Il linguaggio R:
concetti introduttivi ed esempi

versione 1.0 – giugno 2002

Vito M. R. Muggeo
vito.muggeo@giustizia.it

Indice

Note Preliminari ii

Introduzione **1**

1 Nozioni di Sintassi **2**
1.1 Le funzioni in R 4
1.2 Organizzazione del lavoro 5

2 Vettori, Matrici, Array e Liste **6**
2.1 Vettori 6
2.2 Matrici 8
2.3 Array 10
2.4 Liste 11

3 Il dataframe **12**
3.1 Importazione di dati 13
3.2 Valori mancanti 13
3.3 Codifica di variabili 14

4 Sintesi di una distribuzione **16**
4.1 Qualche statistica descrittiva 16
4.2 Le funzioni `tapply()` e `apply()` 19
4.3 Alcune rappresentazioni grafiche 20

5 Cenni ai Modelli Lineari Generalizzati **23**
5.1 Cenni teorici 24
5.2 Aspetti pratici 25

Note preliminari

Una prima bozza di questo materiale risale alle esercitazioni che ho avuto modo di fare con gli studenti del corso di 'Statistica Sociale (Laboratorio)' del Prof. Attanasio dell'Università di Palermo nel 2000, quando frequentavo il dottorato. Il tutto è stato riordinato e sistemato con lo scopo di diffondere R*, soprattutto tra gli studenti di Statistica e gli statistici in generale; esso può essere distribuito gratuitamente, purché nella sua integrità: vedi la nota sotto. Comunque chiunque intenda stampare su carta queste dispense e voglia 'contraccambiare' questo lavoro, può farlo semplicemente stampandolo su carta riciclata. La carta bianca che normalmente si usa viene ottenuta abbattendo alberi ed utilizzando cloro per sbiancarla. Se non hai a disposizione carta riciclata, questa è una buona occasione per iniziare: utilizzando e promuovendo la sua diffusione tra gli amici/colleghi contribuisci a preservare le foreste della Terra. Le generazioni future ringraziano.*

Lecce, giugno 2002

Suggerimenti e osservazioni sono ben accette.

La creazione e distribuzione di copie fedeli di questo manuale è concessa a patto che la nota di copyright e questo permesso stesso vengano distribuiti con ogni copia. Copie modificate di questo manuale possono essere copiate e distribuite alle stesse condizioni delle copie fedeli, a patto che il lavoro risultante venga distribuito con la medesima concessione.

Copyright © 2002 Vito M.R. Muggeo

Introduzione

Queste dispense hanno lo scopo di illustrare i fondamenti logici ed applicativi per l'impiego di R. Piuttosto che definire R come un software statistico, esso deve essere definito come un ambiente, ovvero un insieme di macro, librerie, oggetti che possono essere utilizzati per la gestione, l'analisi dei dati e la produzione di grafici; il termine R o ambiente verrano utilizzati indifferentemente. Una conseguenza immediata di questo è che allorquando ci si appresta a lavorare con l'ambiente la classica domanda "È possibile in R implementare....?", dovrebbe essere sostituita da "Quanto è difficile in R implementare...?"

R è basato sul linguaggio S a cui è strettamente legato un altro 'ambiente' commerciale probabilmente più conosciuto, S-Plus. R, a differenza di S-Plus, è un *GNU-Sofware*, ovvero disponibile gratuitamente sotto i vincoli della GPL (General Public Licence). Nel loro utilizzo base, al quale queste dispense sono rivolte, tali due *dialetti* sono molto simili per cui ciò che è di seguito riportato per R potrebbe essere utilizzato anche per S-Plus, sebbene le differenze tra i due linguaggi non sono evidenziate.

È bene ribadire che, proprio perché si tratta di ambiente, per un determinato problema possono esistere (ed in generale esistono) diverse soluzioni, tutte ugualmente valide. Conseguentemente i metodi ed i risultati che di seguito sono riportati, essendo il frutto di una (appena) triennale esperienza dell'autore, potrebbero essere implementati in modi differenti, tutti ugualmente validi: la flessibilità di R è la sua principale caratteristica.

Il lavoro è organizzato sommariamente in due parti. I primi 3 paragrafi evidenziano i principali elementi che è necessario conoscere per capire come è fatto R e come è stato pensato al fine di iniziare a familiarizzare con l'ambiente ed acquisire un certo livello di 'comprensione di ciò che si sta facendo'. La seconda parte è più a carattere operativo, nel senso che vengono illustrati i procedimenti utilizzati per impostare un'analisi statistica dei dati: indici descrittivi e modellazione. In questo contesto il lettore dovrebbe aver già chiari i concetti di 'matrice dei dati', 'tipo di variabili', 'distribuzioni' e 'modello di regressione'.

Come il titolo evidenzia, lo scopo del presente lavoro è semplicemente fornire nozioni introduttive al linguaggio: dopo una completa lettura, possibilmente accompagnata da un 'parallelo' utilizzo di R, il lettore dovrebbe riuscire (almeno si spera) ad aver chiaro come lavorare con R, e dovrebbe aver acquisito le basi sufficienti per leggere con maggiore semplicità i (numerosi) documenti relativi ad una discussione più approfondita di R: vedi sulla R *home-page*, http://www.r-project.org. Si tenga presente, che poiché la trattazione è mantenuta ad un livello base e cerca di raccogliere gli aspetti essenziali operativi di R, alcuni precisazioni tecniche sono state volutamente omesse ed alcuni termini potrebbero risultare non adeguatamente impiegati ad un lettore esperto che, comunque, potrebbe non trarre alcun vantaggio dalla lettura di queste dispense.

1 Nozioni di Sintassi

Tutti i comandi sono eseguibili dalla linea di comando dell'ambiente, caratterizzata dal prompt `>`. In ciò che segue i comandi digitati e i risultati dell'operazione vengono riportati in `questo font`.

Nel suo utilizzo più semplice, R può essere utilizzato come calcolatrice:

```
> 3+5*3.5
[1] 20.5
```

Quando il comando non è terminato risulta un prompt `+`; a questo punto è sufficiente continuare a scrivere sulla riga successiva

```
> 2^3-
+ 3
[1] 5
```

`[1]` indica il numero della linea in cui compare il risultato dell'operazione. Ad esempio eseguendo dalla stessa riga due operazioni distinte, intervallate con `;` si ha

```
> 3+5*(3.5/15)+5-(2/6*4); 3+2
[1] 7.833333
[1] 5
```

in quanto comunque le due operazioni sono distinte.

Piuttosto che fare eseguire le operazioni e lasciare che i risultati vengano stampati sullo schermo, può essere utile salvarli per poi guardarli, modificarli ed utilizzarli successivamente. Tutto ciò che è richiesto per salvare 'qualcosa' per utilizzarla in qualsiasi modo in seguito è costruire un *oggetto*. Il termine oggetto, d'ora in avanti, verrà impiegato per indicare qualsiasi cosa in R: espressioni, numeri, formule, insomma TUTTO. Per costruire un oggetto viene utilizzato il comando `<-` (o `->`). Le seguenti linee potranno rendere le idee più chiare:

```
> x<-2+(3-4*5)/2 #costruisci un oggetto
> x #guardalo
[1] -6.5
> x-2 #...utilizzalo
[1] -8.5
> x<-5 #...sovrascrivilo
> 2.5->x.x #crea un altro oggetto
> x/x.x #fai qualsiasi operazione con gli oggetti
[1] 2
```

Il simbolo `#` è il commento e viene ignorato dall'ambiente. Nell'esempio di sopra si è creato prima un oggetto `x`, poi un altro `x.x` ed infine è stata eseguita un operazione. Il punto `.` è un normale carattere in R (non ha nulla a che vedere con le estensioni dei file nel sistema operativo Windows).

Quindi, a differenza dei più comuni software tutte le operazioni eseguite possono essere salvate in un oggetto e utilizzate in seguito, semplicemente assegnando qualcosa ad un nome qualsiasi (ad es., `pippo<-`....): questo è un grande

vantaggio di R. Ad ogni modo si tenga presente che, nel momento in cui si decide di creare un oggetto, eventuali oggetti pre-esistenti con quello stesso nome saranno sovrascritti e quindi cancellati: conseguentemente bisogna fare molta attenzione a non creare oggetti con nomi già usati dall'ambiente, perché sia l'utente che R stesso potrebbero confondersi! Ad esempio `T` ha un significato ben preciso (vedi sotto) e quindi operazioni quali `T<-2+10` dovrebbero essere evitate. Con la pratica l'utente imparerà a conoscere i nomi che dovrebbero essere evitati, ma fino a quel momento per maggiore sicurezza si potrebbe interrogare R prima dell'assegnazione. Per esempio

```
> A
Error: Object "A" not found
```

così `A` è un nome da poter essere usato. È forse opportuno osservare che

```
> x<-2
> X
Error: Object "X" not found
```

cioè l'ambiente è *case-sensitive* così `x` e `X` sono differenti oggetti.

