Il linguaggio R:
concetti introduttivi ed esempi

II edizione

– settembre 2005 –

Vito M. R. Muggeo
vmuggeo@dssm.unipa.it

Giancarlo Ferrara
ferrara@dssm.unipa.it

Indice

Note preliminari alla I edizione

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

Note preliminari alla II edizione

I tre anni che sono trascorsi dalla 'pubblicazione' della prima versione di queste dispense hanno visto la diffusione di numerosi altri documenti contribuiti, appunti ed eccellenti libri, sia in lingua inglese che italiana, tutti finalizzati all'utilizzo e alla comprensione di R. *Tuttavia l'aspetto* realmente *introduttivo pare essere stato leggermente tralasciato: per chi si accinge (i così-detti* beginners*) ad imparare* R*, un riferimento di base facilmente accessibile e reperibile può risultare abbastanza utile; nel fornire nozioni e concetti introduttivi, l'intento è che tali dispense possano rappresentare un possibile (e valido) punto di partenza per la lettura di altri libri.*

Con questa idea la prima versione (che pure pare avere avuto una discreta diffusione in Italia) è stata completamente riscritta per cercare di rendere i contenuti più chiari. Pur mantenendo il carattere introduttivo, qualche nuovo elemento, come i cicli while *e* for*, è stato aggiunto.*

Un dovuto ringraziamento va al gruppo di sviluppatori di R *che rendono possibile tutto questo e a tutti gli utenti della* R mailing list *che con domande/risposte e suggerimenti contribuiscono ad accrescere la conoscenza del linguaggio.*

Infine una nota di carattere ambientale: nonostante l'evidenza empirica (arretramento di ghiacciai, aumento delle temperature) i nostri governi sono ancora molto lontani dal mostrare un reale interesse e volontà al mantenimento del nostro ambiente: l'unica cosa che si può fare è aumentare il peso delle associazioni ambientaliste ed assumere atteggiamenti eco-compatibili (come uso di carta riciclata o parsimonioso utilizzo di risorse).

Palermo, settembre 2005

Introduzione

Queste dispense hanno lo scopo di illustrare i fondamenti logici ed applicativi 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 verranno utilizzati indifferentemente. Per questo motivo quando ci si appresta a lavorare la classica domanda "È possibile in R implementare....?", deve 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 siano 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 dell' esperienza degli autori, 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

Una volta che R è stato lanciato, tutte le istruzioni sono eseguibili dalla linea di comando dell'ambiente, tipicamente ma non necessariamente, caratterizzata dal prompt `>`: la linea di comando viene eseguita semplicemente premendo il tasto Invio della tastiera. 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 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 visualizzarli, modificarli ed utilizzarli successivamente. Ciò che è richiesto per salvare un 'qualcosa' che possa essere utilizzato in seguito, in qualsiasi modo, 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 `=` (od anche `->`). Le seguenti linee potranno rendere le idee più chiare:

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

Il carattere `#` è il simbolo di commento e tutto ciò che ad esso segue viene ignorato dall'ambiente. Nell'esempio precedente si è creato prima un oggetto `x` (assegnandogli il risultato dell'espressione `2+(3-4*5)/2`) che è stato successivamente sovrascritto (assegnandogli `5`), poi un altro oggetto `x.x` ed infine è stata eseguita un operazione. Si noti il segno `=` che è del tutto equivalente a `<-`, mentre il punto `.` è un normale carattere in R (non ha nulla a che vedere con le estensioni dei file nel sistema operativo Windows).

A differenza quindi dei più comuni software, i risultati di tutte le operazioni eseguite possono essere salvati in un oggetto e utilizzate in seguito, semplicemente assegnando 'qualcosa' ad un nome 'qualsiasi': questo è un grande VANTAGGIO di R, in quanto diversi oggetti, creati in momenti diversi, saranno sempre disponibili e prontamente richiamabili e utilizzabili. Comunque è bene tenere presente che, nel momento in cui si decide di creare un oggetto, eventuali oggetti pre-esistenti che possiedono lo stesso nome saranno sovrascritti e quindi cancellati: è necessario quindi fare molta attenzione a non creare oggetti con nomi già usati dall'ambiente, perché sia l'utente che R stesso potrebbero confondersi!

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*, ovvero fa distinzione fra lettere minuscole e maiuscole, così `x` e `X` sono oggetti differenti.

Esiste in R una lista di parole "riservate" che non possono essere utilizzate dall'utente a proprio piacimento, ma solo per le finalità per le quali sono state costruite; queste sono:

```
FALSE, TRUE, Inf, NA, NaN, NULL, break, else, for, function, if,
                    in, next, repeat, while
```

Più avanti sarà discusso l'utilizzo di alcune di queste parole, per il momento si osservi cosa succede se l'utente cerca di utilizzarle per creare dei propri oggetti:

```
> TRUE<-2
Error in TRUE <- 2 : invalid (do_set) left-hand side to assignment
> TRUe<-2
> TRUe
[1] 2
```

`TRUE` non può essere utilizzata come nome da assegnare ad un oggetto ma per quanto detto sopra, l'utilizzo di qualsiasi altro nome 'diverso' (anche soltanto per una sola lettera) è consentito.

Le seguenti linee illustrano l'utilizzo dei 6 operatori: `==` (uguale); `>=/<=` (maggiore/minore uguale); `>/<` (maggiore/minore); `!=` (diverso). Tali operatori potranno risultare molto utili in seguito, soprattutto nella selezione di casi dalla matrice dei dati, ma 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
```

Il confronto di due oggetti, che in generale rappresenta una *condizione*, se valutabile, restituisce un `TRUE` o un `FALSE` a seconda del risultato. È appena il caso di far notare che la rappresentazione dei numeri in formato macchina (attraverso una sequenza finita binaria) comporta arrotondamenti e approssimazioni che talvolta possono portare a risultati strani, apparentemente incomprensibili ma in realtà dovuti alla così-detta aritmetica *floating-point*. Infatti:

```
> .2-.1==.1
[1] TRUE
> .5-.4==.1
[1] FALSE
> sqrt(3)*sqrt(3)
[1] 3
> sqrt(3)*sqrt(3)-3
[1] -4.440892e-16
> (sqrt(3)*sqrt(3)-3)==0
[1] FALSE
> (sqrt(4)*sqrt(4)-4)==0
[1] TRUE
```

dove `sqrt()` è una *funzione* (vedi il paragrafo successivo) che calcola la radice quadrata. La questione è delicata, ma dal punto di vista pratico quasi mai porta a differenze sostanziali.

Ad ogni modo, le espressioni logiche `TRUE` e `FALSE` corrispondenti rispettivamente a 1 e 0 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+214
[1] -Inf
[1] Inf
```

Si osservino i risultati delle ultime tre operazioni (le ultime due sono state eseguite sulla stessa linea separandole con un `;`): le forme indeterminate (ad esempio `F/F`, ovvero $0/0$) e quelle tendenti ad infinito sono correttamente prese in considerazione da R che restituisce risultati 'corretti', `NaN` (che sta per *not-a-number*) o `Inf` (infinito). L'altra cosa che balza evidente è l'utilizzo di `T` e `F` che sembrano essere del tutto equivalenti a `TRUE` e `FALSE`, ma che in realtà non lo sono completamente. R all'avvio di ogni sessione assegna a `T` e `F` le variabili `TRUE` e `FALSE` rispettivamente, ma l'utente ha la possibilità di cambiare tale assegnazione e utilizzarle a proprio piacimento:

```
> T<-3
> F<-4
> T+F
[1] 7
> TRUE+FALSE
[1] 1
> (!TRUE)+FALSE
[1] 0
```

per cui l'equivalenza vale fino ad una (consentita) assegnazione su `T` e `F`. Nell'ultima espressione il simbolo `!` rappresenta una negazione, ovvero 'non-`TRUE`' che vale 0, da cui il risultato. Si noti l'uso delle parentesi che anche in R (come in generale in ogni linguaggio di programmazione) risulta essere di fondamentale importanza. Il lettore può facilmente verificare come l'uso delle parentesi possa modificare radicalmente il risultato di una qualsiasi operazione; ad esempio, riprendendo due linee scritte in precedenza si osservi come cambiano i risultati in funzione delle posizioni delle parentesi:

```
> 2*(2==4)
[1] 0
> T/(F+214)
[1] 0.004672897
```

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, e allora si potrà scrivere

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

Oppure utilizzando la nota formula $\sum_{i=1}^{n} i = n(n+1)/2$ potremmo procedere nel seguente modo:

```
> n=5
> n*(n+1)/(2*n)
[1] 3
```

Per calcolare il risultato per diversi valori di n potremmo assegnare a `n` il valore desiderato e poi lanciare ogni volta la seconda linea di istruzioni. Comunque piuttosto che scrivere specificatamente tutte le istruzioni, è possibile 'organizzarle' in modo compatto in maniera da eseguirle velocemente ed ottenere il risultato finale in funzione di uno o più valori impostati: una struttura di linee di codici organizzate in questo modo costituisce una *funzione*.

Per eseguire una qualsiasi funzione `fn()` ed ottenere il relativo risultato è necessario digitare il suo nome specificando fra parentesi tonde l'*argomento* a cui `fn()` deve essere applicata. Ad esempio supponiamo che `fn()` sia una funzione che calcola semplicemente 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. Allora se tale ulteriore argomento (il denominatore, in questo caso) fosse riconosciuto come `d`, ad esempio, si potrebbero scrivere entrambi separandoli con `,`:

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

Generalizzando ancora, un altro argomento potrebbe essere, ad esempio, un numero rispetto al quale effettuare l'elevazione a potenza del rapporto, `e` diciamo:

```
> fn(a=20,d=5,e=3)
[1] 64
```

A questo punto si osservi che:

- I nomi degli argomenti e l'ordine in cui essi appaiono *non* sono arbitrari, ma dipendono dalla funzione in uso; come discusso più avanti, il 'file-di-aiuto' fornisce tutte le informazioni necessarie per un corretto utilizzo.

