Introduzione a R

Salvatore Alaimo, MSc.

Email: alaimos@dmi.unict.it

Panoramica

- Introduzione
- Concetti di base
  - Costrutti fondamentali
  - Strutture dati
  - Grafici
  - L'ambiente RStudio
  - Statistica in R
- Concetti Avanzati
  - Modelli Statistici
  - Programmazione ad Oggetti
  - Programmazione Parallela e Funzionale
  - Grafica Avanzata
    - ggplot2

Introduzione

Parte I

Cosa è R?

- R è una *suite integrata di strumenti software per la manipolazione dei dati, il calcolo e la visualizzazione grafica*.
- Tra le altre cose, R presenta:
  - una gestione efficace dei dati;
  - un insieme di operatori per i calcoli su array, in particolare matrici;
  - una grande raccolta di strumenti intermedi per l'analisi dei dati;
  - servizi grafici per l'analisi e visualizzazione dei dati;
  - un linguaggio di programmazione semplice ed efficace.
- R è molto più di un veicolo per lo sviluppo di nuovi metodi di analisi interattiva dei dati. Si è sviluppato rapidamente, ed è stato esteso con una *vasta collezione di pacchetti per ogni scopo*.
- Tuttavia, la maggior parte dei programmi scritti in R sono essenzialmente scritti allo scopo di analizzare dati.

Cosa è R?

- **R è un software GNU (gratuito sotto licenza GPL).**
- È disponibile on line: http://www.r-project.org
- Le principali repository di pacchetti sono:
  - **CRAN**: http://cran.r-project.org/
    - Circa 4398 pacchetti
  - **Bioconductor**: http://www.bioconductor.org/
    - Circa 600 pacchetti
- Ogni funzione e pacchetto è ampiamente documentato con tutte le informazioni ed esempi che ne facilitano l'utilizzo e l'apprendimento.
- Tutto ciò che serve per apprendere R è disponibile online gratuitamente (anche in italiano):
  - http://cran.r-project.org/manuals.html
- Un IDE per R:
  - http://www.rstudio.com/

I tipi di dati

- R è un linguaggio **object-oriented**.
- Ogni variabile è un **oggetto** a cui è sempre associata una **classe** di appartenenza che ne definisce le proprietà.
- L'unica eccezione sono i **vettori** (gli oggetti più semplici del linguaggio) la cui classe *corrisponde al tipo di dati contenuto in esso*.
- I tipi di dati in R sono in tutto **7** (più uno):
  - *logical*
  - *numeric*
    - *integer*
    - *double o real*
    - *complex*
  - *character*
  - *raw*
  - *function*

I tipi di dati

- Ci sono 5 strutture dati fondamentali (classi predefinite)
  - **vettori** (classe **vector**)
  - **matrici** (classe **matrix**)
  - **array** (classe **array**)
  - **liste** (classe **list**)
  - **Data frame** (classe **data.frame**)
- Come si stabilisce la classe di appartenenza di un oggetto?
  - **class**(), oppure **is.list**(), **is.function**(), **is.logical**(), e così via…
  - **class**() restituisce il nome della classe di appartenenza, mentre…
  - le altre restituisco **TRUE** o **FALSE** se l'oggetto appartiene o meno alla classe controllata dalla particolare funzione.

Eseguire istruzioni

- Le istruzioni possono essere eseguite
  - …o dalla linea di comando digitando al prompt ogni comando (premendo INVIO per eseguirlo)
  - …o caricando uno script .R (file di testo con la lista dei comandi)
- Per caricare uno script:

```
> source('mioscript.R')
>
```

- Per caricare una estensione sotto forma di pacchetto, si può usare il comando **library**.

```
> library(splines)
>
```

- Più istruzioni in una stessa riga devono essere separate da un «**;**».

Operatori Principali

- L'operatore di assegnamento è la freccia
  - **<-** oppure **->**

```
> X <- 5
> X
[1] 5
> 6 -> Y
> Y
[1] 6
>
```

- Non è necessario dichiarare le variabili. Se al primo utilizzo una variabile non esiste un errore sarà restituito.
- **ATTENZIONE: R è case-sensitive!**

```
> X
[1] 5
> x
Errore: oggetto "x" non trovato
>
```

Operatori Principali

- Caratteri speciali:
  - Il nome di una variabile non può contenere i caratteri «**!**», «**"**», «**$**», e «**%**»
  - Il punto «**.**» non è un carattere speciale e l'utilizzo è incoraggiato quando una variabile ha un nome complesso (**number.of.elements** è preferibile a **numberOfElements**)
  - Il carattere «**#**» è il simbolo del commento.
- Le principali parole riservate:
  - **FALSE, TRUE, Inf, NaN, NA, NULL, if, else, for, while, repeat, break, function, in, next, …**
- Particolare attenzione deve essere posta su due «valori» speciali:
  - **NA**: rappresenta un valore ignoto (non disponibile)
  - **NaN**: rappresenta il risultato indeterminato di una computazione numerica (es. **0/0** o **Inf/0**).
- La funzione **is.nan**() restituisce **TRUE** se e solo se il suo argomento è un **NaN**.
- La funzione **is.na**() restituisce **TRUE** se il suo argomento è **NA** o **NaN**.
  - **ATTENZIONE: NA != NA e NaN != NaN**
- Molte funzioni hanno un parametro opzionale **na.rm** che esclude dalla computazione tutti i valori **NA** e **NaN** evitando errori di calcolo.

Operatori Principali

- Operatori aritmetici:
  - + (somma);
  - - (sottrazione);
  - * (moltiplicazione);
  - / (divisione);
  - ^ (potenza);
  - %/% (divisione intera tra due numeri);
  - %% (resto della divisione intera tra due numeri).
- Operatori logici:
  - & (AND logico)
  - | (OR logico)
  - ! (NOT logico)
- Operatori relazionali:
  - == (uguale), != (diverso), < (minore), > (maggiore), <= (minore uguale), >= (maggiore uguale)
  - L'applicazione degli operatori relazionali ha come risultato **TRUE** o **FALSE**.
  - Le espressioni logiche **TRUE** o **FALSE** corrispondono rispettivamente a **1** o **0**, quindi possono essere utilizzate nelle operazioni aritmetiche.

Note

- Alcune funzioni matematiche:
  - **sum(); prod(); sqrt(); exp(); log(); sin(); cos(); max(); min(); round(); sign();**
  - **ceiling(); floor(); mean(); median(); var(); sd(); cov();**
- Il workspace rappresenta il contenuto della memoria di lavoro.
- Per visualizzare gli oggetti definiti nel workspace si può usare il comando **ls()**.
- Tramite il comando **rm**() è possibile rimuovere un oggetto.

```
> ls()
[1] "altra.variabile" "x"
> rm("altra.variabile")
> ls()
[1] "x"
>
```

- Per qualsiasi informazione su una funzione o sul suo utilizzo sono a disposizione i seguenti comandi:
  - **help(nome_funzione)**
  - **help.search("una stringa")**

Costrutti Fondamentali

Parte 2

Informazioni di base

- Le istruzioni devono essere scritte secondo l'ordine in cui verranno eseguite
- Ogni istruzione deve essere separata dalla successiva. Questo lo si può fare in due modi: **usando il «;»** o **andando a capo**

```
t<-1; sum<-0
#Or
product<-1
cat(t)
```

- Attraverso parentesi graffe è possibile isolare blocchi di istruzioni:

```
repeat
{sum<-sum+1
  product<-product*t
  t<-t+1
  if(t==8) break()
}
```

IF-THEN-ELSE

- **if (***espressione-logica***)** *istruzione o blocco* **else** *istruzione o blocco*

```
if (x == 5) print("X = 5") else print("X diverso da 5")

if ((x %% 2) == 0) {
    print("x è pari")
} else {
    print("x è dispari")
}

if ((x %% 2) == 0) {
    print("x è divisibile per 2")
} else if ((x %% 3) == 0) {
    print("x è divisibile per 3")
} else {
    print("x non è divisibile ne per 2 ne per 3")
}

if (!is.numeric(x)) {
    print("x non è un numero")
}
```

CICLO FOR

- **for (***nome.variabile* **in** *vector***)** *istruzione o blocco*

```
sum  <- 0
prod <- 1
for (t in 1:10)
{
    sum  <- sum + t
    prod <- prod * t
}

sum  <- 0
prod <- 1
for (t in seq(from=1, to=10, by=2)) {
    sum  <- sum + t
    prod <- prod * t
}
```

- Il comando **seq(from=1, to=10, by=2)** genera un vettore contenente gli elementi compresi tra «**from**» e «**to**». La distanza tra due elementi consecutiva è specificata dal parametro opzionale «**by**».
- Una forma più compatta del comando è «*from***:***to*»

CICLO WHILE

- **while (***espressione-logica***)** *istruzione o blocco*

```
sum  <- 0
prod <- 0
t    <- 1;
while (t <= 10) {
    sum  <- sum + t
    prod <- prod + t
    t    <- t + 1
}
```

```
sum  <- 0
prod <- 0
t    <- 1;
while (t <= 10) {
    sum  <- sum + t
    prod <- prod + t
    t    <- t + 2
}
```

CICLO REPEAT

- **repeat** *blocco di istruzioni*

```
sum  <- 0
prod <- 1
t    <- 1
repeat {
     sum  <- sum + t
     prod <- prod * t
     t    <- t + 1
     if (t == 10) break()
}
```

SWITCH

- **switch(***espressione***,** *lista di valori***)**

```
num <- x%%2 + 1 #assegna 1 se x è pari, 2 altrimenti
answer <- switch(num, "even", "odd");
cat(answer,"\n")

result = switch(answer, even = {
    # ... altri comandi qui ...
    (x / 2) + 3
}, odd = {
    # ... altri comandi qui ...
    ((x - 1) / 2) + 3
})
```

Funzioni

- Le funzioni contengono argomenti in entrata e in uscita e permettono di utilizzare variabili interne che non interagiscono con le variabili del workspace.
- Struttura generale di una funzione:

```
Function.Name <- function (argomenti.in.input)
{
    # corpo della funzione
    output
}
```

- L'ultima espressione valutata all'interno della funzione costituisce anche il suo output.
- All'interno di una funzione si possono richiamare altre funzioni definite dall'utente purché siano state definite prima della chiamata.
- È anche possibile definire funzioni ricorsive.

Funzioni

- Per definire un argomento opzionale basta indicare dopo il nome il valore predefinito che dovrà assumere se non specificato diversamente.
- Il valore di un argomento opzionale può a sua volta essere una espressione che sarà valutata prima dell'ingresso nel corpo della funzione.

```
seq <- function(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
                length.out = NULL, along.with = NULL, ...) {

    # ... CORPO DEL METODO ...

}
```

- Uno speciale tipo di argomento è «**...**». Esso permette di specificare un elenco non definito di argomenti che potrà essere passato ad altre funzioni richiamate nel corpo del metodo.

Funzioni

```
Valuta <- function (valX, div=2) {
    if ((valX %% div) == 0) {
        t <- 1
    } else {
        t <- 2
    }
    t
}

Main <- function (input, ...) {
    t <- Valuta(input, ...)
    if (t == 1) {
        cat("numero divisibile\n")
    } else {
        cat("numero non divisibile\n")
    }
}
```

```
Main(2)

Main(2, 3)

Main(2, div=3)

Main(input=2, div=3)
```

Strutture dati

Parte 3

Informazioni di base

- **5 strutture dati predefinite:**
  - **vector**
  - **matrix**
  - **array**
  - **list**
  - **data.frame**
- Le funzioni **is.vector**(), **is.matrix**(), **is.array**(), **is.list**() e **is.data.frame**() ritornano **TRUE** se l'argomento è del tipo specificato altrimenti **FALSE**.
- Le funzioni **as.tipo**() permettono il *cast* da un tipo ad un altro.
  - In ambiente R, questa operazione è chiamata «**coercion**».

Vettori

- Un vettore si può definire come una *sequenza ordinata di oggetti dello stesso tipo* (es. number, character, logical).
- Il modo più comune per la costruzione di un vettore è l'utilizzo della funzione **c()**:

```
> vect<-c(1,2,3,4.5,-6) #costruzione di un vettore di cinque
#elementi
> vect[2]
[1] 2
> vect
[1]  1.0  2.0  3.0  4.5 -6.0
> vect[c(1,3,4)] #visualizza gli elementi delle posizioni 1,3,4
[1] 1.0 3.0 4.5
> vect[-c(1,3,4)] #visualizza gli elementi delle posizioni 2,5
[1]  2 -6
> vect[vect<0]+1
[1] -5
```

- Il segno «**-**» che si trova prima del vettore di indici indica gli *elementi che bisogna escludere* durante la selezione.
- **Qualsiasi tipologia di selezione (o qualunque tipo di operazione) che si fa su un vettore è essa stessa un vettore!**

Vettori

- Le operazioni sui vettori (cosi come sulle matrici e sugli array) sono eseguite su ogni elemento contenuto nel vettore (o matrice o array):

```
> vect[vect>0]
[1] 1.0 2.0 3.0 4.5
> vect[vect>0]+c(2,2,2,2)
[1] 3.0 4.0 5.0 6.5
```

- l'operazione deve essere lecita!

```
> vect[vect>0]+c(2,3,4)
[1] 3.0 5.0 7.0 6.5
Warning message:
In vect[vect > 0] + c(2, 3, 4) :
  longer object length is not a multiple of shorter object length
```

- Il risultato viene restituito ugualmente, in quanto il vettore più piccolo viene ripetuto fino al raggiungimento della stessa lunghezza.

Vettori

- **rep**() e **seq**() sono due funzioni, alternative, per la costruzione di un vettore:

```
> rep(c(1,2), times=3) #ripeti il vettore (1,2) 3 volte
[1] 1 2 1 2 1 2
> rep(c(1,2), c(2,3)) #ripeti il vettore "1" 2 volte e "2" 3 volte
[1] 1 1 2 2 2
> seq(3,11,by=2)
[1]  3  5  7  9 11
> seq(3,11,length.out=6)
[1]  3.0  4.6  6.2  7.8  9.4 11.0
```

- **range**() e **length**(), la prima da il range dei valori numerici contenuti nel vettore mentre la seconda restituisce la dimensione dell'oggetto che viene passato come parametro:

```
> range(vect)
[1] -6.0  4.5
> length(vect)
[1] 5
> length(vect[vect>0])
[1] 4
> length(vect[vect>0][4])
[1] 1
```

Vettori

- La funzione **which**() mostra la posizione degli elementi del vettore (o di un qualsiasi altro oggetto di R) che soddisfano una particolare condizione:

```
> vect
[1]  1.0  2.0  3.0  4.5 -6.0
> which(vect<0)
[1] 5
> which.max(vect)
[1] 4
> which.min(vect)
[1] 5
```

- Le funzioni **which.max**() e **which.min**() restituisco, rispettivamente, la posizione dell'elemento massimo e quello dell'elemento minimo all'interno di un qualsiasi oggetto di R.

Sorting

- Esistono due modi per ordirnare un vettore, uno attraverso la funzione **sort**() e tramite la funzione **order**():

```
> vect
[1]  1.0  2.0  3.0  4.5 -6.0
> sort(vect) #ordina in modo crescente
[1] -6.0  1.0  2.0  3.0  4.5
> sort(vect, decreasing=TRUE) #ordina in modo decrescente
[1]  4.5  3.0  2.0  1.0 -6.0
> vect[order(vect)]
[1] -6.0  1.0  2.0  3.0  4.5
> vect[order(vect,decreasing=TRUE)]
[1]  4.5  3.0  2.0  1.0 -6.0
> vectOrd<-c(6,5,4,3,2)
> vect[order(vectOrd)] #l'ordinamento di "vect" avviene rispetto a "vectOrd"
[1] -6.0  4.5  3.0  2.0  1.0
```

- La funzione **order**(), restituendo solo la *sequenza di indici ordinata*, da la possibilità di ordinare un vettore rispetto ad un altro.

Matrici

- Una **matrice** è una *tabella ordinata di elementi dello stesso tipo*. Ciascun elemento è univocamente localizzato tramite *una coppia di numeri interi*: l'indice di riga e quello di colonna.
- La funzione **matrix**() permette la creazione di una matrice:

```
> mtx<-matrix(1:25, ncol=5) #viene costruita una matrice con un numero di 5 cols
> mtx
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> mtx[,3] #viene mostrata la colonna 3
[1] 11 12 13 14 15
> mtx[6,5]
Error: subscript out of bounds
> mtx[,-c(3,4,5)] #vengono selezionate le colonne 1 e 2
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

- Gli elementi sono disposti *dall'alto verso il basso e da sinistra verso destra* (disposizione per colonne), a meno di non specificare il parametro «**byrow=TRUE**».

Matrici

- È possibile eseguire un cast tra matrice e vettore con la funzione **as.matrix**(). Tramite la funzione **as.vector**() avviene il contrario.

```
> vect
[1]  1.0  2.0  3.0  4.5 -6.0
> as.matrix(vect)
     [,1]
[1,]  1.0
[2,]  2.0
[3,]  3.0
[4,]  4.5
[5,] -6.0
> as.vector(mtx[,-c(3,4,5)])
 [1]  1  2  3  4  5  6  7  8  9 10
```

Matrici

- Esistono altre tre funzioni che possono tornare utili per la costruzione di matrici : **cbind**(), **rbind**() e **diag**().

```
> rbind(c(3,4),5:6,c(-5.5,5.5)) #disposizione per righe
     [,1] [,2]
[1,]  3.0  4.0
[2,]  5.0  6.0
[3,] -5.5  5.5
> cbind(c(3,4),5:6,c(-5.5,5.5)) #disposizione per colonne
     [,1] [,2] [,3]
[1,]    3    5 -5.5
[2,]    4    6  5.5
> diag(mtx)
[1]  1  7 13 19 25
> mtxDiag<-diag(2:5)
> mtxDiag
     [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    3    0    0
[3,]    0    0    4    0
[4,]    0    0    0    5
```

Matrici

- Costruire matrici vuote:

```
> mtxEmpty<-matrix(,nrow=4,ncol=3) #creazione di una matrice vuota 4x3
> mtxEmpty
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
[3,]   NA   NA   NA
[4,]   NA   NA   NA
> mtxEmpty[]<-12:23
> mtxEmpty
     [,1] [,2] [,3]
[1,]   12   16   20
[2,]   13   17   21
[3,]   14   18   22
[4,]   15   19   23
```

Matrici

- Come nel caso dei vettori, si possono utilizzare le funzioni **range**() e **length**().
- Per sapere la dimensione della matrice basta eseguire la funzione **dim**(). Per semplicità è possibile utilizzare le funzioni **nrow**() e **ncol**() per restituire rispettivamente il numero di righe e colonne.
- La funzione **apply**() da la possibilità di utilizzare le funzioni matematiche su righe e colonne.

