R per l'analisi statistica multivariata

Unità A: Introduzione ad R

Tommaso Rigon

Università degli Studi di Milano-Bicocca

Informazioni di servizio

Prova

Organizzazione del corso

prova

Cos'è **R**?

- **R** è contemporaneamente un **software statistico**, un **linguaggio di programmazione** ed uno strumento per il **calcolo numerico**.
- **R** è un «read–eval–print loop» (REPL), ovvero un linguaggio di programmazione **interattivo**. E' quindi profondamente diverso per esempio da C/C++.
- Il sito ufficiale è `https://www.r-project.org`.
- **R** è un software **open source** ed è possibile scaricarlo gratuitamente al link `https://cloud.r-project.org`. Per l'installazione è sufficiente seguire le istruzioni.
- **R** è disponibile per tutti i principali sistemi operativi (Windows, Mac Os, Linux).
- Ulteriori informazioni su **R** le trovate tra le FAQ ufficiali `https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-is-R_003f`.

Perché usare **R**?

- **R** è un software **open source** e **gratuito**, a differenza di altri software statistici (SAS, SPSS, Stata, etc.).
- **R** è scritto per gli statistici dagli statistici, con tutti i pro e i contro che ne derivano.
- Esiste un pacchetto **R** per ogni necessità.
- Di **R** sono molto apprezzate le funzionalità legate alla **visualizzazione dei dati**.
- **R** è molto diffuso nelle università e nel mondo accademico.
- **R** è ampiamente utilizzato anche in contesto aziendale ed è ormai diventato uno strumento imprescindibile per un **data scientist**.

Rstudio

- **RStudio** è un'interfaccia utente per il software **R**. Rstudio facilita la scrittura del codice, la visualizzazione dei risultati, la lettura della documentazione.
- L'uso di **Rstudio** è **facoltativo** in questo corso, anche se consigliato. E' possibile scaricarlo al link `https://rstudio.com/products/rstudio/`.
- **RStudio** è disponibile sia nella versione open source (gratuita) che nella versione commerciale. Le due versioni sono identiche, ma la seconda comprende «l'assistenza clienti».
- **Rstudio**, come **R**, è disponibile per tutti i principali sistemi operativi (Windows, Mac Os, Linux).

Operazioni di base

- **R** può essere usato come se fosse una calcolatrice scientifica.
- **Nota**. Tutto quello che viene scritto dopo un cancelletto (#) è considerato un **commento** e non viene eseguito.

Esempi

```
2 + 2
```

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

```
4 * (3 + 5) # La somma entro parentesi viene eseguita per prima
```

```
## [1] 32
```

```
pi / 4 # Pi greco quarti
```

```
## [1] 0.7853982
```

Operazioni di base II

- Per calcolare la potenza a^b si usa la stessa sintassi di una calcolatrice scientifica

```
2^5 # Sintassi alternativa: 2**5
```

```
## [1] 32
```

- Per calcolare $\sqrt{2}$ e $\sin(\pi/4)$ si possono usare le funzioni

```
sqrt(2)
```

```
## [1] 1.414214
```

```
sin(pi / 4)
```

```
## [1] 0.7071068
```

Assegnazione di un valore

- E' possibile salvare un valore assegnandolo ad un oggetto tramite il simbolo <-

```
x <- sqrt(5) # Sintassi alternativa (sconsigliata): x = sqrt(5)
```

- **Nota**. **R** è *case sensitive*, pertanto l'oggetto `x` è diverso dall'oggetto `X`.
- Il valore contenuto in `x` può essere successivamente richiamato e salvato in un nuovo oggetto `y`

```
y <- x + pi # ovvero pi greco + radice quadrata di 5
y
```

```
## [1] 5.377661
```

- Per rimuovere un oggetto dalla memoria, si usa il comando `rm`

```
rm(x) # x non è più presente nel "workspace"
```

Pulizia del «workspace»

