Una Guida all'utilizzo dell'Ambiente Statistico R

Angelo M. Mineo
Dipartimento di Scienze Statistiche e Matematiche "S. Vianelli"
Università degli Studi di Palermo

Prefazione

R, ormai da anni, costituisce, nell'ambito dei software di tipo statistico, una valida alternativa ai più diffusi ambienti statistici. In particolare, **R** è un ambiente statistico distribuito gratuitamente in Internet sotto licenza GPL e sviluppato da un team di ricercatori in ambito statistico e informatico di fama mondiale. Esistono versioni di **R** per diverse piattaforme e quindi anche per i più diffusi sistemi operativi nell'ambito dei Personal Computer, cioè **Windows**, **MAC** e **Linux**. Inoltre, **R**, costituendo un vero e proprio ambiente di programmazione, permette una elevatissima flessibilità nell'implementazione di funzioni di calcolo e di rappresentazione grafica statistica.

R viene inizialmente scritto da Ross Ihaka e Robert Gentelman, del Dipartimento di Statistica dell'Università di Auckland, Nuova Zelanda. In seguito, un folto gruppo di persone comincia a dare il suo contributo, dando vita così all'**R Core Team**, che dal 1997 si occupa dei codici sorgenti di **R**. Quando viene progettato, **R** è sviluppato per sistemi **Unix**. Adesso può essere installato su macchine con diverse architetture e con diversi sistemi operativi.

R è un software *open source*, cioè il suo codice sorgente viene distribuito liberamente, costituendo un sistema aperto a chiunque voglia aumentarne le possibilità di utilizzo e di calcolo. Il codice sorgente è scaricabile da internet dal sito di **The R Project for Statistical Computing**, il cui indirizzo internet è `http://www.r-project.org` e dove è possibile trovare ogni tipo di supporto per l'utilizzo e lo sviluppo di **R**. Assieme a **The R Project for Statistical Computing** vi è il **Comprehensive R Archive Network** (**CRAN**), il cui indirizzo internet è `http://cran.r-project.org`, che è dedicato specificatamente al download del software e di tutta la documentazione collegata.

Questo manuale è rivolto in particolare agli studenti del Laboratorio di Statistica del Corso di Laurea in Statistica e Informatica per la Gestione e l'Analisi dei Dati della Facoltà di Economia di Palermo e in generale a tutti quegli studenti che nei loro corsi si trovano a dovere utilizzare un software di tipo statistico.

Per questo manuale si è fatto esplicito riferimento al manuale *An Introduction to R* che viene fornito assieme al software. E' chiaro, comunque, che qualunque errore presente nel testo è dovuto solo all'autore di questo manuale. Altri riferimenti, che sono risultati utili per la stesura e che comunque sono consigliati per l'apprendimento di **R**, sono riportati in bibliografia.

Ogni suggerimento o segnalazione di errore nel testo è chiaramente benvenuto.

Autore

Angelo M. Mineo
Professore Associato di Statistica
Dipartimento di Scienze Statistiche e Matematiche "Silvio Vianelli"
Università di Palermo
e-mail: elio.mineo@dssm.unipa.it.
Home page: http://dssm.unipa.t/elio.

Indice

Capitolo 1

Per cominciare

1.1 Introduzione

R è un insieme integrato di risorse software per la manipolazione di dati, il calcolo e la visualizzazione di grafici. Ha tra le altre cose

- un efficace manipolatore di dati e un altrettanto efficace dispositivo di memorizzazione;
- un insieme di operatori per i calcoli su array, in particolare matrici;
- una grande, coerente, integrata raccolta di strumenti intermedi per l'analisi dei dati;
- risorse grafiche per l'analisi dei dati con visualizzazione direttamente sul computer o su carta attraverso stampante;
- un ben sviluppato, semplice ed efficace linguaggio di programmazione che include istruzioni condizionali, loop, funzioni ricorsive definite dall'utente e strumenti di input/output.

Il termine "ambiente" è inteso caratterizzarlo come un sistema pienamente pianificato e coerente, piuttosto che un sistema che si incrementa attraverso parti molto specifiche e poco flessibili, come accade frequentemente con altri software statistici.

R è stato inizialmente scritto da Ross Ihaka e Robert Gentelman, del Dipartimento di Statistica dell'Università di Auckland, Nuova Zelanda, ed è considerato un clone o un dialetto di **S**, linguaggio di programmazione statistico sviluppato nel 1980 nei *Bell Labs*, noti tra l'altro per lo sviluppo del sistema operativo **UNIX** e del linguaggio **C**. In particolare, **S** è stato sviluppato da un gruppo di ricercatori guidati da John Chambers, che oggi fa parte anche dell'**R Core Team**, cioè del gruppo principale di sviluppatori di **R** e che nel 1998 ha ricevuto il premio ACM Software System Award proprio per il linguaggio **S**[1].

C'è un'importante differenza filosofica tra **R** (e quindi anche **S**) e gli altri principali ambienti statistici. In **R** un'analisi statistica normalmente è fatta attraverso una serie di passi, con risultati intermedi che sono immagazzinati in

[1] vedi il seguente URL http://www.acm.org/awards/ssaward.html.

oggetti. Così mentre **SAS** o **SPSS** daranno una produzione copiosa di risultati relativi, ad esempio, ad un'analisi di regressione, **R** darà la minima produzione, immagazzinando i risultati in oggetti che possono essere richiamati da altre funzioni di **R**.

1.2 L'Utilizzo dell'Help

Una delle prime cose che è bene conoscere quando per la prima volta si utilizza un software è come cercare aiuti per il corretto funzionamento del software.

R ha un sistema di help interno. Per avere informazioni su qualche specifico nome di funzione, per esempio `solve()`, il comando da utilizzare è `help(solve)`. Un'alternativa a questo comando è `?solve`.

Per una caratteristica specificata attraverso caratteri speciali o per parole riservate, come ad esempio `if`, l'argomento deve essere compreso tra doppie virgolette o singole virgolette, rendendolo in questo modo un vettore stringa: `help("[[")`.

Su molte versioni di **R** è disponibile un help in formato HTML che si richiama con `help.start()` che lancerà il *browser* predefinito.

Se poi si vogliono ricavare maggiori informazioni su come utilizzare l'help basta digitare il comando `?help`.

1.3 I comandi di R

Tecnicamente **R** è un linguaggio basato su espressioni dalla sintassi semplice. E' *case sensitive*, cioè `A` e `a` sono due simboli diversi e si riferiscono a due variabili differenti.

I comandi elementari consistono o di espressioni, o di assegnazioni. Ogni comando viene immesso nella riga di comando che comincia con il simbolo di prompt, che solitamente è `>`. I comandi sono separati o da punto e virgola (`;`), o da un carattere di nuova linea (si immette un carattere di nuova linea digitando il tasto `Invio`). Se un comando non è completo alla fine della linea, **R** darà un prompt differente che per default è il carattere

```
+
```

sulla linea seguente e continuerà a leggere l'input finché il comando non è sintatticamente completo.

R fornisce la possibilità di richiamare e rieseguire i comandi. I tasti freccia verticale, $\uparrow$ e $\downarrow$, sulla tastiera possono essere utilizzati per scorrere avanti e indietro i comandi già immessi. Appena trovato il comando che interessa, lo si può modificare, ad esempio, con i tasti freccia orizzontali, immettendo nuovi caratteri o cancellandone altri.

1.4 Permanenza dei dati e rimozione di oggetti

Le entità che **R** crea e manipola sono note come oggetti. Questi possono essere variabili, array di numeri, caratteri, stringhe, funzioni, o più in generale strutture costruite a partire da tali componenti.

Durante una sessione di **R** gli oggetti sono creati e memorizzati attraverso opportuni nomi. I nomi possono contenere un qualunque carattere alfanumerico e come carattere speciale il punto[2]; in ogni caso non possono mai iniziare con un carattere numerico e non è bene che coincidano con nomi di funzioni o di operatori.

Il comando:

```
> objects()
```

(oppure `ls()`) può essere utilizzato per visualizzare i nomi degli oggetti che sono in quel momento memorizzati in **R**. L'insieme degli oggetti che in quel momento sono memorizzati si chiama spazio di lavoro (*workspace*).

Per eliminare oggetti dallo spazio di lavoro è disponibile la funzione `rm()`; ad esempio

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

cancella tutti gli oggetti indicati entro parentesi. Se si vogliono eliminare tutti gli oggetti presenti nello spazio di lavoro, il comando da utilizzare è il seguente:

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

Tutti gli oggetti creati durante una sessione di **R** possono essere permanentemente immagazzinati in un file per essere utilizzati in futuro in altre sessioni. Alla fine di ogni sessione **R** dà l'opportunità di salvare tutti gli oggetti disponibili. Se si sceglie di salvarli, gli oggetti sono scritti in un file chiamato `.RData` nella cartella corrente.

Quando **R** è riavviato successivamente, ricarica lo spazio di lavoro da questo file, con la storia dei comandi associati, che si trovano memorizzati nel file `.RHistory`, salvato in precedenza assieme al file `.RData`. Il file `.RHistory` può risultare, quindi, molto utile per richiamare, con i tasti freccia su e giù, tutti i comandi digitati nella precedente sessione di lavoro. Si raccomanda inoltre di utilizzare cartelle di lavoro separate per analisi diverse, per evitare che nomi come `x` o `y`, che vengono utilizzati molto spesso, possano portare a risultati in quel contesto privi di senso.

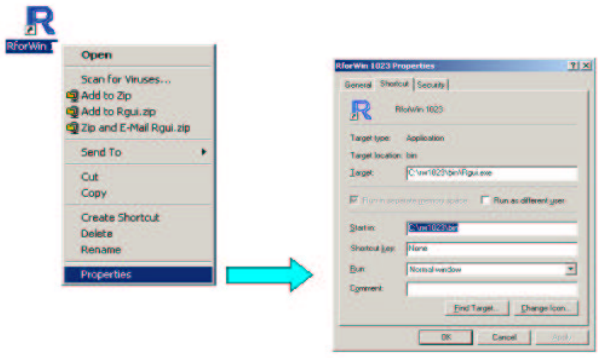
1.5 Suggerimenti su come configurare R sotto Windows

Quelli presentati in questo paragrafo sono alcuni suggerimenti che possono aiutare ad evitare inconvenienti quando si lavora con **R** sotto **Windows**. Ci si renderà conto che una volta presa confidenza con il software, non è detto che tutti concordino sulle procedure qui indicate. Si tratta solo di alcune avvertenze che si ritengono utili.

[2] Questa è una differenza sostanziale con altri linguaggi di programmazione che solitamente ammettono il simbolo di sottolineato, come carattere speciale. In **R** il simbolo di sottolineato è uno dei modi per indicare l'operatore di assegnazione. Attenzione, quindi, a non utilizzarlo all'interno di nomi. E' previsto, comunque, a partire dalla versione 1.8.0 che il carattere di sottolineato non venga più utilizzato come operatore di assegnazione, liberandolo quindi per l'utilizzo come carattere speciale.

- **Scrivere con un editore di testo.**
  Se si vuole evitare di dover digitare nuovamente una serie di comandi, perché magari ci si rende conto di aver commesso un errore 15 linee prima, una buona prassi può essere quella di scrivere tutto il codice in **R** con un editore di testo, ad esempio *Blocco note*, e quindi copiarlo e incollarlo nella *consolle* di **R**. In questo modo, quando si vogliono correggere linee scritte in precedenza, questo è molto facile da fare in un editore di testo. Poi tutto quello che si deve fare è re-incollare le linee opportune nella *consolle* di **R**.

- **Configurare opportunamente la cartella di lavoro.**
  Nell'icona che compare nel *Desktop* per avviare **R**, è possibile cambiare la cartella di lavoro. Per fare questo si clicca con il tasto destro del mouse sull'icona, scegliendo la voce *Proprietà* sul menù che compare (vedi la figura seguente).



Nella finesta *Collegamento* nel campo *Da*, si cambia la cartella con quella che si vuole; per esempio,

```
C:\Cartella Personale\Statistica laboratorio\Compito 1
```

Appena è stata configurata la cartella di lavoro, si possono leggere file da quella cartella senza dovere specificare il percorso assoluto. In altre parole, se si ha un file di dati chiamato `compito.txt` nella cartella di lavoro corrente, bisogna digitare, ad esempio, solo il comando:

```
> traffic <- read.table("compito.txt", header = TRUE)
```

per potere caricare i dati contenuti nel file. Si parlerà del comando `read.table()` più avanti in questo manuale.

- **Commentare il codice.**
  Chi ha esperienza di programmazione, ad esempio in **C**, **Java**, **S-plus** o **SAS**, sa che i commenti sono molto importanti. Fondamentalmente i commenti sono parole in linguaggio naturale (nel nostro caso italiano), che permettono agli utilizzatori di capire il flusso logico del codice e a chi lo ha scritto di ricordare immediatamente il perché di determinate istruzioni. Questo risulta particolarmente importante quando si tenta di scrivere le proprie funzioni in **R**; infatti, non solo i commenti aiuteranno le altre persone a capire quello che si sta facendo, ma anche permetterà di capirlo all'autore che dopo tempo riprenderà il codice. I commenti aiutano anche quando si tenta di correggere errori nel codice (operazione di *debugging*). In **R**, le parole dopo `#` sono considerate commenti e sono ignorate; ad esempio:

  ```
  # Questo è un commento
  ```

  In **R** non è consentito il commento su più linee, cioè viene considerata un commento solo la parte della linea che segue il simbolo `#`.

- **Usare il file .Rprofile.**
  Si possono digitare alcuni comandi di **R** all'interno del file `.Rprofile`, che è uno dei primi file che **R** leggerà ogni volta che parte e che solitamente si trova nella seguente directory:

  ```
  HOME/library/base
  ```

  dove `HOME` sta a rappresentare la directory iniziale in cui è installato **R**. Può essere, per esempio, una buona idea caricare alcuni packages utilizzati di frequente. Un esempio di `.Rprofile` è il seguente:

  ```
  library(mva) # Carica la libreria mva
  library(MASS) # Carica la libreria MASS (VR)
  library(ts) # Carica la libreria di serie storiche
  ```

  Comunque non è una buona idea tentare di caricare molti packages attraverso il file `.Rprofile`, perché questo rallenterebbe significativamente il processo di avvio di **R**.

Capitolo 2

L'oggetto base di R: il vettore

2.1 Vettori e assegnazioni

R opera su strutture di dati; la più semplice di tali strutture è il vettore numerico, che è una singola entità che consiste in un insieme ordinato di numeri[1]; ad esempio:

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Questa è un'istruzione di assegnazione che utilizza la funzione `c()` che in questo contesto può prendere un numero arbitrario di argomenti e il cui valore è un vettore ottenuto concatenando i suoi argomenti.

L'operatore di assegnazione è il seguente:

```
<-
```

e non quello solitamente utilizzato in altri linguaggi di programmazione, cioè

```
=
```

Anche il carattere `_` può essere utilizzato come sinonimo di `<-`, ma non se ne consiglia l'uso per evitare di ottenere un codice non facilmente leggibile (sull'utilizzo di `_` vedi comunque la nota 2 del Capitolo 1).

L'assegnazione può anche essere fatta utilizzando la funzione `assign()`; la seguente istruzione corrisponde esattamente alla precedente:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

Le assegnazioni possono anche essere fatte puntando la freccia nell'altra direzione:

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

ma non se ne consiglia l'uso.