```
> range(mtx)
[1]  1 25
> length(mtx)
[1] 25
> dim(mtx)
[1] 5 5
>
```

```
> mtx
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> apply(mtx,2,max)
[1]  5 10 15 20 25
> apply(mtx,1,max)
[1] 21 22 23 24 25
```

Matrici

- La funzione **t**() permette di calcolare la trasposta di una data matrice.
- Per calcolare l'inversa di una matrice bisogna utilizzare la funzione **solve**() (questa funzione permette, inoltre, di risolvere sistemi di equazioni lineari).

```
> mtxEmpty
     [,1] [,2] [,3]
[1,]   12   16   20
[2,]   13   17   21
[3,]   14   18   22
[4,]   15   19   23
> t(mtxEmpty)
     [,1] [,2] [,3] [,4]
[1,]   12   13   14   15
[2,]   16   17   18   19
[3,]   20   21   22   23
```

```
> mtxDiag
     [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    3    0    0
[3,]    0    0    4    0
[4,]    0    0    0    5
> invMtxDiag<-solve(mtxDiag)
> invMtxDiag
     [,1]      [,2] [,3] [,4]
[1,]  0.5 0.0000000 0.00  0.0
[2,]  0.0 0.3333333 0.00  0.0
[3,]  0.0 0.0000000 0.25  0.0
[4,]  0.0 0.0000000 0.00  0.2
```

Matrici

- Il prodotto matriciale (riga × colonna) si effettua attraverso l'operatore «**%*%**»:

```
> invMtxDiag%*%mtxDiag
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

- mentre l'operatore «*****» esegue semplicemente il prodotto elemento per elemento:

```
> mtx*vect
     [,1]  [,2] [,3]   [,4] [,5]
[1,]    1   6.0   11   16.0   21
[2,]    4  14.0   24   34.0   44
[3,]    9  24.0   39   54.0   69
[4,]   18  40.5   63   85.5  108
[5,]  -30 -60.0  -90 -120.0 -150
>
```

- Operatore «**%in%**»: consente di vedere se un'occorrenza dei valori dell'operando di sinistra si trova nei valori dell'operando di destra. Esso ritorna un vettore logico.

Array

- Gli **array** costituiscono *un'estensione delle matrici*, allo stesso modo le matrici sono considerate un'estensione dei vettori .
- Un elemento di un array è rappresentato da un *vettore di indici*.
- Per costruire un array si utilizza la funzione **array**():

```
> arrayEs<-array(24:49, dim=c(4,2,3)) #creazione di un "array"
> dim(arrayEs)
[1] 4 2 3
> arrayEs[,,1]
     [,1] [,2]
[1,]   24   28
[2,]   25   29
[3,]   26   30
[4,]   27   31
> arrayEs[2,,]
     [,1] [,2] [,3]
[1,]   25   33   41
[2,]   29   37   45
> arrayEs[2,2,2]
[1] 37
```

Indicizzazione tramite nomi

- Gli elementi in un vettore, matrice, o array possono essere **indicizzati** sia numericamente che tramite **nomi**.
- Per assegnare nomi si possono usare le funzioni **names(vettore)** (per i vettori), **rownames(matrix)** e **colnames(matrix)** per le matrici, **dimnames(object)** per gli altri oggetti.
- La funzione **unname(object)** rimuove tutti i nomi assegnati ad un oggetto
- È possibile usare il parametro «**dimnames**» dei costruttori matrix e array per assegnare direttamente i nomi.

```
> vect <- 1:4
> names(vect) <- c("elem_a", "elem_b", "elem_c", "elem_d")
> vect
elem_a elem_b elem_c elem_d
     1      2      3      4
> vect["elem_a"]
elem_a
     1
> vect[c("elem_a", "elem_d")]
elem_a elem_d
     1      4
> unname(vect)
[1] 1 2 3 4
>
```

```
> matr <- matrix(1:6, nrow=3, ncol=2, byrow=TRUE)
> rownames(matr) <- c("val_a", "val_b", "val_c")
> colnames(matr) <- c("exp_1", "exp_2")
> matr
      exp_1 exp_2
val_a     1     2
val_b     3     4
val_c     5     6
> matr[,"exp_1"]
val_a val_b val_c
    1     3     5
> matr[c("val_a", "val_b"),]
      exp_1 exp_2
val_a     1     2
val_b     3     4
>
```

Liste

- Una **lista** in R si può definire come *una raccolta di oggetti (anche di tipi diversi), comprese altre liste*.
- Per creare una lista si utilizza il comando **list**().

```
> lista<-list(rep(1,4),c(1,2,3,4),matrix(2:9, nrow=4),c("Piano","Forte"))
> length(lista) #visualizza la dimensione della lista
[1] 4
> lista[[3]]
     [,1] [,2]
[1,]    2    6
[2,]    3    7
[3,]    4    8
[4,]    5    9
```

Liste

- Il numero di oggetti che compongono una lista ne rappresenta la sua dimensione.
- Per accedere agli elementi di una lista:
  - «**[·]**»: restituisce una lista con gli oggetti selezionati da estrarre tramite il comando **unlist**()
  - «**[[·]]**»: restituisce direttamente un oggetto della lista
  - «lista**$**nome.elemento»: restituisce direttamente un oggetto della lista

```
> length(lista[[3]])
[1] 8
> lista[[3]][3,2]
[1] 8
> lista[[4]] #viene restituito un vettore
[1] "Piano" "Forte"
> lista[4] #viene restituito una lista
[[1]]
[1] "Piano" "Forte"
```

Liste

- È possibile assegnare nomi ad ogni singolo oggetto incluso nella lista durante la sua creazione, in particolare specificando il nome all'interno della funzione **list()** (ad es. ***list(first=rep(1,4)...)***).
- Un altro modo per nominare o rinominare una lista è quello di utilizzare la funzione **names()**.

```
> names(lista)
NULL
> names(lista)<-c("First","Second","Third","Fourth")
>
> #Estrazione del "primo" elemento
> lista$First #estrae come un vettore
[1] 1 1 1 1
> lista["First"] #estrae come una lista
$First
[1] 1 1 1 1
```

Data Frame

- Il **DataFrame** è l'oggetto più importante in R:
  - Simile ad un matrice, ma è usato per raffigurare dati di diversa entità.
  - Ogni riga corrisponde ad un osservazione, mentre ogni colonna è una variabile (*quantitativa* o *categoriale/factor*).
  - Un DataFrame è trattato come lista:
    - Ogni elemento rappresenta una variabile statistica
    - la funzione **length**() restituisce il numero di variabili
    - La funzione **names**() restituisce/imposta i nomi delle variabili
    - La funzione **row.names**() restituisce/imposta I nomi delle osservazioni.
- Per conoscere la dimensione dell'intero dataframe: **dim**()
- Il costruttore dell'oggetto è la funzione **data.frame**()
  - Tra i parametri si può specificare il nome di ogni singola riga del dataframe (**row.names**) e il nome delle osservazioni.

Data Frame

```
> #Viene creato un "dataframe" contenente una variabile 'quantitativa' e una
> #'qualitativa':
> dataFrame<-data.frame(a=0:3, sesso=c("F","M","M","F"))
> dataFrame
  a sesso
1 0     F
2 1     M
3 2     M
4 3     F
> dim(dataFrame) #dimensione del "dataframe" ('osservazioni' e 'variabili')
[1] 4 2
> length(dataFrame)
[1] 2
> dataFrame$altezza=c("172","180","165","163") #Viene aggiunta una nuova
#variabile contenente le altezze in centimetri di ogni singolo individuo.
> dataFrame
  a sesso altezza
1 0     F     172
2 1     M     180
3 2     M     165
4 3     F     163
>
```

Data Frame

```
> names(dataFrame)
[1] "a"       "sesso"   "altezza"
> row.names(dataFrame)
[1] "1" "2" "3" "4"
> row.names(dataFrame)<-c("patient1","patient2","patient3","patient4")
> row.names(dataFrame)
[1] "patient1" "patient2" "patient3" "patient4"
> dataFrame
         a sesso altezza
patient1 0     F     172
patient2 1     M     180
patient3 2     M     165
patient4 3     F     163
```

Data Frame

- Esistono sostanzialmente due diversi modi per selezionare dal dataframe una variabile o un'osservazione:

```
> dataFrame$sesso
[1] F M M F
Levels: F M
> dataFrame[2,3]
[1] "180"
> dataFrame[2,2]
[1] M
Levels: F M
> dataFrame[,2]
[1] F M M F
Levels: F M
> dataFrame$altezza
[1] "172" "180" "165" "163"
```

- L'uso delle parentesi quadre da la possibilità di selezionare le osservazioni che si desiderano. Osservate questi due esempi:

```
> #selezione dei valori della variabile "altezza" per le donne:
> dataFrame[dataFrame$sesso=="F", "altezza"]
[1] "172" "163"
> #selezione dei valori della variabile "sesso" con altezza<164
> dataFrame$sesso[dataFrame$altezza<=164]
[1] F
Levels: F M
```

Operazioni su Data Frame

- Sostanzialmente sono le stesse di quelle utilizzate su un matrice. Ovviamente, devono avere un senso!

```
> dataFrame$altezza
[1] "172" "180" "165" "163"
> mean(dataFrame$altezza)
[1] NA
Warning message:
In mean.default(dataFrame$altezza) :
  argument is not numeric or logical: returning NA
> dataFrame$altezza<-NULL
> dataFrame$altezza<-c(172,180,165,163)
> mean(dataFrame$altezza)
[1] 170
```

- Per cancellare una variabile basta settarla a **NULL**.

Data Frame

- La funzione **as.data.frame**() forza una matrice di dati ad un dataframe.
- I dati possono essere inseriti in maniere semplice tramite un foglio elettronico e caricati in R usando la funzione **read.table**().
- La funzione **write.table**() scrive un **data.frame** in un file formattato come tabella.

```
> esDataframe<-read.table(file="./Esempio.csv", #nomefile
+ header=TRUE, #Intestazione
+ sep="\t", #separatore
+ na.strings = "NA", #utile nel momento in cui un file abbia dei valori mancanti
+ dec="." #viene specificato il tipo di carattere per separare i decimali
+ )
> esDataframe
  ID Sesso Eta
1  1     M  20
2  2     F  30
3  3     F  21
4  4     F  25
5  5     M  28
+ )
> write.table(dataFrame,
+ file="EsDataframe.txt",
+ sep="\t",
+ na="NA",
+ dec=".",
+ row.names=TRUE,
+ col.names=TRUE)
```

GRAFICI

Parte 4

Creazione di un grafico

- Le funzioni principali per la disegnare un grafico sono **plot**(), **point**() e **lines**().

Tipologia grafico:
"b"=*both*
"l"=*lines*
"p"=*points* (default)

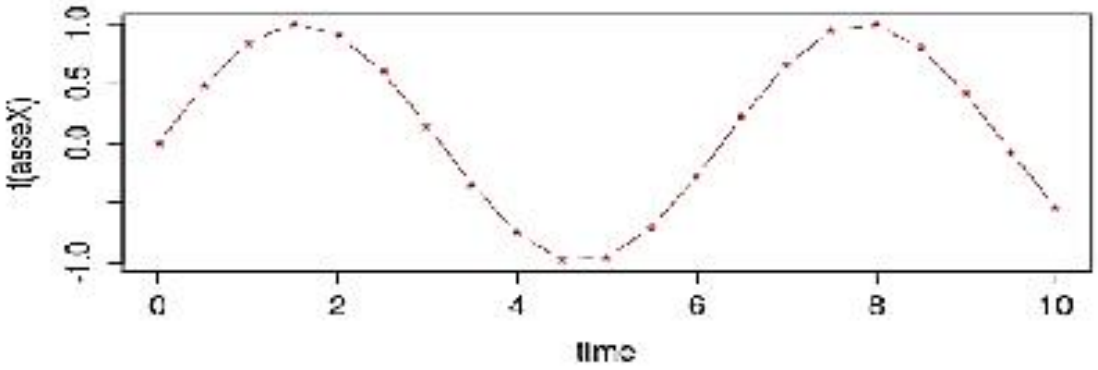
Etichette

Range degli assi

```
> asseX<-seq(0,10,0.5)
> fun1<-sin(asseX)
> fun2<-cos(asseX)
> plot(asseX,fun1,type="b",col=2,pch="*",ylab="f(asseX)",xlab="time", ylim=c(-1,1))
```

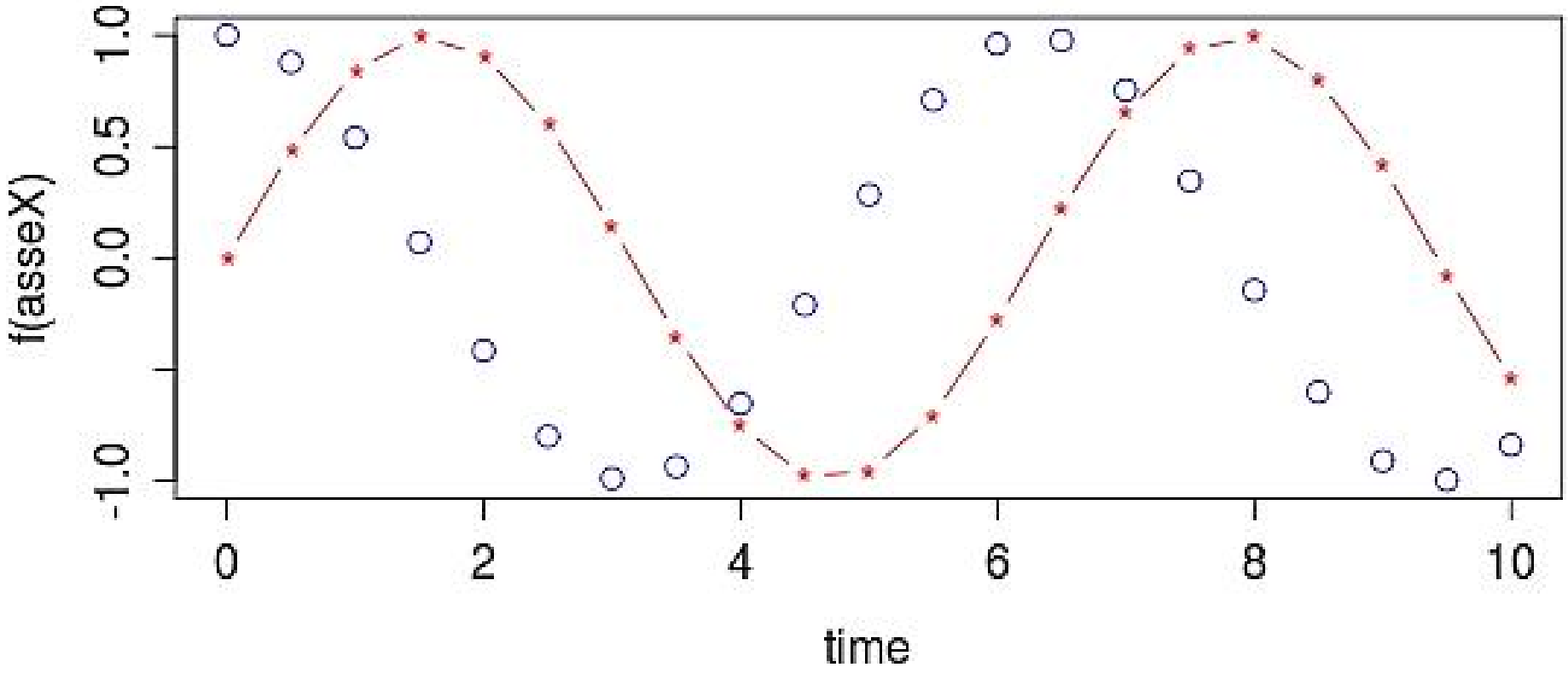
Colore:
defaul 1= *nero*

Tipo di punto:
"o" default



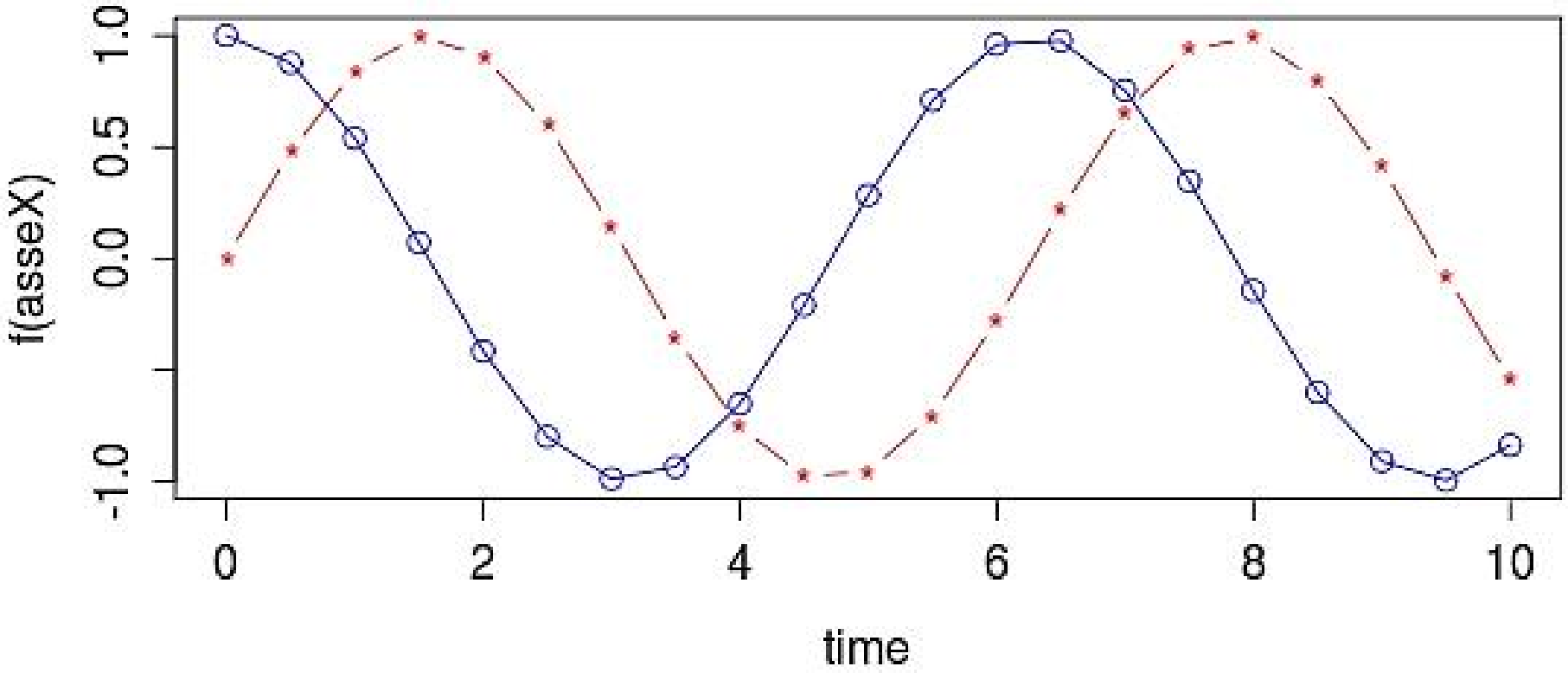
Creazione di un grafico

```
> points(asseX,fun2,col=4)
```