- L'ordine degli argomenti non è importante a meno che non si voglia omettere il loro nome: così `fn(d=5,e=3,a=20)` o `fn(20,5,3)` sono del tutto equivalenti e restituirebbero, come atteso, la quantità $(20/5)^3$, ma scrivendo `fn(3,20,5)` si otterrebbe $(3/20)^5$.
- Non sempre è necessario specificare tutti gli argomenti della funzione; spesso, argomenti 'meno importanti' hanno un proprio valore di *default* che viene utilizzato quando tali argomenti non sono specificati al momento dell'utilizzo della funzione. Ad esempio, se fosse `e=2` per default, allora `fn(15,5)` sarebbe equivalente a `fn(15,5,2)`.

R è dotato di una grande quantità di funzioni che assolvono ai più 'comuni' calcoli matematici (ad esempio integrazione) e soprattutto statistici (quali medie, varianza, modelli di regressione). 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 nome, il loro ordine, quelli necessari e quelli opzionali, ovvero quelli per cui un valore di default è già impostato. Nel corso di queste dispense verranno illustrate e discusse le funzioni più frequentemente utilizzate per i più comuni calcoli statistici e la produzione di grafici; da qui in avanti le parentesi tonde dopo il nome saranno utilizzate per indicare che si tratta di una funzione e non di un qualsiasi altro oggetto.

Forse una delle più semplici funzioni è `rm()` che serve a cancellare gli oggetti; quindi nel suo utilizzo più semplice:

```
> 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`". Quasi sempre una funzione avrà più di un solo argomento (ad esempio, `rm(x,y,Z)` cancellerà i 3 oggetti che figurano come argomenti), ma ci sono alcune funzioni che possono essere invocate senza alcun argomento: ad esempio `ls()` (o `objects()`), quando richiamata senza alcun argomento, consente di elencare sullo schermo gli oggetti presenti nell'ambiente, mentre `Sys.info()` visualizza qualche dettaglio della piattaforma (sistema operativo, utente, *hardware*) su cui R è stato installato.

`all.equal()` e/o `identical()` possono essere utilizzate per verificare la 'quasi' uguaglianza di due oggetti (cioè considerando un'opportuna tolleranza) evitando i problemi '*floating point*' accennati sopra.

La funzione `q()` consente di chiudere R e se eseguita senza alcun argomento, l'utente dovrà successivamente decidere se salvare o meno la sessione di lavoro, discussa nel paragrafo successivo. Alternativamente l'utente può lanciare direttamente i comandi `q("yes")` e `q("no")`.

`options()` viene utilizzata per modificare o definire le tante opzioni che influenzano il modo in cui R mostrerà i risultati: ad esempio è possibile impostare se e come visualizzare sullo schermo i messaggi di errore (*error*) o di attenzione (*warning*) od anche come modificare i caratteri dei prompt (quello 'immediato' `>`, e quello 'di continuazione' `+`). Ad esempio:

```
> options()$prompt #questo è l'attuale prompt
[1] "> "
> options()$continue
[1] "+ "
>
> options(continue="continua: ") #modifica..
> options(prompt="R prompt: ") #modifica..
R prompt: 2**
continua: 3
[1] 8
R prompt: options(continue="+ ") #re-imposta i valori di default
R prompt: options(prompt="> ")
```

Con i codici di sopra sono stati prima visualizzati i prompt, poi sono stati modificati ed infine sono stati re-impostati ai valori di default; più avanti risulterà più chiaro il significato dei codici scritti sopra; in questa sede si è soltanto voluto introdurre la funzione `options()`.

Si provi a lanciare `names(options())` per un elenco delle opzioni oppure si veda `?options` per qualche commento e spiegazione.

1.2 Organizzazione del lavoro

Quando si lavora in R può essere molto utile specificare la *directory* di lavoro, ovvero la cartella dove salvare i file relativi a quella sessione. Spesso uno stesso utente lavora a più progetti contemporaneamente e dunque potrebbe essere molto utile avere una *directory* per ogni progetto. La funzione `getwd()` restituisce il percorso dell'attuale directory di lavoro che è possibile cambiare attraverso `setwd()`:

```
> getwd() #questa è l'attuale dir di lavoro..
[1] "C:/Programmi/R/rw2001"
> setwd("c:/varie") #cambiala..
```

Alternativamente è possibile utilizzare il menù a tendine da "File|Change dir..." specificando poi il percorso desiderato. Nella *directory* di lavoro R salva:

- un file ascii (`.Rhistory`) che riporta i codici digitati sul prompt, senza comunque riportare le 'risposte' di R. Tale file viene creato automaticamente al momento della chiusura della sessione quando questa viene salvata (`q("yes")`), oppure può essere esplicitamente creato attraverso il menù "File|Save History..." in un qualsiasi momento durante la sessione di lavoro. Il classico *file log*, ovvero l'insieme delle istruzioni digitate insieme con le rispettive risposte del sistema, può essere salvato attraverso "File|Save to File..."

- il *workspace* (spazio-di-lavoro) (`.Rdata`) che contiene tutti gli oggetti salvati nell'ambiente durante l'attuale sessione. Come per i file `.Rhistory`, è possibile salvare il *workspace* in un qualsiasi momento utilizzando la funzione `save.image()` (oppure dal menù "File|Save Workspace..."), od anche la funzione `save()` che consente di salvare soltanto alcuni oggetti del *workspace* specificandoli come argomenti. Un importante argomento di tali funzioni è `ascii` che se posto a `TRUE` permette di salvare una 'rappresentazione' ascii di tutti gli oggetti; questo è utile se i dati devono essere letti da differenti versioni di R installate su macchine diverse.

Ogni file di dati (ogni *workspace*) di R, può essere importato in una sessione utilizzando la funzione `load()` (o equivalentemente con "File|Load Workspace..."), oppure semplicemente avviando R dal file `.Rdata` di interesse.

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; ad esempio, $(2, 5, 9.5, -3)$ rappresenta un vettore di dimensione 4 in cui 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 nelle posizioni 2 e 4
[1]  5 -3
> x[-c(1,3)] #escludi quelli nelle posizioni 1 e 3
[1]  5 -3
> x[x>0] #seleziona i sui elementi positivi
[1] 2.0 5.0 9.5
> x[!(x<=0)] #escludi i suoi elementi non strettamente 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 (posizioni) dell'oggetto selezionare: un indice (`[2]`) o un vettore di indici (`[c(2,4)]`) o una condizione (`[x>0]`), sono tutte scritture ammesse. Sono anche ammesse le rispettive 'negazioni': il segno `-` precede il vettore di indici corrispondenti agli elementi da escludere, mentre `!` serve a negare la condizione che precede, invertendo in questo modo i valori del vettore logico generato dalla condizione stessa: gli elementi in corrispondenza dei `TRUE` saranno selezionati.

Naturalmente ogni selezione da un oggetto, in qualunque modo venga effettuata, è essa stessa un oggetto! così `x[x>0]` è un oggetto, in particolare nell'esempio di sopra è un vettore tridimensionale su cui è possibile svolgere una qualsiasi operazione, compreso un'ulteriore estrazione di sue componenti. È importante notare che la riga `x[x>0]-1` sottrae 1 da *ogni* elemento del vettore, così come `x[x>0]*2`, diciamo, moltiplicherebbe ogni elemento per 2. Questa è ancora un'altra caratteristica del linguaggio: le operazioni con oggetti 'multipli' (quali vettori e in seguito vedremo matrici) sono eseguite su ogni elemento dell'oggetto; naturalmente si deve trattare di una operazione ammissibile. Ad esempio:

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

somma gli elementi corrispondenti, mentre `x[x>0]+c(2,5)` è sbagliata, in quanto un vettore di dimensione 3 viene sommato con uno di dimensione 2, ed un messaggio di attenzione (*warning*) appare sullo schermo; comunque in questo caso un risultato viene comunque restituito, perché il vettore più piccolo viene 'riciclato', ovvero viene ripetuto fino al raggiungimento della stessa lunghezza. Quindi

```
> x[x>0]+c(2,5)
[1]  4.0 10.0 11.5
Warning message:
longer object length
        is not a multiple of shorter object length in: x[x>0] + c(2,5)
```

Quanto detto vale anche per gli altri operatori matematici. Si noti che il riciclo di argomenti (spesso vettori) è una operazione abbastanza frequente che R utilizza per cercare di eseguire comandi inizialmente non compatibili; comunque tale comportamento è dichiarato nei file-di-aiuto delle funzioni che fanno questo.

Per conoscere la posizione (gli indici) assunta dagli elementi di un vettore che soddisfano una particolare condizione è possibile utilizzare la funzione `which()`, che richiede come argomento un vettore di tipo logico; ad esempio, dato il vettore `x` precedentemente costruito si vuole conoscere la posizione assunta dagli elementi maggiori di zero:

```
> x
[1]  2.0  5.0  9.5 -3.0
> which(x>0)
[1] 1 2 3
```

Si osservi come le operazioni con `which()` di cui sopra possono essere equivalentemente svolte in modo meno efficiente con `c(1,2,3,4)[x>0]`, ovvero selezionando opportunamente gli elementi del vettore degli indici. Se invece si vogliono conoscere le posizioni dei valori del (primo) minimo e del (primo) massimo è possibile utilizzare le funzioni `which.min()` e `which.max()`:

```
> which.min(x)
[1] 4
> which.max(x)
[1] 3
```

Una utilissima funzione è la funzione `length()` che quando applicata a vettori, ne restituisce la dimensione:

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

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, ad esempio:

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

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 possibile effettuare soltanto qualche operazione:

```
> v<-c("barletta",'bracciano',"palermo")
#si noti l'equivalenza fra i simboli " e '
> 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, i numeri inseriti in vettori 'misti' vengono comunque considerati come dei caratteri:

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

le virgolette indicano che effettivamente si tratta di caratteri; si vedrà 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`. A scopo illustrativo si osservino i seguenti codici:

```
> x<-1:10 #od anche 10:1
> 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
```

Infine riportiamo alcuni codici per ordinare un vettore in ordine crescente o decrescente

```
> x
[1]  2.0  5.0  9.5 -3.0
> sort(x) #in ordine crescente..
[1] -3.0  2.0  5.0  9.5
> sort(x,decreasing=TRUE) #..in ordine decrescente
[1]  9.5  5.0  2.0 -3.0
> x[order(x)]
[1] -3.0  2.0  5.0  9.5
> x[order(x,decreasing=TRUE)]
[1]  9.5  5.0  2.0 -3.0
```

nell'esempio di sopra, `sort()` ha un'uso più diretto, ma se si vuole effettuare un ordinamento rispetto ad un altro vettore (anche più di uno) allora la funzione `order()` diventa molto utile:

```
> xx<-c(100,15,20,21)
> x[order(xx)] #ordina x secondo i valori di xx
[1]  5.0  9.5 -3.0  2.0
```

Concludiamo questo paragrafo facendo osservare che le componenti dei vettori possono essere nominate, o al momento della sua costruzione (inserendo i nomi quando si utilizza la funzione `c()`), oppure attraverso la funzione `names()` per un vettore già esistente:

```
> ax<-1:3
> names(ax)
NULL
> names(ax)<-c("lucius","sergius","catilina") #aggiungi i nomi
>
> a<-c(sic=2,ric=2.4)
```

```
> a
sic ric
2.0 2.4
> names(a) #visualizza i nomi
[1] "sic" "ric"
```

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 interi, che costituiscono l'indice di riga e quello di colonna. Si notino 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 stessa ed il numero delle colonne possibilmente coerente con il primo argomento: ad esempio scrivere `matrix(1:10,nrow=4,ncol=3)` non sarebbe una espressione corretta, ma come discusso in precedenza il vettore `1:10` verrebbe riciclato per ottenere il risultato ed un *warning* verrebbe stampato. Come per i vettori, l'utilizzo di `[ ]` è utile per selezionare elementi della matrice utilizzando la "`,`" per separare gli indici di riga e di colonna.