Le seguenti linee illustrano l'utilizzo dei 5 operatori `== >= <= > < !=`, che potranno risultare molto utili in seguito, soprattutto nella selezione di casi dalla matrice dei dati. Per il momento si osservi:

```
> 2*2==4
[1] TRUE
> 2*3==4
[1] FALSE
> 2*2>=4
[1] TRUE
> 2*3>4
[1] TRUE
> 2*3>=4
[1] TRUE
> 2*3!=4
[1] TRUE
```

`TRUE` (o la forma abbreviata `T`) e `FALSE` (o `F`) sono espressioni logiche corrispondenti a 1 e 0 rispettivamente, che come tali possono essere utilizzate per operazioni numeriche:

```
> TRUE+TRUE
[1] 2
> TRUE*T
[1] 1
> T+FALSE
[1] 1
> F/F
[1] NaN
```

```
> -T/F; T/F+101
[1] -Inf
[1] Inf
```

In particolare si osservino i risultati delle ultime tre operazioni: le forme indeterminate (ad esempio 0/0) e quelle tendenti ad infinito sono correttamente prese in considerazione da R. Tali operatori logici sono molto utilizzati per la gestione dei dati mancanti e soprattutto in fase di programmazione 'piuttosto avanzata'.

1.1 Le funzioni in R

Un insieme di comandi può essere scritto per assolvere un determinato compito. Ad esempio potremmo essere interessati a calcolare la media aritmetica dei primi 5 numeri:

```
> (1+2+3+4+5)/5
[1] 3
```

Piuttosto che scrivere specificatamente tutte le istruzioni, è possibile 'organizzarle' in maniera da ottenere una *funzione*. R è dotato di una miriade di funzioni che assolvono ai più 'comuni' calcoli matematici e soprattutto statistici (quali media aritmetica, varianza, quantili,....). Per eseguire una qualsiasi funzione `fn()` ed ottenere il relativo risultato è necessario digitare il suo nome specificando l'*argomento* fra parentesi tonde a cui `fn()` deve essere applicata. Le parentesi tonde dopo il nome indicano che si tratta di una funzione e non di un qualsiasi altro oggetto. Ad esempio supponiamo che `fn()` sia una funzione che semplicemente calcoli la metà del suo argomento `a`; quindi se `fn()` esistesse in R

```
> fn(a=5) #applica la funzione fn all'argomento 5
[1] 2.5
```

Piuttosto che dividere il suo argomento sempre per 2, `fn()` potrebbe consentire di specificare anche il denominatore, ovvero più in generale, di specificare un altro argomento che è necessario per eseguire la funzione correttamente. Allora se tale ulteriore argomento (il denominatore, in questo caso) fosse riconosciuto come `d`, si potrebbero scrivere entrambi separandoli con `,`:

```
> fn(a=12,d=3)
[1] 4
```

Generalizzando il discorso di sopra, un altro argomento potrebbe essere, ad esempio, un numero al quale il rapporto viene elevato, `e` diciamo:

```
> fn(a=15,d=5,e=3)
[1] 27
```

L'ordine degli argomenti non è importante a meno che non si voglia omettere il loro nome: così `fn(d=5,e=3,a=15)` o `fn(15,5,3)` hanno lo stesso significato.

Spesso argomenti 'meno importanti' hanno un proprio valore di default che viene utilizzato quando tali argomenti non sono specificati. Ad esempio, se fosse `e=2` per default, allora `fn(15,5)` sarebbe equivalente a `fn(15,5,2)`. Per ogni funzione `fn()`, R è dotato di un 'file di aiuto' visionabile digitando `?fn` o `help(fn)` dal prompt. Tale file chiarisce tutti gli argomenti richiesti dalla funzione, il loro ordine, quelli necessari e quelli opzionali, ovvero quelli per cui un valore di default è già impostato.

Forse la più semplice funzione è `rm()` che serve a cancellare gli oggetti, quindi:

```
> X<-2+0
> X
[1] 2
> rm(X)
> X
Error: Object "X" not found
```

Nell'esempio di sopra `X` è l'argomento della funzione `rm()`. La scrittura `rm(X)` può tradursi "applica la funzione `rm()` a `X`". Come è facile intuire, le funzioni sono alla base di ogni operazione in R, e in seguito verranno discusse le più importanti, caratterizzate quasi sempre da più argomenti. Ad esempio, `rm(x,y,Z)` cancellerà i 3 oggetti che figurano come argomenti. Particolari funzioni sono `q()` e `objects()` che non richiedono argomenti e terminano la sessione o stampano sullo schermo gli oggetti salvati nell'ambiente.

1.2 Organizzazione del lavoro

Ogni volta che sia avvia R, può essere molto utile specificare la *directory* di lavoro, ovvero la cartella dove salvare i risultati della sessione di lavoro: "File—Change dir..." consente di specificare il percorso desiderato (*path*). Questo è molto importante perché nella *directory* di lavoro R salva

- un file ascii (`.Rhistory`) che riporta tutto ciò che è stato digitato sul prompt;
- il *workspace* (spazio-di-lavoro) (`.Rdata`) che contiene tutti gli oggetti salvati nel sistema e che quindi possono essere utilizzati ogni volta che lo stesso *workspace* viene richiamato: attraverso `load("c:/....")` o equivalentemente con "File—Load Workspace...", o semplicemente avviando R dal file `.Rdata` di interesse.

Infine "File—Save to File..." consente di salvare su file esterno, non solo i comandi digitati ma anche ciò che il sistema ha restituito durante la sessione che si sta chiudendo; è il classico *file log*.

Tali nozioni possono risultare molto utili ad esempio, nel caso in cui più utenti utilizzano la stessa postazione, o se lo stesso utente lavora con R per progetti differenti.

2 Vettori, Matrici, Array e Liste

2.1 Vettori

I numeri con cui siamo abituati a ragionare sono in realtà un caso particolare di una famiglia più grande, i vettori. Un vettore di dimensione n può essere definito come una sequenza ordinata di n numeri. $(2, 5, 9.5, -3)$ rappresenta un vettore di dimensione 4 in cui, ad esempio, il primo elemento è 2 ed il quarto è -3. Esistono molti modi per costruire un vettore, il più comune è utilizzare la funzione `c()`:

```
> x<-c(2,5,9.5,-3) #costruisci un vettore
> x[2] #seleziona il suo secondo elemento
[1] 5
> x[c(2,4)] #seleziona i suoi elementi nella posizione 2 e 4
[1]  5 -3
> x[x>0] #seleziona i sui elementi positivi
[1] 2.0 5.0 9.5
> x[x>0]-1
[1] 1.0 4.0 8.5
> x[x>0][2]
[1] 5
```

Come è facile notare, le parentesi quadre dopo il vettore definiscono quali componenti dell'oggetto selezionare: un indice (`x[2]`) o un vettore di indici (`x[c(2,4)]`) o una condizione (`x[x>0]`), sono tutte scritture ammesse. Naturalmente ogni selezione da un oggetto è essa stessa un oggetto! così `x[x>0]` è un oggetto, in particolare è un vettore tridimensionale su cui è possibile svolgere una qualsiasi operazione compreso l'estrazione di sue componenti. È importante notare che la riga `x[x>0]-1` sottrae 1 da ogni elemento del vettore. Questa è ancora un'altra caratteristica del linguaggio: le operazioni con oggetti 'multipli' (quali vettori e in seguito vedremo matrici) sono eseguite considerando ogni elemento dell'oggetto; naturalmente si deve trattare di una operazione ammissibile. Ad esempio `x[x>0]+c(2,5)` è sbagliata, mentre

```
> x[x>0]+c(2,5,3)
[1]  4.0 10.0 12.5
```

è una espressione valida. Quanto detto vale anche per gli altri operatori matematici.

Una utilissima funzione per i vettori è la funzione `length()` che restituisce la dimensione del vettore:

```
> length(x[x>0][2])
[1] 1
> length(x[x>0])
[1] 3
> length(x)
[1] 4
```

```
> length(x)/2+3
[1] 5
```

Si vuole ancora ribadire che il risultato di `length()` (cioè in generale di qualsiasi operazione in R) è un oggetto (un numero in questo caso) utilizzabile nell'ambiente in un qualsiasi modo.