- È buona norma mantenere pulito il «workspace», ovvero l'ambiente di lavoro.
- Se un oggetto non è più necessario, è possibile eliminarlo tramite il comando `rm`.
- È possibile visualizzare la lista di oggetti salvati in memoria tramite il comando

```
ls() # Nel workspace è presente l'oggetto y
```

```
## [1] "y"
```

- Pertanto, per **eliminare tutti gli oggetti** salvati, si può usare

```
rm(list = ls())
```

Alcune funzioni matematiche

- Supponiamo che `x` sia un numero reale, ad esempio `x <- 1/2`.

```
exp(x) # Esponenziale e logaritmo naturale
log(x)

abs(x) # Valore assoluto
sign(x) # Funzione segno

sin(x) # Funzioni trigonometriche (seno, coseno, tangente)
cos(x)
tan(x)

asin(x) # Funzioni trigonometriche inverse
acos(x)
atan(x)
```

- Le funzioni di **R** si possono combinare tra di loro, ad esempio: `log(abs(x))`.

Ulteriori funzioni matematiche

- Supponiamo che `x` e `y` siano due numeri reali, ad esempio `x <- 1/2`, `y <- 1/3`.
- Inoltre, siano `n` e `k` due numeri naturali, ad esempio `n <- 5`, `k <- 2`.

```
factorial(n) # n!
choose(n, k) # Coefficiente binomiale
```

```
round(x, digits = 2) # Arrotonda x usando 2 cifre decimali
floor(x) # Arrotonda x all'intero più vicino, per difetto
ceiling(x) # Arrotonda x all'intero più vicino, per eccesso
```

- La funzione $\Gamma(x) = \int_0^\infty s^{x-1} e^{-s} ds$ si calcola in **R** come segue

```
gamma(x) # Funzione gamma
```

- La funzione $\mathcal{B}(x, y) = \int_0^1 s^{x-1}(1-s)^{y-1} ds$ si calcola in **R** come segue

```
beta(x, y) # Funzione beta
```

La documentazione ufficiale (help)

- La documentazione di **R** è la principale fonte di informazione di un utente **R**.
- Cosa fa una funzione? La risposta va sempre cercata nella **documentazione ufficiale** e non in queste slide, che ne sono solamente un riassunto.
- Il comando «`? funzione`» apre una finestra in cui vengono descritta nel dettaglio una `funzione`. Esempio:

```
? log # Documentazione della funzione log
```

- Durante l'esame è legittimo (anzi, è caldamente consigliato!) consultare la documentazione. Non sarà tuttavia concesso utilizzare le slide.

Simboli speciali

- Numeri molto grandi (e molti piccoli) in **R** vengono rappresentati come segue

```
10^15
```

```
## [1] 1e+15
```

```
10^(-15)
```

```
## [1] 1e-15
```

- Quando un numero è troppo grande **R** riporta `Inf`, ovvero *infinito*.

```
10^1000
```

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

- Il simbolo `NaN` significa «Not a Number».

```
log(-1) # Questo comando genera un avviso
```

```
## Warning in log(-1): NaNs produced
```

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

Errori di approssimazione numerica

- È ben noto che $\sin(\pi) = 0$. Tuttavia, in **R** si ottiene un numero molto vicino a 0, ma strettamente positivo

```
sin(pi)
```

```
## [1] 1.224647e-16
```

- **R** è uno strumento di calcolo numerico e pertanto sono sempre presenti **errori di approssimazione numerica**.
- Fortunatamente, nella maggior parte dei casi pratici la differenza tra 0 e 10^{-16} è irrilevante. In altre situazioni, errori di approssimazione numerica possono portare a conclusioni completamente fuorvianti.
- Occorre quindi fare attenzione e valutare caso per caso.
- Ad ogni modo, l'approssimazione numerica potrebbe anche migliorare. Ad esempio:

```
cos(pi)
```

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

Operazioni logiche

```
x <- 5
x < 0 # Il valore di x è minore di 0?
```

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

```
a <- (x == -3) # Il valore di x è uguale a -3?
a
```

