m) <- c(2,2)
v
```

```
m1 <- matrix(1:10, 2, 5)
```

1.13 Fattori

- Una classe aggiuntiva (non base) ma molto comune è `factor`
- Rappresenta variabili categoriche (ordinate o non)

```
(vf <- factor(LETTERS[1:5], levels=LETTERS[c(2, 1, 3,
5, 4)], ordered=T))
```

```
[1] A B C D E
Levels: B < A < C < E < D
```

```
class(vf)
```

```
[1] "ordered" "factor"
```

```
typeof(vf)
```

```
[1] "integer"
```

```
vf[1] < vf[3]
```

```
[1] TRUE
```

1.14 Stringhe

Una stringa può essere pensata come un vettore di caratteri di lunghezza maggiore di 1.

Le funzioni di manipolazione di stringhe più comuni sono `cat()`, `paste()` e `paste0()`. La prima serve a stampare la stringa tale e quale:

```
cat("Ciao!")
```

```
Ciao!
```

Le due funzioni `paste()` e `paste0()` servono a unire due o più stringhe, la prima inserendo uno spazio in mezzo, la seconda senza spazio:

```
paste("Ciao,", "Mondo!")
```

```
[1] "Ciao, Mondo!"
```

```
paste0("Ciao,", "Mondo!")
```

```
[1] "Ciao,Mondo!"
```

1.15 Indicizzazione

- La sintassi di indicizzazione di R è molto flessibile e potente
- si usano sempre le parentesi quadre `[r,c]`, la **base è 1**
- se un indice manca, significa "tutte le righe|colonne"

```
v3[3]
```

```
[1] 2
```

```
m1[1,1]
```

```
[1] 1
```

```
m1[2,]
```

```
[1]  2  4  6  8 10
```

```
m1[,]
```

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

1.16 Indicizzazione

- Un indice può essere anche un vettore di posizioni o un vettore di valori booleani

```
v1[c(2,4,1)] # estrae solo gli elementi 2, 4, e 1
```

```
[1]  2  3 10
```

```
v2[v2 %% 2 == 0] # estrae gli elementi divisibili per 2
```

```
[1]  2  4  6  8 10
```

Il secondo caso funziona grazie all'**operatore modulo**:

```
v2 %% 2 == 0 # operatore modulo (resto)
```

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

> Tip
>
> NOTA: `TRUE` e `FALSE` possono essere abbreviati in `T` e `F`

1.17 Facciamo qualche prova

```
v1 <- c(10, 2, 7.5, 3)
v2 <- 1:10
v1[c(2,4,1)]
v2[v2 %% 2 == 0]
```

```
m1 <- matrix(1:4, nrow = 2, byrow=T)
m2 <- matrix(c(2, 3, 1, 7, 5, 1), nrow = 2, byrow=T)
m1 * m2 # Errore!
```

> Tip
>
> Cercare nell'help in linea funzioni come `%*%`

1.18 Funzioni

- Le funzioni sono *first class objects*, cioè sono variabili come altre
- possono essere assegnate a variabili e passate a funzioni

```
my_fun <- function(x) x^2
my_fun(1:5)
```

```
[1]  1  4  9 16 25
```

```
your_fun <- my_fun
your_fun(6)
```

```
[1] 36
```

```
my_apply <- function(x, f) f(x)
my_apply(10, my_fun)
```

```
[1] 100
```

- Se la definizione richiede più righe, si usa un **blocco** tra `{}`
- Ogni funzione ritorna **sempre** l'ultima espressione valutata
- Oppure esplicitamente mediante `return()`

1.19 Funzioni freccia (*replacement functions*)

- Abbiamo visto cose come `dim(v) <- c(2,3)`: come si dichiarano?

```
`pwr<-` <- function(obj, value) obj ** value
a <- 2
pwr(a) <- 10
a
```

```
[1] 1024
```

- L'ultimo argomento **deve** chiamarsi `value` e rappresenta il lato destro dell'assegnazione!

1.20 Controllo di flusso

R supporta le tipiche istruzioni di controllo di flusso

- per istruzioni condizionali:
  - `if(cond) expr`
  - `if(cond) true.expr  else  false.expr`
  - `ifelse(cond, true.expr, false.expr)`
- e per i cicli:
  - `for(var in seq) expr`
  - `while(cond) expr`
  - `repeat expr`
  - `break`
  - `next`

1.21 Esercizio

1.22 Esercizio

Scrivere una funzione `sum_odd` che sommi tutti gli elementi con indice dispari di un vettore numerico. La funzione deve avere un argomento `x` e restituire la somma di tutti gli elementi di `x`, usando un ciclo `for`.