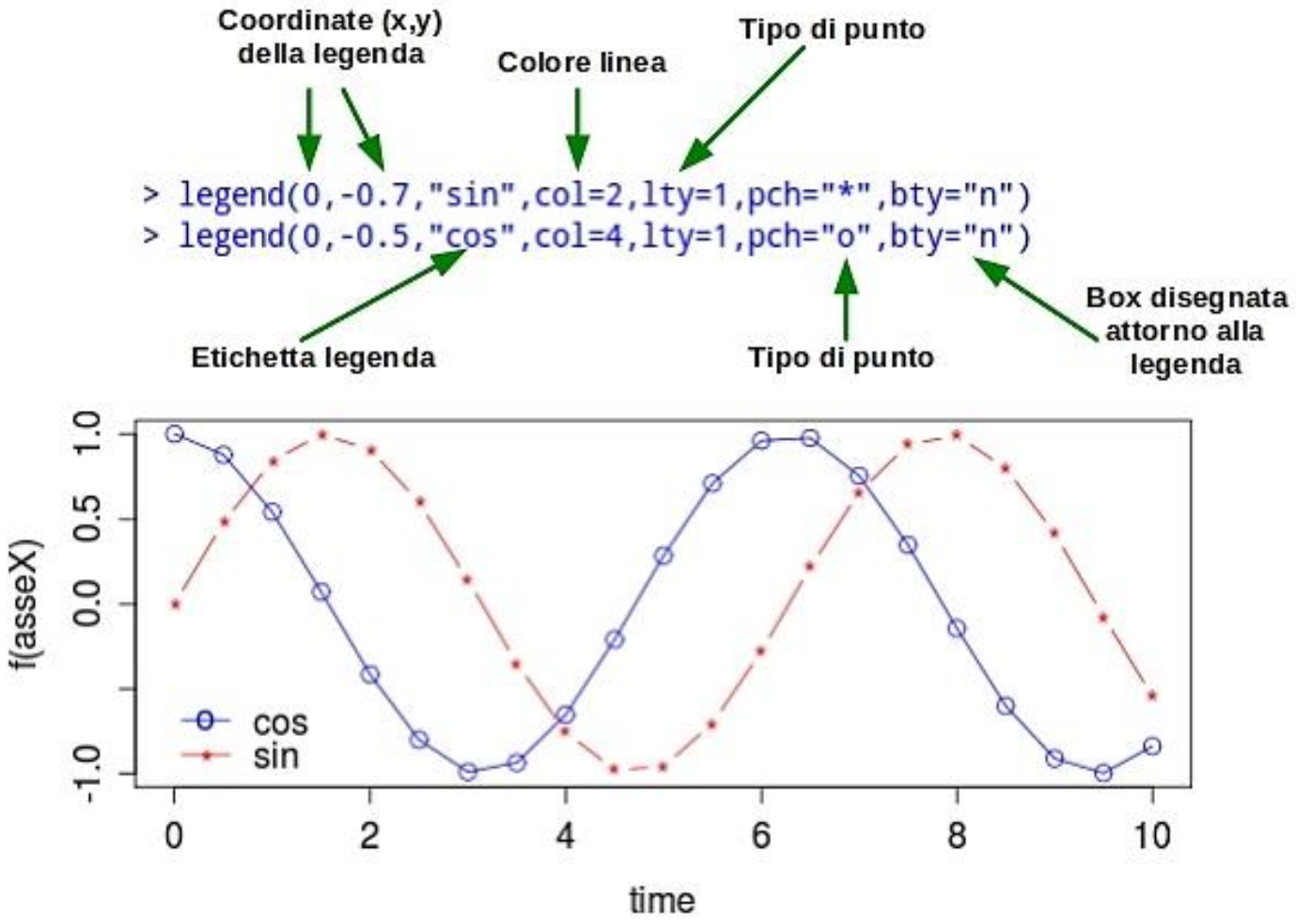
Creazione di un grafico

```
> lines(asseX,fun2,col=4)
```



Legenda

- Per aggiungere una legenda al grafico, c'è la funzione **legend**().



Salvare un grafico

- Per salvare un grafico invece di visualizzarlo si può usare una delle seguenti funzioni:

Funzione	Salva in
pdf("mygraph.pdf")	pdf file
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

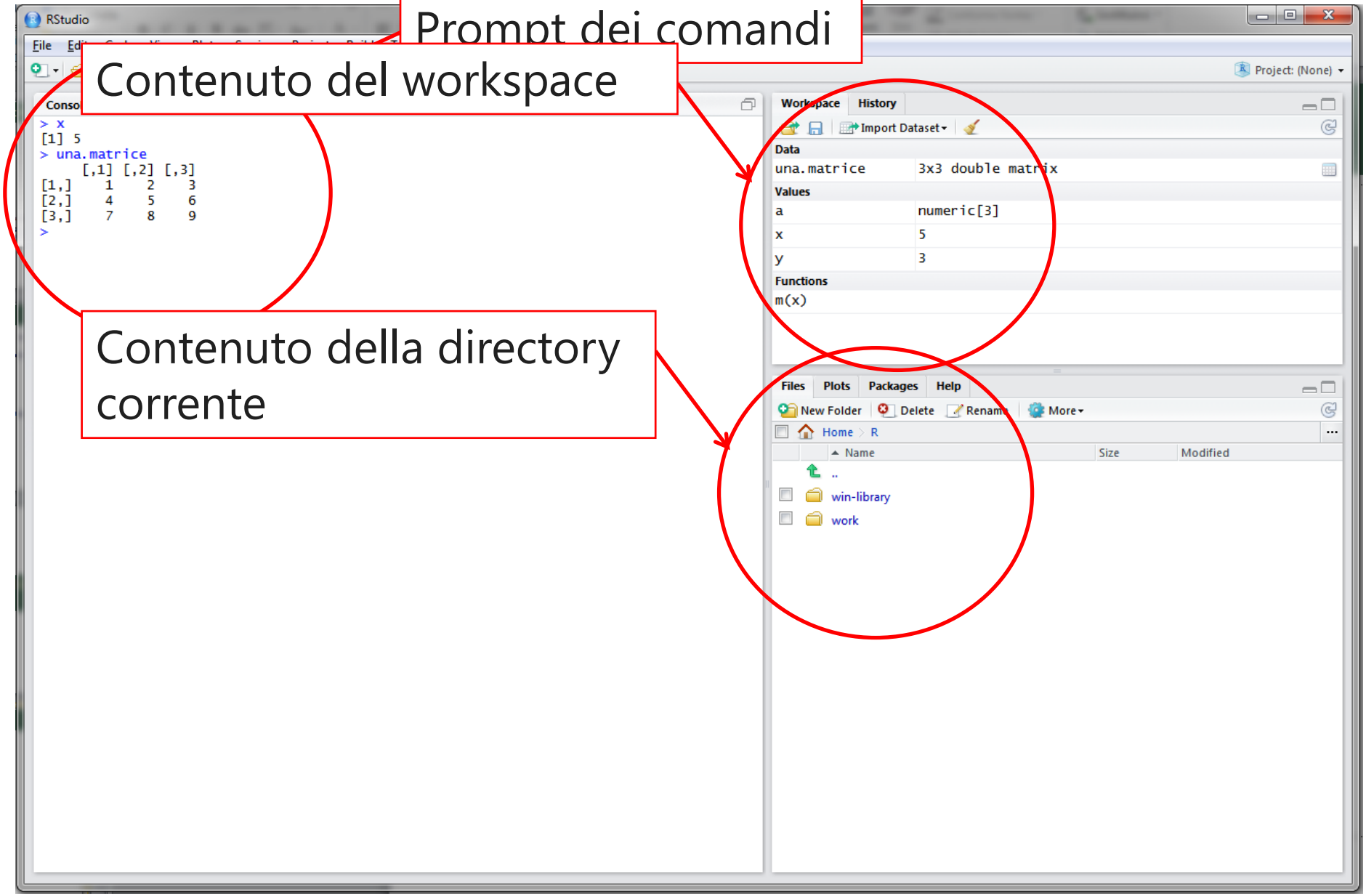
- Esempio:

```
jpeg("myplot.jpg")
plot(x)
dev.off()
```