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

- Il valore di a è un indicatore **binario** o **booleano**, ovvero può essere vero (`TRUE`) oppure falso (`FALSE`).
- Altre funzioni logiche disponibili sono:

```
x >= y # x è maggiore o uguale a y? (Si usa "<=" per minore uguale)
x != y # x è diverso da y?
a & b # a AND b. I valori booleani a e b sono entrambi veri?
a | b # a OR b. Almeno uno tra a ed b è vero?
```

Vettori e successioni

- Un «vettore» in **R** si può scrivere come segue

```
x <- c(4, 2, 2, 8, 10)
x
```

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

- **Nota**. Con il termine generico «vettore» in **R** non si fa riferimento alla nozione dell'algebra lineare ma semplicemente ad una stringa di valori consecutivi. Infatti il seguente oggetto è un «vettore»

```
x <- c("A", "B", 2, 8, 10)
x
```

```
## [1] "A"  "B"  "2"  "8"  "10"
```

```
x <- 5:10 # Equivalente a: x <- c(5, 6, 7, 8, 9, 10)
x
```

```
## [1]  5  6  7  8  9 10
```

Vettori e successioni

- E' possibile anche creare successioni in ordine decrescente

```
x <- 10:-10
x
```

```
##  [1]  10   9   8   7   6   5   4   3   2   1   0  -1  -2  -3
## [15]  -4  -5  -6  -7  -8  -9 -10
```

- Per creare successioni di numeri non interi si usa il comando

```
x <- seq(from = 0, to = 1, by = 0.1)
x
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

- Per creare un vettore di valori ripetuti si usa il comando

```
x <- rep(10, 7) # Vettore in cui il numero 10 è ripetuto 7 volte
x
```

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

Operazioni sui vettori

- La maggior parte delle funzioni matematiche di **R** sono **vettorizzate**, ovvero agiscono su tutti gli elementi di un vettore.

Esempi

```
exp(1:6) + (1:6) / 2 + 1
```

```
## [1]   4.218282   9.389056  22.585537  57.598150 151.913159
## [6] 407.428793
```

```
x <- c(10, 10^2, 10^3, 10^4, 10^5, 10^6)
log(x, base = 10)
```

```
## [1] 1 2 3 4 5 6
```

```
1:8 > 4
```

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

Operazioni sui vettori II

```
x <- c(2, 3, 1, 3, 10, 5)
length(x) # Lunghezza del vettore
```

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

```
sum(x) # Somma degli elementi del vettore
```

```
## [1] 24
```

```
cumsum(x) # Somme cumulate
```

```
## [1]  2  5  6  9 19 24
```

```
prod(x) # Prodotto degli elementi del vettore
```

```
## [1] 900
```

```
cumprod(x) # Prodotti cumulati
```

```
## [1]   2   6   6  18 180 900
```

Operazioni sui vettori III

```
x <- c(2, 3, 1, 3, 10, 5)
sort(x, decreasing = FALSE) # Vettore ordinato in ordine crescente
```

```
## [1]  1  2  3  3  5 10
```

```
min(x) # Valore minimo
```

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

```
which.min(x) # Posizione del valore corrispondente al minimo
```

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

- I risultati dei seguenti comandi sono facilmente intuibili e non riportati.

```
max(x) # Valore massimo
which.max(x) # Posizione del valore corrispondente al massimo
range(x) # Equivalente a: c(min(x), max(x))
```

Manipolazione dei vettori

```
# Concatenazione di vettori
x <- c(rep(pi, 2), sqrt(2), c(10, 7))

x[3] # Estrae il terzo elemento dal vettore x, ovvero sqrt(2)

## [1] 1.414214

x[c(1, 3, 5)] # Estrae il primo, il terzo ed il quinto elemento

## [1] 3.141593 1.414214 7.000000

x[-c(1, 3, 5)] # Elimina il primo, il terzo ed il quinto elemento

## [1]  3.141593 10.000000

x[x > 3.5] # Estrae gli elementi maggiori di 3.5

## [1] 10  7
```

Avvertenze sul calcolo vettoriale in **R**

- Cosa succede quando vengono sommati due vettori di dimensioni diverse? La maggior parte dei linguaggi di programmazione restituisce un errore. **R** invece esegue ugualmente l'operazione «allungando» il vettore più corto.