Ulteriori funzioni utili per la costruzioni di matrici sono `cbind()`, `rbind()` e `diag()` che consente di costruire una matrice diagonale o di estrarre la diagonale da una matrice:

```
> cbind(1:2,c(1,-2),c(0,9))#dispone i tre vettori per colonne
     [,1] [,2] [,3]
```

```
[1,]    1    1    0
[2,]    2   -2    9
> rbind(1:2,c(1,-2),c(0,9))#...li dispone per righe
     [,1] [,2]
[1,]    1    2
[2,]    1   -2
[3,]    0    9
> 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()` consente di risolvere sistemi di equazioni lineari, ma può anche essere utilizzata per il calcolo della matrice inversa

```
> solve(X) #l'inversa di X
     [,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, la dimensione di una matrice è caratterizzata da una coppia di numeri, e la funzione `dim()` è utilizzata per restituire il numero di righe e colonne:

```
> dim(x) #dimensione della matrice
[1] 2 5
> dim(x)[1] #numero di righe
[1] 2
```

Quando `matrix()` viene utilizzata senza il suo primo argomento, R inserisce valori mancanti, etichettati con `NA`, per cui una matrice con tre righe e quattro colonne può essere costruita nel seguente modo:

```
> matrix(,nrow=3,ncol=4)->ma
> ma
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

A questo punto i valori mancanti di tale matrice possono essere sostituiti inserendo righe o colonne direttamente come visto in precedenza; alternativamente disponendo di un vettore numerico di dimensione pari al numero di elementi della matrice (numero di righe $\times$ numero di colonne), è possibile 'riempire' la matrice appena creata con gli elementi del nuovo vettore:

```
> ma[]<-1:12
> ma
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

osservando che R inserisce gli elementi per colonna.

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 matrice trasposta:

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

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

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

Utilizzando un unico vettore $\boldsymbol{x}$, è facile constatare che uno scalare ed una matrice possono essere ottenuti attraverso $\boldsymbol{x'x}$ e $\boldsymbol{xx'}$, ovvero

```
> a1<-t(as.vector(x))%*% as.vector(x)
> a2<-as.vector(x)%*% t(as.vector(x))
```

In realtà i risultati di tali operazioni sono comunque due matrici, la prima avente come unico elemento $\sum x_i^2$, e la seconda di dimensione `length(x)`×`length(x)` con elemento generico dato da $x_i \times x_j$; per 'semplificare' la dimensionalità dell'oggetto (ovvero eliminare le dimensioni pari a uno) è possibile procedere nel seguente modo:

```
> a1
     [,1]
[1,]  185
> drop(a1)
[1] 185
> a1[,,drop=TRUE]
[1] 185
>
```

così `drop()` avrebbe potuto trasformare in un vettore una matrice di dimensione $1 \times n$ o $n \times 1$.

Prodotti matriciali tra matrici e/o vettori, quali `t(x)%*%y`, possono essere più efficientemente effettuati attraverso la scrittura `crossprod(x,y)`; quando un solo argomento viene fornito, `crossprod(x)` è equivalente a $\boldsymbol{x'x}$. Si osservi che poiché il risultato è ancora una matrice, `drop` potrebbe essere utilizzato per ottenere uno scalare.

Una particolare matrice in Statistica è la matrice dei dati definita da $\boldsymbol{X}_{n\times J}$, essendo n il numero di unità d'analisi 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* possano 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(3,4,1,2)) #crea un array
> dim(a) #la sua dimensione
[1] 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
```

Per cercare di fissare le idee, un *array* $3\times4\times2$ 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 due livelli di un confondente. L'uso delle parentesi quadre, alla stregua di vettori e matrici, ha il fine di selezionare sotto-insiemi dell'*array*, così `a[,,2]` è (la seconda) matrice 3×4 e `a[1,,]` è la prima matrice 4×2. Si noti che anche in un array le dimensioni pari ad uno possono essere eliminate:

```
> a<-array(1:24, dim=c(3,4,2))
> dim(a)
[1] 3 4 1 2
> dim(a[,,,,drop=TRUE]) # oppure dim(drop(a))
[1] 3 4 2
```

dove il numero delle virgole `,` all'interno delle parentesi quadre è pari al numero delle dimensioni.

2.4 'Liste'

In R una *list* (elenco o lista) è (naturalmente un oggetto) una raccolta di altri oggetti, anche differenti tra loro, compreso altre liste. Ad esempio questo non è vero per i vettori o le matrici i cui elementi sono tipicamente tutti numeri o tutti caratteri. Una lista può essere creata con il comando `list()` ed il numero degli oggetti che la costituiscono definisce la dimensione della lista, mentre le sue componenti sono individuate con semplici `[]` o doppie `[[ ]]` parentesi quadre:

```
> #crea una lista
> lista<-list(matrix(1:9,nrow=3),rep(0,3),c("buono",'cattivo'))
> length(lista) #la sua dimensione
[1] 3
> lista[[3]] #il suo terzo elemento
[1] "buono"  "cattivo"
> length(lista[[3]])
[1] 2
> lista[[2]]+2 #un'operazione sul secondo elemento
[1] 2 2 2
> lista[[1]][2,2]
[1] 5
```

Come si vede, `lista` contiene una matrice, un vettore numerico ed uno di caratteri; l'estrazione di oggetti appartenenti alla lista stessa è semplice e naturalmente consente di trattare ogni singola (sub-)componente come un oggetto separato e indipendente su cui potere effettuare le operazioni (consentite). Se per estrarre gli oggetti della lista si usano le singole (piuttosto che le doppie) parentesi quadre, l'oggetto verrà comunque estratto ma sarà ancora una lista; la funzione `unlist()` può essere utilizzata per 'rimuovere' dall'oggetto la struttura della lista:

```
> lista[[3]] #è un vettore
[1] "buono"   "cattivo"
> lista[3] #è una lista
[[1]]
[1] "buono"   "cattivo"
> length(lista[3]) #non è 2..
[1] 1
> unlist(lista[3])
[1] "buono"   "cattivo"
```

Avremmo anche potuto nominare ogni elemento della lista o semplicemente specificando il nome all'interno di `list()`, cioè `list(primo=matrix(..),..)`, o attraverso la funzione `names()` che restituisce `NULL` se la lista è stata creata senza nomi.

```
> names(lista) #questa lista è senza nomi
NULL
> names(lista)<-c("primo",'secondo',"terzo") #nomina gli elementi
```

Quando i nomi sono stati assegnati è possibile estrarre ogni elemento della lista invocando direttamente il proprio nome, attraverso il simbolo `$` o con le parentesi quadre, la differenza essendo nel tipo di oggetto restituito:

```
> #estrai il secondo elemento:
> lista$secondo #un vettore
[1] 0 0 0
> lista["secondo"] #una lista
$secondo
[1] 0 0 0
```

I due modi di richiamare gli oggetti non sono del tutto equivalenti in quanto le singole parentesi quadre (con il numero o con il nome) preservano il nome dell'oggetto estratto:

```
> names(lista["secondo"])
[1] "secondo"
> names(lista[2])
[1] "secondo"
```

questo può essere molto utile in fase di programmazione.

Oltre alle liste e vettori visti in precedenza, matrici e array possono essere nominati facendo uso di liste. Per le matrici in particolare, l'assegnazione di etichette alle righe e o colonne può risultare molto utile ed avviene sfruttando liste di nomi in cui ciascuna componente si riferisce ad una dimensione:

```
> 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
```

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 (`dimnames(x)[[1]]`) e quello dei nomi di colonna (`dimnames(x)[[2]]`). A tale lista, comunque, è possibile accedere utilizzando direttamente le funzioni `rownames()` e `colnames()`. Inoltre, così come per `array()`, anche `matrix()` ha un argomento `dimnames` che consente di inserire direttamente i nomi in questione al momento della creazione dell'oggetto. Ad esempio:

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

Si noti l'utilizzo di `NULL`, l'oggetto vuoto (nullo) di R. Concludiamo questo breve paragrafo sulle liste commentando i codici scritti a pagina 8 per modificare i prompt: la semplice `options()` restituisce una lista in cui ciascuna componente è 'un'opzione' di R (ad esempio il prompt). Poiché la lista è nominata, allora `options()$prompt` fornisce la componente "prompt", mentre `names(options())` restituisce tutti i nomi delle componenti dell'intera lista:

```
> a<-options() #salva tutte le opzioni in una lista
> length(a) #il numero delle opzioni (dim della lista)
[1] 38
> names(a)[1:3] #le prime tre..
[1] "prompt"   "continue" "editor"
> is.list(a) #è una lista?
[1] TRUE
```

La funzione `is.list()` verifica che il suo argomento sia effettivamente una lista restituendo `TRUE` o `FALSE`; sebbene non introdotte in precedenza esistono anche le corrispondenti `is.vector()`, `is.matrix()` e `is.array()`.

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* rappresenta la matrice dei dati in cui ad ogni riga corrisponde una osservazione e ad ogni colonna una variabile, e nell'ambiente viene trattato come una lista. Ogni elemento di tale lista rappresenta una variabile statistica, per cui `length()` restituisce il numero delle variabili, mentre `names()` i rispettivi nomi; è anche possibile aggiungere/modificare i nomi di riga (attraverso `row.names()`), ma probabilmente per un dataframe questo potrebbe non essere molto utile. Tra le diverse opzioni disponibili, è possibile costruire un `data.frame` direttamente con la funzione `data.frame()`

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

Il dataframe creato sopra è definito da 4 casi e 3 variabili (due numeriche ed una 'carattere') che possono essere selezionate in tre modi diversi, utilizzando sia il nome (ad es., `X$sesso` o `X[,"sesso"]`), sia il numero di colonna che occupa (ad es., `X[,2]`); il risultato sarà comunque un vettore.

Nelle applicazioni è spesso utile selezionare soltanto una parte del dataframe iniziale, ad esempio solo alcune variabili o soltanto alcuni casi soddisfacenti certi criteri specificati; a tale scopo si notino i seguenti codici:

```
> #seleziona i valori della variabile eta per i maschi:
> X[X$sesso=="M","eta"]
[1] 2.5 6.2
> #seleziona i valori della  variabile sesso per cui eta<=3:
> X$sesso[X$eta<=3]
[1] M F
Levels:  F M
```

L'uso delle parentesi quadre consente di selezionare opportunamente 'i casi' desiderati; nel secondo caso la selezione viene effettuata sugli elementi del

vettore estratto (`X$sesso`) nel primo caso la condizione per selezionare i casi viene specificata all'interno delle parentesi quadre prima della virgola, mentre dopo la virgola appaiano i nomi delle variabili a cui la condizione va applicata. Il discorso può essere facilmente generalizzato per estendere il numero delle variabili da selezionare e per complicare la condizione di inclusione (o esclusione) contemplando contemporaneamente anche altri simboli quali `&` e `|`. Questi ultimi due operatori logici vengono solitamente utilizzati per specificare che le condizioni si verifichino 'simultaneamente' o 'alternativamente'. Ad esempio:

```
> #seleziona i valori di "a" se eta<=5 E eta>3:
> X$a[X$eta<=5 & X$eta>3]
[1] 3
> #seleziona i valori di "a" e "sesso" se eta<3 O eta>5:
> X[X$eta<3 | X$eta>5,c("a","sesso")]
  a sesso
1 1     M
4 4     M
```

Sostanzialmente la condizione di selezione dei record è un vettore logico (ottenuto combinando altri vettori logici) che quando utilizzato come filtro fa in modo che solo gli elementi in corrispondenza di `TRUE` vengano selezionati. Sebbene l'uso dei codici sopra possa servire a selezionare parti di un dataframe, un modo più diretto per l'utente è attraverso la funzione `subset()`, in cui gli argomenti `subset` e `select` specificano rispettivamente i casi e le variabili da includere (o escludere).

```
> subset(X,subset=(eta<3 | eta>5),select=c(a,sesso))
  a sesso
1 1     M
4 4     M
> subset(X,subset=(eta<3 | eta>5),select=-eta)
  a sesso
1 1     M
4 4     M
```

Si noti l'uso del segno '`-`' per escludere le variabili. Comunque se si vuole eliminare soltanto una variabile allora è possibile procedere riassegnando a quella variabile `NULL`, ovvero un oggetto vuoto; perciò `X$eta<-NULL` elimina `X$eta`.

3.1 Importazione di dati

La funzione `data.frame()` così come vista precedentemente non esaurisce le diverse possibilità per affrontare il problema dell'inserimento e gestione di dati. Ad esempio la funzione `as.data.frame()` può essere utilizzata per forzare una matrice di dati ad un *dataframe*, oppure i dati potrebbero essere inseriti più facilmente attraverso un foglio elettronico: in questo caso `X<-data.frame()` crea un dataframe che è poi possibile aprire con `fix(X)` per l'inserimento dei dati direttamente nelle celle.

Un'altra possibilità, che è probabilmente la più frequente nelle applicazioni, è quella di importare i dati che sono disponibili in un file ASCII ottenuto da un qualsiasi altro programma. La funzione `read.table()` può essere utilizzata a tale scopo:

```
> X<-read.table(file="c:/documenti/dati.txt",
+   header=TRUE,
+   sep="\t",
+   na.strings = "NA",
+   dec=".")
```

in questo caso il file `dati.txt` (disponibile nella cartella 'c:\documenti') viene importato e automaticamente convertito nel dataframe `X`. A questo punto sono utili le seguenti osservazioni:

- il percorso del file (*path*) viene scritto come negli ambienti Unix (o Linux) con `/`, oppure con `\\`;
- l'argomento `header=TRUE` specifica che la prima linea del file contiene i nomi delle variabili;
- l'argomento `sep="\t"` indica che i diversi campi sono separati da un tab. Avremmo potuto specificare `sep=","` nel caso di una virgola e così via;
- l'argomento `na.strings="NA"` può essere particolarmente utile se nel file sono presenti valori mancanti, in questo caso individuati con `NA`;
- l'argomento `dec="."` specifica il tipo di carattere utilizzato nel file per separare i decimali, in questo caso un punto.

Tra gli altri argomenti disponibili, `nrows` e `skip` possono essere molto utili quando si stanno importando file contenenti molte righe (record) non tutte di attuale interesse. Il primo argomento specifica il numero massimo di righe da leggere, mentre il secondo il numero di righe iniziali da saltare prima dell'importazione del file.

3.2 Valori mancanti

Molto spesso i dati raccolti sono caratterizzati da valori mancanti che in R sono etichettati 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 se si stanno importando dati da un file esterno, è molto probabile che i valori mancanti siano etichettati con qualcosa di diverso da `NA`; ad esempio un asterisco, oppure valori palesemente insensati, come 999. In questo caso nel *dataframe* tutti i 999 dovranno essere

sostituiti con `NA`. Questo può essere ottenuto facilmente utilizzando l'argomento `na.strings=999` nella funzione `read.table()`, oppure una sostituzione può avvenire facilmente nel seguente modo

```
> 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 il valore logico `TRUE` 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 oppure necessitano di particolare attenzione al momento del loro utilizzo. Per questa ragione è auspicabile avere 'pieno controllo' dei mancanti: ad esempio, è possibile selezionare a priori solo i dati non-mancanti utilizzando la stessa funzione `is.na()`.

```
> x[is.na(x)==FALSE]
[1]  1  2  3  4  6  7  9 10
> x[!is.na(x)==TRUE]
[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, ma la terza, in generale, è quella più compatta e quindi preferita; si noti l'uso della negazione `!` e l'omissione del `==TRUE` che non è necessaria e viene solitamente omessa. Allo stesso modo da un dataframe possono essere selezionati i record con non-mancanti:

```
> X$eta[1]<-X$sesso[4]<-NA
> X1<-X[!is.na(X$sesso),]
> X2<-X[!is.na(X$sesso)&!is.na(X$eta),]
```

La differenza tra `X1` e `X2` è che nel primo vengono eliminati soltanto i record con valori non-mancanti per la variabile `sesso`, mentre nel secondo quelli per cui il dato è mancante anche per la variabile `eta`; in generale quando ci sono dati mancanti relativi a diverse variabili, il sub-dataframe avente soltanto record con valori completi potrà essere ottenuto attraverso la funzione `na.omit()`, cioè semplicemente `na.omit(X)`.

3.3 Codifica di variabili

In Statistica, come è noto, 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",..)`. Sebbene tale procedimento sia del tutto ammissibile, il modo più corretto per specificare una variabile categoriale in R è attraverso la funzione `factor()` che sostanzialmente trasforma i numeri in etichette:

```
> x<-factor(c(1,1,2,3,1)) #crea una variabile categoriale
> x
[1] 1 1 2 3 1
Levels:  1 2 3
> #crea una variabile categoriale e nomina le etichette:
> 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 tra le due linee di comando sopra è nell'apparenza! Con l'argomento `labels` si è semplicemente stabilita una corrispondenza tra l'etichetta (il numero) e le categoria in modo tale che nei risultati R visualizzi i nomi delle categorie invece che i semplici numeri. Questo potrebbe tornare comodo in qualche circostanza, ad esempio al momento di produrre grafici, ma è bene ribadire che l'uso di `factor()` con il suo solo primo argomento specificato è sufficiente per definire una variabile categoriale. `factor()` appare essere particolarmente utile quando il file di dati da importare contiene variabili categoriali con modalità espresse da numeri piuttosto che esplicitamente dalle categorie; in tal caso è sufficiente ri-assegnare alla variabile in questione la sua versione "factor". Ad esempio assumendo la variabile `gruppo` nell'ipotetico *dataframe* `dati`, allora

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

'trasformerà' i numeri della variabile `gruppo` in etichette.

Esistono altre opzioni in `factor()` per codificare variabili categoriali; ad esempio l'argomento `ordered` (o la funzione `ordered()`) per definire variabili ordinali e l'argomento `levels` molto utile per eliminare dall'analisi qualche categoria e considerarla alla stregua di un valore mancante. Ad esempio considerando la variabile categoriale `x` creata in precedenza:

```
> factor(c(1,1,2,3,1),levels=1:2) #escludi la categoria 3
[1] 1    1    2    <NA> 1
Levels: 1 2
> factor(x,levels=c("gruppo1","gruppo3")) #escludi "gruppo2"
[1] gruppo1 gruppo1 <NA>    gruppo3 gruppo1
Levels: gruppo1 gruppo3
```

Come si vede le categorie non specificate in `levels` vengono considerate come valori mancanti e sostituite con `NA`.

A volte può essere necessario '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<-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()` crea una variabile categoriale dividendo la variabile numerica che figura come suo argomento in intervalli tipo $]x_i, x_{i+1}]$ dove gli estremi possono o meno essere inclusi in funzione di come vengono specificati gli argomenti `include.lowest` e `right`; in `breaks` possono essere espressi sia gli estremi degli intervalli (come nell'esempio sopra) o il numero degli intervalli desiderato, mentre `labels` serve soltanto a nominare le etichette della variabile categoriale appena creata; ponendo `labels=FALSE` i semplici interi '1','2',... vengono utilizzati come etichette.