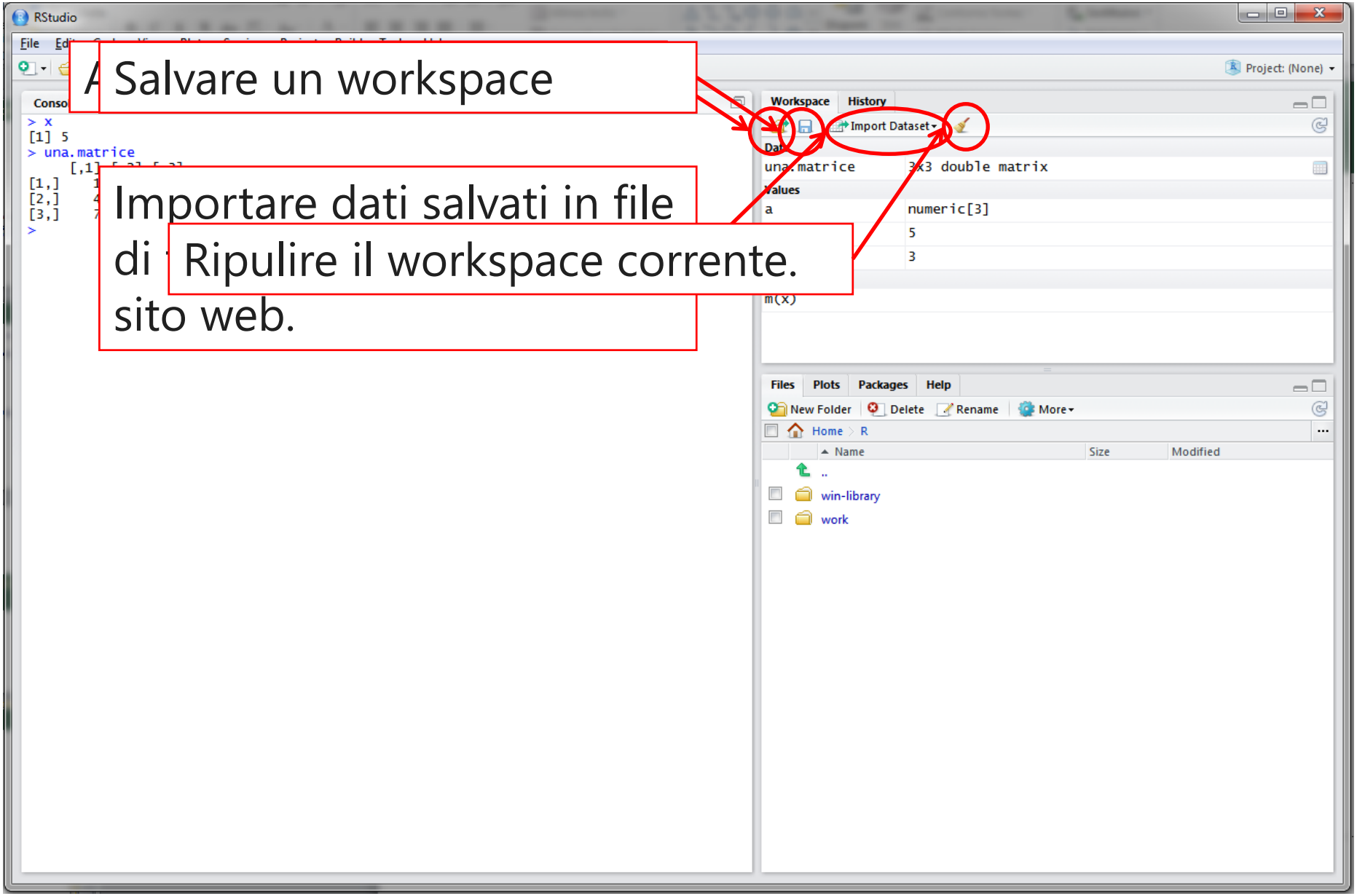
L'ambiente RStudio

Parte 5

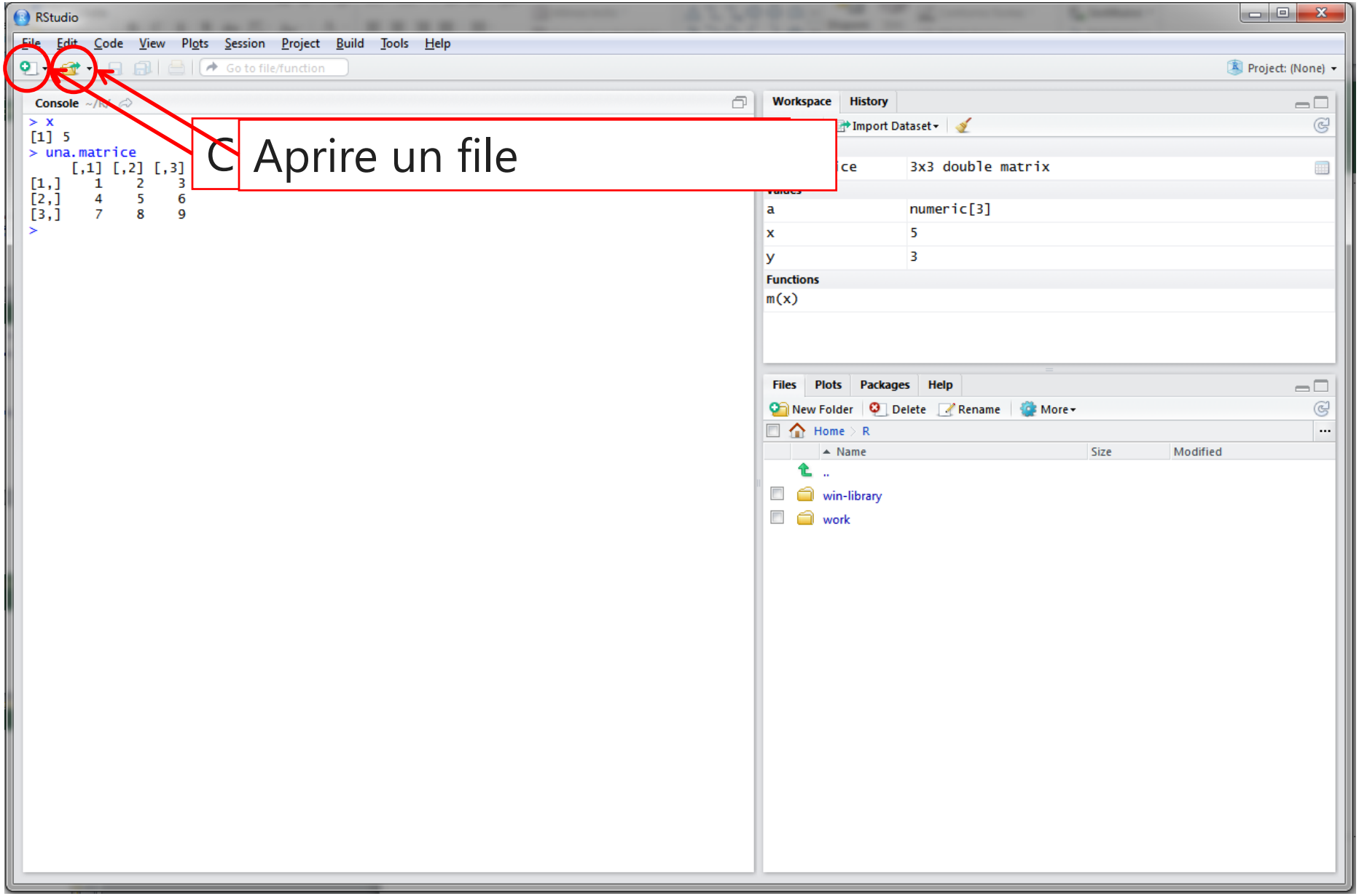
Uno sguardo a RStudio



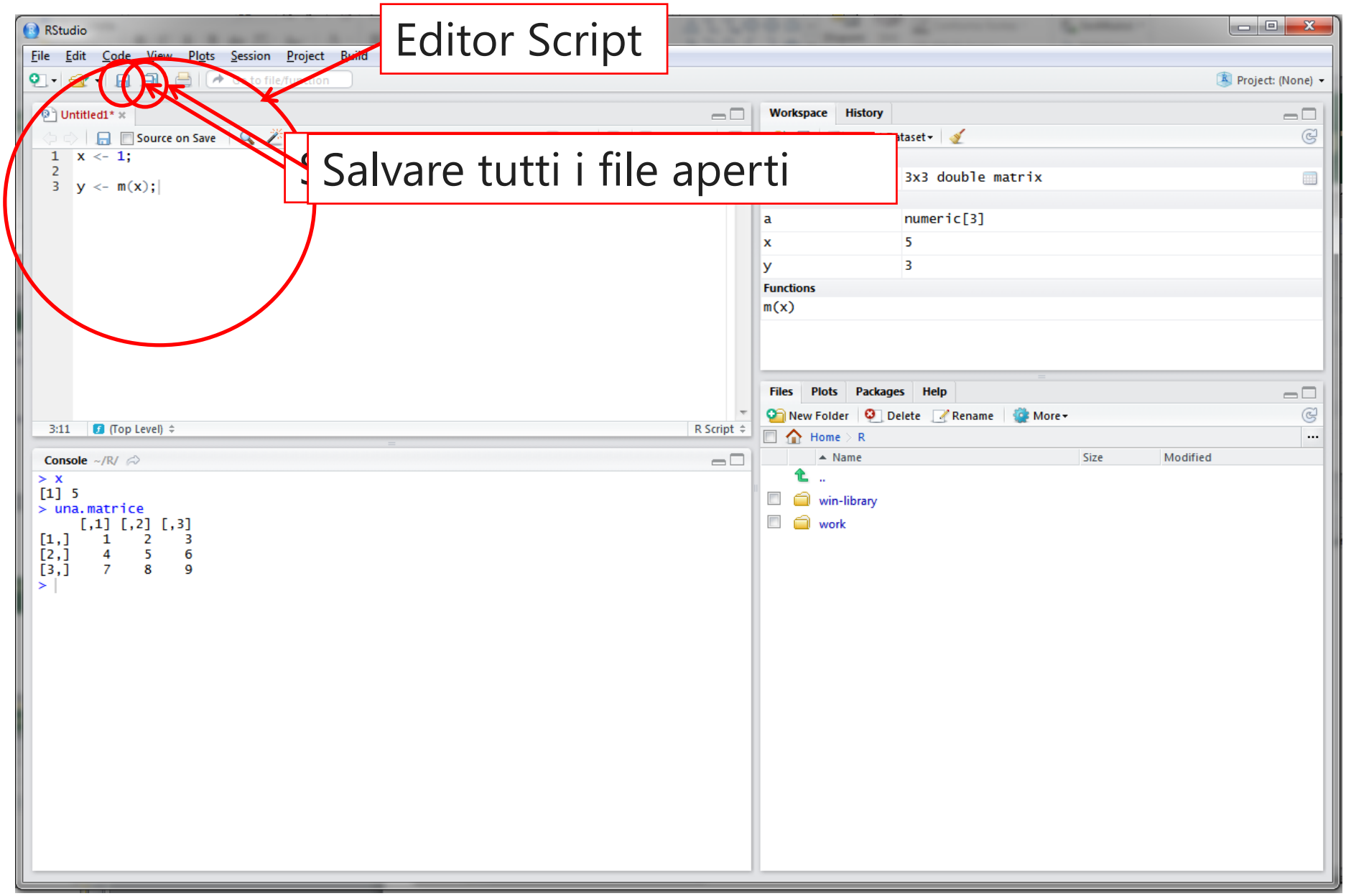
Uno sguardo a RStudio



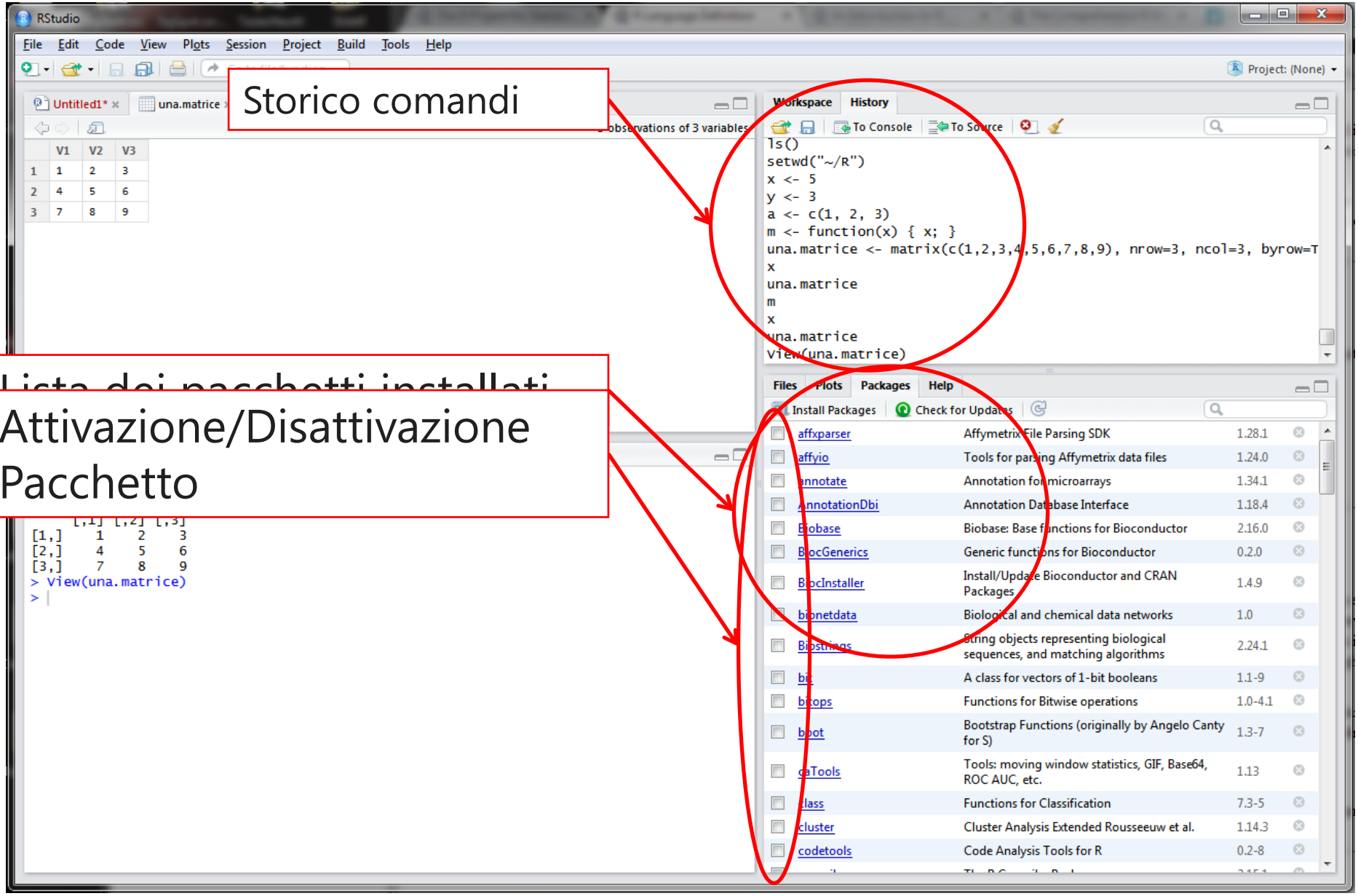
Uno sguardo a RStudio



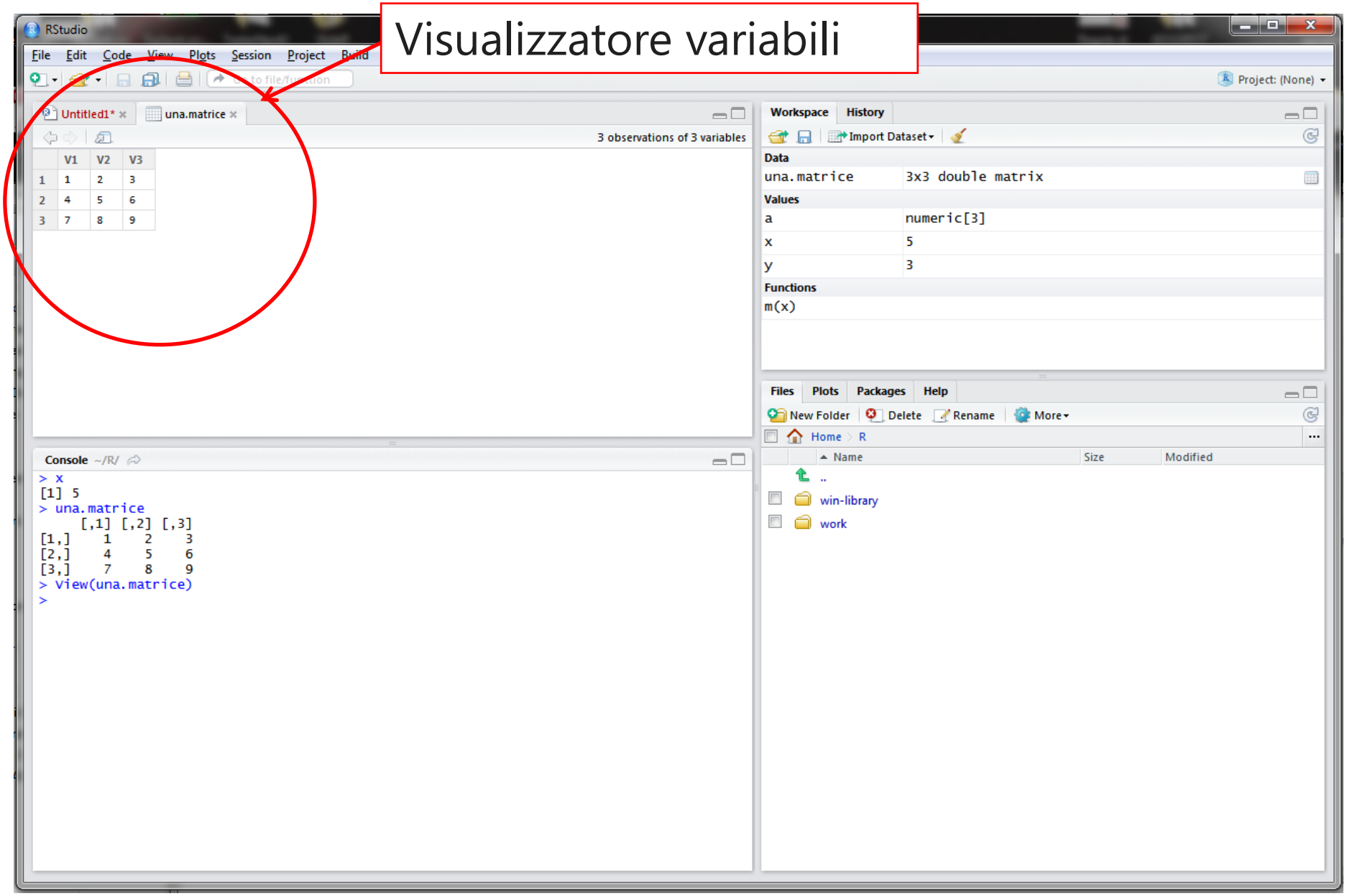
Uno sguardo a RStudio



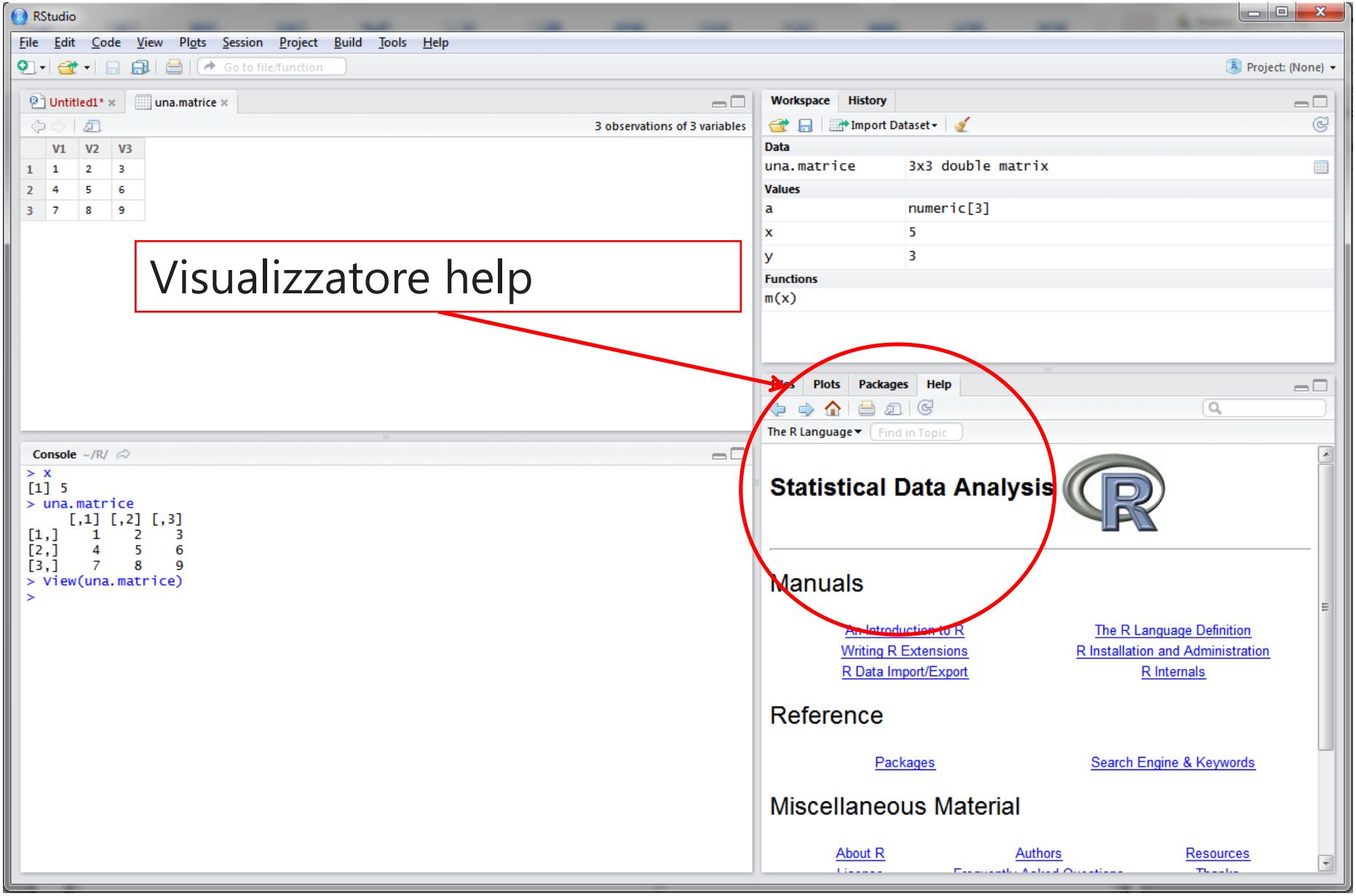
Uno sguardo a RStudio



Uno sguardo a RStudio



Uno sguardo a RStudio



Statistica in R

Parte 6

Indici descrittivi

- R fornisce un gran numero di funzioni per calcoli statistici.
- I più importanti indici descrittivi:
  - **mean(x)**: calcola la media aritmetica di un vettore;
  - **var(x,y=NULL)**: calcola la varianza campionaria di un vettore (o la matrice di varianza, o la varianza tra le colonne di due matrici);
  - **cov(x,y=NULL)**: calcola la matrice di covarianza di una matrice (o la covarianza tra due vettori, o la covarianza tra le colonne di due vettori);
  - **cor(x,y=NULL)**: calcola il coefficiente di correlazione lineare tra due vettori (o la matrice di correlazione, o la correlazione tra le colonne di due vettori);
  - **sd(x)**: calcola la deviazione standard di un vettore (se l'input è una matrice o un dataframe, l'output è il vettore delle deviazioni standard delle colonne dell'input);
  - **median(x)**: calcola la mediana di un vettore in input;
  - **quantile(x)**: calcola i quantili di un vettore in input
- Tutti i metodi hanno il parametro opzionale «**na.rm**» che se settato a **TRUE** rimuove tutti i valori *NA o NaN* prima di eseguire la computazione.

Distribuzioni di probabilità

- Il calcolo delle funzioni associate alle principali distribuzioni di probabilità è estremamente facile in R.
- È, inoltre, possibile generare osservazioni pseudo-casuali con distribuzione diversa da quella uniforme.
- Tutte queste funzioni sono disponibili nel pacchetto «**stats**» tipicamente caricato di default all'avvio dell'ambiente.
- La struttura di tutte le funzioni che implementano le principali distribuzioni è la stessa:
  - **Densità di probabilità**: si antepone **d** al nome della distribuzione;
  - **Funzione di ripartizione**: si antepone **p** al nome della distribuzione;
  - **Quantili**: si antepone **q** al nome della distribuzione;
  - **Osservazioni pseudo-casuali**: si antepone **r** al nome della distribuzione;

Distribuzioni di probabilità

Distribuzione	Nome in R	Parametri
Beta	beta	shape1, shape2, ncp
Binomiale	binom	size, prob
Binomiale negativa	nbinom	size, prob
Cauchy	cauchy	location, scale
Chi-quadro	chisq	df, ncp
Esponenziale	exp	rate
F	f	df1, df2, ncp
Gamma	gamma	shape, scale
Geometrica	geom	prob
Ipergeometrica	hyper	m, n, k
Log-normale	lnorm	meanlog, sdlog
Logistica	logis	location, scale
Normale	norm	mean, sd
Poisson	pois	lambda
T-student	t	df, ncp
Uniforme	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Distribuzioni di probabilità

- Parametri comuni delle funzioni di ripartizione:
  - **lower.tail**: se **FALSE** calcola **P(X > x)**, altrimenti (default) **P(X ≤ x)**
  - **log.p**: se **TRUE** calcola il logaritmo della probabilità
- Parametri comuni delle funzioni densità di probabilità:
  - **log**: se **TRUE** calcola il logaritmo della probabilità
    - …utile per lavorare con prodotti e divisioni di numeri molto piccoli.
- Parametri comuni dei generatori di numeri casuali:
  - **n**: indica il numero di osservazioni pseudo-casuali da generare

Test statistici

- Molti tra i test statistici principali sono disponibili nel pacchetto «**stats**».
- Alcuni tra i più comuni sono:
  - **chisq.test(x)**: esegue il test $\chi 2$ su una tabella di contingenza;
  - **binom.test(x,n,p=0.5)**: esegue il test di ipotesi su una serie di esperimenti di Bernulli;
  - **cor.test(x,y)**: esegue il test di correlazione tra due coppie di input;
  - **fisher.test(x)**: esegue il test di Fisher su una tabella di contingenza;
  - **ks.test(x,y)**: esegue il test di Kolmogorov-Smirnov su uno o due coppie di vettori in input;
  - **t.test(x,y=NULL)**: esegue il T-test su uno o due coppie di vettori in input;
  - **wilcox.test(x,y=NULL)**: esegue il test di Wilcoxon su uno o due campioni.
- L'output di questi test statistici è oggetto classe «**htest**» utilizzabile come una lista.
- Comune a tutti i risultati è la componente «**p.value**» che indica la probabilità che l'ipotesi nulla sia vera sulla base delle condizioni del test.