Fin qui sono stati considerati vettori numerici, ovvero vettori su cui è possibile effettuare operazioni aritmetiche. In realtà R consente la costruzione di vettori non-numerici, sui cui è anche possibile effettuare qualche operazione,

```
> v<-c("barletta","bracciano","palermo")
> length(v)
[1] 3
> v*1
Error in v * 1 : non-numeric argument to binary operator
> v[c(1,2)]
[1] "barletta"  "bracciano"
```

Sebbene sia possibile formare vettori che comprendano sia caratteri sia numeri, comunque i numeri inseriti in vettori 'misti' vengono considerati caratteri:

```
> x<-c(2,"d",5)
> x
[1] "2" "d" "5"
```

le virgolette indicano che effettivamente si tratta di caratteri; vedremo più avanti come creare oggetti 'misti'.

Altri modi di formare i vettori comprendono le funzioni `seq()`, `rep()` e `:`. Vedi i file di aiuto di ogni funzione, digitando, ad esempio, `?seq`. Ad ogni modo si osservi:

```
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x1<-seq(1,1000, length=10) #vettore da 1 a 1000 di ampiezza 10
> x1
 [1]    1  112  223  334  445  556  667  778  889 1000
> x2<-rep(2,times=10) #ripeti 2 10 volte
> x2
 [1] 2 2 2 2 2 2 2 2 2 2
> rep(c(1,3),times=4) #ripeti (1,3) 4 volte
[1] 1 3 1 3 1 3 1 3
> rep(c(1,9),c(3,1)) #ripeti (1,9) 3 e 1 volta rispettivamente
[1] 1 1 1 9
> length(c(x,x1,x2,3))
[1] 31
```

2.2 Matrici

Una matrice può essere definita come un quadro di numeri in cui ciascun elemento è univocamente individuato da una coppia di numeri, che costituiscono l'indice di riga e quello di colonna. Si noti i seguenti codici

```
> x<-matrix(1:10,ncol=5) #costruisci una matrice
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x[,1] #seleziona la prima colonna
[1] 1 2
> x[2,] #seleziona la seconda riga
[1]  2  4  6  8 10
> x[3,2] #seleziona l'elemento [3,2]
Error: subscript out of bounds
> x[2,3] #...e quello [2,3]
[1] 6
> x[,4:5] #seleziona solo le colonne 4 e 5
     [,1] [,2]
[1,]    7    9
[2,]    8   10
> x[,-c(2,4)] #seleziona le colonne 1, 3 e 5
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
```

La funzione `matrix()` viene utilizzata per costruire una matrice e richiede come argomenti gli elementi che devono costituire la matrice ed il numero delle colonne possibilmente coerente con il primo argomento: ad esempio `matrix(1:10, ncol=3)` non sarebbe una espressione corretta ed un *warning* verrebbe stampato. Come per i vettori, l'utilizzo di `[]` è utile per selezionare elementi della matrice utilizzando `,` per separare gli indici di riga e di colonna.

Altre funzioni sono `cbind()`, `rbind()` e `diag()` che costruisce una matrice diagonale o estrae la diagonale da una matrice:

```
> cbind(1:2,c(1,-2),c(0,9)) #dipone per colonne
     [,1] [,2] [,3]
[1,]    1    1    0
[2,]    2   -2    9
> rbind(1:4,c(0,5,-4,6)) #dispone per righe
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    0    5   -4    6
> diag(x[,4:5]) #estrae la diagonale principale
[1]  7 10
> X<-diag(1:3) #costruisce una matrice diagonale
```

```
> X
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

La funzione `solve()` serve per risolvere sistemi di equazioni lineari, ma può essere utilizzata per il calcolo della matrice inversa

```
> solve(X) #l'inversa
     [,1] [,2]      [,3]
[1,]    1  0.0 0.0000000
[2,]    0  0.5 0.0000000
[3,]    0  0.0 0.3333333
> X%*%solve(X) #....verifica
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

essendo `%*%` l'operatore 'prodotto-matriciale' (riga $\times$ colonna)

In una matrice è possibile sostituire completamente una linea (riga o colonna), ammesso che le dimensioni corrispondano. Ad esempio, per la seconda riga:

```
> x[2,]<-rep(2,5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    2    2    2    2
```

A differenza dei vettori, le matrici sono caratterizzate da una coppia di numeri, e la funzione `dim()` è utilizzata per restituire il numero di righe e colonne

```
> dim(x)
[1] 2 5
> dim(x)[1]
[1] 2
```

Matrici e vettori costituiscono gli elementi essenziali di una analisi statistica e molte altre operazioni sono consentite su di essi.

Ad esempio `t()` restituisce la trasposta:

```
> t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

mentre la funzione `as.vector()` può essere utilizzata per 'forzare' una matrice a vettore

```
> as.vector(x)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Infine al lettore potrebbe risultare interessante vedere quali sono i risultati di

`t(as.vector(x))%*% as.vector(x)` e `as.vector(x)%*% t(as.vector(x))`

Una particolare matrice in Statistica è la matrice dei dati definita da $\boldsymbol{X}_{n\times J}$, essendo n il numero di osservazioni e J quello delle variabili. Sebbene in R sia possibile gestire la matrice dei dati con una semplice matrice, esiste comunque un particolare oggetto che serve a tale scopo: il *dataframe*, che verrà trattato nel paragrafo successivo.

2.3 Array

Così come le matrici possono intendersi come estensioni dei vettori, gli *array* costituiscono una estensione delle matrici. In un *array* (multidimensionale) ogni suo elemento è individuato da un vettore di indici (si ricordi che in vettori e matrici gli elementi sono individuati da uno e due indici rispettivamente). Ad esempio, in un *array* tridimensionale, ogni elemento è caratterizzato da una terna (i_1, i_2, i_3). Sebbene in Statistica Applicata gli *array* possono trovare numerose applicazioni, in un approccio base tali elementi possono essere tralasciati. Soltanto per completezza qualche codice per la gestione degli array è riportato sotto.

```
> a<-array(1:24, dim=c(3,4,2))
> a[,,2]
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
> a[1,,]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
> a[1,2,1]
[1] 4
> dim(a)
[1] 3 4 2
```

Per cercare di fissare le idee, un *array* $3 \times 4 \times 2$ può essere pensato come '2 matrici 3×4 una dietro l'altra': ad esempio tali due matrici potrebbero rappresentare una distribuzione doppia in ciascuno dei livelli di un confondente. L'uso delle parentesi quadre, alla stregua di vettori e matrici, ha il fine di selezionare sotto-insiemi dell'*array*.

2.4 Liste

In R una lista è (naturalmente un'oggetto) una raccolta di altri oggetti tra loro anche differenti, compreso altre liste. Ad esempio questo non è vero per i *vettori* i cui elementi devono necessariamente essere numeri. Una lista può essere creata con il comando `list()` e i suoi componenti individuati con doppie parentesi quadre `[[ ]]`:

```
> lista<-list(matrix(1:9,nrow=3),rep(0,3),c("tempesta","angela"))
> length(lista)
[1] 3
> lista[[3]]
[1] "tempesta"  "angela"
> lista[[2]]+2
[1] 2 2 2
> lista[[1]][2,2]
[1] 5
> length(lista[[2]])
[1] 3
```

Come si vede `lista` contiene una matrice, un vettore numerico ed uno carattere; l'estrazione di oggetti appartenenti alla lista stessa è lineare, e naturalmente, ciascuno di essi viene trattato come un oggetto separato. Avremmo potuto nominare ogni elemento della lista o semplicemente specificando il nome all'interno di `list()`, cioè `list(primo=matrix(....)`, o utilizzando il suo argomento `dimnames`, o ancora attraverso funzione `names()` che restituisce `NULL` se la lista è stata creata senza nomi.

```
> names(lista)
NULL
> names(lista)<-c("primo","secondo","terzo")
> lista$secondo #estrai il secondo elemento
[1] 0 0 0
```

Quando i nomi sono stati assegnati è possibile estrarre ogni elemento della lista utilizzando il simbolo `$` come illustrato sopra.