```
sum_odd <-function(x) {
  # scrivi qui il codice
}

sum_odd(1:10)
```

> ⚠ Warning
>
> L'uso del ciclo `for` è sconsigliato in R (perché è più lento di alcune alternative), ma è utile per esercitarsi con il controllo di flusso.

1.23 Soluzione

```
sum_odd <-function(x) {
  odd <- x[c(T, F)]
```

```
  sum <- 0
  for (e in odd) {
    sum <- sum + e
  }
  return(sum)
}

sum_odd(1:10)
```

1.24 Argomenti delle funzioni

- Gli argomenti possono essere indicati per posizione o per nome
- Gli argomenti nominati possono comparire in qualsiasi ordine
- Gli argomenti possono avere un default, in tal caso sono opzionali

```
f <- function(x, y, n=10, test=F) {
  ifelse(test, 0, x^y + n)
}
f(2, 10)
```

```
[1] 1034
```

```
f(test=F, y=10, x=2)
```

```
[1] 1034
```

```
f(test=T)
```

```
[1] 0
```

1.25 Differenza tra `<-` e `=`

- L'operatore `=` come assegnazione è valido solo al *top-level*
- L'operatore `<-` è valido ovunque, anche come argomento di funzione:

```
system.time(m <- mean(1:1E6))
```

```
   user  system elapsed
  0.002   0.000   0.002
```

```
m
```

```
[1] 500000.5
```

1.26 Dataframe

- In R più che matrici si usano `dataframe`
- Si tratta di tabelle organizzate per colonne, internamente omogenee ma potenzialmente di tipi differenti

```
df <- data.frame(A=1:10, B=letters[1:10])
head(df)
```

```
  A B
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
```

1.27 Dataframe

- Un dataframe può essere indicizzato come una matrice (due indici)
- Oppure selezionando una colonna con la notazione `$`

```
df[2,2]
```

```
[1] "b"
```

```
df$B[2]
```

```
[1] "b"
```

Anche in assegnazione:

```
df$C <- LETTERS[1:10]
head(df, 3)
```

```
  A B C
1 1 a A
2 2 b B
3 3 c C
```

1.28 Esercizio

Dal data frame `mtcars`, estrarre le righe corrispondenti ai veicoli con cilindrata (`disp`) maggiore di 200 e peso minore di 3.5 tonnellate (`wt`)

```
head(mtcars)
```

1.29 Liste

Una lista è una sequenza di coppie chiave-valore, cioè una sequenza di valori identificati da un nome, o chiave.

A differenza dei vettori (che sono **sempre** omogenei) possono contenere valori eterogenei.

```
(l <- list(A="uno", B="due", C=1:4))
```

```
$A
[1] "uno"

$B
[1] "due"

$C
[1] 1 2 3 4
```

Una lista può essere indicizzata in tre modi:

- con l'operatore `$`: si estrae un unico elemento per nome
- con l'operatore `[]`: si estraggono elementi per posizione **e si ottiene una lista**
- con l'operatore `[[]]`: si estrae un unico elemento per posizione

1.30 Algoritmi di uso comune

- Ordinamento: `sort`, `rev`, `order`
- Campionamento: `sample`, `expand.grid`
- Aggregazione: `by`, `aggregate`
- Tabelle di contingenza: `table`

1.31 Ordinamento di vettori

Per ordinare un vettore si usa la funzione `sort`:

```
v <- runif(5, 1, 10)
sort(v)
```

```
[1] 3.389578 4.349115 6.155680 9.070275 9.173870
```

```
rev(sort(v))
```

```
[1] 9.173870 9.070275 6.155680 4.349115 3.389578
```

```
sort(v, decreasing = T)
```

```
[1] 9.173870 9.070275 6.155680 4.349115 3.389578
```

1.32 Ordinamento di dataframe

Per riordinare un data frame si estraggono gli indici ordinati:

```
df <- data.frame(A=1:5, B=runif(5))
df[order(df$B),]
```

```
  A         B
1 1 0.2016819
5 5 0.6291140
4 4 0.6607978
2 2 0.8983897
3 3 0.9446753
```

La funzione `order` ritorna appunto gli indici di un vettore ordinati secondo i valori:

```
order(df$B)
```

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

dove il primo è l'indice del valore più piccolo di `df$B` e l'ultimo l'indice del più grande

1.33 Campionamento

Campionare un insieme di dati (un vettore) significa estrarre un sottoinsieme (detto **campione**) di valori in maniera casuale. Si esegue con la funzione `sample`:

```
sample(1:10) # senza reinserimento
```

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