Introduzione a R – Parte 2

Dott. Alaimo Salvatore

Studio 32 – Blocco 2

Email: alaimos@dmi.unict.it

Panoramica

- Introduzione
- Concetti di base
  - Costrutti fondamentali
  - Strutture dati
  - Grafici
  - L'ambiente RStudio
  - Statistica in R
- Concetti Avanzati
  - Modelli Statistici
  - Programmazione ad Oggetti
  - Programmazione Parallela e Funzionale
  - Grafica Avanzata
    - ggplot2

Come installare pacchetti

- Installare un pacchetto in R è estremamente semplice:
  - Se il pacchetto è nella repository **CRAN**:

    install.packages("*nomepacchetto***", dependencies=***TRUE***)**

  - Se il pacchetto è nella repository **BioConductor**:

    source(http://bioconductor.org/biocLite.R)

    biocLite("*nomepacchetto***")**

Modelli Statistici

Parte 7

Modelli Statistici

- R offre una serie di strumenti altamente versatili e potenti per la rappresentazione di modelli statistici
- Inoltre, esistono molti pacchetti che estendono gli strumenti di base forniti dal linguaggio
  - …le potenzialità sono quindi infinite
- Ma prima di iniziare…Cos'è un modello statistico?
  - Un modello statistico è la formalizzazione delle relazioni tra variabili sotto forma di equazioni matematiche.
  - In pratica descrive come una o più variabili random sono correlate ad altre variabili.
  - Il modello si dice statistico perché le variabili non sono deterministiche, ma si suppone che ci sia una specifica distribuzione di probabilità che genera i dati osservati

Modelli Statistici

- R fornisce tutti gli strumenti necessari a:
  - Definire semplicemente un modello (Formulae)
  - A partire da dati: (LM, GLM, NLM)
    - calcolare i parametri del modello che meglio li approssimano
    - eseguire delle predizioni attraverso un modello pre-calcolato
    - aggiornare i parametri di un vecchio modello
  - Eseguire test che consentono di stabilire la bontà di un modello (ANOVA, Kolmogorov-Smirnov)
- LM $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \epsilon$
- GLM $\psi(Y) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \epsilon$

Formulae

- Permettono di rappresentare simbolicamente modelli statistici di ogni tipo
- La notazione può essere estesa da pacchetti che rendono disponibili nuove caratteristiche
- L'operatore che definisce una formula è «**~**»
- Come ogni altro oggetto di R anche le formulae hanno una classe di appartenenza: **formula**
  - Il cast da un oggetto (tipicamente stringa) ad una formula si può eseguire tramite il comando «**as.formula**»
- Sintassi:
  - ***response*** **~** ***op1*** **term1 op2 term2 op3 term3 …**
    - «**response**» definisce le variabili dipendenti del modello
    - «**op**» è un operatore che implica l'inclusione o l'esclusione di una variabile dal modello
    - «**term**» definisce una variabile indipendenti del modello

Formulae

- Operatori:
  - «**+**» inclusione di una variabile
  - «**-**» esclusione di una variabile
  - «**:**» interazione tra due variabili
  - «*****» inclusione di due variabili e della loro interazione
  - «**^***n*» inclusione di tutte le interazioni fino all'n-esimo ordine
  - «**I()**» definisce una porzione del modello in cui gli operatori sono usati nel loro senso aritmetico
- Termini:
  - vettori, matrici, espressioni, factor
  - formulae
  - 1, 0 (nei modelli lineari rappresenta il coefficiente β_0)
- La notazione qui descritta è tipica dei modelli lineari. Per i modelli non lineari quanto scritto nella formula è interpretato direttamente come una equazione.

Formulae

- Esempi:
  - peso ~ altezza + età – genere
  - peso ~ altezza + età * genere
    - → peso ~ altezza + età + genere + età : genere
  - peso ~ (altezza + età + genere) ^ 2 – altezza : età
    - → peso ~ (altezza + età + genere) * (altezza + età + genere) – altezza : età
    - → peso ~ altezza + età + genere + altezza : genere + età : genere

Modelli Lineari

- *model* **<- lm(*formula*, *data*,** *subset*, *weights*, *na.action*)
  - Addestra un modello lineare sui dati in input attraverso la descrizione data sotto forma di una formula;
  - Input:
    - «**formula**» una *formula* che descrive il modello da usare per i dati
    - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
    - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
    - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
    - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
      - Valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
  - L'output consiste in una lista di classe «**lm**» che fornisce le informazioni sull'adattamento del modello ai dati:
    - «**coefficients**» un *vettore* con i coefficienti calcolati dalla funzione
    - «**fitted.values**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
    - «**residuals**» un *vettore* con gli errori di fitting
    - «**rank**» il rango del modello

Modelli Lineari Generalizzati

- *model* **<- glm(*formula*, family, *data*,** *subset*, *weights*, *na.action*)
  - Addestra un modello lineare generalizzato sui dati in input attraverso la descrizione data sotto forma di una formula;
  - Input:
    - «**formula**» una *formula* che descrive il modello da usare per i dati
    - «**family**» la *link function* usata nel modello
      - binomial, gaussian, Gamma, poisson
    - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
    - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
    - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
    - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
      - valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
  - L'output consiste in una lista di classe «**glm**» che fornisce le informazioni sull'adattamento del modello ai dati:
    - «**coefficients**» un *vettore* con i coefficienti calcolati dalla funzione
    - «**fitted.values**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
    - «**residuals**» un *vettore* con gli errori di fitting
    - «**rank**» il rango del modello

Modelli Non Lineari

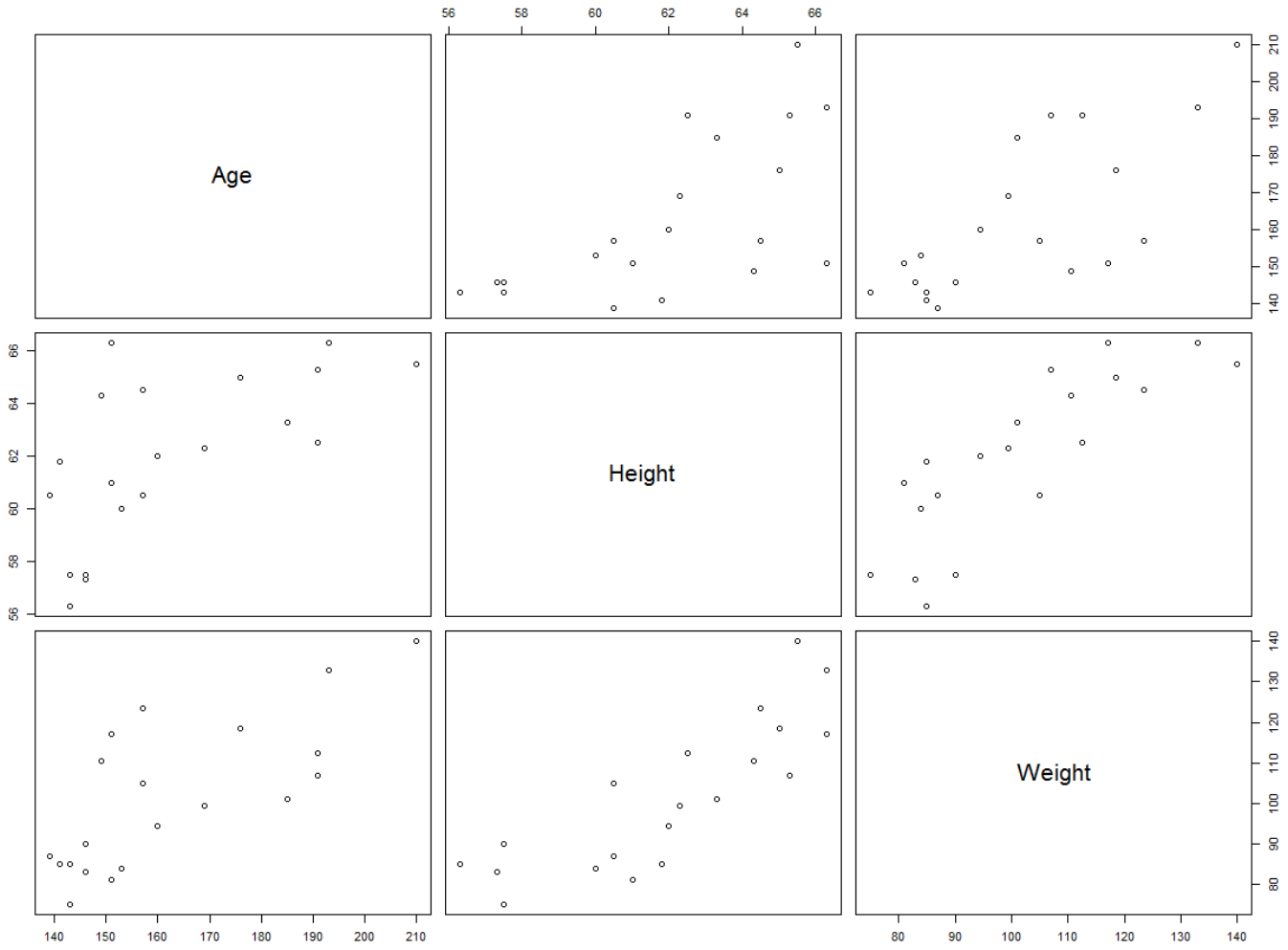
- *model* **<- nls(*formula*, *data*,** *subset*, *weights*, *na.action*)
  - Addestra un modello non lineare sui dati in input usando la forma funzionale descritta dalla formula;
  - Input:
    - «**formula**» una *formula* che descrive l'equazione del modello da usare per i dati
    - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
    - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
    - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
    - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
      - valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
  - L'output consiste in una lista che tra gli elementi ne incorpora uno chiamato «**m**» che consiste in una lista di classe «**nlsModel**» che fornisce le informazioni sull'adattamento del modello ai dati:
    - «**getPars()**» un *vettore* con i coefficienti calcolati dalla funzione
    - «**fitted()**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
    - «**resid()**» un *vettore* con gli errori di fitting

LM, GLM, NLM

- Tutti i metodi elencati restituiscono oggetti su cui possono essere usate le seguenti funzioni:
  - **predict(*model, data*)**
    - esegue una predizione per nuovi dati usando il modello fornito
  - **coefficients(*model*)**
    - restituisce i coefficienti calcolati per il modello fornito
  - **summary(*model*)**
    - calcola e restituisce un elenco di statistiche riassuntive del modello in input

Esempio di Modello Lineare

```
> pop
   Age Height Weight
1  143   56.3   85.0
2  191   62.5  112.5
3  160   62.0   94.5
4  157   64.5  123.5
5  191   65.3  107.0
6  141   61.8   85.0
7  185   63.3  101.0
8  210   65.5  140.0
9  149   64.3  110.5
10 169   62.3   99.5
11 157   60.5  105.0
12 139   60.5   87.0
13 146   57.5   90.0
14 151   66.3  117.0
15 153   60.0   84.0
16 176   65.0  118.5
17 146   57.3   83.0
18 151   61.0   81.0
19 193   66.3  133.0
20 143   57.5   75.0
> plot(pop)
>
```



Esempio di Modello Lineare

```
> model <- lm(weight ~ Age + Height, data=pop)
> summary(model)

Call:
lm(formula = Weight ~ Age + Height, data = pop)

Residuals:
    Min      1Q  Median      3Q     Max
-15.073  -6.688   1.223   7.420  14.675

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -168.1133    45.2984  -3.711  0.00173 **
Age            0.3061     0.1348   2.271  0.03642 *
Height         3.5484     0.9062   3.916  0.00111 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.477 on 17 degrees of freedom
Multiple R-squared:  0.763, Adjusted R-squared:  0.7351
F-statistic: 27.37 on 2 and 17 DF,  p-value: 4.843e-06

> coefficients(model)
(Intercept)         Age      Height
-168.113277    0.306143    3.548435
> predict(model, list(Height=70, Age=220))
       1
147.6286
```

Programmazione ad Oggetti

Parte 8

Classi e Metodi

- Alcune ragioni per cui programmare ad oggetti è una buona pratica:
  - aumento della produttività
  - facilita il mantenimento del codice
  - codice riutilizzabile
  - incapsula la rappresentazione degli oggetti
  - specializza comportamento delle funzioni allo specifico oggetto
- È importante ricordare che:
  - tutto in R è un **oggetto** ed ha, quindi, una **classe** di appartenenza
  - Una **classe** rappresenta la definizione di un oggetto (metodi e proprietà)
  - Un **metodo** è una funzione che esegue una qualche computazione su un oggetto di una specifica classe.
- In R una **funzione generica** è un particolare tipo di funzione che determina la classe dei suoi argomenti e seleziona automaticamente il metodo più appropriato da eseguire (una funzione associata ad una collezione di metodi).

Le classi in R

- A differenza di altri linguaggi OOP, in R ci sono 3 tipi di classi e quindi tre stili di rappresentazione:
  - **Classi S3**: vecchio stile, veloci ma molto informali (Non li studieremo)
  - **Classi S4**: nuovo stile, rigoroso e formale
  - **ReferenceClasses**: classi in stile Java o C++ in contrasto con lo stile funzionale di R (Non li studieremo)
- L'approccio S3 è molto semplice da implementare e non ha requisiti formali sulla struttura degli oggetti. Basta assegnare il nome di una classe ad un qualunque oggetto R per definirla.
- L'approccio S4 richiede invece che di una classe ne sia prima data una definizione rigorosa. Ogni oggetto valido di una classe S4 ne dovrà soddisfare obbligatoriamente tutti i requisiti.
- S4 sta lentamente sostituendo S3 anche se quest'ultimo è ancora molto usato
  - Repository come bioconductor obbligano tutti i nuovi pacchetti ad usare classi di tipo S4
- Tutto ciò che vedremo richiede che il pacchetto «**methods**» sia caricato.

Classi S4

- Le informazioni nelle classi S4 sono organizzate in «**slot**». Ogni slot ha un nome e un preciso tipo (classe).
- Gli slot sono uno dei principali vantaggi delle classi S4.
  - La classe dei dati nello slot deve corrispondere a quella nella definizione.
  - Non è quindi possibile avere un numero in uno slot che ha tipo stringa.

Function	Description
setClass()	Crea una nuova classe
setMethod()	Crea un nuovo metodo
setGeneric()	Crea una funzione generica
new()	Costruisce un nuovo oggetto
getClass()	Restituisce la definizione di una classe
getMethod()	Restituisce la definizione di un metodo
getSlots()	Restituisce il nome e la classe di ogni slot
@	Restituisce o rimpiazza il contenuto di uno slot
validObject()	Controlla se un oggetto è valido

Creare/Instanziare una classe

- Sintassi:
  - **setClass(***Class***,** *slots***,** *contains***,** *prototype***,** *validity***)**
    - «**Class**»: il nome della classe da creare
    - «**slots**»: una lista con i nomi e le classi degli slot
    - «**contains**»: un vettore con i nomi delle superclassi da ereditare
    - «**prototype**»: una lista con i valori di default per ogni slot
    - «**validity**»: una funzione che restituisce TRUE se l'input è un oggetto valido
  - *my.object* **<- new(***Class***, …)**
    - «Class»: il nome della classe da istanziare
    - «**…**»: i parametri che richiede il costruttore o una lista degli slot con i relativi valori

Aggiungere metodi

- Sintassi:
  - **setMethod(***f***,** *signature***,** *definition***)**
    - «**f**»: una funzione generica o il nome di una qualsiasi funzione
    - «**signature**»: un vettore con le classi a cui associare il metodo
    - «**definition**»: la funzione che sarà richiamata
  - **setGeneric(***name***,** *def***,** *valueClass***)**
    - «**name**»: Il nome della funzione generica da creare
    - «**def**»: una funzione opzionale che definisce la funzione di default
    - «**valueClass**»: un vettore opzionale di nomi di classi che costituiscono l'output del generico
  - **callNextMethod()**
    - Chiama il metodo di una delle classi ereditate. Per essere più precisi, il metodo scelto corrisponde a quello che sarebbe stato eseguito se quello corrente non fosse stato definito.

Metodi speciali

- Alcuni esempi di metodi definibili nelle classi:
  - setMethod(f=**"initialize"**, **signature=...**, definition=function (.Object, ...) {

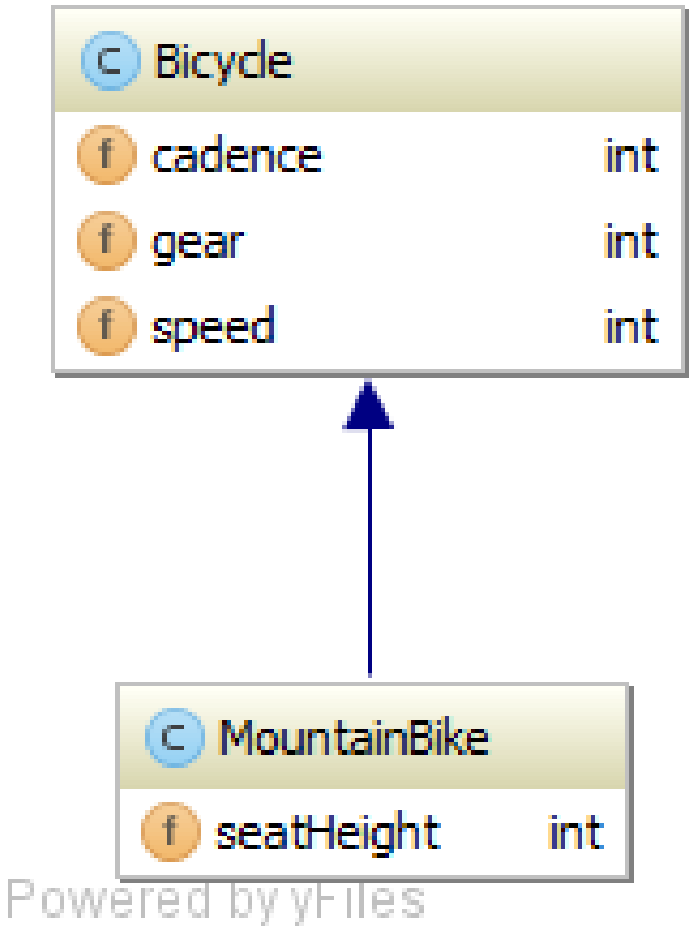
    Object <- callNextMethod(.Object, ...)