Se un'espressione è utilizzata come un comando completo, il valore dell'espressione è visualizzato, ma viene perso; ad esempio, il comando:

[1] Questo è il motivo per cui spesso ci si riferisce ad **R** come ad un linguaggio vettorizzato.

```
> 1/x
```

produrrà solo la visualizzazione del risultato. L'ulteriore assegnazione

```
> y <- c(x, 0, x)
```

creerebbe, invece, un vettore `y` di 11 elementi che consistono di due copie di `x` con uno zero in mezzo, il cui contenuto, però, non viene visualizzato. Per poterlo visualizzare bisogna dare l'ulteriore comando:

```
> y
```

2.2 Vettori aritmetici

I vettori possono essere utilizzati in espressioni numeriche, con le operazioni eseguite elemento per elemento. I vettori presenti nella stessa espressione possono essere di lunghezza diversa. Se è così, il valore dell'espressione è un vettore con la stessa lunghezza del vettore più lungo presente in quella espressione. I vettori con meno elementi sono riconsiderati tante volte fino ad arrivare alla stessa lunghezza del vettore più lungo. Ad esempio

```
> v <- 2*x + y + 1
```

dà un nuovo vettore `v` di lunghezza 11 costruito sommando, elemento per elemento, `2*x` ripetuto 2.2 volte, `y` ripetuto una sola volta e `1` ripetuto 11 volte. Gli operatori aritmetici elementari sono `+`, `-`, `*`, `/` e `^` per l'elevamento a potenza. In aggiunta sono disponibili le più comuni funzioni matematiche: `log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()`, `max()`, `min()` e così via. Altre funzioni di uso comune sono: `range()` che restituisce un vettore `c(min(x), max(x))`; il massimo o il minimo in parallelo, `pmax()` o `pmin()`, che restituisce il massimo o il minimo tra i corrispondenti elementi dei vettori; `which()` dà l'indice per cui la condizione all'interno del comando è `TRUE`; `which.min(x)` e `which.max(x)` che restituiscono la posizione (l'indice) del primo valore minimo e massimo di `x`, rispettivamente; `sort()` che restituisce un vettore ordinato e `order()` che restituisce il vettore di indici corrispondenti all'ordinamento del vettore passato come argomento; `length(x)` che restituisce il numero di elementi di `x`; `sum(x)` che dà la somma degli elementi di `x`, mentre `prod(x)` dà il loro prodotto. Due funzioni statistiche sono `mean(x)`, media aritmetica, e `var(x)`, varianza campionaria. Se l'argomento di `var()` è una matrice $n \times p$, si ottiene la matrice di varianza e covarianza, considerando le righe come vettori di campioni indipendenti a p valori. Queste funzioni statistiche verranno riprese con maggiore dettaglio in un prossimo capitolo.

E' possibile in **R** fare operazioni che coinvolgono i numeri complessi. Per lavorare con numeri complessi, bisogna comunque esplicitare la parte complessa. Quindi

```
> sqrt(-17)
```

darà un errore, ma

```
> sqrt(-17+0i)
```

darà il valore complesso, risultato di questo comando, dato che l'unità immaginaria si indica con `i`.

2.3 Generazione di sequenze regolari

R ha un ampio numero di comandi per generare sequenze di numeri. Ad esempio, `c(1:10)` è il vettore `c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`. L'operatore due punti (`:`) ha la più alta priorità tra gli operatori per cui, ad esempio `c(2*1:5)` è il vettore `c(2, 4, 6, 8, 10)`. L'espressione `c(30:1)` può essere utilizzata per generare una sequenza all'indietro.

Un'altra funzione che genera sequenze è `seq()`. Questa funzione può avere 5 argomenti: i primi due rappresentano l'inizio (`from`) e la fine (`to`) della sequenza, il terzo specifica l'ampiezza del passo (`by`), il quarto la lunghezza della sequenza (`length`) e infine il quinto (`along`), che se utilizzato deve essere l'unico parametro presente, è il nome di un vettore, ad esempio `x`, creando in tal modo la sequenza `1, 2, ...,length(x)`. Esempi di utilizzo della funzione `seq()` sono i seguenti:

```
> seq(2,10)
[1]  2  3  4  5  6  7  8  9 10
> seq(from=1, to=10)
[1]  1  2  3  4  5  6  7  8  9 10
> seq(-5, 5, by=2.5)
[1] -5.0 -2.5  0.0  2.5  5.0
> seq(length=5, from=-5, by=2.5)
[1] -5.0 -2.5  0.0  2.5  5.0
```

Altra funzione utilizzata per generare sequenze è `rep()` che può essere utilizzata per replicare un oggetto in vari modi. Ad esempio:

```
> s5 <- rep(x, times=5)
```

metterà 5 copie di `x` in `s5`.

2.4 Vettori logici

I vettori logici sono quei vettori generati da condizioni logiche. Ad esempio:

```
> temp <- x > 13
```

genera `temp` come un vettore della stessa lunghezza di `x` con valori `FALSE` corrispondenti ad elementi di `x` che non soddisfano la condizione e `TRUE` altrimenti. Gli operatori logici sono `<`, `<=`, `>`, `>=`, `==` per l'uguaglianza e `!=` per l'ineguaglianza. Inoltre per espressioni logiche si ha l'operatore `&` per l'AND, l'operatore `|` per l'OR e l'operatore `!` per la negazione. I vettori logici possono essere usati all'interno di espressioni aritmetiche, nel qual caso `FALSE` assume il valore `0` e `TRUE` assume il valore `1`. Ci sono, però, situazioni in cui non c'è questa esatta corrispondenza, come ad esempio nel caso di dati mancanti.

2.5 Dati mancanti

Quando si è in presenza di un dato mancante, **R** assegna il valore speciale `NA`, che sta per *Not Available*. In generale, un'operazione su un `NA` dà come risultato un `NA`. E' da notare come `NA` non sia un vero e proprio valore, ma è solo un

identificatore: questo può portare a risultati indesiderati, soprattutto quando si effettuano dei confronti logici. La funzione `is.na(x)`[2] dà un vettore logico della stessa ampiezza di `x` con valore `TRUE` se e solo se il corrispondente elemento è `NA`.

Un altro valore mancante è prodotto da calcoli numerici, ad esempio:

```
> 0/0
```

che dà come risultato un `NaN`, cioè *Not a Number*.

La funzione `is.na(x)` dà come risultato `TRUE` in presenza di entrambi i valori `NA` e `NaN`, mentre `is.nan(x)` dà come risultato `TRUE` solo in presenza di `NaN`.

2.6 Vettori di caratteri

I vettori di caratteri sono spesso utilizzati in **R**, ad esempio come etichette nei grafici. Di solito sono indicati da una sequenza di caratteri delimitati da doppie virgolette. I vettori di caratteri possono essere concatenati in un vettore attraverso la funzione `c()`.

La funzione `paste()` prende un numero arbitrario di argomenti e li concatena in un'unica stringa, separandoli con degli spazi, se non specificato altrimenti con l'argomento `sep`; ad esempio:

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

produce il seguente vettore di caratteri:

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10").
```

2.7 Vettori di indici; selezione e modifica di sottoinsiemi di un insieme di dati

Sottoinsiemi degli elementi di un vettore possono essere selezionati collegando al nome del vettore un vettore di indici in parentesi quadre. Tali vettori di indici possono essere di quattro tipi distinti:

1. **Un vettore di interi positivi.**
   In questo caso i valori nel vettore di indici devono essere contenuti nell'insieme {1, 2, ..., `length(x)`}. Per esempio `x[6]` sarà la sesta componente del vettore `x`.

2. **Un vettore di interi negativi.**
   Un tale vettore di indici specifica i valori da escludere piuttosto che quelli da includere. Quindi

   ```
   > y <- x[-(1:5)]
   ```

   assegna ad `y` tutti gli elementi di `x` esclusi i primi cinque.

[2] Le funzioni del tipo *is.nome()* vengono utilizzate in **R** per vedere se l'oggetto considerato è del **modo** specificato in *nome*; le funzioni del tipo *as.nome()* vengono utilizzate, invece, per trasformare l'oggetto considerato nel **modo** specificato in *nome*. Si rimanda al paragrafo 4.6 per il significato di **modo**.

3. **Un vettore logico.**
   In questo caso il vettore di indici deve essere della stessa lunghezza del vettore da cui devono essere selezionati gli elementi. I valori corrispondenti a `TRUE` nel vettore di indici sono selezionati, quelli corrispondenti a `FALSE` sono omessi. Per esempio:

   ```
   > y <- x[!is.na(x)]
   ```

   crea un oggetto `y` che conterrà i valori non-mancanti di `x`, nello stesso ordine.

4. **Un vettore di stringhe di caratteri.**
   Questa possibilità si applica solo se un oggetto ha stringhe per identificare le sue componenti e in questo caso l'utilizzo è simile a quello dei numeri interi positivi visti al punto 1. Un possibile esempio è il seguente:

   ```
   > frutta <- c(5, 10, 1, 20)
   > names(frutta) <- c("arancia", "banana", "mela", "pesca")
   > pranzo <- frutta[c("mela","arancia")]
   > pranzo
      mela arancia
         1       5
   ```

2.8 Fattori ordinati e non ordinati

Un fattore è un oggetto vettore che può essere utilizzato per specificare una classificazione discreta delle componenti di altri vettori della stessa lunghezza. In **R** sono presenti sia fattori ordinati che fattori non ordinati. In buona sostanza, un fattore non ordinato può essere visto come una variabile qualitativa sconnessa, un fattore ordinato come una variabile qualitativa ordinabile. Vediamo il loro utilizzo con un esempio; supponiamo di avere 30 impiegati siciliani e indichiamo nel seguente vettore `prov` le loro province di provenienza:

```
> prov <- c("PA", "PA", "AG", "RG", "ME", "CT", "CT", "SR", "TP",
"EN", "CL", "ME", "AG", "SR", "AG", "RG", "AG", "CT", "ME", "ME",
"RG", "TP", "PA", "CL", "TP", "EN", "TP", "EN", "SR", "PA")
```

Creiamo un fattore utilizzando la funzione `factor()`:

```
> provf <- factor(prov)
```

Se si visualizza `provf` si può notare come la visualizzazione sia leggermente diversa dagli oggetti di tipo vettore visti finora:

```
> provf
[1]   PA  PA  AG  RG  ME  CT  CT  SR  TP  EN  CL  ME  AG  SR  AG
[16]  RG  AG  CT  ME  ME  RG  TP  PA  CL  TP  EN  TP  EN  SR  PA
Levels: AG CL CT EN ME PA RG SR TP
```

Per determinare i livelli di un fattore si può utilizzare la funzione `levels()`:

```
> levels(provf)
[1] "AG" "CL" "CT" "EN" "ME" "PA" "RG" "SR" "TP"
```

Supponiamo ora di avere in un altro vettore i redditi annui espressi in Euro degli stessi impiegati

```
> redditi <- c(16000, 14900, 14000, 16100, 16400, 16000, 15900,
15400, 16200, 16900, 17000, 14200, 15600, 16100, 16100, 16100,
15800, 15100, 14800, 16500, 14900, 14900, 14100, 14800, 15200,
14600, 15900, 14600, 15800, 14300)
```

Per calcolare il reddito medio per provincia di appartenenza possiamo utilizzare la funzione `tapply()`:

```
> redmedi <- tapply(redditi, provf, mean)
> redmedi
      AG       CL       CT       EN       ME       PA       RG
15375.00 15900.00 15666.67 15366.67 15475.00 14825.00 15700.00
      SR       TP
15766.67 15550.00
```

La funzione `tapply()` è usata per applicare una funzione, nel nostro caso `mean()`, a ogni gruppo di elementi del primo argomento, nel nostro caso `redditi`, definito dai livelli del secondo argomento, qui `provf`, come se fossero strutture vettore separate. Il risultato è una struttura della stessa lunghezza del numero di livelli del fattore. Alla stessa famiglia di `tapply()` appartengono le funzioni `apply()`, `lapply()` e `sapply()`. Rimandiamo all'help per l'utilizzo specifico di queste funzioni.

Quando abbiamo a che fare con fattori ordinati, i livelli dei fattori sono memorizzati in ordine alfabetico, o nell'ordine in cui sono stati specificati al fattore, se sono stati specificati esplicitamente. Talvolta i livelli avranno un ordinamento naturale che vogliamo registrare e vogliamo che la nostra analisi statistica ne faccia uso. La funzione `ordered()`, applicata ai livelli dei fattori, crea tali fattori ordinati.

Capitolo 3

Altri tipi di oggetti: array e matrici

3.1 Introduzione

I vettori sono il tipo di oggetto più importante in **R**, ma di oggetti ne esistono altri: matrici o più generalmente array che sono generalizzazioni multidimensionali di vettori; liste che sono una forma generale di vettore in cui i vari elementi non hanno bisogno di essere dello stesso tipo; data frames che sono strutture tipo matrice, in cui le colonne possono essere vettori di tipi differenti. Si può pensare ad un data frame come ad una classica matrice di dati (*righe=unità*, *colonne=variabili*). Anche le funzioni sono oggetti di **R**. In questo capitolo tratteremo gli array, con particolare riferimento alle matrici, e alcune funzioni legate al loro utilizzo. Nei prossimi capitoli tratteremo le liste, i data frame e le funzioni.

3.2 Array e matrici

Un array può essere considerato come una collezione di numeri con indici multipli. Una matrice è un particolare array con due indici. Il limite inferiore per gli indici di un array è 1. Un vettore può essere trasformato in un array modificando il suo attributo `dim()`. Ad esempio, se abbiamo un vettore `z` di 1500 elementi, allora

```
> dim(z) <- c(3,5,100)
```

modificherà il suo attributo `dim` che permette ad **R** di trattarlo con un array $3 \times 5 \times 100$. I valori assegnati ad un array seguono le regole utilizzate ad esempio nel linguaggio **Fortran**, cioè facendo muovere il primo indice più velocemente degli altri e di conseguenza l'ultimo più lentamente. Ad esempio, se abbiamo un array di dimensione $3 \times 4 \times 2$, allora i dati sono immessi in questo ordine: `a[1,1,1]`, `a[2,1,1]`, ..., `a[2,4,2]`, `a[3,4,2]`. La quantità `a[2, ,]` è un array 4×2, cioè di dimensioni `c(4,2)`, con valori pari al seguente vettore

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1], a[2,1,2], a[2,2,2],
      a[2,3,2], a[2,4,2])
```

La quantità `a[ , , ,]` sta quindi ad indicare l'intero array e corrisponde all'utilizzo di `a`. Un array può anche essere inizializzato nel seguente modo:

```
> x <- array(1:20,dim=c(4,5)) # Genera un array 4 per 5.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

La stessa matrice `x` poteva ottenersi utilizzando il comando `matrix()`:

```
> x <- matrix(1:20,nrow=4)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

In generale la funzione `array()` ha la seguente sintassi:

```
> Z <- array(vettore di dati, vettore dim)
```

Ad esempio

```
> Z <- array(0, c(3,4,2))
```

genera l'array `Z` con tutti zero di dimensione $3 \times 4 \times 2$.

Gli array possono essere utilizzati in espressioni aritmetiche e il risultato è un array formato dalle operazioni fatte elemento per elemento sul vettore di dati. L'attributo `dim()` degli operandi generalmente deve essere lo stesso e questo sarà anche l'attributo `dim()` del risultato. Così se `A`, `B` e `C` sono array simili tra loro allora

```
> D <- 2*A*B + C + 1
```

rende `D` un array con dimensione uguale a quella di `A`, `B` e `C` e con dati ottenuti dalle operazioni fatte elemento per elemento sui tre array e sulle costanti.

Quando non si hanno array con le stesse dimensioni, allora per prevedere che risultato si otterrà, è bene tenere in mente le seguenti regole:

- le espressioni sono esaminate da sinistra a destra;
- alcuni operandi costituiti da vettori corti sono estesi ripetendo i loro valori fino a quando non si ha una corrispondenza tra le dimensioni degli operandi coinvolti;
- fino a quando si incontrano solo vettori e array, gli array devono tutti avere lo stesso attributo `dim()` o si ha un errore;
- qualsiasi operando vettore più lungo di un operando matrice o array genera un errore;
- se sono presenti strutture array e nessun errore o forzatura è stata visualizzata, il risultato è una struttura array con attributo `dim()` comune a quello degli operandi di tipo array.