```
x <- 1:5
y <- 1:6
x + y # Equivalente a: c(x, x[1]) + y
```

```
## Warning in x + y: longer object length is not a multiple of
## shorter object length
```

```
## [1]  2  4  6  8 10  7
```

- Attenzione invece al contesto seguente. Nessun avviso da parte di **R**.

```
x <- 1:3
y <- 1:6
x + y # Equivalente a: c(x, x) + y
```

```
## [1] 2 4 6 5 7 9
```

Matrici

- Una matrice $\boldsymbol{A}$ è una collezione di elementi $(a)_{ij}$ per $i = 1, \ldots, n$ e $j = 1, \ldots, m$.
- Per esempio, la matrice quadrata di dimensione 2×2

$$\boldsymbol{A} = \begin{pmatrix} 5 & 2 \\ 1 & 4 \end{pmatrix},$$

si può definire in **R** come segue:

```
A <- matrix(c(5, 1, 2, 4), nrow = 2, ncol = 2)
A
```

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

```
# Definizione equivalente
A <- matrix(c(5, 2, 1, 4), nrow = 2, ncol = 2, byrow = TRUE)
```

Manipolazione di matrici

```
dim(A) # Restituisce la dimensione della matrice
```

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

```
a <- c(A) # Converte la matrice in un vettore
a
```

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

Estrazione di elementi

```
A[1, 2] # Estrazione di elemento in posizione (1,2)
A[, 2] # Estrazione seconda colonna
A[1, ] # Estrazione prima riga
```

- Per operazioni più complesse si procede come per i vettori, con le dovute ma ovvie modifiche.

Manipolazione di matrici II

```
diag(A) # Restituisce la diagonale della matrice
```

```
## [1] 5 4
```

```
t(A) # Calcola la matrice trasposta A'
```

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

```
sum(A) # Somma di tutti gli elementi di A
```

```
## [1] 12
```

- Come per i vettori, le operazioni elementari (somma, prodotto, `log`, `exp`, etc.) vengono eseguite **elemento per elemento**.

Manipolazione di matrici III

- Siano $\boldsymbol{A}$ e $\boldsymbol{B}$ due matrici aventi lo stesso numero di colonne e definiamo

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{B} \end{pmatrix}.$$

- In **R** questa operazione si esegue tramite il comando:

```
C <- rbind(A, B)
```

- Siano $\boldsymbol{A}$ e $\boldsymbol{B}$ due matrici aventi lo stesso numero di righe e definiamo

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \end{pmatrix}.$$

- In **R** questa operazione si esegue tramite il comando:

```
C <- cbind(A, B)
```

Vettori riga e vettori colonna

- Gli oggetti `matrix` consentono di definire **vettori riga** e **vettori colonna**

```
x_col <- matrix(c(1, 10, 3, 5), ncol = 1)
x_col
```

```
##      [,1]
## [1,]    1
## [2,]   10
## [3,]    3
## [4,]    5
```

```
x_row <- matrix(c(1, 10, 3, 5), nrow = 1)
x_row
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   10    3    5
```

Vettori riga e vettori colonna

- Nella maggior parte dei casi, l'oggetto `x_row` è intercambiabile col vettore `x`.