    ...

    return (Object)

    })
  - setMethod(f=**“show"**, **signature=...**, definition=function (object) { ... })
  - setMethod(f=**“["**, **signature=...**, definition=function (x,i,j) { ... })
  - setMethod(f=**“[<-"**, **signature=...**, definition=function (x,i,j,value) { ... })
- Per rendere lecito un cast da un tipo a un altro:
  - **setAs(***from***,** *to***,** *def***)**
    - «**from**»: il tipo di partenza
    - «**to**»: il tipo a cui è possibile fare il cast
    - «**def**»: la funzione che esegue la conversione degli oggetti

Un semplice esempio – Biciclette



Un semplice esempio – Biciclette

```
setClass("Bicycle",
         slots=list(cadence="numeric",
                    gear="numeric",
                    speed="numeric"),
         prototype=list(cadence=0,
                        gear=1,
                        speed=0))

setGeneric("setCadence", function(object, cadence) standardGeneric("setCadence"))
setGeneric("changeGear", function(object, gear) standardGeneric("changeGear"))
setGeneric("applyBrake", function(object, decrement) standardGeneric("applyBrake"))
setGeneric("speedUp", function(object, incremnt) standardGeneric("speedUp"))

setMethod("setCadence", signature(object="Bicycle", cadence="numeric"), function (object, cadence) {
  object@cadence <- cadence
  return (object)
})

setMethod("changeGear", signature(object="Bicycle", gear="numeric"), function (object, gear) {
  object@gear <- gear
  return (object)
})

setMethod("applyBrake", signature(object="Bicycle", decrement="numeric"), function (object, decrement) {
  object@speed <- object@speed - decrement
  return (object)
})

setMethod("speedUp", signature(object="Bicycle", incremnt="numeric"), function (object, incremnt) {
  object@speed <- object@speed + incremnt
  return (object)
})

setMethod("show", "Bicycle",
          function (object) {
            cat("A bicycle \n")
            cat("  Cadence:", object@cadence, " revolutions per minute \n")
            cat("  Speed:", object@speed, " meters per hour \n")
            cat("  Gear:", object@gear, "\n")
            cat("\n")
          })
```

Un semplice esempio – Biciclette

```
setClass("MountainBike",
         slots=list(seatHeight="numeric"),
         prototype=list(seatHeight=1),
         contains="Bicycle")

setGeneric("setSeatHeight", function(object, height) standardGeneric("setSeatHeight"))

setMethod("show", "MountainBike",
          function (object) {
            cat("A Mountain Bike \n")
            cat("  Seat Height:", object@seatHeight, " cm \n")
            cat("  Cadence:", object@cadence, " revolutions per minute \n")
            cat("  Speed:", object@speed, " meters per hour \n")
            cat("  Gear:", object@gear, "\n")
            cat("\n")
          })

setMethod("setSeatHeight", signature(object="MountainBike", height="numeric"), function (object, height) {
  object@seatHeight <- height
  return (object)
})
```

Un semplice esempio – Biciclette

```
> my.bike <- new("Bicycle")
> print(my.bike)
A bicycle
  Cadence: 0  revolutions per minute
  Speed: 0  meters per hour
  Gear: 1
```

Programmazione Parallela e Funzionale

Parte 9

Prima di iniziare

- Prima di poter parlare di **programmazione parallela**, è necessario parlare di **vettorizzazione** del codice.
- Infatti lo scopo principale della parallelizzazione è lo speed-up del codice in modo da sfruttare al massimo tutte le capacità del sistema di calcolo.
- In un ambiente come R, però, un primo speed-up significativo si ottiene eseguendo una **vettorizzazione** del codice

Prima di iniziare

- Un tipico codice non vettorizzato che esegue calcoli su matrici è:

```
for (i in ...) {
    for (j in ...) {
        my.matrix <- func(mymatric, i, j)
    }
}
```

- Il principale svantaggio di questo codice è che la maggior parte del tempo si perde nelle iterazioni e nel copiare le variabili
  - … piuttosto che fare calcoli
- Infatti in R non esiste il passaggio per riferimento (paradigma funzionale) e ogni operazione che modifica una qualche variabile richiede che questa venga prima copiata e poi modificata
  - …l'unica eccezione sono le Reference Classes

Prima di iniziare

- Quando si sviluppa codice R occorre seguire alcune semplici regole:
  - Evitare cicli che processano un elemento per iterazione
  - Usare funzioni che processano intere strutture dati in una singola chiamata, eliminando i cicli e il processo di copia
  - Processare vettori interi piuttosto che singoli elementi (vettorizzazione)
  - Usare le funzioni della serie «**apply**» per semplificare la vettorizzazione, ovvero il processing di intere righe, colonne o liste
  - Programmazione Funzionale!!

Le funzioni «apply»

- La più semplice funzione per vettorizzare il codice in R è la funzione «**apply**» che opera sulle matrici:
  - **apply(M,1,fun)** – applica fun alle righe di M
  - **apply(M,2,fun)** – applica fun alle colonne di M
- Sintassi:
  - *result* **<- apply(***X***,** *MARGIN***,** *FUN***, …)**
    - «**x**»: un array o una matrice o un dataframe (da convertire in matrice)
    - «**MARGIN**»: la dimensione a cui applicare FUN (es. 1 righe, 2 colonne per una matrice)
    - «**FUN**»: la funzione da applicare
    - «**…**»: lista opzionale di argomenti da passare alla funzione
  - Esempio:

```
> M <- matrix(runif(20), nrow=5, ncol=4)
> apply(M, 1, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
> apply(M, 2, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
>
```

Altre funzioni «apply»

- «**apply**» non funziona sulle liste per questo è disponibile la funzione «**lapply**»
- Sintassi:
  - *result* **<- lapply(***X*, *FUN*, ..., *simplify*, *USE.NAMES***)**
    - «**X**»: un vettore o una lista o un oggetto che può essere convertito in lista
    - «**FUN**»: la funzione da applicare ad ogni elemento di X
    - «**...**»: una lista opzionale di argomenti da passare alla funzione
    - «**simplify**»: se TRUE cerca di trasformare l'output in un vettore o una matrice altrimenti restituisce una lista
    - «**USE.NAMES**»: se TRUE e X è un vettore di stringhe usa X come vettore dei nomi per la lista dei risultati
  - Esempio:

```
> x <- list(a = 1:10, beta = exp(-3:3))
> lapply(x, mean)
$a
[1] 5.5

$beta
[1] 4.535125
```

Altre funzioni «apply»

- Per il caso speciale «**lapply(…, simplify=TRUE, USE.NAMES=TRUE)**» è stata definita la funzione «**sapply**»
- Sintassi:
  - *result* **<- sapply(***X*, *FUN*, …**)**
    - «**X**»: un vettore o una lista o un oggetto che può essere convertito in lista
    - «**FUN**»: la funzione da applicare ad ogni elemento di X
    - «**…**»: una lista opzionale di argomenti da passare alla funzione
  - Esempio:

```
> x <- list(a = 1:10, beta = exp(-3:3))
> x
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$beta
[1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183  7.38905610 20.08553692

> y <- sapply(x, mean)
> y
       a     beta
5.500000 4.535125
```

Altre funzioni «apply»

- Altre due funzioni che permettono di vettorizzare facilmente il codice sono:
  - *result* **<- mapply(***FUN***, …)**
    - Applica parallelamente «**FUN**» a più liste
  - *result* **<- tapply(***X***, *INDEX*, *FUN*, …)**
    - Applica «**FUN**» ad un vettore o lista o oggetto «**X**» suddividendolo in gruppi specificati dalla variabile categoriale «**INDEX**»
- Esempi:

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

```
> groups
 [1] 15 11 11 13 19 12 9  13 15 14
Levels: 9 11 12 13 14 15 19
> tapply(groups, groups, length)
 9 11 12 13 14 15 19
 1  2  1  2  1  2  1
```

Parallelizzazione del codice

- Per parallelizzare un codice vettorizzato, tutte le funzioni necessarie sono disponibili nel pacchetto «**parallel**»
- Tutte le funzioni *apply* hanno un corrispettivo nel pacchetto *parallel*:
  - apply(X, MARGIN, FUN, …) → parApply(cl, X, MARGIN, FUN, …)
  - apply(X, 1, FUN, …) → parRapply(cl, X, FUN, …)
  - apply(X, 2, FUN, …) → parCapply(cl, X, FUN, …)
  - lapply(X, FUN, …) → parLapply(cl, X, fun, …)
  - sapply(X, FUN, …) → parSapply(cl, X, fun, …)
  - mapply(FUN, …) → clusterMap(cl, fun, …, SIMPLIFY=FALSE)
- Hanno la stessa sintassi della corrispondente versione non parallela con l'aggiunta di un parametro «cl» che rapppresenta un oggetto che descrive il cluster computazionale

Oggetto cluster

- Per costruire un oggetto cluster si utilizza la funzione «**makeCluster**»
  - la sintassi completa è disponibile nell'help del pacchetto parallel
  - per una macchina con processore multi-core:
    - **cl <- makeCluster(detectCores())**
- Al termine delle operazioni è necessario distruggere l'oggetto cluster con la funzione «**stopCluster**»:
  - **stopCluster(cl)**
- Se un cluster usa il generatore di numeri random è necessario inizializzarlo con il comando «**clusterSetRNGStream(***cl***, *iseed*)**»
  - clusterSetRNGStream(cl, 123) # imposta un seed riproducibile
  - clusterSetRNGStream(cl, NULL) # imposta un seed casuale

Esempio

```
> cl <- makeCluster(detectCores())
>
> M <- matrix(runif(20), nrow=5, ncol=4)
> x <- list(a = 1:10, beta = exp(-3:3))
>
> parApply(cl=cl, M, 1, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
> parRapply(cl=cl, M, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
>
> parApply(cl=cl, M, 2, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
> parCapply(cl=cl, M, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
>
> parLapply(cl=cl, x, mean)
$a
[1] 5.5

$beta
[1] 4.535125

> parSapply(cl=cl, x, mean)
       a     beta
5.500000 4.535125
>
> clusterMap(cl, rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4

>
> stopCluster(cl)
```

Programmazione Funzionale

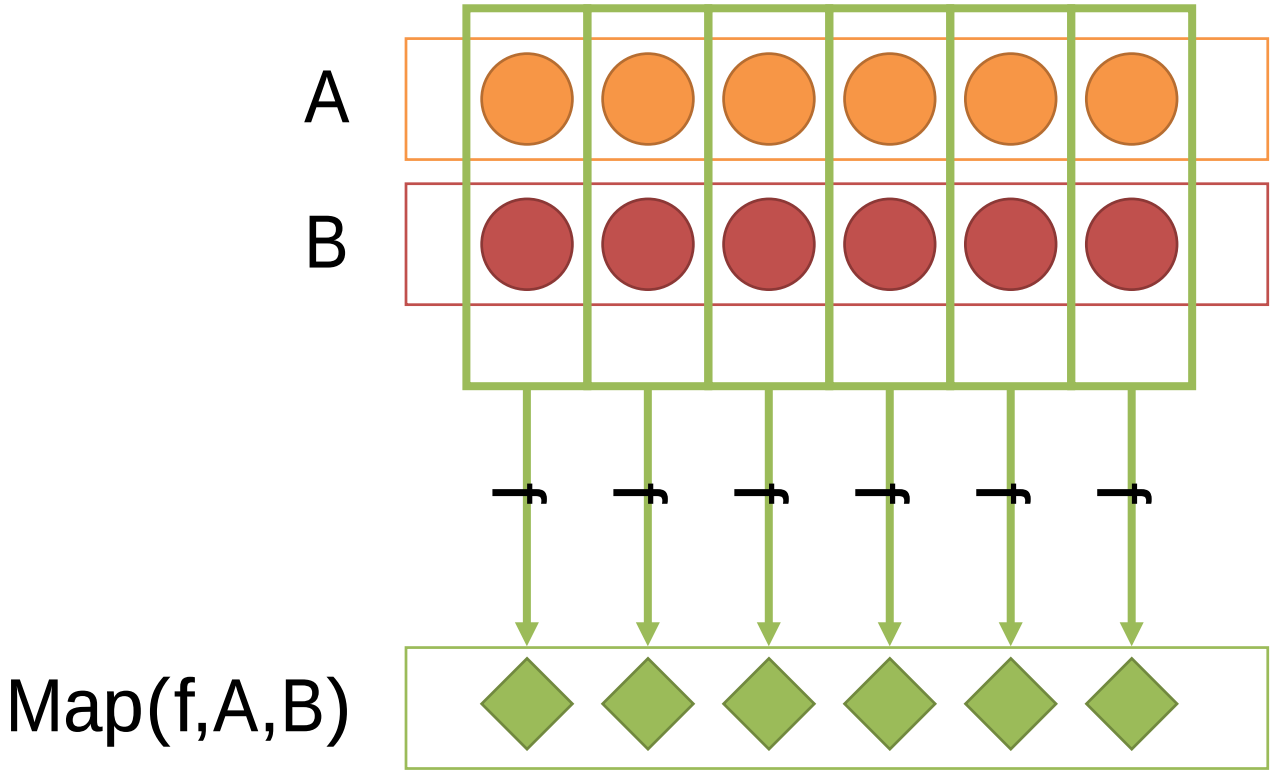
- Un'altra strategia per evitare cicli ed ottenere uno speed-up significativo del codice consiste nell'usare le caratteristiche della programmazione funzionale.
- Supponiamo di definire un predicato (funzione) ***f***, che restituisce ***TRUE*** per gli elementi che si desidera elaborare, ***FALSE*** altrimenti.
- Filter(f, x) – restituisce gli elementi del vettore ***x*** per cui ***f*** è ***TRUE***.
  - Il parametro opzionale «***right***» se TRUE impone alla funzione di analizzare gli elementi dall'ultimo al primo, se FALSE (default) dal primo all'ultimo.

Programmazione Funzionale

- Supponendo di avere una funzione ***f(x,y)*** qualsiasi
- Reduce(f, x) – è un modo per iterare una lista o vettore ***x*** applicando una funzione ***f*** a risultati successivi.
- Supponiamo che x = x1, x2, x3, …
- Reduce(f,x) calcola:
  - f(x1, x2)
  - f(f(x1, x2), x3)
  - f(f(f(x1,x2), x3), x4)
- Il risultato di *default* è il valore dell'ultima chiamata a ***f***, ma impostando il parametro «***accumulate=TRUE***» tutti i risultati intermedi saranno restituiti.

Programmazione Funzionale

- Supponiamo di avere una funzione f(a,b,c…)
- **Map**(f, …) – applica la funzione f a tutti gli elementi passati come parametri



Programmazione Funzionale – Esempio 1

```
x <- as.double(1:1000000)

f1 <- function(x) {
  s <- 0
  for (i in 1:length(x)) {
    s <- s + x[i]
  }
  print(s)
}

f2 <- function(x) {
  s <- Reduce("+", x)
  print(s)

}


print(system.time(f1(x)))

print(system.time(f2(x)))
```

```
> source('D:/[redacted]')
[1] 500000500000
   user  system elapsed
   0.51    0.00    0.55
[1] 500000500000
   user  system elapsed
   0.25    0.00    0.25
>
```

Programmazione Funzionale – Esempio 2

```
Reduce(c,
       Map(
         function (x) mean(as.vector(unlist(x))),
         split(cars, 1:nrow(cars))
       )
      )
```

Programmazione Funzionale – Esempio 3

```
v <- c("aardvark", NULL, "aluminum", "bail", "", "a", "zero")

Reduce("+",
       Map(
         function (w) (1),
         Filter(
           function (w) (substr(w, 1,1) == "A"),
           Map(
             function (w) toupper(w),
             Filter(function (w) (!is.null(w) && w != ""), v)
             )
           )
         )
       )
```

Programmazione Funzionale – Esempio 3bis

```
v <- c("aardvark", NULL, "aluminium", "bail", "", "a", "zero")

Reduce("+",
       Map(
         function (w)
           ifelse((!is.null(w) && w != "" && substr(toupper(w), 1,1) == "A"), 1, 0),
         v
       )
      )
```

Programmazione Funzionale

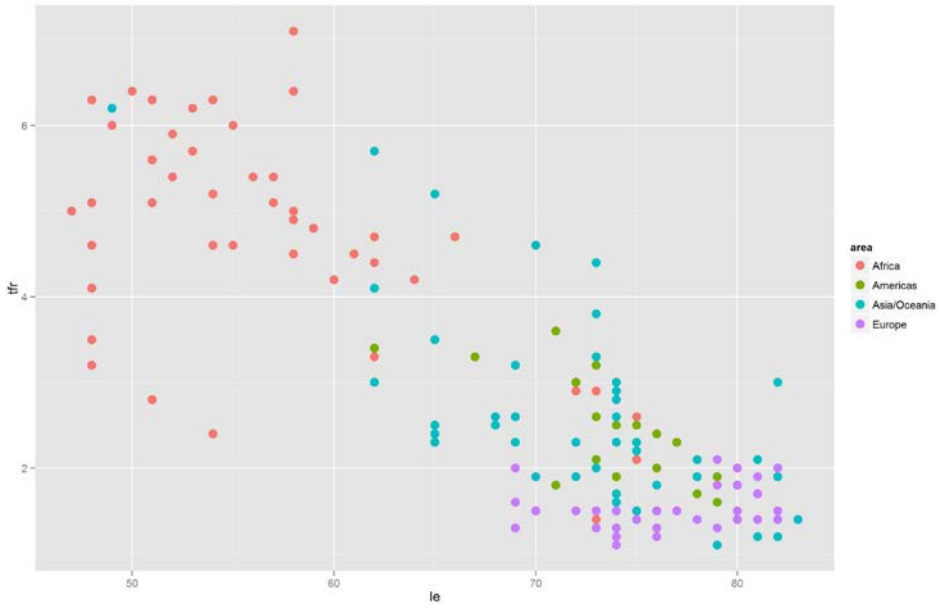
- **Find**(f, x) – trova il **primo** elemento di ***x*** che soddisfa il predicato ***f***
- **Find**(f, x, right=TRUE) – trova l'**ultimo** elemento di ***x*** che soddisfa il predicato ***f***
- **Position**(f, x) – trova la posizione del **primo** elemento di ***x*** che soddisfa il predicato ***f***
- **Position**(f, x, right=TRUE) – trova la posizione dell'ultimo elemento di ***x*** che soddisfa il predicato ***f***
- **Negate**(f) – calcola la negazione di un predicato ***f***

GGPLOT2

Parte 10

Cosa è ggplot2?