Oltre alle liste è anche possibile nominare i singoli elementi di vettori e matrici o liste. Per le matrici l'assegnazione di etichette alle righe e o colonne può risultare molto utile:

```
> x<-matrix(1:10, ncol=5)
> dimnames(x)<-list(c("nome1","nome2"),NULL)#nomina solo le righe
> x
      [,1] [,2] [,3] [,4] [,5]
nome1    1    3    5    7    9
nome2    2    4    6    8   10
> dimnames(x)[[2]] <-c("g","h","j","j","k")#nomina le colonne
> x
      g h j j  k
```

```
nome1 1 3 5 7  9
nome2 2 4 6 8 10
> x[,"g"]==x[,1]#una banale verifica
nome1 nome2
 TRUE  TRUE
```

Quindi per una matrice i nomi di riga e di colonna sono salvati in una lista di 2 componenti, il vettore dei nomi di riga e quello dei nomi colonna. Così come per `array()`, anche `matrix()` ha un argomento `dimnames` che consente di inserire direttamente i nomi in questione. Ad esempio:

```
x<-matrix(1:10, ncol=5, dimnames=list(c("h","k"),NULL))
```

3 Il dataframe

Il *dataframe* costituisce forse l'oggetto più importante di tutto l'ambiente R, almeno in una sua ottica di gestione e analisi dei dati. Il *dataframe* è una particolare lista e rappresenta la matrice dei dati in cui ad ogni riga corrisponde una osservazione e ad ogni colonna una variabile. Tra le diverse opzioni disponibili, è possibile costruire un `data.frame` direttamente con la funzione `data.frame()`

```
> X<-data.frame(a=1:4, sesso=c("M","F","F","M"))#crea un dataframe
    #con una variabile 'quantitativa' e una 'qualitativa'.
> X
  a sesso
1 1     M
2 2     F
3 3     F
4 4     M
> dim(X)
[1] 4 2
> X$eta<-c(2.5,3,5,6.2) #aggiungi una variabile
> X
  a sesso eta
1 1     M 2.5
2 2     F 3.0
3 3     F 5.0
4 4     M 6.2
```

Come è facile notare, ogni elemento di tale lista può essere sia numerico che carattere e può essere individuato sia con `X$sesso` che con `X[,"sesso"]`, od anche specificando il numero della colonna, `X[,2]`.

Si notino ora i seguenti codici:

```
> X[X$sesso=="M","eta"] #guarda eta per i maschi
[1] 2.5 6.2
> X$sesso[X$eta<=3] #guarda sesso per cui eta<=3
```

```
[1] M F
Levels:  F M
```

L'uso delle parentesi quadre consente di selezionare opportunamente 'i casi' desiderati; il discorso può essere facilmente generalizzato per includere anche contemporaneamente gli altri simboli quali `!=`, insieme a `&` e `|`. Questi ultimi due operatori logici vengono solitamente utilizzati fra condizioni per specificare che queste si verifichino 'simultaneamente' o 'alternativamente' Ad esempio

```
> X$a[X$eta<=5 & X$eta>3] #seleziona casi se eta<=5 e eta>3
[1] 3
> X$sesso[X$eta<3|X$eta>5] #seleziona se eta<3 o eta>5
[1] M M
Levels:  F M
```

Infine una colonna, ovvero una variabile, può essere eliminata attraverso l'utilizzo di `NULL`; ad esempio `X$eta<-NULL` elimina `X$eta`.

3.1 Importazione di dati

La funzione `data.frame()`, non esaurisce le diverse possibilità di R che esistono per risolvere il problema della gestione dei dati. Ad esempio la funzione `as.data.frame()` forza una matrice a *dataframe*, oppure i dati potrebbero essere inseriti più facilmente attraverso un foglio elettronico; in questo caso `X<-data.frame()` crea un data.frame che è possibile aprire con `fix(X)`per l'inserimento dei dati.

Ancora un'altra possibilità, che è probabilmente la più frequente, è quella di caricare i dati che sono salvati in un formato ASCII ottenuto da un qualsiasi altro programma. La funzione `read.table()` costituisce uno strumento per l'importazione di file ascii:

```
X<-read.table(file="c:/documenti/dati.txt", header=T, sep="\t")
```

in questo caso il file `dati.txt` viene importato e automaticamente convertito nel data.frame `X`. A questo punto sono utili le seguenti osservazioni:

- il percorso del file (*path*) viene scritto come negli ambienti Unix (o Linux), cioè con `/` o con `\\`;
- l'argomento `header=T` specifica che la prima linea del file contiene i nomi delle variabili;
- l'argomento `sep="\t"` definisce che i diversi campi sono separati da un tab. Avremmo potuto specificare `sep=","` nel caso di una virgola, ad esempio.

3.2 Valori mancanti

(Molto) spesso i dati raccolti sono caratterizzati da valori mancanti. In R un dato mancante viene etichettato con `NA`. Inserire direttamente un dato mancante è facile:

```
> x<-1:10
> x[8]<-NA #inserisci un dato mancante
> x
 [1]  1  2  3  4  5  6  7 NA  9 10
```

od anche `x<-c(2.5,3,NA,4,....)`. Comunque più frequente è il caso di aver inserito i dati altrove ed aver etichettato i mancanti con numeri particolari, ad esempio 999. In questo caso nel *dataframe* tutti i 999 dovranno essere sostituiti con `NA`. Questo può essere ottenuto facilmente

```
> x[c(5,8)]<-999
> x
 [1]   1   2   3   4 999   6   7 999   9  10
> x[x==999]<-NA
> x
 [1]  1  2  3  4 NA  6  7 NA  9 10
```

Una funzione utilissima per i dati mancanti è `is.na()` che restituisce un valore logico `T` se il dato è mancante

```
> is.na(x)
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

Riconoscere i dati mancanti è fondamentale per R (oltre che per noi, naturalmente), in quanto alcune funzioni non possono essere utilizzate con dati mancanti. Comunque è possibile selezionare a priori solo i dati non-mancanti utilizzando la stessa funzione `is.na()`.

```
> x[is.na(x)==F]
[1]  1  2  3  4  6  7  9 10
> x[!is.na(x)==T]
[1]  1  2  3  4  6  7  9 10
> x[!is.na(x)]
[1]  1  2  3  4  6  7  9 10
```

Le precedenti tre scritture sono equivalenti; la terza, in generale, è quella più utilizzata. Qual è il risultato di `x[is.na(x)]`? In ultimo la funzione `na.omit()` potrebbe anche essere utilizzata, vedi paragrafo 4.1.

3.3 Codifica di variabili

Come è noto in Statistica si distinguono sommariamente due principali gruppi di variabili: quelle numeriche e quelle categoriali. Come si è visto sopra, una variabile categoriale potrebbe essere inserita direttamente con le etichette, ovvero ad esempio `sesso<-c("M","M","F",....)`. In pratica è più frequente che anche tali variabili siano inserite attraverso numeri, quindi specificare che si tratta di etichette e non di numeri e probabilmente, infine, assegnare una etichetta ad un numero (ad esempio 1=maschio e 2=femmina). È possibile costruire una variabile categoriale facilmente attraverso la funzione `factor()`:

```
> x<-factor(c(1,1,2,3,1))
> x
[1] 1 1 2 3 1
Levels:  1 2 3
> x<-factor(c(1,1,2,3,1),labels=c("gruppo1","gruppo2","gruppo3"))
> x
[1] gruppo1 gruppo1 gruppo2 gruppo3 gruppo1
Levels:  gruppo1 gruppo2 gruppo3
```

L'unica differenza è nell'apparenza! anche senza specificare a cosa le etichette corrispondono R è in grado di riconoscere che si tratta di una variabile non-numerica. Alternativamente, avendo importato una colonna numerica, è possibile trasformarla in variabile categoriale utilizzando ancora la funzione `factor()`. Ad esempio assumendo la variabile `sesso` nell'ipotetico *dataframe* `dati`

```
dati[,"sesso"]<-factor(dati[,"sesso"])
```

A volte può essere conveniente 'categorizzare' una variabile quantitativa, ad esempio formando classi di età da una variabile che riporta l'età esatta di ogni osservazione.

```
> eta<-c(2,4.3,5,.2,6,8,9.8,4,10,9.5)
> eta.cat<- factor(cut(eta, breaks=c(0,3,5,10),
+ labels=c("basso","medio","alto")))
> eta.cat
 [1] basso medio medio basso alto  alto  alto  medio alto  alto
Levels:  basso medio alto
```

La funzione `cut()` divide la variabile numerica in intervalli tipo $]x_i, x_{i+1}]$ dove gli estremi sono specificati nell'argomento `breaks`. Con tale argomento è anche possibile specificare il numero degli intervalli piuttosto che gli estremi; ad esempio si provi a chiamare `cut()` con l'argomento `breaks=3`

Con variabili categoriali può risultare interessante guardare le distribuzioni, sia quella univariata, sia quella doppia con qualche altra variabile categoriale.

```
> table(eta.cat)
eta.cat
basso medio  alto
    2     3     5
> table(eta.cat, sesso=rep(1:2,5))
       sesso
eta.cat 1 2
  basso 1 1
  medio 1 2
  alto  3 2
> table(eta.cat, sesso=rep(c("m","f"),c(5,5)))
       sesso
eta.cat f m
```

```
basso 0 2
medio 1 2
alto  4 1
```

Naturalmente il discorso può estendersi a distribuzioni multivariate, sebbene il commento e l'interpretazione dei risultati attraverso le semplici tabelle, potrebbero diventare un po' problematici. Infatti volendo considerare una distribuzione tripla, che tipo di oggetto è `a` ottenuto come sotto?