4 Sintesi di una distribuzione

4.1 Qualche statistica descrittiva

Supponiamo di avere rilevato una o più variabili $X_1, X_2, \ldots$, su n unità e che tali dati siano disponibili in R sotto forma di un dataframe `X`. In generale questo potrebbe essere stato creato in R inserendo i dati attraverso la funzione `data.frame()` oppure importato con `read.table()`, ma qui per semplicità si assumerà un dataframe simulato; in questo modo il lettore potrà anche 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=factor(rep(1:3,length=150)))
> dim(X)
[1] 150   2
```

La seconda variabile del dataframe è una variabile categoriale con tre livelli, mentre la prima è stata creata con la funzione `rnorm()` che 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 pseudo-casuali; poiché ad ogni nuova generazione di numeri casuali (da una qualsiasi densità) il seme cambia, uno stesso seme è necessario per ottenere gli stessi valori simulati; Infatti:

```
> set.seed(95)
> rnorm(5, 2.5, 1)
[1] 1.4708796 0.8844742 2.4721205 2.1788724 4.3803713
> rnorm(5, 2.5, 1)
[1] 3.196807 1.635123 1.425115 3.494412 2.269954
> set.seed(95)
> rnorm(5, 2.5, 1)
[1] 1.4708796 0.8844742 2.4721205 2.1788724 4.3803713
```

cioè solo anteponendo uno stesso seme alla generazione di quantità pseudo-casuali si hanno gli stessi risultati. R ha molte altre funzioni per la generazioni di numeri pseudo-casuali, ad esempio `rpois()` per variabili poissoniane o `rchi()` per v.c. χ^2 con argomenti che naturalmente dipendono 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 tutti con un vettore di
#stringhe di lunghezza pari al numero di variabili del data frame
> names(X)
[1] "y"      "gruppo"
> names(X)[2]<-"eta" #modifica solo la seconda variabile
> names(X)
[1] "y"     "eta"
```

Si noti che poiché le variabili sono contenute in un *dataframe*, esse possono avere nomi uguali ad altri oggetti contenuti nell'*ambiente globale*, senza che ci sia una qualche sovrapposizione; infatti:

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

`y` è contenuto nell'ambiente globale, mentre la scrittura `X$y` indica la componente `y` di una lista `X` dell'ambiente globale. Per cui per richiamare ed utilizzare una variabile in un dataframe è necessario specificare il dataframe di appartenenza, o alternativamente è possibile 'attaccare' il dataframe in memoria, come discusso più avanti.

Quando viene chiesto ad R di valutare una linea di comando, R nell'interpretare i codici cercherà i diversi oggetti che figurano nella scrittura (compreso funzioni) in una serie di 'posti' ordinati che definiscono il così-detto *search-path*. La funzione `search()` (o `searchpaths()`, entrambe chiamate senza alcuno argomento) restituisce un vettore con i nomi dei diversi posti del *search-path* in cui trovare gli oggetti necessari per eseguire l'espressione. Ad esempio i primi tre sono:

```
> search()[1:3]
[1] ".GlobalEnv"      "package:methods" "package:stats"
```

Tralasciando gli altri, notiamo che il primo posto è riservato all'ambiente globale (`.GlobalEnv`) e poi via di seguito ai successivi, tra i quali non figura il *dataframe* in questione. Quindi per poter richiamare e utilizzare le variabili contenute al suo interno è necessario aggiungere al search-path anche il dataframe; questa operazione viene effettuata attraverso la funzione `attach()`:

```
> attach(X) #attacca X in memoria
> search()[1:3] #parte del search-path
[1] ".GlobalEnv"      "X"               "package:methods"
```

Comunque, come si evince dalla 'risposta' di `search()`, tale operazione metterà il dataframe `X` in posizione 2, in quanto la prima è sempre riservata all'ambiente globale; come conseguenza se esiste un oggetto con lo stesso nome di una variabile (in questo esempio `y`), digitando tale nome verrà invocato l'oggetto nell'ambiente globale (in posizione 1) piuttosto che quello nel dataframe (in posizione 2). Infatti:

```
> length(y) #non è 150..
[1] 2
> length(eta)
[1] 150
```

Un altro effetto indesiderato è che poi eventuali modifiche delle stesse variabili del dataframe creeranno nuove variabili nell'ambiente globale e non nel dataframe. Infine se `X` è di grandi dimensione, attaccandolo in memoria appesantirebbe il sistema in modo sensibile. Per queste ragioni l'uso di `attach()` non è, in generale, consigliabile.

Lo studio di una distribuzione rappresenta un aspetto comune di un'analisi statistica, anche se questa non è 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. La prima potrebbe essere calcolata molto facilmente attraverso:

```
> sum(X$y)/length(X$y)
[1] 169.6664
```

dove `sum()` calcola la somma di tutti gli elementi del suo argomento. Comunque R è già dotato di specifiche funzioni: `mean()` e `median()`, rispettivamente per la media e mediana:

```
> mean(X$y)
[1] 169.6664
> median(X$y)
[1] 169.4219
```

È evidente che per la variabile `X$y` media e mediana coincidano, sebbene questo in generale può non verificarsi poichè, come è noto, le due misure dipendono 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 l'argomento `prob` nella funzione `quantile()` che restituisce per default minimo, massimo e quartili:

```
> quantile(X$y)
      0%      25%      50%      75%     100%
158.5055 166.2774 169.4219 172.6615 182.3126
> quantile(X$y, prob=c(.5,.6,.9)) #calcola altri quantili
     50%      60%      90%
169.4219 170.7881 176.0861
```

Di contro, il percentile di un dato valore può essere calcolato contando la frazione dei valori maggiori

```
> sum(X$y>=170)/length(X$y)
[1] 0.4533333
```

dove il risultato di `X$y>=170` è un vettore logico la cui somma si riferisce al numero dei `TRUE`.

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

```
> X$y[c(2,56,90)]<-NA #introduci qualche mancante in X$y
>
> set.seed(113)
> id<-sample(x=1:150,size=2)
> id
[1] 83 95
> X$eta[id]<-NA #altri mancanti in X$eta
```

In `y` sono stati introdotti `NA` nelle posizioni 2,56 e 90, mentre nell'altra variabile si è proceduto attraverso una estrazione casuale degli indici. La funzione `sample()` estrae casualmente `size` (2 in questo caso) elementi dal vettore `x` (`1:150`); analogamente al caso di generazione di numeri casuali, `set.seed()` viene utilizzata per assicurarsi che ulteriori chiamate a `sample()` portino alla stessa estrazione.

L'argomento `na.rm=TRUE` indica che gli eventuali valori mancanti in `X$y` devono essere omessi nel calcolo della media, altrimenti (cioè se `na.rm=FALSE`, il default) il risultato sarebbe anche mancante:

```
> mean(X[,"y"])
[1] NA
> mean(X[,"y"], na.rm=TRUE)
[1] 169.6914
> mean(na.omit(X$y)) #modo alternativo per eliminare i mancanti
[1] 169.6914
```

Anche `median()` e `quantile()` hanno l'argomento `na.rm`, ma in generale altre funzioni possono richiedere un argomento differente per la gestione dei dati mancanti; i file di aiuto sono molto chiari a questo riguardo. Infine, ricordando la funzione `is.na()` (vedi paragrafo 3.2) un'altra possibilità per eliminare i mancanti è data da

```
> mean(X$y[!is.na(X$y)])
[1] 169.6914
```

dove il filtro di riga `!is.na(X$y)` può essere utilizzato anche come argomento `subset` nella funzione `subset()` per ottenere un dataframe completo. Alternativamente a come illustrato sopra, è possibile utilizzare la funzione `na.omit()` che elimina direttamente i valori 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] 145   2
```

cioè ogni riga con almeno un mancante (relativo a qualsiasi colonna del dataframe) viene eliminata; in tal caso si hanno rispettivamente 3 e 2 mancanti per `X[,1]` e `X[,2]` e quindi il dataframe completo avrà 145 righe.

Piuttosto che calcolare singolarmente le diverse misure di sintesi per ogni variabile, la funzione `summary()` applicata ad un vettore numerico, quali sono le singole variabili statistiche, restituisce diverse misure di sintesi, incluso il numero dei mancanti

```
> summary(X$y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  158.5   166.4   169.4   169.7   172.6   182.3     3.0
> summary(X$eta)
bambino  adulto anziano    NA's
     50      48      50       2
```

Si noti quale è il risultato quando la variabile è categoriale. Comunque `summary()` è una funzione 'più generica'; infatti se applicata ad un intero dataframe automaticamente restituisce la sintesi di ciascuna variabile costituente il dataframe stesso:

```
> summary(X)
       y              eta
 Min.   :158.5   bambino:50
 1st Qu.:166.4   adulto :48
 Median :169.4   anziano:50
 Mean   :169.7   NA's   : 2
 3rd Qu.:172.6
 Max.   :182.3
 NA's   :  3.0
```

Come è noto, la 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=TRUE)
[1] 22.76201
```

Più in generale, la funzione `var()` applicata ad una matrice (o un dataframe) consente di calcolare la matrice di varianze-covarianze tra le coppie di colonne della matrice (o del dataframe). Analogamente le covarianze o le correlazioni fra le coppie di variabili possono essere immediatamente ottenute con la funzione `cov()` e `cor()`.

```
> set.seed(72)
> X$z<-runif(n=150,10,20) #un'altra esplicativa
> X$x<-runif(n=150) #un'altra ancora..
> var(X[,c("y","z")],na.rm=TRUE)
          y        z
y 22.762012 2.789044
z  2.789044 9.705090
> cov(X[,c("y","z","x")],use="complete.obs")
           y           z           x
y 22.7620121  2.78904383  0.13837163
z  2.7890438  9.70508970 -0.03537063
x  0.1383716 -0.03537063  0.08462638
> cov(X[,c("y","z","x")],use="pairwise.complete.obs")
           y           z           x
y 22.7620121  2.78904383  0.13837163
z  2.7890438  9.58326373 -0.02690087
x  0.1383716 -0.02690087  0.08482712
```

Quando utilizzata con un solo argomento, `cov()` restituisce pure una matrice di varianze-covarianze, ma l'argomento per tener conto dei mancanti non è `na.rm`. Invero ponendo `use="complete.obs"` tutte le righe con mancanti vengono eliminate prima dei calcoli (il risultato è simile a quello ottenuto con `cov(na.omit(X[,c("y","z","x")]))`), mentre con `"pairwise.complete.obs"` l'eliminazione dei mancanti avviene valutando singolarmente ogni coppia di variabili in questione: così per calcolare la covarianza tra `X$x` e `X$z` nessuna osservazione è stata eliminata, poiché nessun mancante è presente tra i valori di tali due variabili.