```
x_row <- matrix(c(1, 10, 3, 5), nrow = 1)
x <- c(1, 10, 3, 5) # Simile, ma non identico, a x_row
```

- Ad esempio, le funzioni `sum(x_row)` e `sum(x)` forniscono lo stesso risultato. Ci sono tuttavia alcune lievi distinzioni. Ad esempio:

```
dim(x_row)
```

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

```
dim(x)
```

```
## NULL
```

- Il simbolo `NULL` significa «non definito», perché non esiste la nozione di «dimensione» per un generico vettore **R**.

Il prodotto incrociato

Siano $\boldsymbol{x}$ e $\boldsymbol{y}$ due vettori colonna in $\mathbb{R}^p$. Allora, il loro prodotto incrociato è pari a

$$\boldsymbol{x}^\intercal \boldsymbol{y} = \sum_{i=1}^{p} x_i y_i.$$

```
x <- matrix(c(-4, 2, 6, 10, 22), ncol = 1)
y <- matrix(c(3, 2, 2, 7, 9), ncol = 1)
crossprod(x, y) # Equivalente a: sum(x * y)
```

```
##      [,1]
## [1,]  272
```

- Il comando `crossprod` funziona correttamente anche con «vettori» **R**.
- Inoltre, il comando `crossprod` può essere usato anche per calcolare $\boldsymbol{A}^\intercal \boldsymbol{B}$, dove $\boldsymbol{A}$ e $\boldsymbol{B}$ sono due matrici di dimensioni compatibili.

Operazioni di algebra lineare

- In algebra lineare il prodotto tra matrici compatibili ***AB*** è chiamato prodotto righe per colonne. In **R** si usa il comando %*%.

```
A <- rbind(c(1, 2, 3), c(4, 9, 2), c(2, 2, 2))
B <- rbind(c(5, 2, 5), c(3, 3, 7), c(-2, -8, 10))
A %*% B # Prodotto righe per colonne AB
```

```
##      [,1] [,2] [,3]
## [1,]    5  -16   49
## [2,]   43   19  103
## [3,]   12   -6   44
```

- **Nota**. Il comando `A * B` indica il prodotto elemento per elemento e non il prodotto righe per colonne!
- Se le matrici non sono compatibili **R** produce un errore (provateci!)

Operazioni di algebra lineare II

- Sia $\boldsymbol{A}$ una matrice quadrata $n \times n$ a valori reali. La sua inversa $\boldsymbol{A}^{-1}$, quando esiste, è l'unica matrice tale per cui $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = I_n$.

```
A <- rbind(c(1, 2, 3), c(4, 9, 2), c(2, 2, 2))
A1 <- solve(A) # Matrice inversa di A
A1
```

```
##            [,1]        [,2]        [,3]
## [1,] -0.5833333 -0.08333333  0.95833333
## [2,]  0.1666667  0.16666667 -0.41666667
## [3,]  0.4166667 -0.08333333 -0.04166667
```

```
round(A %*% A1, digits = 5) # Operazione di controllo
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Operazioni di algebra lineare III

```
det(A) # Calcola il determinante della matrice A
```

```
## [1] -24
```

```
# Esempio di matrice NON invertibile
A <- rbind(c(1, 2, 3), c(2, 4, 6), c(2, 2, 2))
det(A) # Deteminante pari a 0, solve(A) produce un errore
```

```
## [1] 0
```

- Ci sono numerose funzioni per la decomposizione di matrici, il cui output è a volte una **lista** (vedi prossima slide).

```
A <- matrix(c(4, 1, 1, 8), ncol = 2)
chol(A) # Decomposizione di Cholesky
qr(A) # Decomposizione QR
eigen(A) # Decomposizione spettrale
```

Liste

- Una lista è una collezione di oggetti (numeri, vettori, matrici, altro).
- Per salvare o per estrarre un oggetto da una lista si usa il simbolo $.

```
# Creazione di una lista
new_list <- list(
  A = matrix(c(4, 1, 1, 8), ncol = 2),
  x = c(1, 2, 6, 6, 9)
)
new_list
```

```
## $A
##      [,1] [,2]
## [1,]    4    1
## [2,]    1    8
##
## $x
## [1] 1 2 6 6 9
```