```
> a<-table(eta.cat, rep(1:2, c(5,5)), rep(1:2, 5))
```

Una conferma della risposta può essere ottenuta attraverso la funzione `is.array()` per cui esistono, naturalmente, anche le corrispondenti versioni `is.vector()`, `is.matrix()` e `is.list()` e `is.data.frame()`.

4 Sintesi di una distribuzione

4.1 Qualche statistica descrittiva

Supponiamo di avere rilevato una variabile o più variabili $X_1, X_2, \ldots$, su n unità e che tali dati siano disponibili in R sotto forma di un *data-frame* `X`, creato ad esempio, con `read.table()`. Comunque per semplicità, qui si assume un *data-frame* simulato, in questo modo il lettore potrà riprodurre i comandi illustrati e confrontare i risultati ottenuti.

```
> set.seed(11) #poni il seme per riprodurre i risultati
> X<-data.frame(y=rnorm(n=150, mean=170, sd=5),
+ x=rep(1:3,length=150))
> dim(X)
[1] 150   2
> X$x<-factor(X[,"x"]) #converti in fattore
```

Si ricordi l'uguaglianza delle scritture `X$x`, `X[,"x"]` e `X[,2]`, essendo 2 la colonna di `X` relativa a `x`, tutte ugualmente valide per estrarre l'oggetto `x` (in tal caso una variabile) di `X`. È forse opportuno precisare che gli oggetti contenuti in altri oggetti (in questo caso una variabile nel *dataframe*) possono avere nomi diversi da altri oggetti contenuti nell'ambiente generico, in quanto sono due elementi distinti. Infatti

```
> y<-c("a","b")
> length(y)
[1] 2
> length(X$y)
[1] 150
```

`y` è contenuto nell'ambiente generico, mentre `X$y` è contenuto nell'oggetto `X` dell'ambiente. Ad ogni modo è sempre possibile 'attaccare il data-frame' in memoria e chiamare i relativi elementi direttamente, anche se questo può portare a 'pericolosi equivoci tra oggetti':

```
> attach(X) #attacca X in memoria
> length(y) #chiama i suoi elementi direttamente....
[1] 150
> detach() #....rimuovilo
```

La funzione `rnorm(n,mean,sd)` genera un vettore di n valori gaussiani con media `mean` e deviazione standard `sd`. Soltanto `n` è richiesto in quanto per default `rnorm()` genera valori da una normale standardizzata; ovvero `rnorm(n)` e `rnorm(n,0,1)` sono espressioni perfettamente equivalenti. `set.seed()` pone il 'seme' per la generazione dei numeri casuali al fine di riprodurre quanto di seguito riportato; se omessa i risultati non potrebbero essere riprodotti. Infatti:

```
> set.seed(11)
> rnorm(5, 2.5, 1)
[1] 2.933413 2.880623 3.585841 1.190895 3.203150
> rnorm(5, 2.5, 1)
[1] 2.833709 1.958368 3.466403 1.039843 3.952333
> set.seed(11)
> rnorm(5, 2.5, 1)
[1] 2.933413 2.880623 3.585841 1.190895 3.203150
```

cioè anteponendo uno stesso seme alla generazione di quantità casuali, i risultati sono immutati.

R ha molte altre funzioni per la generazioni di numeri casuali, ad esempio `rpois()` per variabili poissoniane o `rchi()` per v.c. χ^2. Il numero degli argomenti richiesti dipende, naturalmente, dalla distribuzione di interesse: si vedano i rispettivi file di aiuto.

Nella creazione del *dataframe* i nomi delle variabili sono stati impostati direttamente nella funzione `data.frame()`, ma questi possono essere facilmente modificati in qualsiasi momento:

```
> names(X) #i nomi delle variabili
[1] "y" "x"
> names(X)<-c("y","gruppo") #....modificali
> names(X)
[1] "y"      "gruppo"
> names(X)[2]<-"eta"
> names(X)
[1] "y"     "eta"
```

Lo studio di una distribuzione rappresenta un aspetto comune di un'analisi statistica, anche se questa non sia di principale interesse. Le misure più utilizzate per sintetizzare una distribuzione sono probabilmente gli indici di tendenza centrale, tra cui, naturalmente, figurano la media (aritmetica) e la mediana, calcolabili facilmente attraverso le apposite funzioni,

```
> mean(X$y)
[1] 170.2691
> median(X$y)
[1] 170.4347
```

È evidente che per la variabile `X$y`, le due misurano coincidano sebbene questo in generale può non verificarsi dipendendo, come è noto, dalla asimmetria della distribuzione. Tale aspetto può essere, in un primo approccio, facilmente studiato attraverso i quantili che possono essere calcolati per una qualsiasi frazione di dati attraverso la funzione `quantile()` che restituisce per default minimo, massimo, mediana e quartili:

```
> quantile(X$y)
      0%      25%      50%      75%     100%
158.0243 166.8741 170.4347 173.2240 184.9731
> quantile(X$y, prob=c(.5,.6,.9))
     50%      60%      90%
170.4347 171.3035 176.8117
```

Un aspetto comune a molti *data-set* è la presenza dei dati mancanti, che possono essere facilmente considerati nel calcolo delle quantità di interesse. Ad esempio supponiamo che in `X$y` ci siano dei valori mancanti,

```
X$y[c(2,56,90)]<-NA
```

L'argomento `na.rm=T`, indica che gli eventuali valori mancanti in `X$y` devono essere omessi nel calcolo della media altrimenti il risultato è anche mancante.

```
> mean(X[,"y"])
[1] NA
> mean(X[,"y"], na.rm=T)
[1] 170.2307
> mean(na.omit(X$y)) #modo alternativo per eliminare i mancanti
[1] 170.2307
```

Anche per `median()` e `quantile()` il discorso è lo stesso, ma in generale altre funzioni possono richiedere un argomento differente per la gestione dei dati mancanti. Comunque i file di aiuto, sono molto chiari a questo riguardo. Ricordando la funzione `is.na()` (vedi paragrafo 3.2) come calcolare la media senza ricorrere a `mean(....,na.rm=T)`?

Alternativamente è possibile utilizzare la funzione `na.omit()` che elimina direttamente i dati mancanti, evitando così eventuali problemi susseguenti. Si noti cosa avviene per un *dataframe*

```
> dim(X)
[1] 150   2
> dim(na.omit(X)) #elimina righe con almeno un mancante
[1] 147   2
```

Naturalmente il calcolo di media o mediana, non ha senso per variabile categoriali, per le quali potrebbe essere calcolate una tabella di frequenza (`table()`) od anche la moda.

Come è noto, il calcolo della tendenza centrale non è assolutamente sufficiente a dare un'idea anche sommaria di una distribuzione. Tra le misure di variabilità che è possibile utilizzare, `var()` calcola la varianza di un vettore.

```
> var(X$y, na.rm=T)
[1] 26.51844
```

Più in generale data una matrice (o un data-frame), `var()` consente di calcolare la matrice di varianze-covarianze, tra le colonne della matrice (o del data-frame). Alternativamente la correlazione può essere immediatamente ottenuta con `cor()`.

```
> var(cbind(X$y,1:150), na.rm=T)
          [,1]        [,2]
[1,] 26.518442    1.774417
[2,]  1.774417 1884.950890
> cor(X$y, 1:150,use="complete.obs")
[1] 0.007936548
> cor(cbind(X$y, 1:150), use="complete.obs")
            [,1]        [,2]
[1,] 1.000000000 0.007936548
[2,] 0.007936548 1.000000000
```

4.2 Le funzioni `tapply()` e `apply()`

Spesso, in funzione dalla struttura dei dati, potrebbe essere auspicabile 'ottimizzare' l'utilizzo delle funzioni che, per loro natura, richiedono come argomento soltanto un vettore. `mean()` e `median()`, ad esempio, appartengono a questa 'famiglia' di funzioni. Considerando il *dataframe* `X`, uno potrebbe essere interessato al calcolo della media di `X$y` per ciascuno dei livelli di `X$eta`. Una tale struttura dei dati è chiamata *ragged array* in R. La funzione `tapply()` consente di ottenere velocemente il risultato ottenuto:

```
> tapply(X$y, X$eta, mean)
       1        2        3
170.5327       NA       NA
> tapply(X$y, X$eta, mean, na.rm=T)
       1        2        3
170.5327 170.0565 170.0931
```

`tapply()` richiede la variabile di interesse (`X$y`), la variabile che individua i gruppi (`X$eta`), e la funzione seguita dai sui argomenti. Ogni funzione che richiede come argomento un vettore è valida; ad esempio, per calcolare la distribuzione di `X$eta` utilizzando `tapply()`,

```
> tapply(X$eta, X$eta, length)
  1  2  3
 50 50 50
```

E se il primo argomento fosse stato `X$y`?

`apply()` è invece disegnata per applicare una funzione alle righe o colonne di una matrice. Ad esempio, relativamente alla prima e quarta colonna di `X` le varianze possono essere facilmente ottenute:

```
> apply(X[,c(1,3)],2,var,na.rm=T)
       y        z
26.51844 65.50607
```

Il secondo argomento `2`, specifica se applicare la funzione alle colonne o alle righe, per le quali si sarebbe dovuto specificare `1`. Naturalmente se le colonne della matrice sono variabili, l'operazione per righe potrebbe non aver senso! Il risultato dell'uso di `apply()` dipende dalla funzione che viene applicata alle righe (o colonne) della matrice. I seguenti codici possono chiarire quanto detto

```
> length(apply(X[,c(1,3)],1,mean,na.rm=T))
[1] 150
> a<-apply(X[,c(1,3)],1,quantile,na.rm=T)
> dim(a)
[1]   5 150
```

Nel primo esempio vengono restituite 150 medie, perché `mean` ne restituisce una per ogni riga; nel secondo poiché il risultato di `quantile` è un vettore di dimensione 5, il risultato è costituito da 5×150 valori.

4.3 Alcune rappresentazioni grafiche

Medie, varianze e correlazione possono essere inteso come misure di sintesi delle distribuzioni, ma in pratica le rappresentazioni grafiche possono fornire utili informazioni aggiuntive. Come è facile immaginare, esistono svariati modi che possono essere impiegati per rappresentare graficamente le distribuzioni, e moltissimi possono essere implementati in R.

La creazione di grafici di alta qualità è un'altra caratteristica di R: è possibile impostare parametri per modificare ogni aspetto di un grafico, simboli, colori ed esportare nei più comuni formati, sia vettoriali (quali `.ps` o `.pdf`) che bitmap (`.bmp`,`.jpg`) che *metafile*.

Coerentemente all'impostazione di queste dispense, di seguito riportiamo soltanto alcune funzioni, che probabilmente risultano essere le più utilizzate in un primo approccio all'analisi di dati. Ulteriori dettagli sono reperibili dai manuali specifici nella R *home-page*

La funzione `hist()` può essere utilizzata per rappresentare graficamente una distribuzione: l'argomento `breaks` può essere utilizzato per specificare il numero di categorie da disegnare. Così:

```
> hist(X$y, breaks=20) #istogramma
```

disegnerà un istogramma con 20 barre. Per salvare il grafico è possibile utilizzare il menù a tendine, attraverso File—Export Graph.... e scegliere l'estensione di interesse.

Prima di procedere nell'illustrare le altre funzioni alcuni dei più importanti argomenti opzionali che possono essere impostati nella funzione e possono tornare molto utili nel creare e personalizzare i grafici sono:

- `xlab="...."` e `ylab="...."` per i nomi degli assi su cui è disegnato il grafico;

- `xlim=c(x1, x2)` e `ylim=c(y1, y2)` per i limiti dei due assi;
- `main="...."` per aggiungere il titolo al grafico.

Così ad esempio, un'espressione valida potrebbe essere

```
> hist(X$y,xlab="Frequenza",ylab="Valori di Y")
```

La funzione `par()` viene utilizzata per impostare i parametri della rappresentazione grafica. Un suo impiego potrebbe essere quello di dividere l'area del grafico per ottenere figure accostate; così `par(mfrow=c(2,3))` divide l'area del grafico in 2×3 aree su cui riportare altrettanti grafici.

Se `hist()` serve per rappresentare una distribuzione univariata, la funzione `plot()` può essere utilizzata per costruire i diagrammi di dispersione (*scatter plot*) che vengono solitamente utilizzati per guardare alla relazione esistente tra due variabili quantitative. Ad esempio, costruiamo una variabile `z` dipendente linearmente da `y` perturbata da valori gaussiani:

```
> set.seed(100)
> X$z<-2+1.5*X$y+rnorm(150,0,2)
```

Il seguente comando costruisce uno *scatter plot* per le due variabili di interesse, `y` e `z` con qualche argomento:

```
> plot(X$y, X$z, xlab="valori di Y",ylab="valori di X",
+ xlim=c(150,200))
```

Quando un solo argomento è specificato allora `plot()` può essere utilizzato per disegnare serie temporali; ovvero i due comandi `plot(1:length(X$z), X$z)` e `plot(X$z)` sono perfettamente equivalenti. Se i valori del vettore da disegnare, `X$z` per esempio, sono già ordinati, allora `plot(X$z,type="l")` disegna i valori del vettore uniti da una linea. `type` è un altro argomento che specifica come contrassegnare i valori sul grafico. Il default è `type="p"` (points).

La funzione `points()` è simile a `plot()`, ma disegna i punti dei vettori del suo argomento su un grafico già esistente; questo può essere molto utile per sovrapporre punti su uno stesso grafico, ed evidenziare, per esempio, differenti gruppi; si veda l'esempio sotto. Degna di nota è la funzione `pairs()` che richiede come unico argomento una matrice. Il risultato di questa funzione è un unico grafico comprendente tutti i possibili $n(n-1)/2$ diagrammi di dispersione fra le colonne della matrice; fra i diversi argomenti opzionali meritevole di menzione è `lower.panel=panel.smooth` che riporta nella parte inferiore le stime *smoothing* delle relazioni tra le variabili. Per riassumere quanto detto si osservino i seguenti comandi in cui altri argomenti opzionali sono impostati:

```
> par(mfrow=c(2,2)) #dividi l'area in 2x2
> hist(X$y, breaks=10, ylab="Frequenza",col=4)
> plot(X$y, X$z, xlab="valori di Y",ylab="valori di X")
> points(X$y[X$eta==1], X$z[X$eta==1], col=2, pch="+")
> plot(X$z, ylim=c(200,300),type="l") #nota i buchi dei mancanti!
> abline(h=240, lty=3, col=3) #una linea orizzontale
```

```
> plot(X$z, type="l", lty=2)
> points(X$z, pch=2, col=3)
> title(main="Un grafico") #titolo
> dev.off() #chiudi (dopo aver salvato ;-))
null device
          1
> pairs(X,lower.panel=panel.smooth) #Figura 2
```

I risultati sono le Figure 1 e 2.

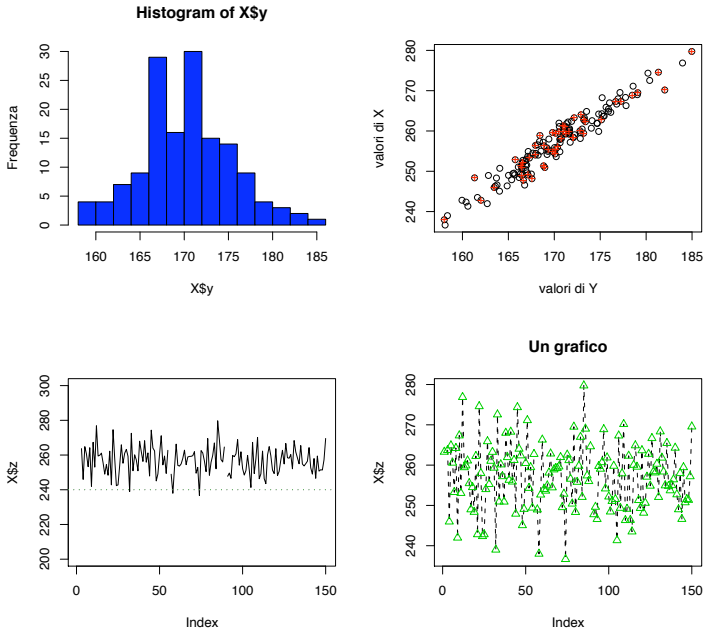


Figura 1: Alcuni grafici ottenuti con `hist()`, `plot()`, e `points()`.

Concludiamo accennando ad un'altra non-trascurabile caratteristica di R: l'inserimento di 'espressioni matematiche' nel grafico. Ad esempio si provi ad inserire come argomento di una qualsiasi funzione grafica la seguente scrittura: `xlab=expression(F[g]^alpha-hat(beta)[(x)])`

Il risultato è che come etichetta dell'asse delle ascisse appare qualcosa come $\mathrm{F}_g^{\alpha} - \hat{\beta}_{(x)}$. In un'ottica di esportazione e presentazione di risultati, questo può essere molto utile, sebbene gli utenti di LaTeX potrebbero trovare il pacchetto psfrag molto più produttivo.

Tutti i grafici prodotti possono essere esportati utilizzando dal 'menù a tendina' File — Save as— e selezionando l'estensione. Alternativamente, soprattutto per grafici molto complessi, per i quali la memoria del sistema potrebbe non essere sufficiente durante il processo di esportazione è possibile disegnare il grafico direttamente sul file esterno, chiamando prima il *device* su cui indirizzare il risultato. Ad esempio per creare un file in postcript