- **ggplot2** è una libraria grafica per R
  - basata sulla «**grammar of graphics**» (Leland Wilkinson, 2005)
  - cerca di prendere gli aspetti positivi dell'ambiente grafico di base ma nessuno di quelli negativi
  - Si prende cura di molti dettagli che rendono il plotting complicato
    - fornendo un potente sistema grafico che rende semplice la produzione delle rappresentazioni più complesse



Cosa è ggplot2?

- Un grafico è suddiviso in 7 componenti principali:
  - **Dati**: i dati che devono essere visualizzati sul grafico, sotto forma di un data frame.
  - **Coordinate System**: descrive lo spazio in cui i dati saranno proiettati
    - coordinate cartesiane, polari, …
  - **Geoms**: descrivono gli oggetti usati per rappresentare i dati
    - punti, linee, rettangoli, …
  - **Aesthetics**: descrivono le caratteristiche visuali del grafico e dei geoms
    - posizioni, dimensioni, colori, forme, trasparenze, …
  - **Scales**: descrive, per ogni aesthetic, come convertire una caratteristica in un valore visualizzato
  - **Stats**: descrivono trasformazioni statistiche da applicare per riassumere i dati
  - **Facets**: descrivono come suddividere i dati in gruppi da rappresentare in sotto grafici diversi

I dati di esempio:

158 observations of 9 variables

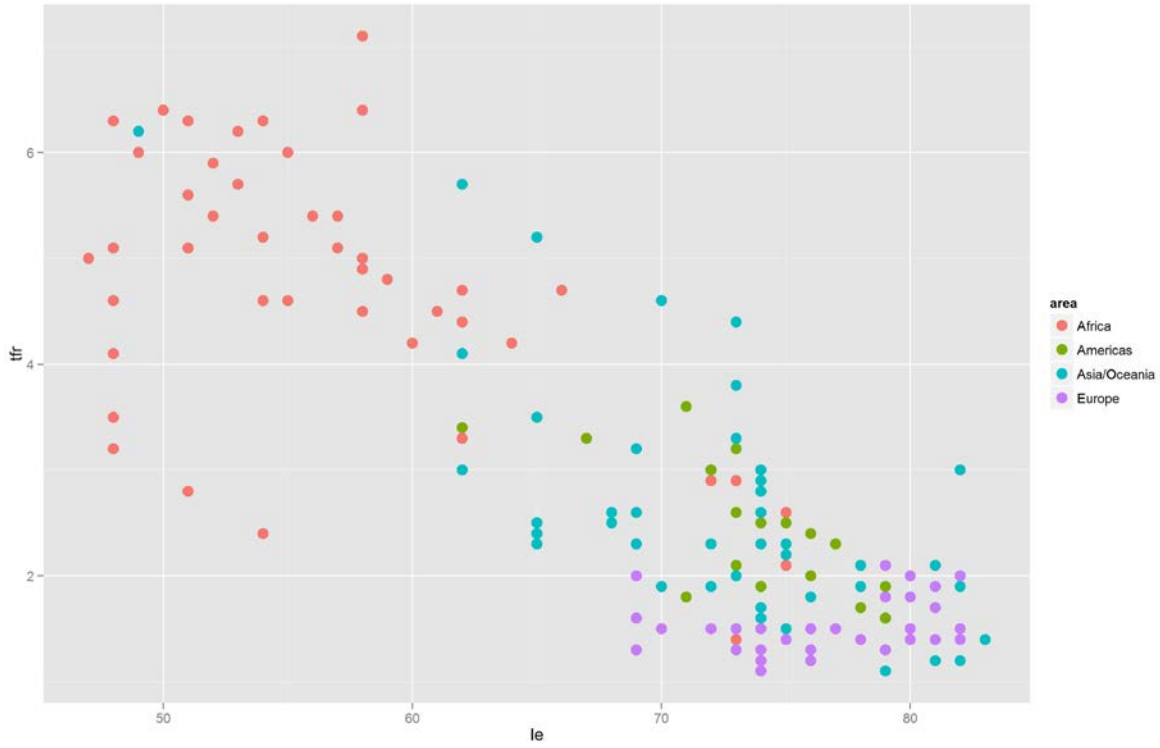
	country	pop2012	imr	tfr	le	leM	leF	region	area
1	Algeria	37.4	24	2.9	73	72	75	Northern Africa	Africa
2	Egypt	82.3	24	2.9	72	70	74	Northern Africa	Africa
3	Libya	6.5	14	2.6	75	72	77	Northern Africa	Africa
4	Morocco	32.6	30	2.3	72	70	74	Northern Africa	Africa
5	South Sudan	9.4	101	5.4	52	50	53	Northern Africa	Africa
6	Sudan	33.5	67	4.2	60	58	62	Northern Africa	Africa
7	Tunisia	10.8	20	2.1	75	73	77	Northern Africa	Africa
8	Benin	9.4	81	5.4	56	54	58	Western Africa	Africa
9	Burkina Faso	17.5	65	6.0	55	54	56	Western Africa	Africa
10	Cote d'Ivoire	20.6	73	4.6	55	54	56	Western Africa	Africa
11	Gambia	1.8	70	4.9	58	57	59	Western Africa	Africa
12	Ghana	25.5	47	4.2	64	63	65	Western Africa	Africa
13	Guinea	11.5	89	5.2	54	52	55	Western Africa	Africa
14	Guinea-Bissau	1.6	103	5.1	48	47	50	Western Africa	Africa
15	Liberia	4.2	83	5.4	56	55	57	Western Africa	Africa
16	Mali	16.0	97	6.3	51	50	52	Western Africa	Africa
17	Mauritania	3.6	74	4.5	58	57	60	Western Africa	Africa
18	Niger	16.3	81	7.1	58	56	60	Western Africa	Africa
19	Nigeria	170.1	77	5.6	51	48	54	Western Africa	Africa
20	Senegal	13.1	47	5.0	58	57	59	Western Africa	Africa
21	Sierra Leone	6.1	109	5.0	47	47	48	Western Africa	Africa
22	Togo	6.0	78	4.7	62	60	65	Western Africa	Africa
23	Burundi	10.6	63	6.4	58	57	60	Eastern Africa	Africa
24	Eritrea	5.6	51	4.5	61	59	63	Eastern Africa	Africa
25	Ethiopia	87.0	59	4.8	59	57	60	Eastern Africa	Africa

Comandi di base

- **ggplot**(*data*=NULL, ...)
  - Inizializza un oggetto ggplot da usare per le successive operazioni
  - Parametri:
    - **data** il dataframe con i dati da visualizzare
    - **...** altri argomenti come un aesthetic per definire le caratteristiche visive di base
- **layer**(*geom*, *geom_params*, *stat*, *stat_params*, *data*, *mapping*)
  - Aggiunge un nuovo livello ad un grafico esistente (mostra i dati)
  - Parametri:
    - **geom**, **geom_params**: il nome e i parametri per l'aggiunta al livello di un geom
    - **stat**, **stat_params**: il nome e i parametri di per l'aggiunta al livello di una trasformazione statistica
    - **data**, **mapping**: i dati da visualizzare e il relativo aesthetic (può essere ereditato)

Un primo grafico di esempio

```
library("ggplot2")
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(data=w, aes(x=le, y=tfr, color=area))
p + layer(geom="point", geom_params=list(size=4))
```



Scorciatoie

- Al posto del comando «layer» sono fornite una serie di scorciatoie per aggiungere direttamente geom e stat ad un grafico.
- Hanno tutte la sintassi:
  - **geom**_*nome*(*parametri*)
  - **stat**_*nome*(*parametri*)
- L'esempio precedente ad esempio diventa:

```
library("ggplot2")
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(data=w, aes(x=le, y=tfr, color=area))
p + geom_point(size=4)
```

geoms

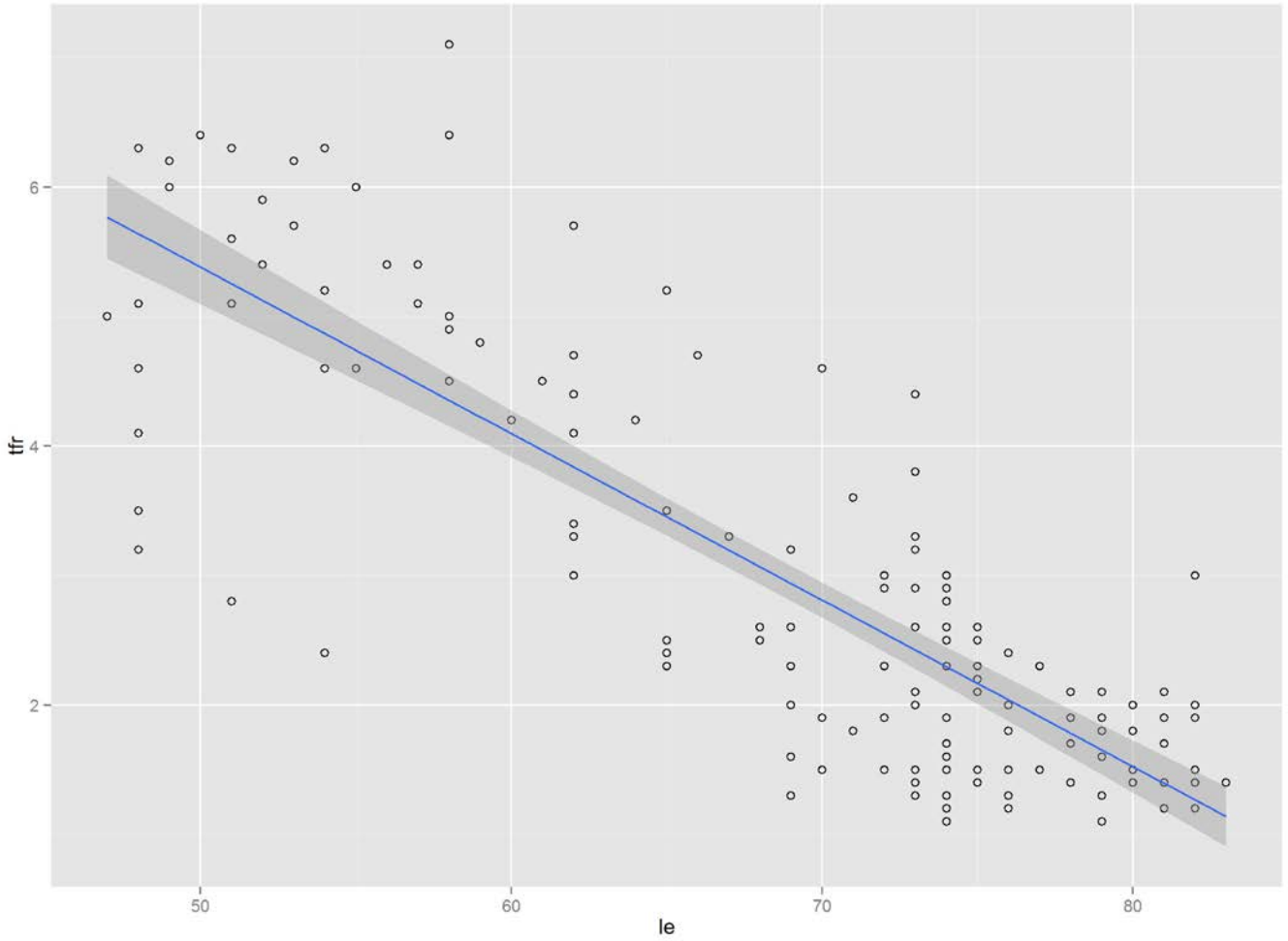
geom_abline	Line specified by slope and intercept.
geom_area	Area plot.
geom_bar	Bars, rectangles with bases on x-axis
geom_bin2d	Add heatmap of 2d bin counts.
geom_blank	Blank, draws nothing.
geom_boxplot	Box and whiskers plot.
geom_contour	Display contours of a 3d surface in 2d.
geom_crossbar	Hollow bar with middle indicated by horizontal line.
geom_density	Display a smooth density estimate.
geom_density2d	Contours from a 2d density estimate.
geom_dotplot	Dot plot
geom_errorbar	Error bars.
geom_errorbarh	Horizontal error bars
geom_freqpoly	Frequency polygon.
geom_hex	Hexagon bining.
geom_histogram	Histogram
geom_hline	Horizontal line.
geom_jitter	Points, jittered to reduce overplotting.
geom_line	Connect observations, ordered by x value.
geom_linerange	An interval represented by a vertical line.
geom_map	Polygons from a reference map.
geom_path	Connect observations in original order
geom_point	Points, as for a scatterplot
geom_pointrange	An interval represented by a vertical line, with a point in the middle.
geom_polygon	Polygon, a filled path.
geom_quantile	Add quantile lines from a quantile regression.
geom_raster	High-performance rectangular tiling.
geom_rect	2d rectangles.
geom_ribbon	Ribbons, y range with continuous x values.
geom_rug	Marginal rug plots.
geom_segment	Single line segments.
geom_smooth	Add a smoothed conditional mean.
geom_step	Connect observations by stairs.
geom_text	Textual annotations.
geom_tile	Tile plane with rectangles.
geom_violin	Violin plot.
geom_vline	Line, vertical.

stats

stat_abline	Add a line with slope and intercept.
stat_bin	Bin data.
stat_bin2d	Count number of observation in rectangular bins.
stat_bindot	Bin data for dot plot.
stat_binhex	Bin 2d plane into hexagons.
stat_boxplot	Calculate components of box and whisker plot.
stat_contour	Calculate contours of 3d data.
stat_density	1d kernel density estimate.
stat_density2d	2d density estimation.
stat_ecdf	Empirical Cumulative Density Function
stat_ellipse	Plot data ellipses.
stat_function	Superimpose a function.
stat_hline	Add a horizontal line
stat_identity	Identity statistic.
stat_qq	Calculation for quantile-quantile plot.
stat_quantile	Continuous quantiles.
stat_smooth	Add a smoother.
stat_spoke	Convert angle and radius to xend and yend.
stat_sum	Sum unique values. Useful for overplotting on scatterplots.
stat_summary	Summarise y values at every unique x.
stat_summary2d	Apply funciton for 2D rectangular bins.
stat_summary_hex	Apply funciton for 2D hexagonal bins.
stat_unique	Remove duplicates.
stat_vline	Add a vertical line
stat_ydensity	1d kernel density estimate along y axis, for violin plot.

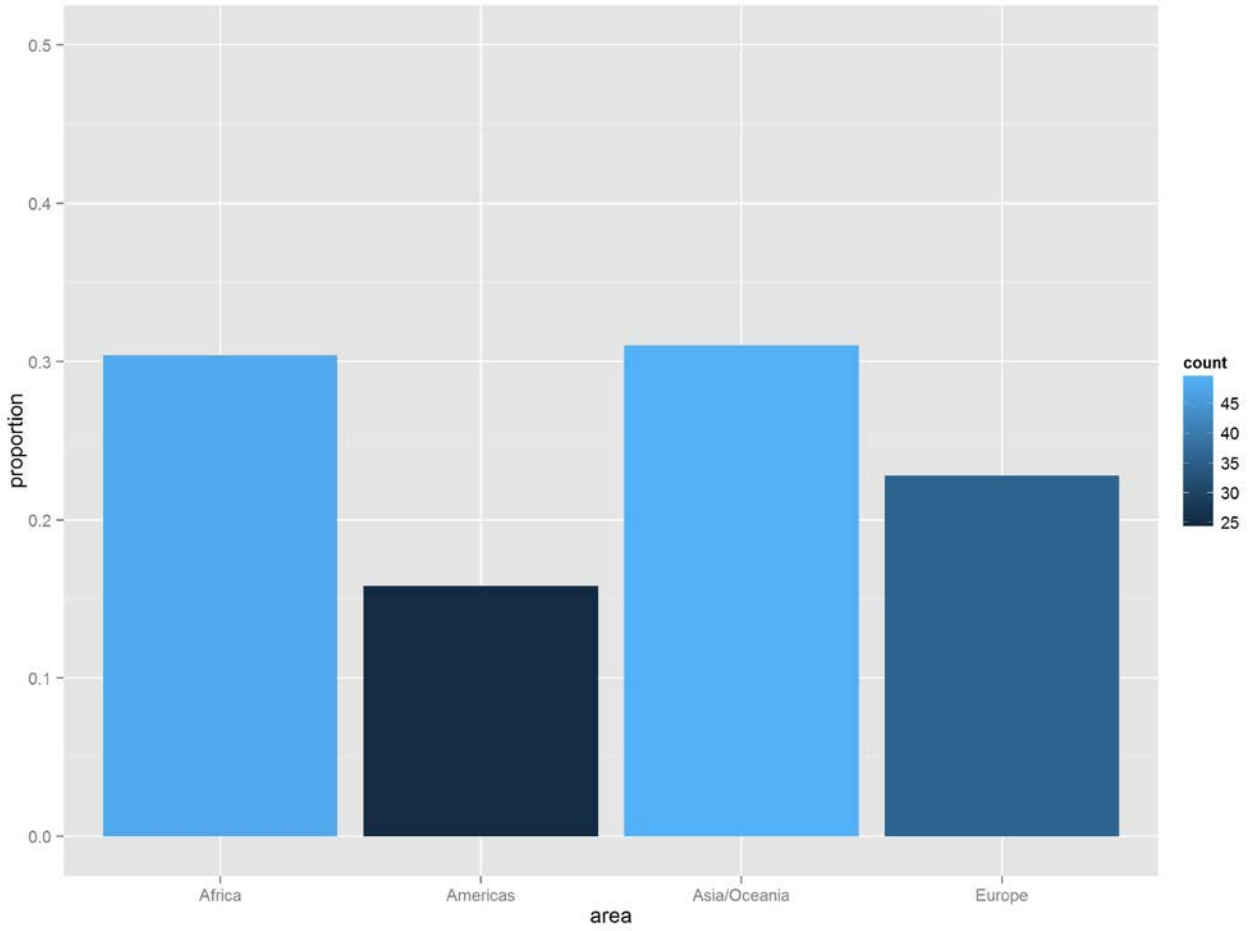
Esempio 1

```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(data=w, aes(x=le, y=tfr))
p + geom_point(shape=1) + stat_smooth(method="lm", se=TRUE)
```



Esempio 2

```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(data=w, aes(x=area))
p +
  stat_bin(aes(y = ..count../sum(..count..), fill=..count..)) +
  ylab("proportion") +
  ylim(0,.5)
```