Quanto finora esposto su media e varianza afferisce alle variabili numeriche; la mediana può anche essere calcolata per variabili ordinali, ma per variabili categoriali lo studio di una distribuzione solitamente avviene attraverso le tabelle di frequenze che possono essere ottenute con `table()`:

```
> table(X$eta)

 1  2  3
```

```
50 48 50
> #aggiungi le etichette:
> X$eta<-factor(X$eta,labels=c('bambino','adulto','anziano'))
> table(X$eta)

bambino  adulto anziano
     50      48      50
> #aggiungi un'altra variabile categoriale:
> X$sesso<-gl(n=2,k=2,length=150,labels=c("M","F"))
> table(X$eta,X$sesso) #tab. di conting. doppia

          M  F
  bambino 25 25
  adulto  26 22
  anziano 25 25
```

Nell'esempio di sopra la funzione `gl()` è stata utilizzata per generare una variabile categoriale di `length` osservazioni con `n` livelli, ciascuno ripetuto `k` volte; tale risultato è ottenibile attraverso le funzioni `rep()` e `factor()` (su cui la stessa `gl()` è basata), ma talvolta il suo utilizzo può risultare comodo in quanto più immediato. Da notare inoltre come la tabulazione di variabili categoriali sfrutti le etichette delle variabili stesse, una cosa che può portare a risultati più leggibili soprattutto quando si devono considerare più variabili. Distribuzioni bivariate o multivariate sono facilmente ottenibili semplicemente aumentando l'elenco degli argomenti in `table()`; le tabelle di contingenza multiple risultanti sono array con dimensione e nomi-di-dimensione dipendenti dal numero e dalle etichette delle categorie delle variabili in questione:

```
> #una distribuzione tripla
> a<-table(eta=X$eta,sesso=X$sesso,gl(2,1,150))
> is.array(a)
[1] TRUE
> dim(a)
[1] 3 2 2
>
> dimnames(a)
$eta
[1] "bambino" "adulto"  "anziano"

$sesso
[1] "M" "F"

[[3]]
[1] "1" "2"
```

L'uso di array potrebbe risultare talvolta un po' problematico, soprattutto al crescere del numero delle variabili in esame. Un modo alternativo e forse più

semplice per rappresentare tabelle di contingenza è il così-detto ordine lessicografico, che è sostanzialmente una ri-organizzazione dell'array in una matrice avente come righe tutte le possibili combinazioni di categorie delle varie variabili. La funzione `as.data.frame()` applicata ad un array assolve tale compito:

```
> d=as.data.frame(a) #la tabella in ordine lessicografico
> d
       eta sesso Var3 Freq
1  bambino     M    1   13
2   adulto     M    1   13
3  anziano     M    1   12
4  bambino     F    1   12
5   adulto     F    1   10
6  anziano     F    1   13
7  bambino     M    2   12
8   adulto     M    2   13
9  anziano     M    2   13
10 bambino     F    2   13
11  adulto     F    2   12
12 anziano     F    2   12
```

La colonna `Freq` viene creata automaticamente, mentre i nomi delle altre colonne e relative etichette sono desunti dai nomi delle dimensioni dell'array (`dimnames`, se esistenti) che a loro volta sono stati ottenuti dai nomi delle variabili utilizzate in `table()` per costruire l'array. A tale riguardo si noti come, sebbene non strettamente necessario ai fini dei risultati, l'uso dei nomi e delle etichette nelle gestione delle variabili categoriali porti a risultati più prontamente e facilmente leggibili e presentabili; invero incrociare più variabili magari con lo stesso numero di modalità senza nomi e/o etichette (come è stato volutamente fatto nell'esempio di sopra con la terza variabile) porterebbe a oggetti di difficile interpretazione e lettura. Come sarà discusso più avanti, l'ordine lessicografico di una tabella multidimensionale sarà necessario per l'analisi di tabelle di contingenza attraverso i modelli di regressione.

L'operazione inversa a quella appena descritta, ovvero costruire una tabella di contingenza in forma di array a partire da una in ordine lessicografico organizzata in un dataframe, può essere effettuata attraverso la funzione `xtabs()` come illustrato nel seguente esempio:

```
> a1<-xtabs(formula=Freq~eta+sesso+Var3,data=d)
> a1
, , Var3 = 1

         sesso
eta       M  F
  bambino 13 12
  adulto  13 10
  anziano 12 13
```

```
, , Var3 = 2

         sesso
eta       M  F
  bambino 12 13
  adulto  13 12
  anziano 13 12
```

i cui argomenti principali sono il dataframe (`data`) che contiene le informazioni necessarie e la 'formula' (`formula`) in cui vengono specificate le variabili per le quali si vuole costruire la tabella di contingenza. La formula in R è un costrutto fondamentale e ricorrente, soprattutto in ambito di costruzione di modelli. Non è questa la sede per approfondire, ma in generale una formula è specificata da qualcosa come 'risposta~termini'; da notare il simbolo separatore `~`. 'risposta' rappresenta il nome della variabile oggetto principale di analisi, mentre 'termini' definisce una scrittura dove figurano altre variabili. Usualmente i nomi delle variabili in 'termini' saranno separate da un `+`, ma in generale altri simboli sono consentiti, ciascuno con un proprio significato. Riguardo a `xtabs()` la risposta è la variabile che definisce le frequenze, nell'esempio di sopra è una sola, `Freq`, mentre i termini sono le variabili da incrociare. Un numero inferiore di variabili è consentito per valutare le distribuzioni marginali; ad esempio

```
> xtabs(Freq~eta+sesso,data=d)
         sesso
eta       M  F
  bambino 25 25
  adulto  26 22
  anziano 25 25
> xtabs(Freq~eta,data=d)
eta
bambino  adulto anziano
     50      48      50
```

Si noti che `xtabs()` può essere utilizzata partendo direttamente da un *dataframe* di singole osservazioni; per esempio per la distribuzione dell'età si ha

```
> xtabs(rep(1,nrow(X))~eta,data=X)
eta
bambino  adulto anziano
     50      48      50
```

ovvero la variabile risposta in questo caso è solo una colonna di 1, necessaria per effettuare i conteggi ed ottenere poi le frequenze.

Concludiamo questa breve digressione sulla rappresentazione tabellare di variabili categoriali introducendo un'altra funzione che può essere utilizzata a tale scopo: `ftable()`. Il vantaggio di `ftable()` rispetto a suoi 'competitori' è che tale funzione consente di ottenere tabelle in cui siano distinte le variabili

'di riga' e quelle 'di colonna'. Fatta questa distinzione, la tabella restituita da `ftable()` ha per righe e per colonne tutte le combinazioni di categorie delle variabili di riga e di colonna rispettivamente. Tale rappresentazione può essere molto utile talvolta, per ottenere tabelle più facilmente interpretabili:

```
> ftable(X[,2],X[,5],gl(2,1,150),row.vars=c(1,2),col.vars=3)
            1  2

bambino M  13 12
        F  12 13
adulto  M  13 13
        F  10 12
anziano M  12 13
        F  13 12
```

dove i numeri in `row.vars` e `col.vars` che definiscono i due gruppi (riga e colonna) di variabili si riferiscono all'ordine delle variabili specificate come argomento. È possibile chiamare `ftable()` in modo diverso, utilizzando un dataframe o un array restituito da `table()`

```
> #un dataframe come argomento:
> ftable(X[,c(2,5)],row.vars="eta",col.vars="sesso")
> #equivalente ma più lento:
> ftable(X,row.vars="eta",col.vars="sesso")
>
> #un array come argomento:
> ftable(a,row.vars="eta",col.vars="sesso")
> ftable(a,row.vars=1,col.vars=2:3)
```

dove i numeri in `row.vars` e `col.vars` nell'ultima linea sono relativi alle dimensioni dell'array.

4.2 Le funzioni `tapply()`, `apply()`, `lapply()`, `sapply()`

Spesso, in funzione della 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 questo tipo di funzioni. Considerando sempre il *dataframe* `X`, si potrebbe essere interessati 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=X$y, INDEX=X$eta, FUN=mean)#ci sono dei mancanti quindi
 bambino   adulto  anziano
170.4221       NA       NA
> tapply(X$y, X$eta, mean, na.rm=TRUE) #..poni na.rm=TRUE
 bambino   adulto  anziano
170.4221 169.0742 169.5222
```

`tapply()` richiede di specificare la variabile di interesse (`X`), la variabile che individua i gruppi (`INDEX`), e la funzione da applicare (`FUN`) eventualmente seguita dai sui argomenti. La variabile di raggruppamento può essere costituita anche da una variabile ottenuta come 'interazione' di due o più variabili o una lista di fattori:

```
> tapply(X$y, X$eta:X$sesso, mean, na.rm=TRUE)
bambino:M bambino:F  adulto:M  adulto:F anziano:M anziano:F
 170.5986  170.2456  169.6816  168.3510  169.4371  169.6040
> tapply(X$y, list(X$eta,X$sesso), mean, na.rm=TRUE)
                M        F
bambino 170.5986 170.2456
adulto  169.6816 168.3510
anziano 169.4371 169.6040
```

Il risultato differisce soltanto da come i risultati sono presentati. Poiché ogni funzione che richiede come argomento un vettore è valida, è interessante osservare come ottenere tabelle di contingenza: utilizzando `tapply()`,

```
> tapply(X$eta,X$eta,length)
bambino  adulto anziano
     50      48      50
> tapply(X$eta,list(X$eta,X$sesso),length)
         M  F
bambino 25 25
adulto  26 22
anziano 25 25
```

naturalmente in questo caso il primo argomento non è rilevante.

Simile a `tapply()` è la funzione `by()` con la differenza che il primo argomento non è un vettore ma un dataframe (od anche una matrice) e di conseguenza la funzione da specificare dovrà essere relativa ad un dataframe e non ad un vettore. Ad esempio, come è stato visto in precedenza, `summary()` applicata ad un dataframe restituisce un vettore di misure sintetiche per ogni variabile del dataframe; per ottenere un tale risultato per i diversi livelli di una variabile di raggruppamento è possibile digitare