```
sample(1:10, replace = T) # con reinserimento
```

```
 [1]  9  5  5  9  9  5  5  2 10  9
```

La dimensione del campione può essere uguale (caso sopra) o più piccola dell'insieme iniziale:

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

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

```
sample(10)  # generazione interi casuali senza
ripetizione
```

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

1.34 Griglie

Una **griglia** è una matrice che contiene tutte le combinazioni (ordinate) tra n vettori di dimensioni possibilmente diverse. In R viene rappresentata come un data frame e costruita con la funzione `expand.grid`:

```
(df <- expand.grid(A=1:2, B=c("-", "+"), D=c("a", "b",
"c")))
```

```
  A B D
1 1 - a
2 2 - a
3 1 + a
```

```
4  2 + a
5  1 - b
6  2 - b
7  1 + b
8  2 + b
9  1 - c
10 2 - c
11 1 + c
12 2 + c
```

1.35 Esercizio

Riordinare il dataframe `df` in maniera casuale:

```
df <- expand.grid(A=1:3, B=LETTERS[1:2])
# aggiungo una colonna di numeri casuali
# riordino df per quella colonna
df # stampo df
```

1.36 Aggregazione

Per aggregazione si intende raggruppare righe aventi elementi comuni in un data frame e applicare ad ogni gruppo una data funzione. È utile ad esempio per il calcolo di sub-totali.

In R può essere eseguita mediante la funzione `by` o la funzione `aggregate` (cambia il tipo di output):

```
by(df$A, INDICES = df$B, FUN=sum)
```

```
df$B: -
[1] 9
------------------------------------------------
df$B: +
[1] 9
```

```
aggregate(A~B, data = df, FUN = sum)
```

```
  B A
1 - 9
2 + 9
```

1.37 Tabelle di contingenza

Una tabella di contingenza conta le occorrenze tra una coppia di colonne in un data frame:

```
head(airquality, n = 3)
```

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
```

```
with(airquality, table(OzHi = Ozone > 80, Month,
                       useNA = "ifany"))
```

```
       Month
OzHi     5  6  7  8  9
  FALSE 25  9 20 19 27
  TRUE   1  0  6  7  2
  <NA>   5 21  5  5  1
```

> Tip
>
> **NOTA**: `with()` serve per risparmiarsi di scrivere `airquality$Ozone` e `airquality$Month`: rende *implicito* il nome del data frame

1.38 Tabelle di contingenza

- È anche utile `tapply()`, che opera su una tabella analogamente alle funzioni di aggregazione:

```
round(with(airquality,
           tapply(Ozone, Month, mean, na.rm=T)), 1)
```

```
   5    6    7    8    9
23.6 29.4 59.1 60.0 31.4
```

Con `aggregate()` si farebbe:

```
aggregate(Ozone~Month,   data=airquality,   FUN=mean,
ra.rm=T)
```

```
  Month    Ozone
1     5 23.61538
2     6 29.44444
3     7 59.11538
4     8 59.96154
5     9 31.44828
```

1.39 Input/output su file

Dato che la statistica si occupa generalmente di grandi quantità di dati è fondamentale poter importare ed esportare dati in formati generici.

Generalmente i dati sono presentati in forma tabulare (per righe e colonne)

I formati più semplici e comuni sono:

- *Flat File*: un file di testo ASCII contenente valori in riga e colonna; le colonne possono essere separate
  - a lunghezza fissa
  - mediante caratteri separatori
- CSV (*Comma-Separated Values*): una versione speciale di FF in cui i campi colonna sono separati da virgole

1.40 Input da file

Un Flat File con campi separati da spazi può avere questo aspetto:

```
# Dati raccolti il 10/8/2023
x y z
1.2 3.7 2.7
2.1 2.5 3.9
3.8 2.2 6.8
```

Un simile file può essere importato come data frame in questo modo:

```
df <- read.table("data_file.txt", header=T, sep=" ",
comment.char="#")
```

La funzione `read.table()` dispone di numerose opzioni che consentono di gestire tutti i possibili casi in cui file contenga campi separati da caratteri specifici (spazi od altro)

1.41 Input da file

Un Flat File con campi a larghezza fissa può avere questo aspetto:

```
# Dati raccolti il 10/8/2023
x     y     z
1.2   3.7   2.7
2.1   2.5   3.9
3.8   2.2   6.8
```

Un simile file può essere importato come data frame in questo modo:

```
df <- read.fwf("data_file.txt", widths=5, header=T,
skip=1)
```

Il parametro `skip=1` richiede di saltare la prima linea (commento)

1.42 Input da file CSV

I file CSV sono FF speciali in cui il separatore di campo è la virgola. In questi casi si usa la funzione `read.csv()` che opera analogamente a `read.table()` ma non richiede di specificare il separatore.