Liste II

```
Spec_A <- eigen(A) # Decomposizione spettrale
Spec_A

## eigen() decomposition
## $values
## [1] 8.236068 3.763932
##
## $vectors
##            [,1]       [,2]
## [1,] 0.2297529 -0.9732490
## [2,] 0.9732490  0.2297529

Spec_A$values # Estrazione degli autovalori

## [1] 8.236068 3.763932
```

Programmazione in **R**

- **R** è anche e soprattutto un linguaggio di programmazione. È possibile quindi definire nuove funzioni.

```
cube <- function(x) {
  out <- x^3
  out
}
```

- L'ultimo oggetto (in questo caso chiamato `out`) viene restituito come risultato.

```
cube(4)
```

```
## [1] 64
```

Funzioni

- È possibile definire funzioni con molteplici argomenti, scegliendo eventuali valori di predefiniti.

```
power <- function(x, p = 2) {
  out <- x^p
  out
}

power(x = 4)

## [1] 16

power(x = 4, p = 3)

## [1] 64
```

If e else

```
condizione <- pi^2 < 10 # Valore booleano

if (condizione) {
  print("La condizione è vera")
  # Alcuni comandi da eseguire
  # ...
} else {
  print("La condizione è falsa")
  # Altri comandi da eseguire
  # ...
}

## [1] "La condizione è vera"
```

- **Nota**. Non è obbligatorio usare la funzione `else` dopo una condizione `if`.

If e else: esempio

```
square_root <- function(x) {
  if (x < 0) {
    print("Il valore di x deve essere positivo")
    out <- NaN
  } else {
    out <- sqrt(x)
  }
  out
}
square_root(-2)
```

```
## [1] "Il valore di x deve essere positivo"
```

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

```
square_root(36)
```

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

Cicli

- La funzione `while` esegue ripetutamente le operazioni all'interno delle parentesi fino a quando la condizione non è soddisfatta.

```
i <- 5
while (i <= 25) {
  print(i) # Mostra a schermo il valore di i
  i <- i + 5

  # Altre operazioni da eseguire
  # ...
}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 25
```

Cicli

- Attenzione a non creare delle ripetizioni senza fine!
- Il codice seguente richiede l'interruzione forzata della sessione di **R** oppure, nella peggiore delle ipotesi, il riavvio del computer.

```
# NON ESEGUIRE IL SEGUENTE CODICE!
#
# La condizione i <= 25 è sempre vera
# perché i non viene aggiornato
i <- 5
while (i <= 25) {
  print(i)

  # Altre operazioni da eseguire
  # ...
}
```

Cicli `for`

- In alternativa ai cicli `while`, si può usare la sintassi più esplicita dei cicli `for`.

```
values <- seq(from = 5, to = 25, by = 5)
for (i in values) {
  print(i) # Mostra a schermo il valore di i

  # Altre operazioni da eseguire
  # ...
}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 25
```

Cicli `for`: esempio

- Vogliamo calcolare gli elementi di una matrice quadrata $\boldsymbol{D}$, i cui elementi sono pari a

$$d_{ij} = (x_i - x_j)^2, \qquad i, j \in \{1, \ldots, n\},$$

  dove $\boldsymbol{x} = (x_1, \ldots, x_n)^\intercal$ è un vettore.

- Per fare questo, useremo due cicli `for` annidati.

- Questa implementazione è inefficiente (è un po' lenta soprattutto se n è grande), ma corretta.

- Come regola indicativa, in **R** i cicli `for` andrebbero limitati o evitati, ove possibile, in quanto poco efficienti.

Cicli `for`: esempio

```
distances <- function(x) {
  n <- length(x) # Ottengo la lunghezza del vettore x
  D <- matrix(0, nrow = n, ncol = n) # Creazione matrice vuota

  for (i in 1:n) {
    for (j in 1:n) {
      D[i, j] <- (x[i] - x[j])^2
    }
  }
  D
}
x <- c(5, 2, 1, 24)
distances(x)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    9   16  361
## [2,]    9    0    1  484
## [3,]   16    1    0  529
## [4,]  361  484  529    0
```