3.3 Operatori e funzioni per il calcolo matriciale

R contiene diversi operatori e funzioni che possono essere utilizzati con matrici. Per esempio la funzione `t(X)` dà la trasposta della matrice `X`; le funzioni `nrow(A)` e `ncol(A)` danno il numero di righe e di colonne nella matrice `A`, rispettivamente. L'operatore `%*%` è utilizzato per il prodotto matriciale *righe* × *colonne*; se uno degli operandi è un vettore, allora viene considerato come vettore riga o come vettore colonna a seconda di quale formato sia coerente con il prodotto. L'operatore `*` effettua il prodotto elemento per elemento. La funzione `crossprod(X, y)` esegue l'operazione `t(X)%*%y`. Se il secondo argomento viene omesso, allora viene ripetuto il primo e quindi `crossprod(X)` corrisponde a `t(X)%*%X`.

E' possibile calcolare il determinante di una matrice quadrata con il comando `det()`, mentre per calcolare l'inversa di una matrice quadrata non singolare, cioè con determinante diverso da zero, il comando da utilizzare è `solve()`. E' da notare come il calcolo dell'inversa di una matrice quadrata possa dare dei problemi numerici quando la matrice pur non essendo singolare è quasi singolare, cioè ha un valore del determinante prossimo a zero.

Altra funzione utile per il calcolo matriciale è la funzione `diag()`. Il funzionamento di `diag()` dipende dal suo argomento:

- `diag(v)`, con `v` vettore, dà una matrice diagonale con gli elementi del vettore come elementi diagonali;
- `diag(M)`, con `M` matrice, dà il vettore formato dagli elementi della diagonale principale di `M`;
- `diag(k)`, con `k` singolo numero intero, dà la matrice identità $k \times k$.

La funzione `eigen(Sm)` calcola gli autovalori e gli autovettori della matrice simmetrica `Sm`. Il risultato è una lista di due componenti chiamate `values` e `vectors`. L'assegnazione

```
> ev <- eigen(Sm)
```

assegnerà questa lista a `ev`. Quindi `ev$val` ed `ev$vec` daranno rispettivamente gli autovalori e gli autovettori di `Sm`. Si potrebbe utilizzare l'assegnazione:

```
> evals <- eigen(Sm)$values
```

per assegnare direttamente ad `eval` gli autovalori. Vedremo meglio il funzionamento delle liste nel prossimo capitolo.

Una funzione interessante dal punto di vista statistico è `lsfit()` che dà i risultati di una procedura di adattamento dei minimi quadrati; in particolare:

```
> ans <- lsfit(X, y)
```

dà i risultati dell'adattamento dei minimi quadrati con `y` il vettore delle osservazioni e `X` la matrice disegno. In seguito, vedremo come, per un adattamento dei minimi quadrati in un modello lineare, venga solitamente utilizzata la funzione `lm()`.

3.4 Le funzioni cbind() e rbind()

Le matrici possono essere costruite a partire da altri vettori o da altre matrici attraverso le funzioni `cbind()` e `rbind()`. In termini generali, si può dire che `cbind()` forma matrici legando insieme vettori o matrici orizzontalmente, o per colonne, e `rbind()` verticalmente, o per righe. Nell'assegnazione

```
> X <- cbind(arg 1, arg 2, arg 3, ...)
```

gli argomenti di `cbind()` devono essere vettori di qualsiasi lunghezza o matrici con la stessa lunghezza per ciascuna colonna, cioè con lo stesso numero di righe. Se alcuni argomenti non sono della dimensione esatta, allora vengono ciclicamente estesi fino a far combaciare l'ampiezza delle colonne della matrice.

La funzione `rbind()` fa le stesse operazioni di `cbind()` sulle righe invece che sulle colonne. Così ad esempio, se `X1` e `X2` sono due variabili esplicative in un problema di regressione, la matrice disegno `X` può essere determinata in questa maniera:

```
> X <- cbind(1, X1, X2)
```

Inoltre l'uso di `cbind(x)` e `rbind(x)` è probabilmente il modo più semplice per forzare un vettore `x` ad essere trattato come vettore colonna o come vettore riga, rispettivamente. E' da notare che mentre `cbind()` e `rbind()` rispettano l'attributo `dim()`, la funzione di base `c()` non lo rispetta, cancellando negli oggetti numerici gli attributi `dim()` e `dimnames`. Il modo migliore per forzare un array a ritornare ad essere un semplice vettore è forse quello di utilizzare la funzione `as.vector()`:

```
> vec <- as.vector(X)
```

Comunque, un risultato simile si può ottenere anche nel seguente modo

```
> vec <- c(X)
```

Capitolo 4

Liste e data frame

4.1 Liste

Una lista in **R** è un oggetto che consiste di un insieme ordinato di altri oggetti, che vengono chiamati componenti della lista. Un semplice esempio di lista è:

```
> Lst <- list(nome="Ugo", moglie="Maria", nu.figli=3,
                        eta.figli=c(4,7,9))
```

Le componenti sono sempre numerate e ci si può riferire ad esse singolarmente attraverso degli indici. In particolare, per l'esempio visto si ha `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` e `Lst[[4]]`. Inoltre per la quarta componente si possono individuare le sottocomponenti `Lst[[4]][1]`, `Lst[[4]][2]` e `Lst[[4]][3]`. Le componenti di una lista possono essere richiamate anche specificando il loro nome; ad esempio `Lst$nome` è lo stesso di `Lst[[1]]` e corrisponde alla stringa `Ugo`. Si può anche utilizzare la seguente espressione: `Lst[["nome"]]`. La funzione `length(Lst)` dà il numero di componenti della lista `Lst`. E' importante distinguere `Lst[[1]]` da `Lst[1]`. Con `[[ ]]` si seleziona un singolo componente della lista, mentre `[ ]` è un operatore per indici. Quindi `Lst[[1]]` è il primo oggetto della lista `Lst` e il nome della componente non viene incluso. `Lst[1]` è una sottolista della lista `Lst` che consiste solo della prima entrata, con il nome della componente incluso nella sottolista.

I nomi delle componenti possono essere usati anche abbreviati; l'abbreviazione è valida quando è univoca.

Nuove liste possono essere formate da oggetti già esistenti utilizzando la funzione `list()`:

```
> Lst <- list(nome 1=oggetto 1, . . . , nome m=oggetto m)
```

Se i nomi sono omessi, le componenti sono solo numerate. E' da notare come permangano nello spazio di lavoro i vari oggetti che hanno formato la nuova lista. Le liste, come qualsiasi altro oggetto indicizzato, possono essere estese specificando componenti addizionali. Ad esempio:

```
> Lst[5] <- list(matrix=Mat)
```

aggiunge una quinta componente, che nella fattispecie è una matrice, alla lista `Lst`. Anche le liste possono essere concatenate:

```
> lista.ABC <- c(lista.A, lista.B, lista.C)
```

4.2 Data frames

In termini pratici un data frame può essere visto come una matrice con colonne, in generale, di modi e attributi differenti. In termini più formali, un data frame è una lista di classe `data.frame`. Ci sono restrizioni che vengono fatte nei data frame e specificatamente nelle componenti che devono essere vettori (numerici, di caratteri o logici), fattori, matrici numeriche, liste, o altri data frame; matrici, liste e data frame forniscono tante variabili al nuovo data frame per quante sono le colonne, gli elementi o le variabili che hanno, rispettivamente; i vettori numerici e i fattori sono inclusi così come sono, i vettori non numerici sono forzati ad essere fattori, i livelli dei quali sono gli unici valori che appaiono nel vettore; le strutture **vettore** che appaiono come componenti del data frame devono tutte avere la stessa lunghezza, le strutture **matrice** devono tutte avere la stessa ampiezza di riga. Gli oggetti che soddisfano queste condizioni possono formare un data frame, ad esempio, in questo modo:

```
> conti <- data.frame(provincia=provf,proventi=redditi,
                                        classe=redditif)
```

Si possono utilizzare liste come data frame utilizzando la funzione

```
as.data.frame()
```

Se si vuole eliminare una componente da un data frame, basta utilizzare il comando:

```
> conti$provincia <- NULL
> conti$classe <- NULL
```

Lo stesso vale nel caso di vettori o liste.

4.3 Le funzioni attach() e detach()

La funzione `attach()` con argomento un data frame serve a posizionare il data frame al posto 2 nel cammino di ricerca, permettendo di utilizzare immediatamente le variabili che compongono il data frame senza definirle, verificato che non esistono già nello spazio di lavoro delle variabili con quel nome. In ogni caso le operazioni fatte su queste variabili non cambiano il contenuto delle corrispettive componenti del data frame. Per effettuare un cambio permanente nel data frame bisogna utilizzare il simbolo `$`, che già poteva essere utilizzato per richiamare le componenti del data frame senza far ricorso alla funzione `attach()`. Per rilasciare il data frame si utilizza la funzione `detach()`. La funzione `detach()` può essere anche utilizzata per rilasciare packages caricati in precedenza. Ad esempio se si vuole rilasciare il package `ctest` il comando da utilizzare è il seguente:

```
> detach(package:ctest)
```

4.4 Lavorare con i data frame

Per poter analizzare diversi problemi contemporaneamente nella stessa cartella di lavoro possono essere utili i seguenti consigli:

- raggruppare tutte le variabili per un particolare problema in un unico data frame assegnando alle variabili nomi adatti;
- utilizzare la funzione `attach()` per posizionare il data frame relativo al problema sotto studio in posizione 2 e utilizzare la cartella di lavoro al livello 1 per fare operazioni o per utilizzare variabili temporanee;
- prima di abbandonare il problema caricare tutte le variabili di cui interessa conservare memoria nel data frame usato per quel problema, utilizzando il simbolo `$` e quindi "staccare" il data frame con la funzione `detach()`;
- rimuovere tutte le variabili non necessarie per le analisi successive da effettuare con quegli stessi dati.

4.5 Gestire il cammino di ricerca

La funzione `search()` mostra il cammino di ricerca corrente. All'inizio di una sessione solitamente si ha:

```
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

dove `.GlobalEnv` rappresenta lo spazio di lavoro. Supponendo di aver "attaccato" il data frame `pippo`, che contiene le variabili `u`, `v` e `w`, al livello 1 del cammino di ricerca si ha:

```
> search()
[1] ".GlobalEnv" "pippo" "Autoloads" "package:base"
```

mentre al livello 2 si ha:

```
> ls(2)
[1] "u" "v" "w"
```

con la funzione `ls()` che può, quindi, essere utilizzata per esaminare il contenuto di una qualunque altra posizione del cammino di ricerca. Alla fine si possono utilizzare i seguenti comandi:

```
> detach("pippo")
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

4.6 Attributi intrinseci degli oggetti: modo e lunghezza

Come già abbiamo visto, le entità su cui **R** opera sono tecnicamente note come oggetti. I vettori sono note come **strutture atomiche**, perché le loro componenti sono tutte dello stesso tipo, o modo. Le liste, invece, sono note come **strutture ricorsive**, dato che le loro componenti possono essere esse stesse delle liste. Le altre strutture ricorsive sono quelle di modo funzione e espressione. Per **modo** di un oggetto si intende quindi il tipo base dei suoi costituenti fondamentali. Questo è un caso speciale di **attributo** di un oggetto. Gli attributi

di un oggetto forniscono informazioni circa l'oggetto stesso. Un altro tipo di attributo di un qualsiasi oggetto è la sua **lunghezza**, che consiste nel numero di elementi che compongono l'oggetto. Ogni oggetto in **R** deve possedere almeno questi due attributi. In ogni caso nuove componenti possono essere aggiunte ad un qualunque oggetto semplicemente considerando un'operazione di assegnazione con un valore dell'indice più elevato del massimo indice valido in quel momento. Così ad esempio, se definiamo un vettore vuoto di tipo numerico:

```
> e <- numeric()
```

l'assegnazione

```
> e[3] <- 17
```

rende il vettore `e` di lunghezza 3, con le prime due componenti di tipo `NA`. Viceversa, per troncare l'ampiezza di un oggetto è richiesta solo un'assegnazione. Se, ad esempio, `alpha` è un oggetto di lunghezza 10, allora

```
> alpha <- alpha[2 * 1:5]
```

lo rende un oggetto di lunghezza 5 che consiste delle sole vecchie componenti con indice pari. E' chiaro che in questo caso i vecchi indici vengono persi e i nuovi andranno da 1 a 5.

4.7 Visualizzare e settare gli attributi

La funzione `attributes(oggetto)` dà una lista di tutti gli attributi non intrinseci attualmente definiti per quell'oggetto. La funzione `attr(oggetto, nome)` può essere utilizzata per selezionare uno specifico attributo. In ogni caso queste funzioni non sono molto utilizzate. Comunque deve essere posta particolare cura quando si assegnano o si cancellano attributi, dato che sono parte integrante dell'oggetto utilizzato. Quando la funzione `attr()` è utilizzata sul lato sinistro di un'assegnazione, può essere utilizzata o per associare un nuovo attributo o per cambiarne uno esistente. Ad esempio

```
> attr(z,"dim") <- c(10,10)
```

permette ad **R** di trattare `z` come una matrice 10×10. Come già abbiamo visto, lo stesso risultato si sarebbe ottenuto con la seguente istruzione:

```
> dim(z) <- c(10,10)
```

4.8 La classe di un oggetto

Un attributo speciale noto come **classe** dell'oggetto è utilizzato per permettere uno stile di programmazione orientata ad oggetti in **R**. Per rimuovere temporaneamente gli effetti della classe, si usa la funzione `unclass()`. Per esempio, se `inverno` possiede la classe **data frame**, allora il comando:

```
> inverno
```

lo visualizzerà in formato data frame, mentre

```
> unclass(inverno)
```

lo visualizzerà come un'ordinaria lista. In ogni caso questa funzione viene utilizzata raramente.

Capitolo 5

La lettura di dati da file

5.1 La funzione read.table()

Per leggere i dati da file in **R** è conveniente preliminarmente generare un file di dati in formato ASCII, disponendoli come si farebbe in una matrice di dati, e mettere questo file nella cartella di lavoro corrente. Fatto questo, si può utilizzare la funzione `read.table()` per leggere l'intero data frame. Se la prima riga del file contiene l'intestazione delle variabili e ogni riga del file è preceduta dall'identificatore di riga (01, 02, ecc.), allora `read.table()` interpreterà la prima riga del file come una riga dove sono contenuti i nomi delle variabili, assegnando ciascun nome alle variabili del data frame:

```
> prova <- read.table("mio.txt")
```

Se gli identificatori di riga non sono presenti, ma nella prima riga si hanno sempre i nomi delle variabili, allora si può utilizzare il seguente comando per importare i dati:

```
> prova <- read.table("mio.txt", header=TRUE)
```

Se invece nel file di dati non si hanno i nomi delle variabili nella prima riga, ma sono presenti gli identificatori di riga all'inizio di ogni riga, allora il comando da utilizzare è:

```
> prova <- read.table("mio.txt", row.names=1)
```

Se nel file di dati non si hanno nè i nomi delle variabili, nè gli identificatori di riga allora il comando da utilizzare per importare un file di dati in formato testo è comunque:

```
> prova <- read.table("mio.txt")
```

Alla fine, per tutti gli esempi visti in `prova`, si avrà il data frame con i dati importati.