```
> by(X,X$eta,summary)
X$eta: bambino
       y              eta           z               x           sesso
 Min.   :160.2   bambino:50   Min.   :10.13   Min.   :0.03312   M:25
 1st Qu.:167.0   adulto : 0   1st Qu.:12.53   1st Qu.:0.18980   F:25
 Median :170.0   anziano: 0   Median :15.39   Median :0.39429
 Mean   :170.4                Mean   :15.18   Mean   :0.42505
 3rd Qu.:173.4                3rd Qu.:18.62   3rd Qu.:0.64815
 Max.   :182.3                Max.   :19.92   Max.   :0.99656
------------------------------------------------------------------------
X$eta: adulto
       y              eta           z               x           sesso
 Min.   :159.1   bambino: 0   Min.   :10.18   Min.   :0.01766   M:26
 1st Qu.:165.9   adulto :48   1st Qu.:13.19   1st Qu.:0.31909   F:22
```

```
 Median :168.8   anziano: 0   Median :16.35   Median :0.50721
 Mean   :169.1                Mean   :15.46   Mean   :0.53908
 3rd Qu.:171.9                3rd Qu.:17.82   3rd Qu.:0.81372
 Max.   :181.7                Max.   :19.98   Max.   :0.98355
 NA's   :  2.0
------------------------------------------------------------------------
X$eta: anziano
       y             eta           z               x           sesso
 Min.   :158.5   bambino: 0   Min.   :10.46   Min.   :0.05055   M:25
 1st Qu.:166.6   adulto : 0   1st Qu.:12.50   1st Qu.:0.21255   F:25
 Median :169.7   anziano:50   Median :15.39   Median :0.52601
 Mean   :169.5                Mean   :15.43   Mean   :0.52461
 3rd Qu.:173.4                3rd Qu.:18.34   3rd Qu.:0.74377
 Max.   :178.5                Max.   :19.88   Max.   :0.98326
 NA's   :  1.0
```

In sostanza è come se si fossero costruiti tre dataframe (ad esempio, attraverso `subset()`) e poi si fosse applicata `summary()` a ciascun sub-dataframe ottenuto.

`apply()` è invece disegnata per applicare una funzione alle righe o colonne di una matrice o in generale di un array. Comunque anche dataframe sono consentiti, ammesso che la trasformazione in matrice sia consentita. Ad esempio, per calcolare la deviazione standard di ciascuna variabile numerica di `X` si potrebbe utilizzare la seguente linea:

```
> apply(X=X[,c(1,3,4)],MARGIN=2,FUN=sd,na.rm=TRUE)
        y         z         x
4.7709551 3.0956847 0.2912510
```

ma se il dataframe avesse incluso una variabile categoriale allora la trasformazione del dataframe in matrice non sarebbe stata del tutto ammissibile e quindi si sarebbe ottenuti risultati 'inattesi'. Il secondo argomento `MARGIN` specifica se applicare la funzione alle colonne (`MARGIN=2`) o alle righe (`MARGIN=1`), ed in tale caso il risultato con `MARGIN=1` sarebbe stato un vettore di lunghezza pari a `nrow(X)`. Naturalmente se le colonne della matrice fossero variabili, l'operazione per righe potrebbe non aver senso! Il risultato dell'uso di `apply()` dipende anche dalla funzione che viene applicata. I seguenti codici possono chiarire quanto detto:

```
> length(apply(X[,c(1,3,4)],1,mean,na.rm=TRUE))
[1] 150
> apply(X[,c(1,3,4)],2,quantile,na.rm=TRUE)
            y        z          x
0%   158.5055 10.13424 0.01765792
25%  166.3703 12.62007 0.21971266
50%  169.3998 15.62737 0.48362839
75%  172.6295 18.20833 0.74579169
100% 182.3126 19.97941 0.99655941
```

Nel primo esempio vengono restituite 150 medie, perché ne viene calcolata una per ogni riga; nel secondo, poiché il risultato di `quantile()` è un vettore di

dimensione 5, il risultato è costituito da 5×3 valori perché `quantile()` viene applicata su tre variabili.

Quando si lavora con array allora l'argomento `MARGIN` può essere anche un vettore; utilizzando l'array `a` creato in precedenza con `table()`, si osservi in questo caso i risultati che si ottengono

```
> apply(a,1,sum)
bambino  adulto anziano
     50      48      50
> apply(a,3,sum)
 1  2
73 75
> apply(a,1:2,sum)
         sesso
eta        M  F
  bambino 25 25
  adulto  26 22
  anziano 25 25
```

cioè un altro modo per ottenere distribuzioni marginali.

Le funzioni equivalenti a `tapply()` e `apply()`, quando l'oggetto considerato è una lista, sono `lapply()` `sapply()` che consentono di applicare una stessa funzione a ciascun elemento di una lista. Come esempio si consideri la lista sotto costituita da un vettore numerico di interi e un vettore logico; i risultati di `lapply()` e `sapply()` sono:

```
> x<-list(numeri=9:1,logico=sample(c(TRUE,FALSE),10,replace=TRUE))
>
> lapply(x,range) #restituisce una lista
$numeri
[1] 1 9

$logico
[1] 0 1
>
> sapply(x,range) #restituisce una matrice
     numeri logico
[1,]      1      0
[2,]      9      1
>
> unlist(lapply(x,mean)) #un vettore
numeri logico
   5.0    0.2
```

Come è possibile notare, il risultato restituito da `lapply()` è ancora una lista della stessa lunghezza della lista iniziale, mentre `sapply()` semplifica (se possibile) tale procedimento restituendo un vettore o una matrice a seconda della funzione considerata. Si osservi la funzione `unlist()` che consente di trasformare

le liste in vettori. Comunque si noti che il risultato di `sapply()` può essere ancora una lista se i risultati relativi alle diverse componenti non sono 'omogenei'. Ad esempio `unique()` restituisce i soli elementi distinti del vettore dato come suo argomento; poiché i risultati di `unique(x[[1]])` e `unique(x[[2]])` sono due vettori di diversa dimensione, allora `sapply()` non potendo organizzare i risultati in modo compatto restituisce ancora una lista:

```
> sapply(x,unique) #uguale a lapply(x,unique)
$numeri
[1] 9 8 7 6 5 4 3 2 1

$logico
[1] FALSE  TRUE
```

4.3 Alcune rappresentazioni grafiche

Medie, varianze, correlazioni e tabelle di contingenza possono essere intese come misure di sintesi delle distribuzioni, ma in pratica le rappresentazioni grafiche possono fornire utili informazioni aggiuntive. Come è facile immaginare, esistono svariati modi 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 'standard', che probabilmente risultano essere le più utilizzate in un primo approccio all'analisi di dati; il termine 'standard' suggerisce che l'altra ampia categoria di funzioni grafiche 'non-standard' (quelle reperibili da CTAN, ad esempio per produrre grafici *trellis*) non verrà assolutamente menzionata. Ulteriori dettagli sono reperibili dai manuali specifici nella R *home-page*

La funzione `hist()` può essere utilizzata per rappresentare graficamente una distribuzione attraverso un istogramma: l'argomento `breaks` può essere impostato per suggerire il numero di categorie da disegnare. Così:

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

cercherà di disegnare un istogramma con 20 barre, oppure

```
> plot(x=X$y,y=X$z) #un diagramma di dispersione
```

disegnerà un diagramma di dispersione (*scatter-plot*) con i valori di `X$y` in ascissa.

Lanciando una qualsiasi funzione 'grafica' (ad esempio `hist()` o `plot()`) viene aperta una periferica (grafica) (*device*) su cui vengono indirizzati i risultati dei diversi comandi (grafici) eseguiti dalla linea di comando. In R, per default, ogni nuovo grafico creato cancellerà quello precedente andando ad occupare la stessa periferica attiva. Per creare un nuovo grafico evitando la cancellazione

di quelli già esistenti è necessario aprire una nuova periferica con `windows()` (o `x11()` o `X11()`) e quindi eseguire i normali comandi di grafica che produrrano i risultati sulla nuova periferica aperta. In questo modo gli altri grafici creati in precedenza continueranno ad essere visibili sulle altre periferiche che rimaranno aperte ma disattivate; la differenza tra periferica attiva e inattiva (condizione visibile sulle barre delle finestre di ciascun grafico con il termine 'active' o 'inactive') è che l'utente può operare soltanto sulle periferiche attive. Non sembra esserci limite al numero delle periferiche aperte, sebbene questo possa richiedere una notevole quantità di risorse in funzione anche della complessità dei grafici prodotti.

L'aspetto e le dimensioni di ogni grafico prodotto possono essere modificate semplicemente con il mouse trascinando i lati della relativa finestra per rendere, ad esempio, una figura più schiacciata o più allungata. Quando la figura risulta essere pronta solitamente l'utente ha interesse nel salvarla; a questo punto due alternative sono possibili.

La figura attiva può essere salvata in R come un oggetto, `o` diciamo, attraverso una semplice assegnazione `o<-recordPlot()`; naturalmente tale procedimento può essere applicato progressivamente ad ogni figura resa attiva dalla chiusura della periferica precedente. Ogni figura così salvata può essere nuovamente visualizzata con `replayPlot(o)` o semplicemente digitando il nome dell'oggetto, `o`, sulla linea di comando.

Per salvare il grafico in un file esterno è possibile utilizzare il menù a tendine: essendo evidenziata la figura in questione, selezionare "File|Save as.." e scegliere l'estensione di interesse. Si noti che questo procedimento riguarderà una qualsiasi periferica (attiva o inattiva) che risulta essere selezionata (ad esempio cliccandoci sopra con il mouse). Comunque se il grafico è complesso e 'pesante' e quindi richiede molta memoria (ad esempio se ci sono decine di migliaia di punti) è conveniente indirizzare l'*output* direttamente su un file esterno. Per fare questo, ciò che è richiesto è creare prima il file di destinazione, eseguire i diversi comandi per produrre il grafico e poi chiudere la periferica. Ad esempio per creare un file in postscript,