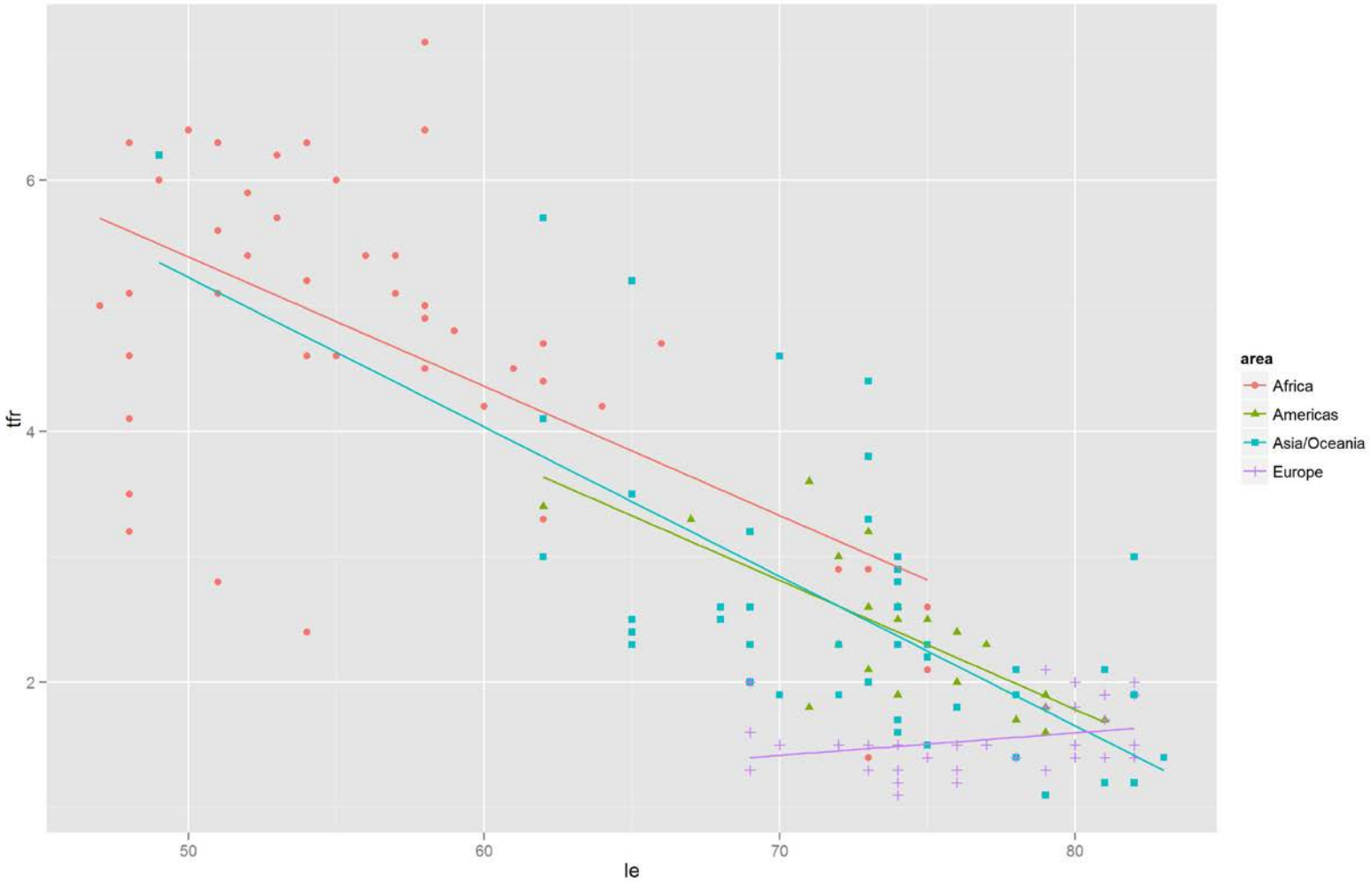


Aesthetics

- **aes**(*x*, *y*, ...)
  - Descrive le caratteristiche visuali che rappresentano i dati. Ogni layer *eredita* le caratteristiche dell'oggetto plot a meno di non modificarle esplicitamente.
  - Parametri:
    - **x**: posizione x (o variabile dei dati)
    - **y**: posizione y (o variabile dei dati)
    - **color**: il colore del contorno (o variabile dei dati)
    - **fill**: il colore di riempimento (o variabile dei dati)
    - **alpha**: percentuale di trasparenza (o variabile dei dati)
    - **size**: dimensione in millimetri (o variabile dei dati)
    - **linetype**: tipo di linea
      - blank, solid, dashed, dotted, dotdash, longdash, twodash

Esempio 3

```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(data=w, aes(x=le, y=tfr, color=area))
p +
  geom_point(aes(shape=area)) +
  geom_smooth(method="lm",se=FALSE)
```

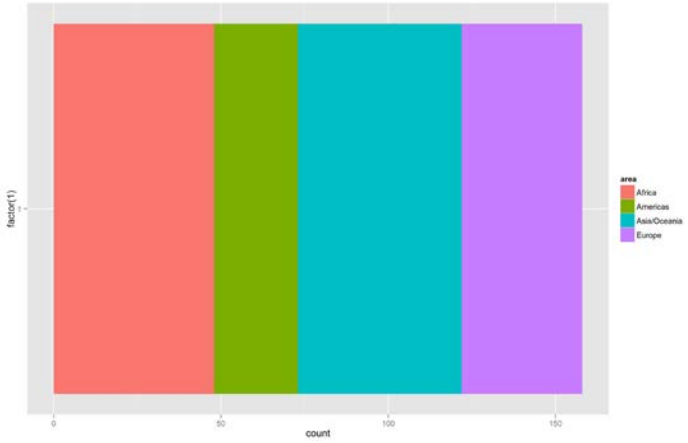


Coordinate System

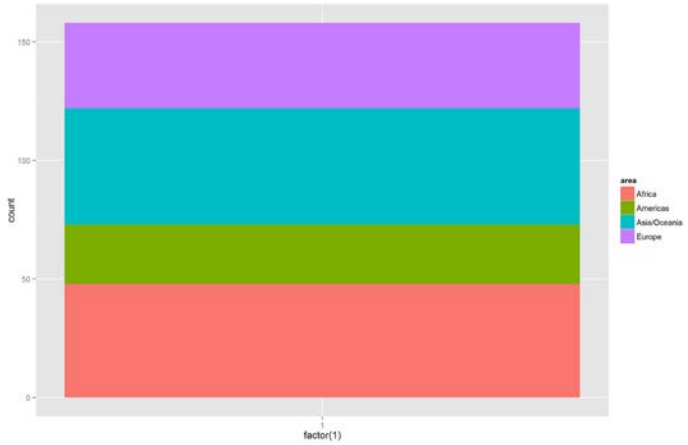
- Permette di gestire con facilità le più comuni tipologie di coordinate nei grafici.
- Ognuna delle istruzioni va posta dopo aver creato il layer su cui le coordinate vanno applicate.
- Esempi:
  - *Cartesiane*
  - *Polari*
  - *Mappe*
  - *Cartesiane trasfrormate (logaritmiche)*

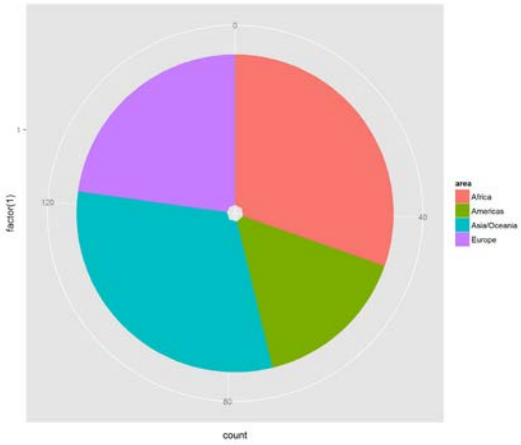
coord_cartesian	Cartesian coordinates.
coord_expand_defaults	Set the default expand values for the scale, if NA
coord_equal	Cartesian coordinates with fixed relationship between x and y scales.
coord_flip	Flipped cartesian coordinates.
coord_map	Map projections.
coord_polar	Polar coordinates.
coord_quickmap	Cartesian coordinates with an aspect ratio approximating Mercator projection.
coord_trans	Transformed cartesian coordinate system.

Esempio 4




```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
p <- ggplot(w, aes(x=factor(1), fill=area))
p + geom_bar()
p + geom_bar() + coord_flip()
p + geom_bar() + coord_polar(theta="y")
```





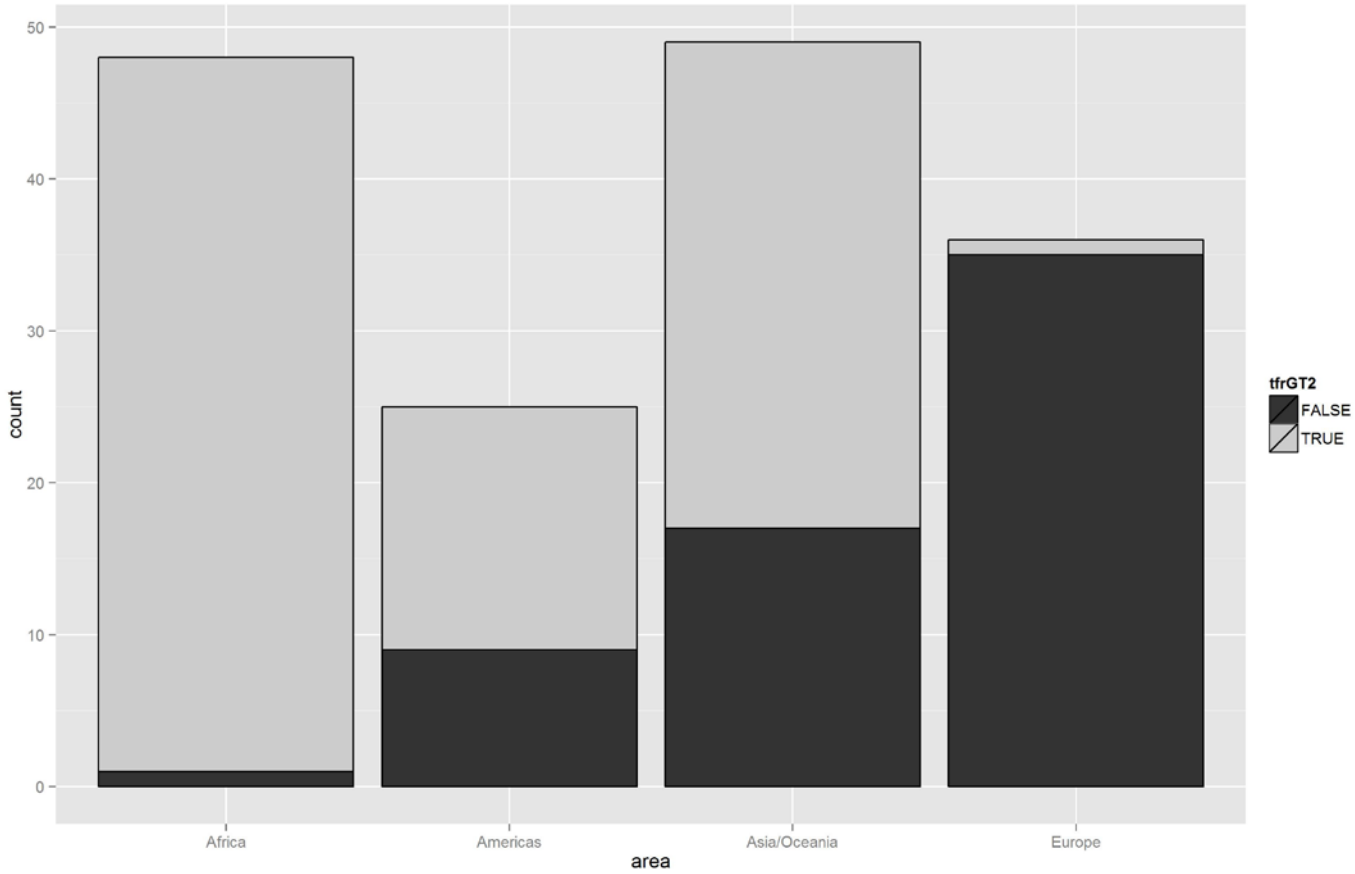
Scales

- Controllano il mapping dai dati agli aesthetics, ovvero prendono i dati e li convertono in qualcosa che può essere visualizzato.
- Fornisce anche un supporto automatizzato alle legende per facilitare la comprensione dei grafici.

scale_alpha	Alpha scales.
scale_area	Scale area instead of radius (for size).
scale_color_brewer	Sequential, diverging and qualitative colour scales from colorbrewer.org
scale_x_continuous	Continuous position scales (x & y).
scale_x_date	Position scale, date
scale_x_datetime	Position scale, date
scale_x_discrete	Discrete position.
scale_color_continuous	Smooth gradient between two colours
scale_color_gradient2	Diverging colour gradient
scale_color_gradientn	Smooth colour gradient between n colours
scale_color_grey	Sequential grey colour scale.
scale_color_discrete	Qualitative colour scale with evenly spaced hues.
scale_alpha_identity	Use values without scaling.
scale_linetype	Scale for line patterns.
scale_alpha_manual	Create your own discrete scale.
scale_shape	Scale for shapes, aka glyphs.
scale_size	Size scale.
scale_size_area	Scale area instead of radius, for size.

Esempio 5

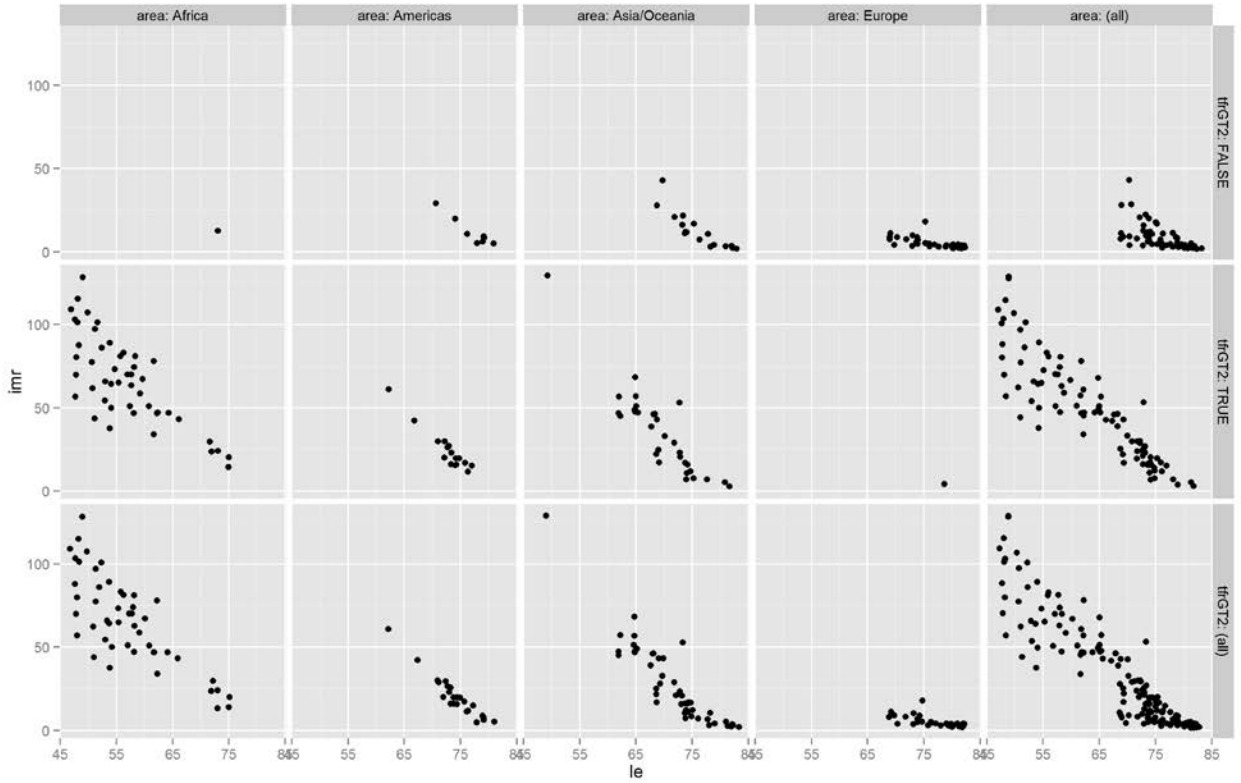
```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
w$tfrGT2 <- w$tfr > 2
p <- ggplot(data=w, aes(x=area, fill=tfrGT2))
p +
  geom_bar(color="black") +
  scale_fill_grey()
```



Facets

- Suddividono i dati in sottoinsiemi e plottano ognuno di essi su un pannello differente

```
w <- read.csv(file="WDS2012.csv", head=TRUE, sep=",")
w$tfrGT2 <- w$tfr > 2
p <- ggplot(data=w, aes(x=le, y=imr)) + geom_jitter()
p + facet_grid(tfrGT2 ~ area, labeller="label_both", margins=TRUE)
```



Salvare un plot

- **ggsave**(*filename*, *scale*, *width*, *height*, *units*, *dpi*)
  - Salva l'ultimo plot eseguito in un file.
  - Tipi supportati: *ps, eps, tex, pdf, jpg, tiff, png, bmp, svg*
- **Parametri**:
  - **filename**: il nome del file di output
  - **scale**: fattore di scala usato nel renderizzare l'output
  - **units**: unità di misura usata per altezza e largezza (Default: **in**, Supportati: **in, cm, mm**)
  - **width**, height: larghezza e altezza dell'immagine output
  - **dpi**: per le immagini rasterizzate (jpg, tiff, png, bmp, pdf) indica la risoluzione dell'immagine (dots per inch)

Libri per approfondire

- The **R Graphics Cookbook** by *Winston Chang* provides a set of recipes to solve common graphics problems. Read this book if you want to start making standard graphics with ggplot2 as quickly as possible.
- **ggplot2: Elegant Graphics for Data Analysis** by *Hadley Wickham* describes the theoretical underpinnings of ggplot2 and shows you how all the pieces fit together. This book helps you understand the theory that underpins ggplot2, and will help you create new types of graphic specifically tailored to your needs. You can read sample chapters and download the book code from the book website.

THE END