5.2 La funzione scan()

Un'altra funzione che si può utilizzare per leggere dati da file è la funzione `scan()`. Supponiamo di avere vettori di dati di uguale lunghezza da leggere in

parallelo dal file **input.dat**; supponiamo inoltre che i vettori siano tre: il primo di tipo carattere e i rimanenti tre numerici. Il primo passo è usare `scan()` per leggere i quattro vettori come una lista:

```
> in <- scan("input.dat", list("",0,0,0))
```

Il secondo argomento è una struttura lista dummy che stabilisce il modo in cui i tre vettori devono essere letti e quindi importati. Per separare i tre vettori si può utilizzare la seguente assegnazione:

```
> label <- in[[1]]; x <- in[[2]]; y <- in[[3]]; z <- in[[4]]
```

Il tutto poteva farsi più convenientemente con la seguente assegnazione:

```
> in <- scan("input.dat", list(id="", x=0, y=0, z=0))
```

Se poi si vuole accedere alle variabili separatamente si può utilizzare la seguente assegnazione:

```
> label <- in$id; x <- in$x; y <- in$y; z <- in$z
```

oppure possiamo "attaccare" la lista alla posizione 2 del cammino di ricerca con la funzione `attach()`. Se il secondo argomento di `scan()` è un singolo valore e non una lista, viene letto un singolo vettore con tutte le componenti dello stesso tipo del valore dummy; ad esempio, con la seguente istruzione `X` sarà una matrice:

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

In **R** esiste comunque un package, `foreign`, che importa file di dati salvati attraverso altri software statistici, quali, ad esempio, **Minitab**, **S**, **SAS**, **SPSS** e **Stata**.

5.3 File di dati forniti da R

In **R** esistono comunque oltre 50 insiemi di dati contenuti nel package `base` e altri sono disponibili in altri packages. A differenza di **S-Plus** è necessario caricare esplicitamente questi insiemi di dati utilizzando la funzione `data()`. Per vedere l'elenco degli insiemi di dati disponibili nel package `base` basta usare il comando `data()`; per caricare un particolare insieme di dati, ad esempio `cars`, basta utilizzare il comando

```
> data(cars)
```

Nella maggior parte dei casi questo corrisponde a caricare un oggetto, solitamente un data frame, dello stesso nome: per l'esempio considerato si avrebbe un data frame di nome `cars`. Per vedere l'elenco e per caricare insiemi di dati contenuti in altri packages si possono usare i seguenti comandi:

```
> data(package="nls")
> data(Puromycin, package="nls")
```

Se in precedenza si era provveduto a caricare un package, allora tutti gli insiemi di dati di quel package possono essere caricati direttamente:

```
> library(nls)
> data()
> data(Puromycin)
```

Capitolo 6

Tabelle e Grafici

6.1 Introduzione

Le rappresentazioni tabellari e grafiche rivestono un ruolo fondamentale in Statistica. Si tratta infatti solitamente delle prime elaborazioni che vengono fatte su un insieme di dati per trarne le prime informazioni. Solitamente un grafico deve essere sempre accompagnato da una tabella che descrive i dati che hanno prodotto quel grafico. Nei prossimi paragrafi vedremo come è possibile in **R** costruire tabelle e grafici.

6.2 Distribuzioni di frequenza semplici e doppie

6.2.1 Distribuzioni di frequenza semplici

La costruzione di una distribuzione di frequenza semplice e di una distribuzione di frequenza doppia in **R** viene effettuata utilizzando il comando `table()`.

Per quanto riguarda le distribuzioni di frequenza semplice distingueremo tre casi a seconda delle natura della variabile presa in considerazione, cioè a seconda se abbiamo una mutabile, una variabile ordinabile oppure una variabile quantitativa.

Se abbiamo a che fare con una mutabile, la costruzione di una distribuzione di frequenza semplice risulta particolarmente facile. Supponiamo infatti di avere come mutabile il colore della confezione di un prodotto venduto in un determinato giorno in un supermarket e che i dati siano i seguenti (si opererà una generazione di valori fittizi per la mutabile `colore`):

```
> colore<-c(rep(c("verde","giallo","rosso"),10),
     rep(c("azzurro","viola"),3),rep("giallo",3),
     rep("rosso",5),"azzurro","azzurro",rep("arancione",3),"grigio")
> colore
 [1] "azzurro"   "azzurro"   "arancione" "arancione" "arancione"
 [6] "grigio"    "rosso"     "rosso"     "rosso"     "rosso"
[11] "rosso"     "azzurro"   "viola"     "azzurro"   "viola"
[16] "azzurro"   "viola"     "giallo"    "giallo"    "giallo"
[21] "verde"     "giallo"    "rosso"     "verde"     "giallo"
[26] "rosso"     "verde"     "giallo"    "rosso"     "verde"
```

```
[31] "giallo"    "rosso"     "verde"     "giallo"    "rosso"
[36] "verde"     "giallo"    "rosso"     "verde"     "giallo"
[41] "rosso"     "verde"     "giallo"    "rosso"     "verde"
[46] "giallo"    "rosso"     "verde"     "giallo"    "rosso"
> table(colore)
colore
arancione   azzurro    giallo    grigio     rosso     verde     viola
        3         5        13         1        15        10         3
```

Come si può notare, le modalità della variabile `colore` sono state ordinate in ordine alfabetico. In ogni caso, `table()` riporterà le modalità della mutabile che presentano valore di frequenza assoluta diverso da zero. Se si vuole ottenere la distribuzione di frequenza semplice con frequenze relative, basta utilizzare il seguente comando:

```
> table(colore)/length(colore)
colore
arancione   azzurro    giallo    grigio     rosso     verde     viola
     0.06      0.10      0.26      0.02      0.30      0.20      0.06
```

Se abbiamo a che fare con una variabile ordinabile, bisogna considerarla come un fattore e ordinare i livelli di questo fattore prima di costruire la distribuzione di frequenza. Supponiamo di avere rilevato il titolo di studio posseduto da un certo gruppo di persone; costruiamo anche in questo caso la distribuzione di frequenza per la relativa variabile:

```
> titolo.studio<-c(rep(c("diploma","laurea"), 15), rep("nessuno",2),
      rep("elementare",5),rep("media",7),rep("diploma",6))
> titolo.studio
 [1] "diploma"    "laurea"     "diploma"    "laurea"     "diploma"
 [6] "laurea"     "diploma"    "laurea"     "diploma"    "laurea"
[11] "diploma"    "laurea"     "diploma"    "laurea"     "diploma"
[16] "laurea"     "diploma"    "laurea"     "diploma"    "laurea"
[21] "diploma"    "laurea"     "diploma"    "laurea"     "diploma"
[26] "laurea"     "diploma"    "laurea"     "diploma"    "laurea"
[31] "nessuno"    "nessuno"    "elementare" "elementare" "elementare"
[36] "elementare" "elementare" "media"      "media"      "media"
[41] "media"      "media"      "media"      "media"      "diploma"
[46] "diploma"    "diploma"    "diploma"    "diploma"    "diploma"
```

Se costruiamo direttamente la distribuzione di frequenza semplice non si avrà l'ordinamento delle modalità della variabile:

```
> table(titolo.studio)
titolo.studio
   diploma elementare     laurea      media    nessuno
        21          5         15          7          2
```

Per ottenere la distribuzione corretta bisogna utilizzare i seguenti comandi:

```
> titolo.studio<-factor(titolo.studio,
      levels=c("nessuno","elementare","media","diploma","laurea"))
```

```
> table(titolo.studio)
titolo.studio
   nessuno elementare      media    diploma     laurea
         2          5          7         21         15
```

Anche in questo caso possiamo determinare la distribuzione di frequenza semplice a partire dalle frequenze relative:

```
> table(titolo.studio)/length(titolo.studio)
titolo.studio
   nessuno elementare      media    diploma     laurea
      0.04       0.10       0.14       0.42       0.30
```

Vediamo, invece, come si può ottenere una distribuzione di frequenza semplice a partire da una variabile quantitativa. Faremo riferimento al caso di distribuzione di frequenza con classi di ampiezza costante ma, come si vedrà, il procedimento che seguiremo è del tutto generale. Supponiamo di avere dati relativi alle altezze espresse in *cm* rilevate su un gruppo di persone (anche in questo caso considereremo dati generati direttamente; inoltre, per il significato del comando `rnorm()` si veda il prossimo capitolo):

```
> altezze<-rnorm(100,170,10)
```

Nel caso specifico, il vettore `altezze` è costituita da 100 osservazioni. Per determinare l'estremo inferiore della prima classe e l'estremo superiore dell'ultima, utilizziamo il seguente comando:

```
> range(altezze)
[1] 145.4411 191.2725
```

E' chiaro che riprovando a generare nuove osservazioni, in generale i valori ottenuti saranno diversi. A questo punto fissiamo in 5 il numero di classi da prendere in considerazione e di conseguenza in 10 l'ampiezza di ciascuna classe, prendendo come estremo inferiore della prima classe il valore 145 e estremo superiore dell'ultima 195:

```
> ampiezza.classi<-(195-145)/5
> ampiezza.classi
[1] 10
```

A questo punto costruiamo gli estremi delle varie classi:

```
> classi<-145+ampiezza.classi*(0:5)
> classi
[1] 145 155 165 175 185 195
```

Si potrebbe vedere ora come vengono assegnati i valori della variabile `altezze` alle varie classi, utilizzando la funzione `cut()`:

```
> cut(altezze,breaks=classi)
```

Quindi, per ottenere la nostra distribuzione di frequenza basta digitare il comando:

```
> table(cut(altezze,breaks=classi))

(145,155] (155,165] (165,175] (175,185] (185,195]
        7        19        38        32         4
```

In questo caso le classi sono costruite considerando l'estremo inferiore escluso e quello superiore incluso. Anche in questo caso è semplice determinare la distribuzione di frequenza semplice per frequenze relative:

```
> table(cut(altezze,breaks=classi))/length(altezze)

(145,155] (155,165] (165,175] (175,185] (185,195]
     0.07      0.19      0.38      0.32      0.04
```

6.2.2 Distribuzioni di frequenza doppie

Per determinare distribuzioni di frequenza doppie il comando da utilizzare è sempre `table()`. Supponiamo di avere due mutabili rilevate su n soggetti e di volere determinare la relativa tabella di contingenza. Nel caso specifico le due mutabili saranno il colore degli occhi e il colore dei capelli:

```
> occhi<-c(rep("neri",18),rep("castani",24),rep("azzurri",3),
                   rep("verdi",5))
> capelli<-c(rep("neri",10),"biondi","biondi",
         rep("castani",23),"rossi",rep("biondi",5),
         rep("castani",5),rep("rossi",3),"castani")
> table(capelli,occhi)
         occhi
capelli   azzurri castani neri verdi
  biondi   0       5       2    0
  castani  3      18       6    2
  neri     0       0      10    0
  rossi    0       1       0    3
```

Se una delle variabili coinvolte dovesse essere quantitativa, si può seguire il procedimento visto nel caso di distribuzioni di frequenza semplici per determinare le classi relative a questa variabile e quindi procedere alla costruzione della tabella a doppia entrata, così come visto in precedenza.

Spesso è utile calcolare, nel caso di tabelle a doppia entrata, l'indice X^2 di indipendenza di Pearson; in questo caso il comando da utilizzare è:

```
> chisq.test(capelli,occhi)

        Pearson's Chi-squared test

data:  capelli and occhi
X-squared = 43.1023, df = 9, p-value = 2.064e-06

Warning message:
Chi-squared approximation may be incorrect in: chisq.test(capelli, occhi)
```

Il messaggio che compare alla fine ci avverte solamente che in questo caso la distribuzione χ^2, che è la distribuzione asintotica dell'indice X^2, potrebbe non funzionare bene, a causa delle diverse celle vuote presenti nella nostra tabella.

6.3 Le principali rappresentazioni grafiche

Per poter fare un grafico potrebbe essere necessario inizializzare una speciale finestra dove verranno mostrati in modo interattivo i grafici: in ambiente **Windows** si utilizza il comando `windows()`, mentre in **Linux** in ambiente grafico si utilizza il comando `x11()`. Solitamente, comunque, la finestra grafica viene aperta in modo automatico alla prima esecuzione di un comando per effettuare un grafico.

I comandi per i grafici possono essere divisi in tre gruppi:

- funzioni di alto livello, che creano un nuovo grafico sulla finestra grafica;
- funzioni di basso livello, che aggiungono parti ad un grafico già esistente;
- funzioni per grafici interattivi, che consentono di aggiungere interattivamente informazioni, o di estrarne, da un grafico esistente.

In ogni caso **R** ha una lista di parametri grafici, che possono essere cambiati attraverso il comando `par()`.

Per vedere una panoramica dei grafici che è possibile realizzare in **R**, basta eseguire il comando:

```
> demo(graphics)
```

6.3.1 Comandi di alto livello

La funzione di alto livello più utilizzata è `plot()`, che permette di effettuare diversi grafici:

`plot(x, y)` `plot(xy)`	Se `x` e `y` sono vettori, `plot(x,y)` produce uno scatterplot di `y` su `x`. Lo stesso effetto si ha con `plot(xy)` se l'argomento è di tipo lista con due elementi o una matrice a due colonne.
`plot(x)`	Se `x` è una serie temporale, si ha un grafico temporale; se `x` è un vettore, si ha un grafico dei valori del vettore sui rispettivi indici, se `x` è un vettore di complessi, si ha un grafico delle parti immaginarie su quelle reali.
`plot(f)` `plot(f, y)`	Se `f` è un fattore e `y` un vettore, la prima forma dà un grafico a barre di `f`, la seconda un boxplot di `y` per ogni livello di `f`.

`plot(df)`
`plot(~expr)`
`plot(y~expr)`
	Se `df` è un data frame, `y` un oggetto, `expr` una lista di nomi di oggetti separati da `+`, le prime due forme danno grafici distribuzionali delle variabili nel data frame (primo caso) o degli oggetti indicati in `expr` (secondo caso). La terza forma dà grafici di `y` su ciascun oggetto indicato in `expr`.

Esistono anche due funzioni molto utili per rappresentare dati multivariati. Se `X` è una matrice o un data frame, il comando

```
> pairs(X)
```

produce una matrice di scatterplot accoppiati delle variabili definite dalle colonne di `X`. Quando sono coinvolte tre o quattro variabili, si può usare la funzione `coplot()`; se `a` e `b` sono due vettori e `d` è o un vettore o un fattore, allora

```
> coplot(a ~ b | d)
```

produce un numero di scatterplot di `a` su `b` per dati valori di `d`. Se `d` è un fattore, allora avremo i grafici di `a` su `b` per ogni livello di `d`; quando `d` è numerico, allora è diviso in un numero di intervalli e per ogni intervallo si ha il grafico di `a` su `b` per i valori di `d` nell'intervallo. Il numero e le posizioni dell'intervallo possono essere controllate con

```
given.value = argomento
```

per `coplot()`. Le funzioni `coplot()` e `pairs()` hanno l'argomento `panel` che indica il tipo di grafico che deve apparire in ogni pannello; per default si ha uno scatterplot.

Esistono comunque delle funzioni che danno luogo a particolari grafici:

`qqnorm(x)`
`qqline(x)`
`qqplot(x,y)`
	Grafici per confronti di distribuzione. Il primo dà il grafico di `x` contro i punteggi normali attesi, il secondo aggiunge una retta disegnandola tra la distribuzione e i quartili dei dati, il terzo confronta la distribuzione di `x` con quella di `y`.

`hist(x)`
`hist(x,nclass=n)`
`hist(x,breaks=b, ...)`
	Dà istogrammi di `x`. Il numero delle classi viene scelto per default o può essere indicato, oppure si possono dare i punti estremi degli intervalli. Se `probability=TRUE` allora vengono rappresentate le frequenze relative.

`dotchart(x, ...)`	Dà uno scatterplot dei dati `x` su sfondo con puntini. In questo tipo di grafico nell'asse `y` abbiamo la label dei dati, nell'asse `x` il loro valore.
`barplot(x, ...)`	Dà un grafico a colonne con altezza delle colonne prese da `x`.
`boxplot(x, ...)`	Dà un boxplot per i valori del vettore `x`.

Per le funzioni di alto livello si possono avere i seguenti argomenti:

`add=TRUE`	Forza la funzione ad agire come funzione di basso livello, con il risultato di sovrapporre il grafico corrente.
`axes=FALSE`	Sopprime la generazione degli assi.

`log="x"`
`log="y"`
`log="xy"`
	Considera gli assi come logaritmici.