```
> postscript(file="c:/documenti/file.ps") #crea il file
```

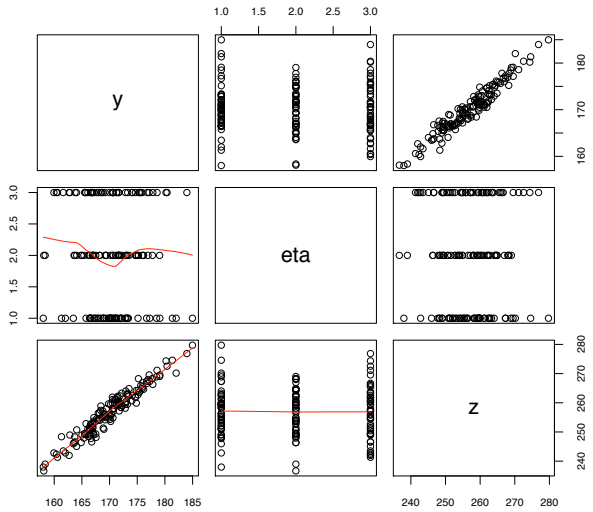


Figura 2: Grafico ottenuto con `pairs()`

```
> plot(....) #disegna ciò che vuoi
> dev.off() #chiudi
```

Si veda `?Devices` per le altre possibilità.

5 Cenni ai Modelli Lineari Generalizzati

Dopo aver organizzato ed ordinato il *dataframe*, ed aver calcolato uno o più misure di sintesi delle variabili, raramente lo scopo dell'analisi si esaurisce con la descrizione dei dati. Spesso il passo successivo riguarda lo studio delle relazioni tra variabili a cui seguono problemi di Statistica inferenziale.

Tra i diversi modi in cui tale argomento può essere affrontato, un approccio multivariato basato sulla modellazione è sicuramente quello più difficile, ma anche quello più completo in quanto consente di ottenere una visione più ampia e quindi più chiara della problematica: infatti, come noto, tutti i più familiari test 'bivariati' (quali, test t per il confronto di medie e test X^2 per tabelle di contingenza) possono essere derivati da opportuni modelli di regressione.

I Modelli Lineari Generalizzati, da qui in avanti GLM (*Generalized Linear Models*), sono un'ampia famiglia di modelli di regressione che hanno rivoluzionato il modo di studiare le relazioni tra variabili, questo anche grazie alla crescente disponibilità di software di alto livello (quale, ad esempio R). L'argomento è talmente vasto e notevole che sarebbe impossibile racchiuderlo in qualche pagina, nè è mia intenzione farlo! Tuttavia data la sua importanza nelle applicazioni pratiche, i GLM stanno entrando nel bagaglio culturale di chiunque 'stia studiando Statistica', e così alcuni aspetti concettuali e operativi dell'argomento sono discussi.

Sebbene alla fine di questo paragrafo il lettore potrebbe essere in grado

(almeno si spera) di 'stimare un suo GLM' si badi che i risultati che ne derivano e i fondamenti teorici che ne costituiscono la base devono essere tenuti sempre in alta considerazione. In sostanza: non si pensi di essere in grado di applicare un GLM e fornire i relativi risultati semplicemente dopo aver letto i paragrafi seguenti.

5.1 Cenni teorici

I modelli di regressione (univariati) hanno lo scopo di studiare la relazione tra una variabile dipendente Y e una o più variabili esplicative $X_1, X_2, \ldots, X_p$. Assumendo una relazione lineare per l'osservazione $i = 1, 2, \ldots, n$ risulta:

$$y_i = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p + \epsilon_i \tag{1}$$

In sostanza il dato osservato y_i viene scomposto in una parte sistematica $\mu_i = \eta_i = \sum_j \beta_j X_j$ dovuta alle (possibili) variabili influenti X_j ed una parte casuale ϵ_i che definisce la variabilità implicita di Y. Lo scopo di un modello di regressione è quello di cercare di spiegare quanta parte della variabilità di Y sia dovuta a fattori esterni sistematici e quanta invece sia attribuibili alla componente *random* di Y non eliminabile. Il problema, detto più comunemente, è individuare quali X_j hanno effetto su Y. Quindi il modello di interesse può anche scriversi solo per la sua parte sistematica

$$\mu_i = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

Si noti che sebbene di fatto non si è interessati alla componente *random*, questa è necessaria per fare inferenza sulla parte sistematica!

Spesso, problemi legati alla natura di Y (insieme di definizione, relazioni tra media e varianza,....) possono portare all'invalidità di un modello quale (1). Per chiarire, molto semplicemente si osservi che se $Y = \{0, 1\}$ (presenza o assenza di malattia), $\eta = \sum \beta X$ potrebbe portare a valori non consentiti, ad esempio negativi. Conseguentemente il modello più generico deve scriversi

$$g(\mu) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p \tag{2}$$

dove $g(\cdot)$ è la funzione che 'lega' la parte sistematica μ del dato osservato y con il *predittore lineare* η. L'equazione (2) definisce un GLM con funzione *link* $g(\cdot)$. Da quanto detto ne segue che per definire un GLM sono necessarie almeno

- variabile risposta Y (parte casuale)
- funzione legame $g(\cdot)$ (funzione *link*)
- predittore lineare (parte sistematica)

Sebbene in generale la distribuzione di Y e la funzione *link* siano aspetti distinti, talvolta nelle applicazioni più frequenti esse risultano particolarmente legate in quanto a certe distribuzioni corrispondono certe funzioni *link*. Queste funzioni *link* a 'diretta corrispondenza' con Y si dicono *link canonici*.

In un approccio classico (frequentista) i parametri vengono solitamente stimati attraverso il metodo della *massima-verosimiglianza* (ML, *maximum likelihood*) che è il metodo impiegato da R.

Indicato con $\hat{\beta}_j$ la stima del generico parametro, ne segue che $\hat{\eta} = \sum_j \hat{\beta}_j X_j$ e quindi $\hat{\mu} = g^{-1}(\hat{\eta})$.

5.2 Aspetti pratici

In R esistono molte funzioni per stimare modelli di regressioni, anche non lineari, ma `glm()` è specificatamente disegnata per GLM e consente di ottenere (quasi) tutte le informazioni di interesse. Nel suo utilizzo più semplice `glm()` prevede di specificare soltanto gli elementi essenziali discussi nel paragrafo precedente. In generale un suo semplice impiego potrebbe essere del tipo

$$\texttt{glm(}Y \sim X_1 + \ldots + X_p\texttt{,family=}\text{distribuzione}\texttt{(link="}\text{link}\texttt{"))}$$

Il primo argomento è la formula dell'equazione del modello, in cui variabile risposta e variabili esplicative sono separate dal simbolo `~`. `family` con `link` definiscono la distribuzione ipotizzata della variabile risposta e la funzione *link*. Il default per `family` è `gaussian`, mentre quello di `link` dipende da `family`, in quanto `glm()` seleziona il legame canonico per la distribuzione impostata. Molti altri argomenti possono essere aggiunti: si veda `?glm` per dettagli.

Supponiamo di mettere in relazione alcune variabili del *dataframe* `X`, e di costruire un GLM gaussiano specificando alcune argomenti opzionali ma utili in questo caso:

```
> glm(z~y, family=gaussian, data=X, na.action=na.omit)

Call:  glm(formula = z ~ y, family = gaussian, data = X,
na.action = na.omit)

Coefficients:
(Intercept)            y
     -1.735        1.520

Degrees of Freedom: 146 Total (i.e. Null);  145 Residual
Null Deviance:     9564
Residual Deviance: 618.8  AIC: 634.5
```

Il risultato viene stampato sullo schermo alla stregua dei più frequenti software statistici con le informazioni più importanti: stima dei parametri di regressione, gradi di libertà e la *Residual Deviance*, che misura la variabilità residua del modello. Gli argomenti `data` e `na.action` specificano rispettivamente il *dataframe* a cui si riferiscono le variabili del modello e come devono essere trattati i dati mancanti. È interessante osservare che, per quanto fino ad ora detto, è chiaro che il modello di sopra (gaussiano con *link* identità) potrebbe essere stimato più semplicemente con

```
> attach(na.omit(X))
> glm(z~y+x)
  ....
```

Piuttosto che stampare immediatamente i risultati, la programmazione ad oggetti, quale quella utilizzata da R consente di salvare il risultato in una lista:

```
> ogg<-glm(z~y+x, family=gaussian, data=X, na.action=na.omit)
> is.list(ogg)
[1] TRUE
```

La lista contiene molti elementi utili per la valutazione del modello, quali coefficienti, valori attesi, residui, matrice di varianze-covarianze, ..., oltre ad alcune informazioni relative al processo di convergenza dell'algoritmo. Ad esempio le prime due componenti della lista sono i coefficienti delle variabili esplicative ed i residui.