Un CSV ha questo aspetto:

```
# Dati raccolti il 10/8/2023
x,y,z
1.2,3.7,2.7
2.1,2.5,3.9
3.8,2.2,6.8
```

Software che usano lingue latine (Italiano, Spagnolo, Portoghese e Francese) adottano la virgola come separatore decimale. Di conseguenza quando questi software (ad. es. MS Excel) generano dei CSV usano il punto e virgola come separatore di campo.

In questo caso da R è necessario utilizzare la funzione `read.csv2()`, che assume la virgola come separatore decimale e il punto e virgola come separatore di campo.

1.43 Output su file

L'operazione opposta all'importazione di un file in un data frame è l'esportazione di un data frame su un file.

Questa operazione viene eseguita con le funzioni opposte alle precedenti:

- `write.table()`
- `write.fwf()`
- `write.csv()` e `write.csv2()`

Tutte queste funzioni hanno due argomenti obbligatori: il data frame da salvare e il file di destinazione:

```
write.csv(df, "data.csv")
```

Altri argomenti opzionali servono per personalizzare il risultato.

1.44 Tidyverse

Assieme a RStudio è emersa una *new wave* di librerie R che modificano radicalmente l'approccio. Vanno sotto il nome collettivo di `tidyverse`

- `ggplot2`: grafici
- `purrr`: programmazione funzionale
- `dplyr`: manipolazione dati
- `stringr`: manipolazione stringhe
- `tibble`: data frame migliorati
- `readr`: importazione dati
- `tidyr`: preparazione dati
- `lubridate`: manipolazione date

1.45 Tidyverse

L'approccio `tidyverse` ha alcune caratteristiche comuni:

- dati in formato **tidy** (un'osservazione per riga; una variabile, o **osservando**, per colonna)
- composizione di funzioni grafiche con `+` (`ggplot(...) + geom_line()`), ogni funzione è un **layer**
- notazione prefissa con `%>%` (`a %>% str()` invece di `str(a)`)

> **Tip**
>
> È utile consultare i cheat sheet: https://posit.co/resources/cheatsheets/

1.46 Notazione infissa

```
# creo l'istogramma di un campione di 10 elementi da
100 numeri casuali
```

```r
# infissa:
hist(sample(rnorm(100), 10))
```

Poco leggibile, il primo passo dell'algoritmo è quello più interno

1.47 Notazione infissa, sequenziata

```r
# creo l'istogramma di un campione di 10 elementi da
100 numeri casuali
# infissa:
hist(sample(rnorm(100), 10))

# infissa sequenziata:
s <- rnorm(100)
c <- sample(s, 10)
hist(c)
```

Più leggibile, l'algoritmo è più evidente, ma richiede la creazione di variabili intermedie

1.48 Notazione prefissa

```r
# creo l'istogramma di un campione di 10 elementi da
100 numeri casuali
# infissa:
hist(sample(rnorm(100), 10))

# infissa sequenziata:
s <- rnorm(100)
c <- sample(s, 10)
hist(c)

# prefissa con pipe:
rnorm(100) %>% sample(10) %>% hist()
```

Molto più leggibile, l'algoritmo sequenziale è evidente, non sono necessarie variabili intermedie

1.49 Notazione prefissa

```r
# creo l'istogramma di un campione di 10 elementi da
100 numeri casuali
# infissa:
hist(sample(rnorm(100), 10))
```

```r
# infissa sequenziata:
s <- rnorm(100)
c <- sample(s, 10)
hist(c)

# prefissa con pipe:
rnorm(100) %>% sample(10) %>% hist()

# anche su più righe:
rnorm(100) %>%
  sample(10) %>%
  hist
```

Line 15

le righe successive alla prima vanno **indentate**

Line 16

solo quando si usa pipe, se non ci sono argomenti le parentesi sono opzionali

1.50 Help in linea

- L'help in linea per unafunzione può essere ottenuto con il comando `?<nome funzione>`, ad esempio `?summary` (senza parentesi)
- Alcune funzioni però, come `summary()`, sono **funzioni generiche**, e il loro help è scarno
- Le funzioni generiche dirottano la chiamata a una *funzione specifica* definita dalla classe del primo argomento
- Il nome della funzione specifica è del tipo `summary.lm()`, che è la funzione realmente chiamata quando a passo un modello lineare (es. `summary(df.lm)`)
- Alla fine dell'help per la generica (*See also*) c'è l'elenco delle funzioni specifiche note, che vanno consultate per i dettagli di funzionamento desiderati
- In alternativa si può cercare direttamente `?summary.lm`