`type=`	Con `"p"` si avranno punti individuali, con `"l"` linee, con `"b"` punti connessi da linee, con `"o"` punti uniti da linee che passano sopra i punti, con `"h"` linee verticali dai punti all'asse delle ascisse, con `"s"` o `"S"` una funzione a gradini (nel primo caso la linea verticale sta in cima, nel secondo in basso), con `"n"` nessun grafico.
`xlab=stringa` `ylab=stringa`	Nomi per l'asse delle `x` e per l'asse delle `y`.
`main=stringa`	Titolo della figura posto in testa al grafico.
`sub=stringa`	Sottotitolo, posto appena sotto l'asse delle `x`.

6.3.2 Comandi di basso livello

I comandi di basso livello sono usati per aggiungere caratteristiche a grafici già costruiti. Alcuni dei comandi più utili sono i seguenti:

`points(x,y)` `lines(x,y)`	Aggiunge punti o linee di collegamento al grafico corrente.
`text(x,y,labels, ...)`	Aggiunge testo al grafico nei punti dati da `x` e `y`. `labels` può essere un intero o un vettore di caratteri.

`abline(a,b)`
`abline(h=y)`
`abline(v=x)`
`abline(lm.obj)`
	Aggiunge una retta di pendenza `b` e intercetta `a`. `h=y` può essere usato per specificare le coordinate `y` delle linee orizzontali lungo il grafico; `v=x` fa lo stesso per le coordinate `x` per le linee verticali. `lm.obj` contiene i risultati dell'adattamento di un modello e dà la retta stimata.

`polygon(x,y, ...)`	Traccia un poligono definito dai vertici ordinati (`x`, `y`).
`legend(x,y,legend, ...)`	Aggiunge una legenda nella posizione specificata. Caratteri per il grafico, stili delle linee, colori, ecc., sono identificati con le label nel vettore caratteri `legend`. Almeno un altro argomento `v`, vettore con la stessa lunghezza di `legend`, deve essere dato: `legend(... , fill=v)` colora box pieni, `legend(... , col=v)` colora punti e linee che saranno disegnate, `legend(... , lty=v)` stile di linee, `legend(... , lwd=v)` ampiezza delle linee, `legend(... , pch=v)` caratteri per il grafico.
`title(main,sub)`	Aggiunge il titolo (sopra) e il sottotitolo (sotto).
`axis(side, ...)`	Aggiunge un asse sul lato dato dal primo argomento (da 1 a 4)

6.3.3 Interagire con i grafici

Le principali funzioni per interagire con un grafico sono:

`locator(n,type)`	Aspetta che l'utilizzatore selezioni le posizioni del grafico usando il tasto sinistro del mouse. Continua finchè `n` punti non sono stati selezionati, o il bottone centrale del mouse non è pressato. L'argomento `type` permette di disegnare i punti selezionati: il default è nessun grafico (`type="n"`).
`identify(x,y,labels)`	Permette che l'utilizzatore evidenzi alcuni punti definiti da (`x`, `y`), usando il tasto sinistro del mouse, disegnando le corrispondenti label vicino.

6.3.4 Configurare le caratteristiche di un grafico

Per potere configurare le caratteristiche di un grafico il comando da utilizzare è `par()`. Con questo comando si possono configurare le caratterstiche di un grafico passando come argomento uno o più dei possibili parametri che si vogliono configurare. Ad esempio, se si desidera dividere la finestra grafica in quattro parti in modo tale da potere rappresentare contemporaneamente quattro grafici il comando da utilizzare è:

```
> par(mfrow=c(2, 2))
```

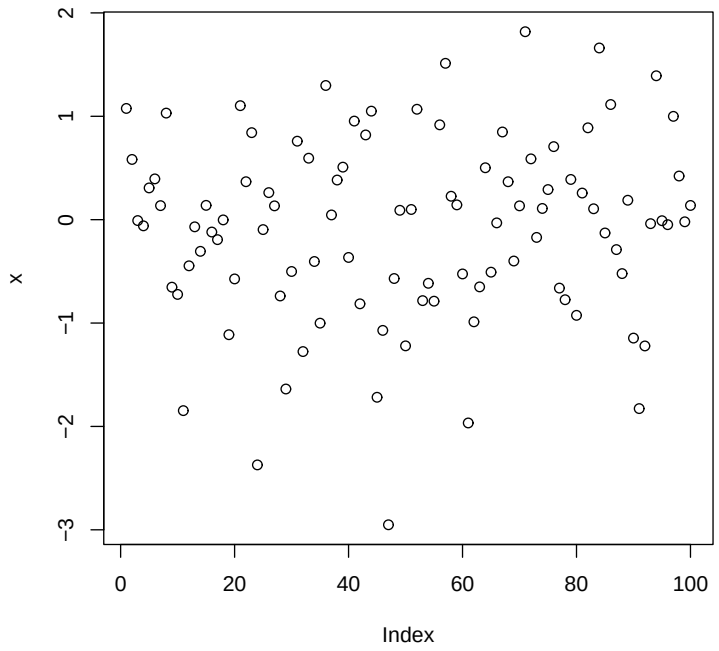
Per avere un elenco completo dei parametri grafici che è possibile configurare con il comando `par()` è bene consultare l'help in linea. Qui possiamo dire che tra le caratteristiche che possono essere controllate abbiamo il tipo di assi che si vogliono utilizzare, il font e il colore delle etichette che costituiscono il nome degli assi o il titolo del grafico, l'area grafico su cui visualizzare il grafico, ecc.

6.3.5 Esempi di grafico

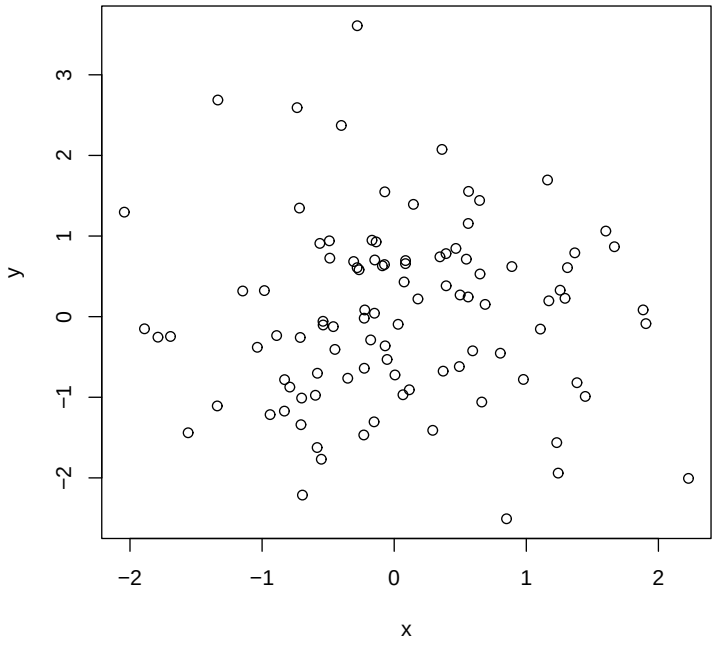
Nel seguente paragrafo verranno descritti alcuni esempi di grafico con i relativi comandi in **R** che lo hanno prodotto.

Supponendo di avere un vettore `x` e un vettore `y`, generati casualmente da una normale standardizzata, si possono avere i seguenti grafici (per il significato della funzione `rnorm()` si veda il prossimo capitolo):

```
> x <- rnorm(100)
> plot(x)
```

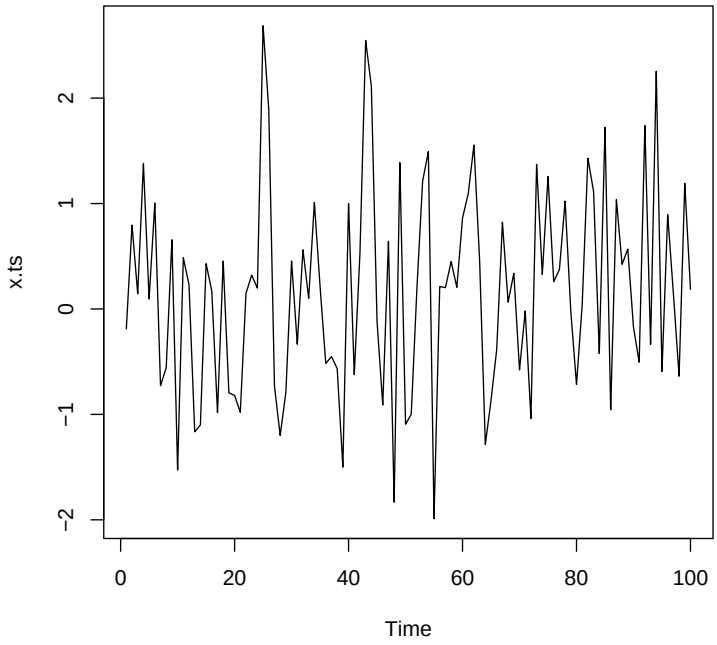



```
> y <- rnorm(100)
> plot(x, y)
```



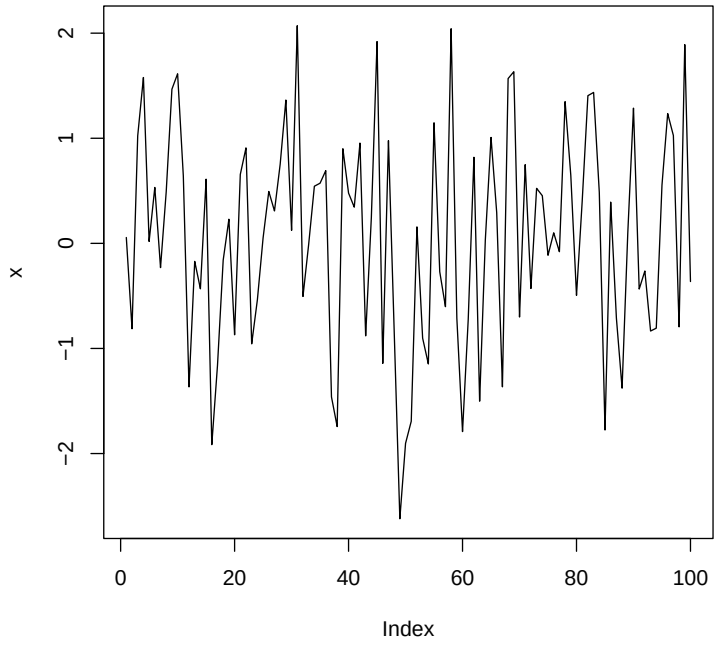
Se si vuole rappresentare una serie storica bisogna prima avere un oggetto `ts`:

```
> x <- rnorm(100)
> x.ts<-as.ts(x)
> plot(x.ts)
```



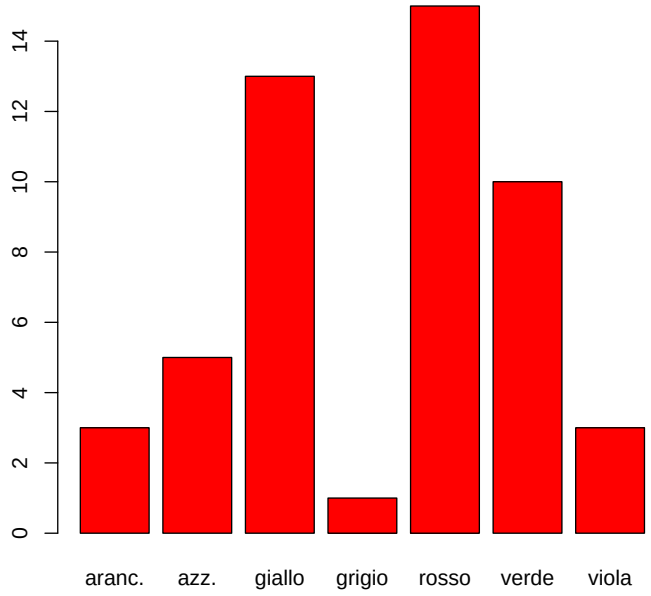
Si sarebbe ottenuto un grafico simile utilizzando direttamente il vettore `x`, settando nel comando `plot()` l'opzione `type`:

```
> x <- rnorm(100)
> plot(x, type="l")
```



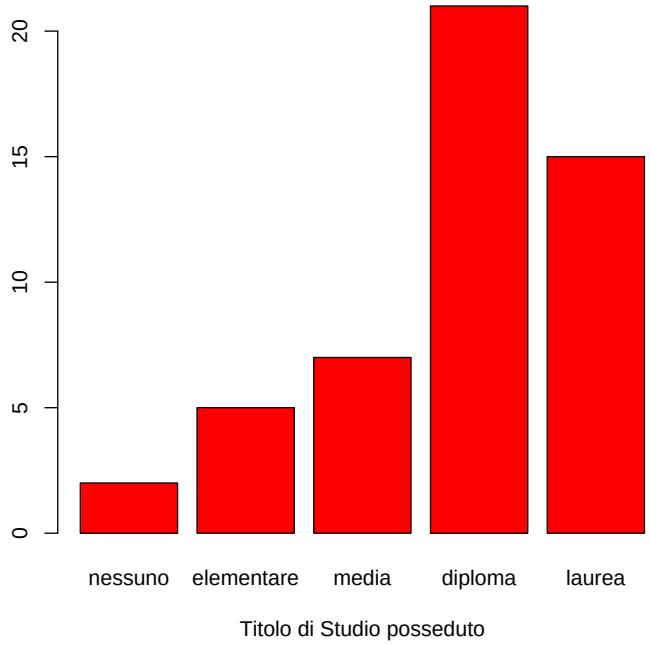
Consideriamo ora alcuni semplici esempi relativi ai grafici a colonne e agli istogrammi; prendiamo in particolare gli esempi visti in precedenza quando si sono introdotte le distribuzioni di frequenza. Consideriamo il caso di una distribuzione di frequenza per una variabile qualitatitiva:

```
> colore<-c(rep(c("verde","giallo","rosso"),10),
     rep(c("azzurro","viola"),3),rep("giallo",3),
     rep("rosso",5),"azzurro","azzurro",rep("arancione",3),"grigio")
> colore <- as.factor(colore)
> table(colore)
colore
arancione   azzurro    giallo    grigio     rosso     verde     viola
        3         5        13         1        15        10         3
> levels(colore)[1]<-"aranc."
> levels(colore)[2]<-"azz."
> plot(colore)
```



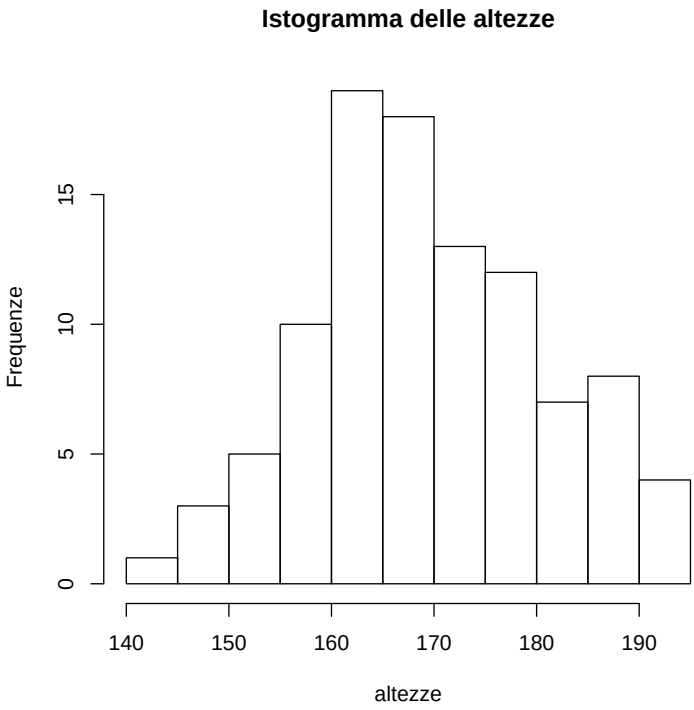
Per ottenere il grafico a colonne per la variabile `colore` si è prima dovuto forzare questa variabile ad essere un fattore. La stessa procedura si può utilizzare per disegnare un grafico a colonne a partire da una variabile qualitativa ordinabile:

```
> titolo.studio<-c(rep(c("diploma","laurea"), 15), rep("nessuno",2),
      rep("elementare",5),rep("media",7),rep("diploma",6))
> titolo.studio<-factor(titolo.studio,
      levels=c("nessuno","elementare","media","diploma","laurea"))
> table(titolo.studio)
titolo.studio
   nessuno elementare      media    diploma     laurea
         2          5          7         21         15
> plot(titolo.studio, xlab="Titolo di Studio posseduto")
```