```
> names(ogg)[1:2]
[1] "coefficients" "residuals"
```

Si digiti `names(ogg)` per una visione più completa. La funzione `summary()` è molto utile per ottenere un sommario del modello:

```
> summary(ogg)

Call:
glm(formula = z ~ y, family = gaussian, data = X,
na.action = na.omit)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-5.17747  -1.36439   0.01021   1.27081   4.89910

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.7348     5.6543  -0.307     0.76
y             1.5200     0.0332  45.782   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 4.267659)

    Null deviance: 9563.89  on 146  degrees of freedom
Residual deviance:  618.81  on 145  degrees of freedom
AIC: 634.46

Number of Fisher Scoring iterations: 2
```

Sebbene questo non sia del tutto corretto, l'output che cattura immediatamente chi ha appena stimato un modello è probabilmente quello dei coefficienti di regressione, forse nella speranza di riscontrare significativo uno o più parametri di interesse! Per ogni variabile X_j, figurano le stime dei coefficienti $\hat{\beta}_j$, i loro errori standard ed i valori della statistica di Wald che, sotto certe condizioni, può essere utilizzata per saggiare la significatività della stima. La matrice di varianze-covarianze delle stime può essere ottenuta nel seguente modo:

```
> summary(ogg)$cov.scaled
             (Intercept)            y
(Intercept)  31.9712062 -0.187640531
y            -0.1876405  0.001102272
```

cioè è contenuta nella lista restituita da `summary()`.

È forse opportuno ribadire che, come tutti gli oggetti in R, ogni elemento di `ogg` è un oggetto 'indipendente' che può essere estratto e utilizzato secondo le proprie esigenze. Si noti ciò che segue:

```
> ogg$coef #estrai i coefficienti
(Intercept)           y
  -1.734755    1.519993

> ogg$fit[1:5] #i valori attesi
       1        3        4        5        6
259.9580 264.9164 246.7149 262.0080 259.2002

> ogg$fit[1:5]-ogg$y[1:5] #semplici residui
         1          3          4          5          6
-3.3213163  1.2090572  0.7460385 -2.8416262 -1.3609540

> all.equal(ogg$dev, sum((ogg$fit-ogg$y)^2)) #una verifica
[1] TRUE
```

L'ultima riga riporta un modo per ottenere la *Deviance* utilizzando `ogg$y` che costituisce il vettore dei valori osservati utilizzati nella stima del modello. `all.equal()` controlla i suoi argomenti (o ciascuno degli elementi dei suoi argomenti) e restituisce `T` se sono tutti uguali.

Per cercare di rendere più chiaro l'utilizzo dei GLM, supponiamo di disporre di una variabile di conteggio, e simuliamo tale variabile attraverso la funzione `rpois()`. A differenza della distribuzione Normale che richiede 3 argomenti (numero elementi da simulare, media e deviazione standard), la distribuzione di Poisson ne richiede solo due, numero di elementi da simulare e media che, come è noto, eguaglia la varianza. Inoltre poiché per definizione la distribuzione di Poisson ammette solo valori positivi utilizziamo la funzione `exp()`, ottenendo in questo modo valori log-dipendenti da `X$x`.

```
> X$x<-1:150 #crea una semplice variabile
> set.seed(20)
> X$w<-rpois(150, exp(3-.025*X$x)) #simula valori poissoniani
```

Può risultare interessante guardare i dati simulati, attraverso `plot(X$x,X$w)`, ad esempio. Il GLM solitamente utilizzato per l'analisi di dati di conteggio è

$$\log \mu = \eta$$

cioè un GLM con log-*link*. Così un tale modello potrebbe essere stimato e salvato in un'opportuna lista con

```
> ogg1<-glm(w~x,family=poisson(link="log"), data=X)
```

dove anche `family=poisson(link=log)` o più semplicemente `family=poisson` sono ammessi, essendo log il *link* canonico per la distribuzione di Poisson. Prima di analizzare più attentamente il modello si supponga di volere inserire nel predittore lineare la variabile categoriale `X$eta`. Non è banale evidenziare che se una variabile non è riconosciuta come categoriale, attraverso `factor()` ad esempio, R la riconoscerebbe come variabile quantitativa e conseguentemente il relativo parametro non avrebbe alcun significato. Ancora, anche dopo che la variabile è stata 'categorizzata' è necessario conoscere il modo in cui è stata parametrizzata per interpretare correttamente i relativi parametri. In R esistono 3 opzioni, `helmert, sum, treatment` (il default) che consente di ottenere parametri interpretabili come la differenza di ogni categoria dal livello *baseline*. La funzione `C()` viene utilizzata direttamente nella formula del GLM.

```
> ogg1<-glm(w~x +C(eta, treatment),family=poisson, data=X)
> summary(ogg1)

Call:
glm(formula = w ~ x + C(eta, treatment), family = poisson,
data = X)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.4264  -0.9139  -0.1127   0.5306   2.4788

Coefficients:
                    Estimate Std. Error z value Pr(>|z|)
(Intercept)         3.084867   0.070061  44.031   <2e-16 ***
x                  -0.024355   0.001065 -22.867   <2e-16 ***
C(eta, treatment)2 -0.117443   0.085419  -1.375    0.169
C(eta, treatment)3 -0.032463   0.084084  -0.386    0.699
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 840.76  on 149  degrees of freedom
Residual deviance: 148.61  on 146  degrees of freedom
AIC: 582.7
```

```
Number of Fisher Scoring iterations: 4
```

Come è noto, i parametri di un modello poissoniano canonico sono interpretabili come log-rischi relativi, e quindi la trasformazione e^{β} definisce il rischio relativo. Le stime puntuali ed i relativi intervalli di confidenza possono essere facilmente ottenuti con:

```
> rr<-exp(ogg1$coef) #calcola i rischi relativi
> se.rr<-summary(ogg1)$coef[,2] #estrai gli se
> cbind(rr, lim.inf=exp(log(rr)-1.96*se.rr),
+ lim.sup=exp(log(rr)+1.96*se.rr)) #guarda i risultati
                            rr    lim.inf    lim.sup
(Intercept)        21.8645638 19.0591528 25.0829172
x                   0.9759391  0.9739039  0.9779785
C(eta, treatment)2  0.8891914  0.7521160  1.0512492
C(eta, treatment)3  0.9680582  0.8209709  1.1414981
```

I valori di sopra non hanno alcun significato per l'intercetta e così la prima riga potrebbe essere facilmente eliminata; il secondo coefficiente definisce il rischio relativo per incrementi unitari di `x`, mentre gli ultimi valori definiscono i rischi delle categorie 2 e 3 di `eta` rispetto alla categoria 1 assunta come riferimento (*baseline*). È bene ribadire che tale interpretazione è strettamente legata alla parametrizzazione (`C(....,treatment)`) della variabile categoriale inserita nel modello; invero le altre parametrizzazione portano ai relativi parametri completamente differenti:

```
> coef(glm(w~x +C(eta, treatment),family=poisson, data=X))[3:4]
C(eta, treatment)2 C(eta, treatment)3
       -0.11744279        -0.03246305
> coef(glm(w~x +C(eta, sum),family=poisson, data=X))[3:4]
C(eta, sum)1 C(eta, sum)2
  0.04996861  -0.06747418
> coef(glm(w~x +C(eta, helmert),family=poisson, data=X))[3:4]
C(eta, helmert)1 C(eta, helmert)2
    -0.058721395      0.008752783
```

Naturalmente i tre modelli (`ogg1`, `ogg2`, `ogg3`) sono equivalenti:

```
> all.equal(ogg1$dev, ogg2$dev, ogg3$dev)
[1] TRUE
```

Concludiamo questa brevissima trattazione sui GLM, osservando che nei comandi di sopra, per estrarre i coefficienti da un modello `ogg` abbiamo utilizzato `coef(ogg)` piuttosto che `ogg$coef`. Questa differenza è riscontrabile anche per i residui (`residual(ogg)` o `ogg$res`) ed i valori attesi (`fitted(ogg)` o `ogg$fitted`). In generale tali due metodi di estrazione non sono del tutto

equivalenti, ma la discussione delle loro differenze unitamente a moltissimi altri aspetti esulano dall'aspetto introduttivo di queste dispense.

Approfondimenti e ulteriori informazioni possono essere trovati altrove: lo scopo qui è stato quello di fornire concetti introduttivi all'ambiente R che dovrebbero avviare il lettore a familiarizzare con il linguaggio e potrebbero semplificare la lettura di materiale più ampio e specifico in seguito (non italiano, probabilmente): *tutorials* di qualche altro utente o i manuali del *R core team*.