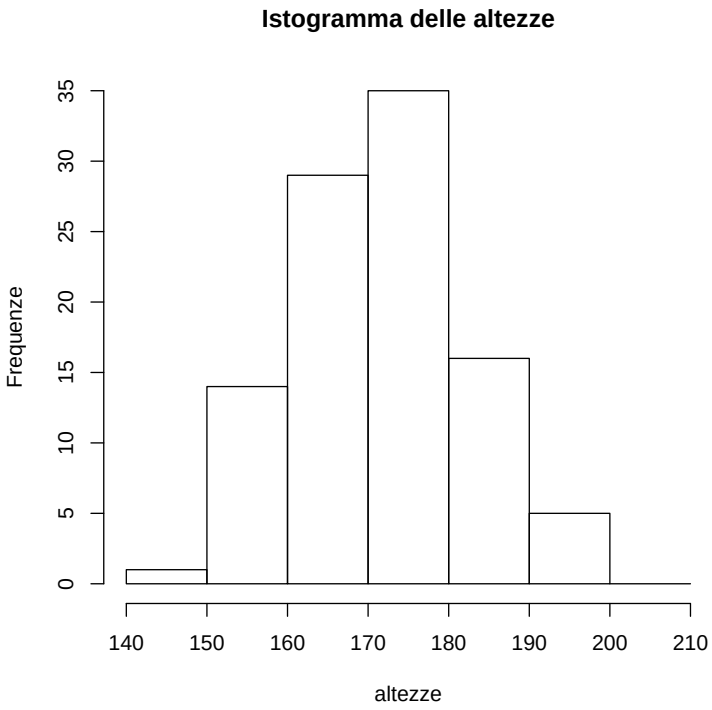
Nel caso in cui si ha una variabile quantitativa allora si deve fare ricorso ad un istogramma:

```
> altezze<-rnorm(100,170,10)
> hist(altezze,ylab="Frequenze",main="Istogramma delle altezze")
```



In questo caso l'ampiezza delle classi nell'istogramma è stata determinata da **R**, ma è data la possibilità all'utente di fissarle:

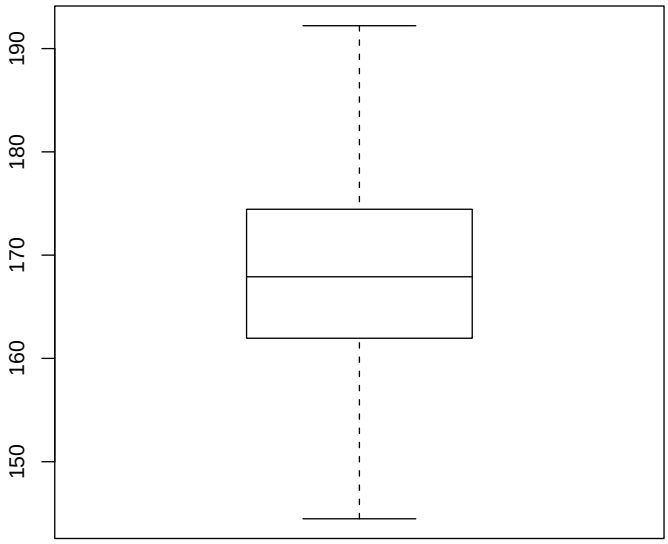
```
> classi<-140+10*(0:7)
> hist(altezze,breaks=classi,ylab="Frequenze",main="Istogramma delle altezze")
```



E' da notare come in quest'ultima rappresentazione grafica alcuni dati non siano stati visualizzati, a causa della scelta degli estremi delle classi.

Se si vuole costruire un boxplot a partire da una variabile quantitativa si possono utilizzare i seguenti comandi:

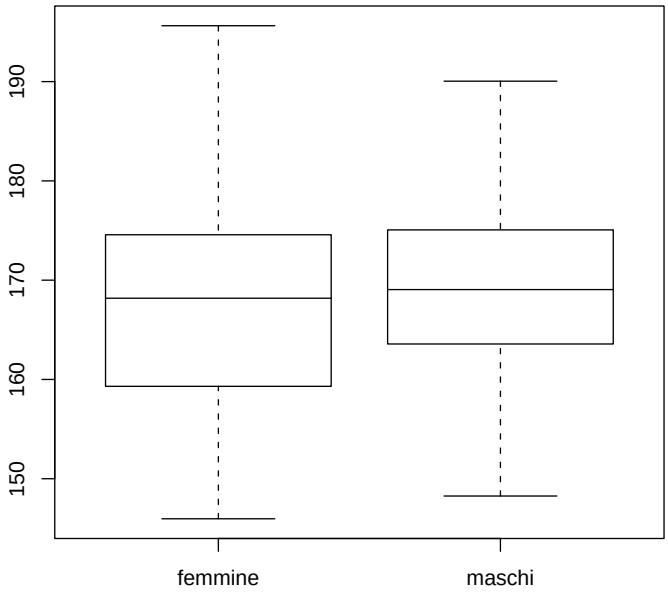
```
> altezze<-rnorm(100,170,10)
> boxplot(altezze, xlab="Altezze")
```



Altezze

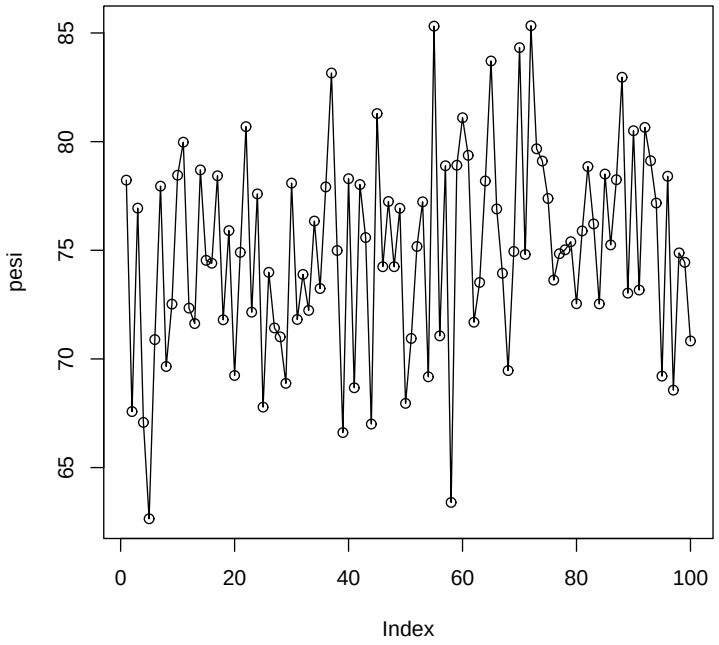
Possiamo avere diversi boxplot per una stessa variabile quantitativa categorizzata secondo i livelli di un fattore:

```
> sesso<-factor(c(rep("maschi",56), rep("femmine",44)))
> plot(sesso,altezze)
```



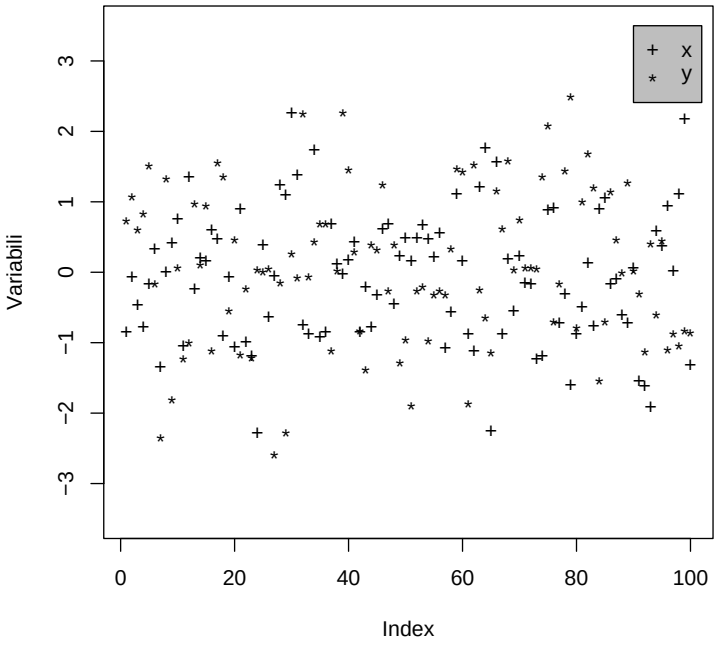
Molto utili sono i comandi di basso livello `points()` e `lines()`, che vengono solitamente utilizzati assieme al comando `plot()`:

```
> pesi<-rnorm(100,75,5)
> plot(pesi)
> lines(pesi)
```



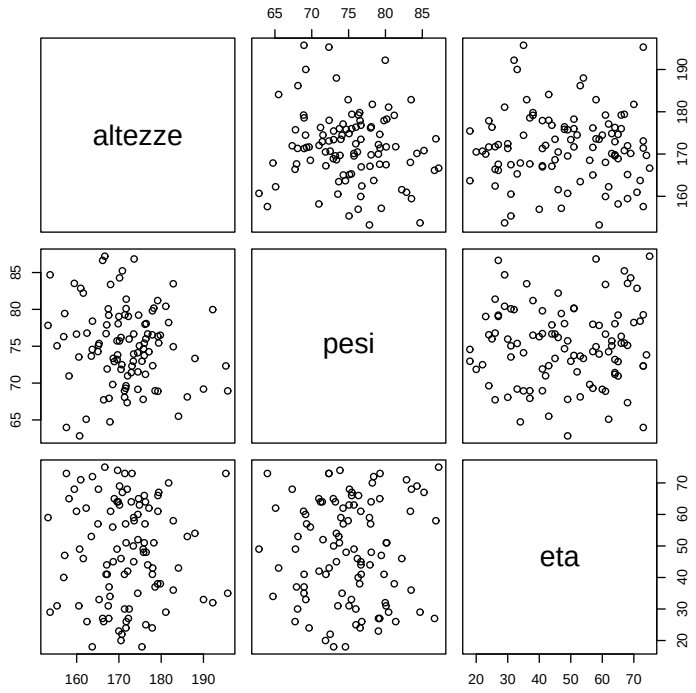
Rappresentiamo ora in uno stesso grafico le due variabili `x` e `y`, indicandole con due simboli diversi e inserendo una legenda:

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, pch="+",ylim=c(-3.5,3.5),ylab="Variabili")
> points(y, pch="*")
> legend(90,3.5,c("x","y"),pch=c("+","*"),bg="gray")
```



Se si ha a che fare con un data frame può essere utile avere una matrice di grafici:

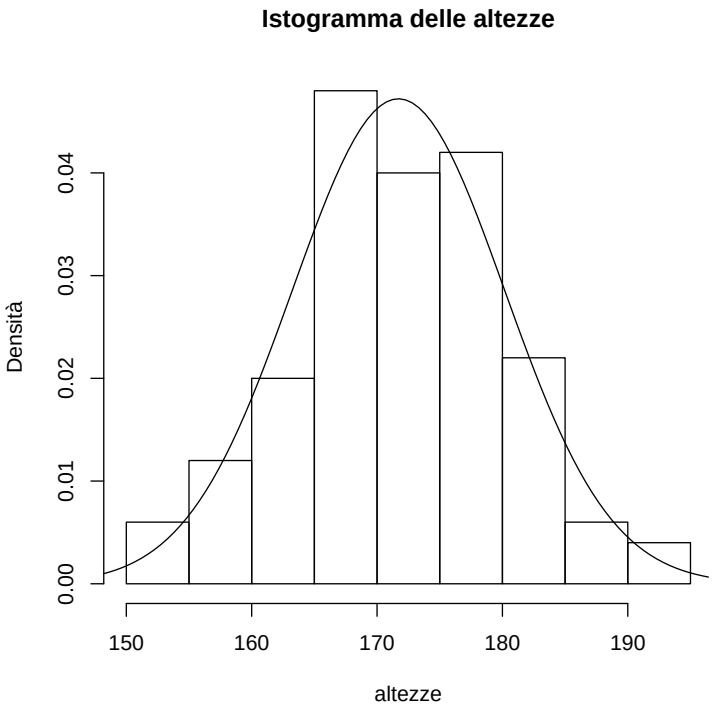
```
> altezze<-rnorm(100,170,10)
> pesi<-rnorm(100,75,5)
> x<-18:75
> eta<-sample(x,100,replace=TRUE)
> dati<-data.frame(altezze,pesi,eta)
> pairs(dati)
```



Il comando `sample()`, utilizzato sopra, prende un campione di ampiezza 100 nel nostro caso tra gli elementi del vettore `x`, supponendo che tutti gli elementi di `x` abbiano la stessa probabilità di essere estratti.

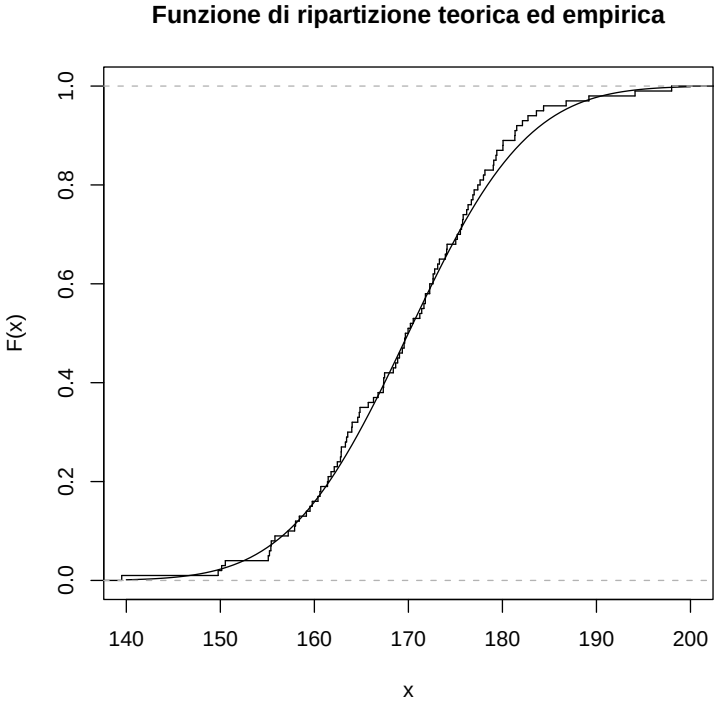
Come esempi finali di questa sezione si consideri il caso di dover costruire un grafico in cui si sovrappongono un istogramma costruito su un insieme di dati con la distribuzione normale di parametri la media e la varianza calcolati direttamente sui dati:

```
> altezze<-rnorm(100,170,10)
> media.altezze<-mean(altezze)
> sd.altezze<-sd(altezze)
> min.altezze<-min(altezze)
> max.altezze<-max(altezze)
> hist(altezze,freq=FALSE,ylab="Densità",main="Istogramma delle altezze")
> plot(function(x) dnorm(x,media.altezze,sd.altezze),
          min.altezze-5, max.altezze+5,add=TRUE)
```



Supponiamo ora di voler costruire un grafico che contiene la funzione di ripartizione empirica costruita sui dati e la corrispondente funzione di ripartizione teorica; esiste un unico comando che permette di rappresentare graficamente la funzione di ripartizione empirica e questo comando è contenuto nel package `stepfun`:

```
> library(stepfun)
> altezze<-rnorm(100,170,10)
> plot(function(x) pnorm(x,170,10), 140, 200, ylab = "F(x)",
          main="Funzione di ripartizione teorica ed empirica")
> plot(ecdf(altezze),verticals=TRUE,do.points=FALSE,add=TRUE)
```



6.3.6 Consigli per l'utilizzo di grafici sotto Windows

Anche in questo caso si dà qualche semplice consiglio utile quando si effettuano dei grafici in **R** sotto ambiente **Windows**.

- **Salvare/Copiare un grafico come Metafile piuttosto che come Bitmap.**
  Dopo avere prodotto un grafico in **R**, si potrebbe copiare e incollare il grafico nel processore di testi utilizzato (solitamente **Word**). In questo caso il formato grafico utilizzato è il *bitmap* che non risulta idoneo per riadattamenti del grafico ed inoltre prende molta memoria il mantenerlo negli appunti. D'altronde, i grafici in formato *JPEG* e *GIF* hanno qualità molto scarsa. Così si consigliano i grafici in formato *metafile*, considerato che vi è anche il vantaggio che questi grafici occupano meno spazio rispetto ai grafici in formato *bitmap*. Per copiare un file in formato *metafile* in **Word** è possibile utilizzare gli opportuni menù per fare operazioni di taglia/incolla.

- **Salvare i grafici nel drive C.** Quando si copia/incolla un grafico da **R** in **Word**, è una buona idea salvare il file su cui si sta lavorando molto spesso e possibilmente sul drive `C:` (ad esempio sul Desktop). Se si tenta di salvarlo direttamente su floppy, si potrebbe verificare un blocco del programma, perdendo in questo modo tutto il lavoro. Una delle Leggi di Murphy dice che *la probabilità che un computer si blocchi diventa sempre più alta all'avvicinarsi della fine del lavoro*, quindi si cerchi di evitare di perdere il lavoro fatto, salvando spesso su disco fisso.

Capitolo 7

Calcoli di tipo statistico

7.1 Alcune funzioni statistiche di frequente utilizzo

In questo paragrafo vedremo alcune delle funzioni che permettono il calcolo dei più importanti indici descrittivi. Abbiamo già visto in un precedente paragrafo le funzioni `mean()` e `var()`: in particolare,

- la funzione `mean()` calcola la media aritmetica di un vettore di osservazioni oppure, specificando l'argomento `trim`, una media α-trimmed;
- la funzione `var()` calcola la varianza campionaria di un vettore di osservazioni, se come argomento della funzione viene passato appunto un vettore; collegata alla funzione `var()` è la funzione `cov()`: se come argomenti queste due funzioni hanno due vettori `x` e `y` allora, utilizzando una delle due funzioni, viene restituita la covarianza di `x` e `y`; se, invece, l'argomento di `var()` e `cov()` è una matrice `X`, si avrà il calcolo della matrice di varianza e covarianza, considerando come variabili le colonne della matrice `X`; se, invece, argomenti di `var()` e `cov()` sono due matrici `X` e `Y`, si avrà il calcolo della matrice di varianza e covarianza tra le colonne di `X` e le colonne di `Y`;
- la funzione `cor()` permette di calcolare il coefficiente di correlazione lineare ed ha un utilizzo simile a quello della funzione `cov()`: in particolare, `cor()` calcola il coefficiente di correlazione di due vettori `x` e `y`, se argomenti di questa funzione sono questi due vettori; se, invece, l'argomento di `cor()` è una matrice `X`, si avrà il calcolo della matrice di correlazione, considerando come variabili le colonne della matrice `X`; se, invece, argomenti di `cor()` sono due matrici `X` e `Y`, si avrà il calcolo della matrice di correlazione tra le colonne di `X` e le colonne di `Y`;
- se si vuole calcolare lo scarto quadratico medio su un campione di osservazioni contenute nel vettore `x`, si può utilizzare la funzione `sd()`; se, invece, come argomento di `sd()` si utilizza una matrice o un data frame si otterrà il vettore di scarti quadratici medi delle colonne della matrice o del data frame.

- altra funzione molto utilizzata è `median()`, che restituisce la mediana di un vettore;
- se di un vettore di osservazioni `x` si vogliono calcolare il minimo, il massimo, i quartili e la media aritmetica si può utilizzare il comando `summary(x)`;
- per calcolare i quantili si può utilizzare la funzione `quantile()`; risultati simili a quelli ottenuti con la funzione `quantile()` si hanno con la funzione `fivenum()`, che fornisce i cosiddetti cinque numeri di Tukey, che approssimativamente corrispondono ai tre quartili più il minimo e il massimo. Tra l'altro i valori calcolati con la funzione `fivenum()` sono le quantità che **R** utilizza per la costruzione dei boxplot. Per maggiori dettagli sulle lievi differenze nel risultato utilizzando `quantile()` e `fivenum()` si rimanda alla consultazione dell'help in linea.

7.2 Distribuzioni di probabilità

In **R** è possibile calcolare la funzione di densità di probabilità, la funzione di ripartizione o i quantili delle principali distribuzioni di probabilità. E' inoltre possibile generare osservazioni pseudo-casuali da queste distribuzioni. Un elenco delle distribuzioni disponibili nel package `base` è il seguente:

Distribuzione	Nome in R	Argomenti addizionali
beta	`beta`	`shape1`, `shape2`, `ncp`
binomiale	`binom`	`size`, `prob`
binomiale negativa	`nbinom`	`size`, `prob`
Cauchy	`cauchy`	`location`, `scale`
chi-quadrato	`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 di Student	`t`	`df`, `ncp`
uniforme	`unif`	`min`, `max`
Weibull	`weibull`	`shape`, `scale`
Wilcoxon	`wilcox`	`m`, `n`

Per ottenere la funzione di densità, si antepone al nome della distribuzione una `d`, per la funzione di ripartizione una `p`, per il quantile una `q` e per generare osservazioni pseudo-casuali una `r`. Ad esempio, con il seguente comando viene calcolato il p-valore sulle due code di una distribuzione t di Student con 15 gradi di libertà in corrispondenza dei punti -2.43 e 2.43:

```
> 2*pt(-2.43, df = 13)
```

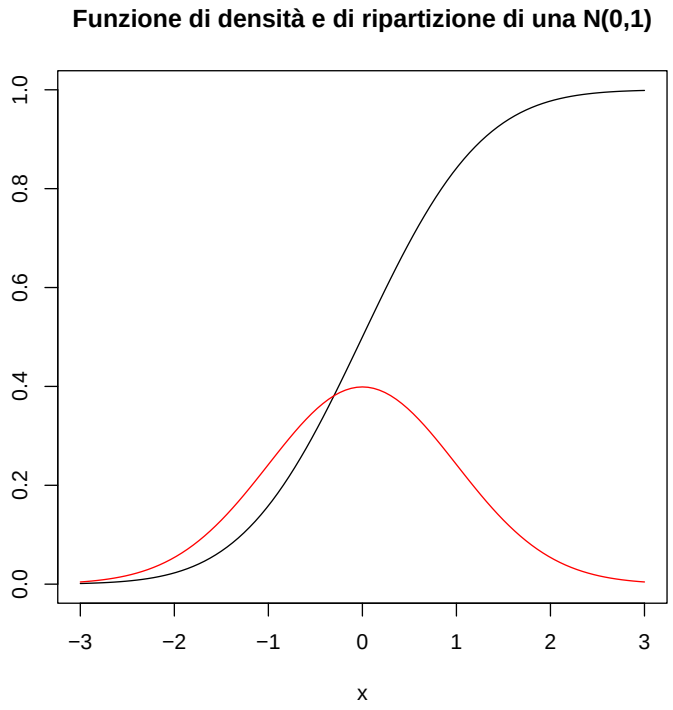
```
[1] 0.0303309
```

Con il seguente comando viene invece calcolato il quantile in corrispondenza di una probabilità nella coda superiore di una distribuzione F(2, 7) pari all'1%:

```
> qf(0.99, 2, 7)
[1] 9.546578
```

E' facile rappresentare graficamente la funzione di densità di una variabile casuale. Consideriamo il caso in cui si vogliano rappresentare la funzione di densità e la funzione di ripartizione di una variabile casuale normale standardizzata:

```
> plot(function(x) pnorm(x), -3, 3, ylab = "",
      main = "Funzione di densità e di ripartizione di una N(0,1)")
> plot(function(x) dnorm(x), -3, 3, col = "red", add = TRUE)
```



7.3 Modelli lineari

Indichiamo con `y`, `x`, `x0`, `x1`, `x2`, ... dei vettori numerici, con `X` una matrice, con `A`, `B`, `C`, ... dei fattori. Allora si potrebbero avere i seguenti modelli:

`y ~ x` `y ~ 1 + x`	indicano il modello di regressione lineare semplice di `y` su `x`. Il primo ha implicita l'intercetta, il secondo esplicita.

`y ~ 0+x`
`y ~ -1+x`
`y ~ x-1`
	regressione lineare semplice di `y` su `x`
	attraverso l'origine, cioè con intercetta nulla.

`log(y) ~ x1+x2`	regressione multipla della variabile trasformata, `log(y)`, su `x1` e `x2`, con intercetta implicita.
`y ~ poly(x,2)` `y ~ 1+x+I(x^2)`	regressione polinomiale di `y` su `x` di grado 2. Il primo usa polinomi ortogonali, il secondo potenze esplicite.
`y ~ X+poly(x^2)`	regressione multipla di `y` con matrice del modello formata da `X` e dai termini polinomiali in `x` fino al grado 2.
`y ~ A`	ANOVA ad una via con classi determinate da `A`.
`y ~ A+x`	ANCOVA con classi determinate da `A` e covariata `x`.

`y ~ A*B`
`y ~ A+B+A:B`
`y ~ B%in%A`
`y ~ A/B`
	modello non additivo a due fattori di `y` su `A` e `B`.
	I primi due specificano la stessa classificazione incrociata,
	i secondi due specificano la stessa classificazione annidata.

`y ~ (A+B+C)^2` `y ~ A*B*C-A:B:C`	esperimento a tre fattori con un modello che contiene solo gli effetti principali e le interazioni a due fattori.

`y ~ A*x`
`y ~ A/x`
`y ~ A/(1+x)-1`
	modelli di regressione lineare semplice di `y` su `x` separati
	nei livelli di `A`, con differente codifica. L'ultima forma dà stime
	esplicite di tante intercette e pendenze quanti sono i livelli di `A`.

L'operatore `~` è usato per definire un modello in **R**. La forma, per un ordinario modello lineare, è

```
risposta ~ op1 term1 op2 term2 ...
```

dove `risposta` è un vettore o una matrice che definisce la o le variabili di risposta; `opi` è l'operatore i-esimo, `+` o `-`, che implica l'inclusione o l'esclusione di un termine nel modello; `termi` è o un vettore o una matrice (compreso 1), o un fattore, o una formula comprendente fattori, vettori o matrici connesse agli operatori. La funzione `I()` in questo contesto è utilizzata per inibire l'interpretazione di operatori come `+`, `-`, `*` e `^` come operatori della formula, ma per utilizzarli come operatori aritmetici.

La funzione base per adattare modelli lineari multipli ordinari è `lm()`:

```
> modello.adattato <- lm(formula, data = data.frame)
```

Il valore di `lm()` è l'oggetto modello adattato, che tecnicamente consiste di una lista di risultati della classe `lm`. Per effettuare un'analisi della varianza si può utilizzare anche la funzione `aov(formula, data=data.frame)`. Le informazioni sul modello adattato possono essere visualizzate utilizzando opportune funzioni, elencate di seguito:

`add1`	`coef`	`effects`	`kappa`	`predict`	`residuals`
`alias`	`deviance`	`family`	`labels`	`print`	`step`
`anova`	`drop1`	`formula`	`plot`	`proj`	`summary`

I comandi più comunemente utilizzati sono i seguenti:

`anova(oggetto)`	Restituisce la tavola di analisi della varianza.
`deviance(oggetto)`	Somma dei quadrati residua, nel caso pesata.
`formula(oggetto)`	Estrae l'espressione del modello.
`plot(oggetto)`	Produce quattro grafici, utilizzati per l'analisi dei residui.
`predict(oggetto, newdata=data.frame)`	Il data frame fornito deve avere variabili specificate con le stesse etichette delle originali. Restituisce un vettore o una matrice di valori predetti che corrispondono ai valori che determinano le variabili in `data.frame`.
`print(oggetto)`	Stampa una versione concisa dell'oggetto.
`residuals(oggetto)` `resid(oggetto)`	Estrae la matrice di residui, nel caso pesati.
`step(oggetto)`	Seleziona un opportuno modello aggiungendo o eliminando termini e preservando gerarchie. Viene restituito il modello con il più grande valore di AIC (Akaike Information Criterion) ottenuto nella ricerca stepwise.
`summary(oggetto)`	Stampa un ampio riassunto dei risultati dell'analisi di regressione.

7.3.1 Aggiornare i modelli adattati

La funzione `update()` permette di adattare un modello che differisce da uno precedentemente adattato per qualche termine aggiuntivo o rimosso:

```
nuovo.modello <- update(vecchio.modello, nuova.formula)
```

In `nuova.formula` il carattere speciale `.` può essere usato per indicare la corrispondente parte della vecchia formula. Per esempio

```
> fm5 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = production)
> fm6 <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

adatterà una regressione multipla a cinque variabili, con le variabili prese dal data frame `production`, adatta un ulteriore modello che include un sesto regressore `x6` e adatta una variante sul modello prendendo come variabile di risposta la radice quadrata di `y`. Il simbolo `.` può essere usato anche in altri contesti. Ad esempio

```
> fmpieno <- lm(y ~ . , data = production)
```

adatterà rispettivamente un modello con risposta `y` e regressori tutte le altre variabili presenti nel data frame `production`.

7.3.2 Esempio di un'analisi di regressione lineare semplice

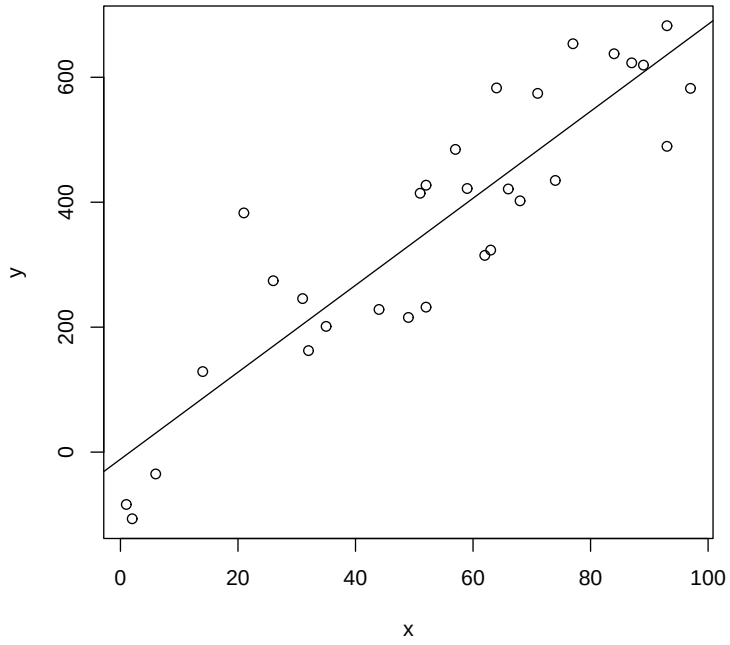
I dati che utilizzeremo in questa sezione sono dati fittizi creati con alcune delle funzioni di generazione fornite da **R**: è scontato quindi il fatto che se si cerca di ripetere questa analisi utilizzando gli stessi comandi che vedremo, si otterranno

in generale risultati diversi. Indichiamo al solito con `y` la variabile di risposta e con `x` la variabile esplicativa:

```
> x <- 1:100
> x <- sample(x, 30, replace = TRUE)
> x
 [1]  76   5   5  62  16  49 100  84  61  44  20  94  80  42  85  90
[17]  88  29  82  48  41  48  77  23  32  43  74   3  47  74
> y <- 3 + 7 * x + rnorm(30, 0, 100)
> y
 [1] 630.61936 150.28477  64.92860 468.33847 174.66836 392.49618 881.98688
 [8] 368.02340 331.03066 329.55150 244.94382 576.24187 594.90326 190.57372
[15] 742.37795 616.81531 800.78326 193.52706 528.43952 320.81466 198.79310
[22] 437.28956 575.91533 258.07386 257.14912 378.82705 721.18819 -36.18574
[29] 203.67833 563.60960
> res.reg <- lm(y ~ x)
> summary(res.reg)
Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-251.581  -68.392    8.107   73.430  173.170

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   18.281     39.752    0.46    0.649
x              7.159      0.651   11.00 1.14e-11 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 101.2 on 28 degrees of freedom
Multiple R-Squared: 0.812,  Adjusted R-squared: 0.8053
F-statistic: 120.9 on 1 and 28 DF,  p-value: 1.135e-11
```

Dal precedente `summary()` si può vedere, tra l'altro, come le stime dei parametri siano pari a 18.281 per l'intercetta e a 7.159 per il coefficiente angolare, mentre la parte di varianza spiegata dal modello risulta elevata ($R^2 = 0.812$). Se si vuole effettuare il grafico che contiene i dati e la retta stimata, i comandi da utilizzare sono i seguenti:

```
> plot(x, y)
> abline(res.reg)
```

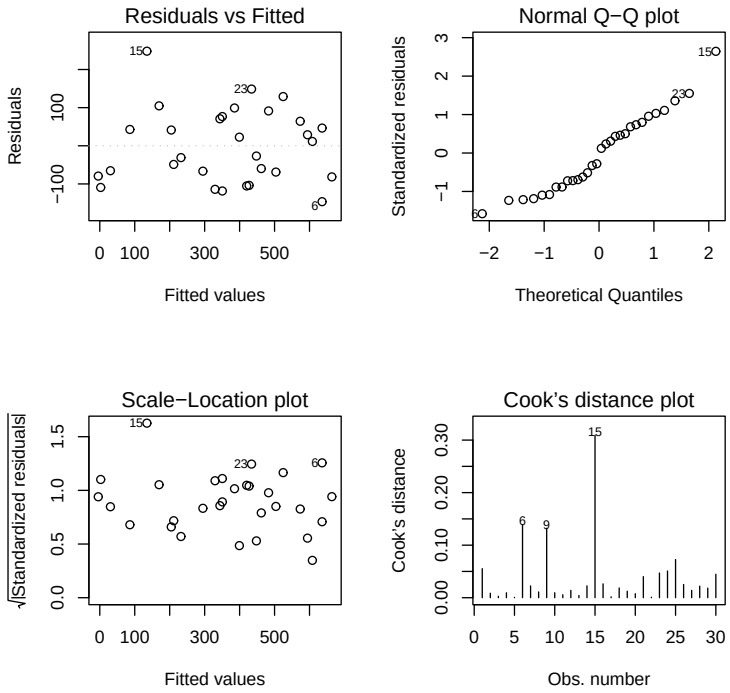


dal quale si può notare un buon adattamento della retta ai dati. Se si vuole visualizzare la tabella di analisi della varianza per questo modello di regressione, il comando da utilizzare è il seguente:

```
> anova(res.reg)
Analysis of Variance Table

Response: y
          Df  Sum Sq Mean Sq F value    Pr(>F)
x          1 1238499 1238499  120.92 1.135e-11 ***
Residuals 28  286778   10242
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Se si vuole effettuare un'analisi dei residui di tipo grafico, allora si possono utilizzare i seguenti comandi:

```
> par(mfrow = c(2, 2))
> plot(res.reg)
> par(mfrow = c(1, 1))
```



Il comando `par(mfrow=c(2,2))` è servito per rappresentare i quattro grafici forniti da **R** in un'unica finestra divisa in quattro parti (due righe e due colonne); l'ultimo comando ha ristabilito la situazione originaria. In particolare, i grafici visualizzati ci danno informazioni sugli errori accidentali relativamente all'indipendenza (grafico in alto a sinistra), all'ipotesi di normalità (grafico in alto a destra), alla omoschedasticità (grafico in basso a sinistra) e alla eventuale presenza di valori anomali (grafico in basso a destra). Nel caso specifico, non sembrano esserci forti violazioni delle ipotesi di base fatte sugli errori accidentali.

Capitolo 8

Elementi di programmazione in R

8.1 Loops ed esecuzioni condizionali

In **R** i comandi possono essere raggruppati usando parentesi graffe

```
{expr 1;
...;
expr n;}
```

nel qual caso il valore del gruppo di istruzioni è il risultato dell'ultima espressione del gruppo. E' possibile usare espressioni condizionali del tipo:

```
> if (expr1) expr2 else expr3
```

dove `expr1` ha come risultato un valore logico. Nell'espressione logica si possono utilizzare gli operatori relazionali: `x < y`, `x > y`, `x <= y`, `x >= y`, `x == y` e `x != y`. Si possono anche utilizzare gli operatori NOT (`!`), AND (`&` e `&&`) e OR (`|` e `||`): la forma più corta esegue confronti elemento per elemento in modo simile agli operatori aritmetici, la forma più lunga valuta da sinistra a destra l'espressione esaminando solo il primo elemento di ogni vettore. Il risultato del precedente comando è pari a `expr2` se il risultato di `expr1` è `TRUE`, altrimenti il risultato di questa istruzione è `expr3`. Esiste una versione vettorizzata di *if/else* che ha la forma

```
ifelse(condizione, a, b)
```

dove il risultato è un vettore pari alla componente di `a` o di `b` a seconda se di volta in volta il risultato di `condizione` è `TRUE` o `FALSE`. Per chiarire meglio il funzionamento dei due costrutti `if else` e `ifelse()`, vediamo due esempi:

```
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> if (x<5) y<-1 else y<-2
Warning message:
the condition has length > 1 and only the first element will be
```

```
used in: if (x < 5) y <- 1 else y <- 2
> y
[1] 1
```

In questo caso è stato valutato solo il primo elemento del vettore `x` e, risultando `TRUE` la condizione, l'espressione che viene presa in considerazione è `y<-1`, per cui alla fine il valore di `y` sarà proprio pari ad 1. E' da notare come comunque **R** abbia prodotto un messaggio di avvertimento che spiega come nel valutare la condizione sia stato utilizzato solo il primo elemento di `x`. Un esempio di utilizzo per il costrutto `ifelse()` è invece il seguente:

```
> ifelse(x<5, 1, 2)
[1] 1 1 1 1 2 2 2 2 2 2
> y<-ifelse(x<5, 1, 2)
> y
[1] 1 1 1 1 2 2 2 2 2 2
```

Come si può notare, in questo caso viene valutata la condizione `x<5` per ogni valore di `x` per cui alla fine il risultato è un vettore di 1 e di 2 che può, ad esempio, essere assegnato al vettore `y`.

Esiste anche il costrutto loop che ha la forma:

```
> for(nome in expr1) expr2
```

dove `nome` è la variabile di loop e `expr1` è di solito una sequenza del tipo `1:20`.

Altri comandi per eseguire loop sono:

```
> repeat expr
> while(condizione) expr
```

Il comando `break` può essere usato per terminare qualsiasi loop ed è il solo modo per terminare un loop con `repeat`. Il comando `next` può essere usato per interrompere un particolare ciclo e passare alle istruzioni successive al `next`.

8.2 Scrivere proprie funzioni

Una funzione è definita da un'assegnazione del tipo:

```
> nome <- function(arg1, arg2, ...) espressione
```

dove `espressione` può essere in realtà un gruppo di espressioni racchiuse da parentesi graffe:

```
> nome <- function(arg1, arg2, ...)
{
espressione1
espressione2
espressione3
...
}
```

Il risultato della funzione sarà in generale l'ultimo valore calcolato.

Una chiamata di funzione è fatta di solito nel seguente modo:

```
> nome(arg1, arg2, ...)
```

Scrivere delle funzioni è il modo più semplice in **R** per memorizzare una serie di comandi. Scritta una funzione si può utilizzare il comando `edit(nomefunzione)` per richiamare un editore di testo (sotto **Windows** solitamente il *Blocco Note*) che visualizza il corpo della funzione che quindi, con piccole modifiche, può essere salvata. In una successiva sessione le funzioni possono essere richiamate utilizzando il comando `source("nomefile")`, oppure utilizzando la voce "Source R code" dal menu "File". Ogni assegnazione ordinaria fatta in una funzione è locale e temporanea e viene persa quando si esce dalla funzione.

8.3 Come creare un proprio package

R è accompagnato da diversi packages che coprono le più svariate analisi statistiche. Ogni utilizzatore, però, può scrivere un package di proprie funzioni, con relativi file di help, esempi e dati, che poi può essere reso di pubblico dominio, sottoponendolo al **CRAN**.

In questo paragrafo si farà esplicito riferimento alla creazione di un package sotto **Linux**, rimandando al prossimo paragrafo per eventuali differenze che si hanno nel caso in cui si disponesse del sistema operativo **Windows**.

Il primo passo per la realizzazione di un proprio package è la creazione nel filesystem del proprio sistema di una cartella, a cui bisogna dare lo stesso nome del package. In questa cartella verrano creati tutti i file e le sottocartelle, necessari alla compilazione del package.

I file contenuti nella cartella principale sono `DESCRIPTION` ed `INDEX`. Nel primo file sono racchiuse tutte le principali informazioni sul package, come il nome, la versione, la data di creazione, gli autori, il tipo di licenza e altre informazioni. Molte delle informazioni qui contenute verranno utilizzate in fase di compilazione del package. Il file `INDEX` contiene i nomi dei comandi ed una loro breve descrizione. Questo file può essere creato manualmente, ma anche attarverso il comando

```
$ R CMD Rdindex man > INDEX
```

dove `$` sta ad indicare uno dei simboli di prompt utilizzati da **Linux**.

Dopo questi due file, bisogna creare delle sottocartelle alcune delle quali potrebbero anche non essere strettamente necessarie. Le sottocartelle sono `R`, `man`, `data`, `inst`, `src`, `tests`. La cartella `R`, che è tra quelle che comunque deve essere sempre presente, conterrà i file contenenti il codice in **R**. Il nome dei file di codice dovrà iniziare per lettera, ed avere una tra queste estensioni: `.R`, `.S`, `.q`, `.r`, `.s`. E' consigliato l'utilizzo dell'estensione `.R` perché non viene utilizzata solitamente da nessun altro software.

Un'altra cartella che non è possibile omettere è la cartella `man`, che contiene i file di help per ogni comando incluso nel package. Questi file dovono avere estensione `.Rd` o `.rd`. L'elaboratore di testi utilizzato per scrivere i file di help è LaTeX, col quale è stato scritto anche questo manuale. Questo elaboratore di testi permette al compilatore di **R** di creare per ogni file `.Rd`, dei file di help in formato LaTeX, `HTML` e `testo`.

Accennando brevemente alle altre sottocartelle che non sono indispensabili per la compilazione di un package, `data` contiene dei file di dati e tabelle, file che

possono essere di tre tipi differenti: codice, tabella ed immagine; quest'ultima viene creata con il comando `save()`. Questi file di dati vengono caricati da **R** con il comando `data()`. Nella sottodirectory `tests` vanno salvati dei file con estensione `.R` o `.Rin`, contenenti dei codici per saggiare il funzionamento del package.

Dopo avere creato i file e le sottocartelle, verifichiamo che non vi siano errori. Per fare ciò, digitiamo da terminale l'istruzione

```
$ R CMD check /path/nomelib/
```

Questo comando prima di tutto verifica se il package si installa correttamente. Poi analizza il file `DESCRIPTION`, verificando che nulla sia stato omesso. Dopo di ciò controlla che per ogni comando del package corrisponda un file di help, e che i file di help siano stati scritti correttamente, verificando il corretto funzionamento degli esempi, presenti alla fine dei file con estensione `.Rd`. Il controllo del package termina con la creazione di una cartella denominata `/nomelib.check`. In questa cartella troveremo il file di log `00check.log`, dove vi è la traccia di tutte le verifiche fatte dal programma, registrando quindi ogni possibile errore occorso durante la compilazione. Essa contiene inoltre la libreria compilata ed un manuale di riferimento in formato LaTeX.

Se il controllo è andato a buon fine, bisogna a questo punto creare il package. Il comando per creare il package è

```
$ R CMD build /path/nomelib
```

Alla fine della compilazione si avrà un file compresso, con estensione `tar.gz`, il cui nome è composto dal nome del package, unito al numero di versione specificato nel file `DESCRIPTION`. Questo file contiene la cartella, le sottocartelle e i file creati per il package, compressi con il software **gzip**.

A questo punto non rimane che l'installazione. Il comando da utilizzare è

```
$ R CMD INSTALL nomelib.tar.gz
```

Per poter utilizzare il package, aprendo una sessione di **R** e digitando

```
> library(nomelib)
```

si potranno utilizzare i nuovi comandi forniti dal package.

8.4 Creare un package sotto Windows

Perché si riescano a compilare correttamente i package sotto **Windows**, è necessario che siano installati alcuni compilatori che solitamente non si trovano già installati in ambiente **Windows**, come il **Perl**, il **C++** ed il **Fortran**. Tutte le utility necessarie per la creazione e l'installazione di package sotto **Windows**, oltre alle indicazioni su come procedere, si trovano nel sito `http://www.stats.ox.ac.uk /pub/Rtools/`.

I passi sostanziali per la creazione di un package a questo punto diventano simili a quelli visti nel paragrafo precedente; in particolare, se si vuole effettuare un controllo per verificare che tutto funzioni bene, il comando da digitare nella finestra **DOS** è il seguente:

```
> Rcmd check percorso_alla_cartella_del_package
```

dove > indica il prompt della finestra **DOS**.

E' possibile anche creare una versione del package immediatamente installabile sotto **Windows**. Sotto questo sistema operativo, digitando nella finestra **DOS** l'istruzione

```
> Rcmd build --binary percorso_alla_cartella_del_package
```

si otterrà un file con estensione `.zip` che è immediatamente installabile, ad esempio, con il comando

```
> Rcmd install nome_package
```

In ogni caso, nella versione di **R** per **Windows**, l'installazione di un package è molto semplice, se si è avviato **R** con interfaccia grafica: basta selezionare la voce "Install package from local zip file" dal menù "packages". Dalla finestra che apparirà, si seleziona il file con estensione `.zip` che contiene il package. Terminata questa operazione, il package potrà essere utilizzato richiamandolo con il comando `library()`.

E' da notare comunque come dalla versione 1.7.0 di **R** è possibile compilare un package per **Windows** anche utilizzando il sistema operativo **Linux**. Per maggiori dettagli si veda Jun Yan e Rossini (2003).

Indice analitico

Bibliografia

[1] M. J. Crawley. *Statistical Computing. An Introduction to Data Analysis using S-plus.* Wiley, Chichester, England, 2002.

[2] P. Dalgaard. *Introductory Statistics with R.* Springer-Verlag, New York, 2002.

[3] J. Fox. *An R and S-plus Companion to Applied Regression.* SAGE Pubblications, Thousand Oaks, California, 2002.

[4] J. Maindonald. Using R for Data Analysis and Graphics. Published on the URL: http://wwwmaths.anu.edu.au/~johnm, 2000.

[5] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer-Verlag, New York, 2002.

[6] W. N. Venables, D. M. Smith, and the R Development Core Team. An Introduction to R. Published on the URL: http://CRAN.R-project.org/manuals.html, 2003.

[7] Ko-Kang Wang. R for Windows Users. Published on the URL: http://www.stat.auckland.ac.nz/ kwan022/pub/R/WinBook/, 2003.

[8] Jun Yan and A.J. Rossini. Building Microsoft Windows Versions of R and R packages under Intel Linux. *R News*, 3(1):15–17, June 2003.