```
> postscript(file="c:/documenti/file.eps") #crea il file
> plot(..) #disegna ciò che vuoi
> dev.off() #chiudi
```

Invece che `postscript()` si sarebbe potuto utilizzare un'altra funzione di periferica, quale `pdf()` o `jpeg()` o altre, ma il procedimento sarebbe rimasto immutato; si veda `?Devices` per le altre possibili periferiche. Si noti che quando il file viene direttamente esportato in un file esterno, le funzioni di periferica (come `pdf()` o `postscript()`) hanno argomenti che possono essere opportunamente impostati per regolarne l'aspetto e le dimensioni (`width` e `height`), od anche il numero di pagine in cui andranno disegnati i grafici (argomento `onefile` in `pdf()` o `postscript()`). Una periferica particolare è `pictex()` che non produce un'immagine, ma un file di testo con i codici che possono essere inseriti in un file LaTeX (o anche TeX) con l'ausilio del pacchetto pictex. Comunque tale periferica è un po' limitata, ad esempio non consente l'uso di colori, e sarà esclusa dalla trattazione.

D'ora in avanti nelle illustrazioni dei codici per creare grafici non verrà fatta alcuna distinzione tra il tipo di periferica utilizzata, poiché per quanto trattato in questa sede non c'è alcuna differenza sul prodotto finale; l'uso di una periferica 'esterna', saltando la visualizzazione in R, può essere utile soprattutto per grafici molto complessi, per i quali la memoria del sistema potrebbe non essere sufficiente durante il processo di esportazione.

I grafici ottenuti con `hist()` e `plot()` sopra verranno prodotti con una serie di impostazioni che R ha per default, ovvero colori, nomi degli assi, grandezza dei punti. Prima di procedere nell'illustrare le altre funzioni, è importante elencare alcuni dei più importanti argomenti opzionali che possono essere impostati nel momento della creazione del grafico e che possono risultare molto utili nell'ottenere risultati 'personalizzati'. I più comuni parametri grafici sono:

- `xlab` e `ylab` che accettano singoli caratteri per nominare l'asse delle ascisse e quello delle ordinate;
- `xlim` e `ylim` per impostare i limiti dei due assi;
- `type` per decidere come disegnare i valori sul grafico: `type='p'` per punti (il default); `"l"` per linee; `"o"`,`"b"` e `"c"` per punti e linee con diverse modalità; `"h"` per barre verticali; `"s"` per funzioni a gradino (*step*); `"n"` per non disegnare nulla ma preparare solo l'area del grafico;
- `lty` per impostare il tipo di linea da disegnare (ad esempio `lty=1` continua, `lty=2` tratteggiata);
- `pch` per definire il tipo di punto o simbolo da disegnare;
- `col` per specificare i colori dei punti e/o linee da disegnare;
- `main` un carattere per aggiungere il titolo al grafico.

Non è il caso di dilungarsi su altri parametri grafici, anche perché sarebbe veramente un'impresa ardua e forse non molto utile. Adesso è opportuno soltanto sapere cosa è possibile fare e come 'muoversi', poi l'utente con il tempo e la pratica imparerà molti altri modi per personalizzare i propri grafici.

Così, alla luce di quanto detto sopra, alcune espressioni valide (di cui non si riportano i risultati) potrebbero essere le seguenti. Il lettore può eseguire i codici e osservare l'effetto dei diversi argomenti.

```
> hist(X$y,breaks=20,ylab="Frequenza",
+   xlab="Valori di Y",main="") #oppure main=NULL
> hist(X$y,freq=FALSE,ylab="Frequenza Relativa",xlab="Y",col=2)
> plot(X$y,X$z,pch="2")
> plot(X$y,X$z,pch=2,xlab="variabile Y")
> plot(X$y,X$z,pch=3,col=3,main="diagramma di dispersione",
+   ylim=c(0,20))
```

Quando un solo argomento è specificato in `plot()` (cioè `x` è mancante), allora tale funzione può essere utilizzata per disegnare una sequenza di valori

ordinati, ad esempio serie temporali; i comandi `plot(1:length(X$z), X$z)` e `plot(X$z)` sono perfettamente equivalenti. Per cui se i valori del vettore da disegnare, `X$z` per esempio, sono ordinati, allora può avere senso disegnare i punti unendoli con una linea, ovvero con `plot(X$z,type="l")`, oppure con altre opzioni quali `type="o"` o `"b"` o `"c"`.

La funzione `points()` è simile a `plot()`, ma disegna i suoi punti su un grafico che necessariamente deve essere stato già creato (per esempio da `plot()`). Aggiungere punti su un grafico già esistente può essere molto utile per evidenziare gruppi differenti, si veda l'esempio sotto. Comunque se devono essere sovrapposte linee piuttosto che semplici punti, un utilizzo di `points(..,type="l")` può essere sostituito da `lines()`.

```
> par(mfrow=c(1,2)) #dividi l'area in 1x2
> plot(X$y, X$z, xlab="valori di Y",ylab="valori di X",type="n")
> points(X$y[X$eta=="bambino"],X$z[X$eta=="bambino"],pch=19)
> points(X$y[X$eta=="adulto"],X$z[X$eta=="adulto"],col=2,pch=22)
> points(X$y[X$eta=="anziano"],X$z[X$eta=="anziano"],col=3,pch=24)
> #linee orizzontali:
> abline(v=mean(X$y,na.rm=TRUE),lty=2,col=4)
> abline(h=mean(X$z),lty=3,col=5)
> title(main="un diagramma di dispersione",cex.main=.8) #titolo
NULL
> ma<-apply(a,2:1,sum)
> barplot(height=ma,ylab="frequenza",beside=TRUE)
> title(main="un diagramma a barre",cex.main=.6)
NULL
> title(main="Due grafici",outer=TRUE,line=-1) #titolo generale
NULL
> dev.off() #chiudi (dopo aver salvato ;-))
null device
          1
```

I grafici in Figura 1 sono stati ottenuti con i codici riportati sopra; proviamo a commentarli.

`par()` è una funzione che viene utilizzata per impostare i diversi 'parametri' grafici; `par()` può essere associata a `options()`, nel senso che servono entrambi a regolare le opzioni di R, ma la differenza è che `par()` concerne specificatamente la produzione di grafici. Un suo impiego, ad esempio, è quello di dividere l'area per ottenere figure accostate; così `par(mfrow=c(l,k))` divide l'area del grafico in `l` righe e `k` colonne per un totale di `l`×`k` sub-aree su cui riportare altrettanti grafici. Per il primo grafico, dopo aver disegnato soltanto gli assi e le relative etichette ponendo `type="n"`, i punti per i diversi gruppi sono stati aggiunti con `points()` utilizzando colori differenti; infine una linea orizzontale ed una verticale vengono tracciate in corrispondenza delle medie. Il secondo grafico è un diagramma a barre e viene solitamente utilizzato per visualizzare i valori di una qualche quantità (ad esempio la frequenza) per diverse categorie. Quando il suo argomento principale (`height`) è un vettore allora il risultato sono delle barre

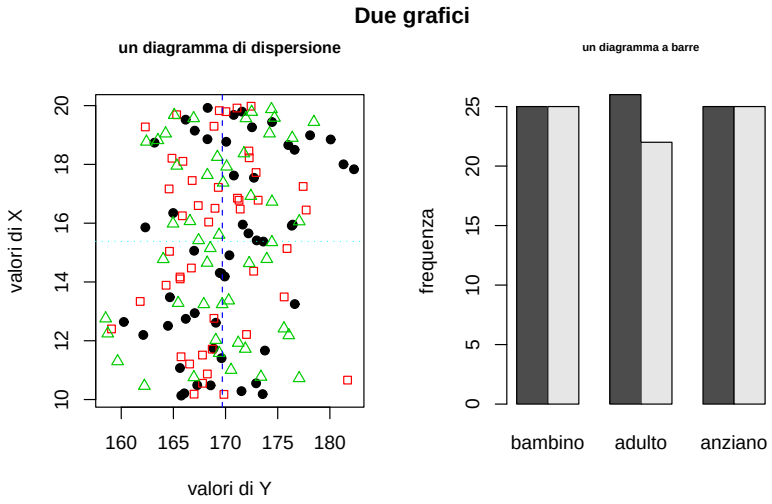

Figura 1: Alcuni grafici ottenuti con `hist()`, `plot()`, e `points()`.

con altezze date dai valori del vettore; quando è una matrice (come nell'esempio sopra) allora `barplot()` interpreta le righe come ulteriori categorie e riporta le barre per ogni combinazione di riga e colonna. Nell'esempio di sopra `ma` è una matrice dove le colonne sono le categorie di età, mentre le righe il sesso, per cui sono state disegnate due barre (quella dei maschi/femmine) per ciascuna delle tre categorie di età. Ponendo `beside=FALSE` le barre all'interno di ciascuna categoria vengono sovrapposte piuttosto che affiancate. La funzione `title()` è stata utilizzata per aggiungere titoli dei grafici: sia quelli specifici con font più piccoli attraverso il fattore di proporzionalità `cex.main` (default è `cex.main=1`) sia quello generale, relativo a tutti i grafici, ottenuto ponendo `outer=TRUE` e riducendo la distanza con i grafici attraverso valori negativi dell'argomento `line`; un esempio più avanti illustrerà ulteriormente il ruolo di `line` nell'aggiunta di testo ad un grafico.

L'utilizzo dell'argomento `mfrow` ha il limite di dover necessariamente dividere l'area in un numero di 'celle' (`l`×`k`) perfettamente allineate (per riga e colonna) e tale che ogni grafico occupi poi lo stesso spazio. La funzione `layout()` ha un utilizzo molto più ampio in quanto consente di specificare quanto spazio riservare alle diverse figure senza che queste siano necessariamente allineate. Si osservi come fare:

```
> m<-matrix(c(1,1,0,2,3,0),ncol=3) #una matrice
> m
     [,1] [,2] [,3]
[1,]    1    0    3
[2,]    1    2    0
> layout(mat=m,widths=c(1,1.5,.5),heights=c(1,.5))
> layout.show(max(m))#visualizza la divisione dell'area
> replicate(n=max(m),expr=plot(runif(10),type="l"))
[[1]] NULL
```

..

La matrice `m` è costituita da una serie di interi che rappresentano gli indici delle figure che devono essere disegnate; chiamando `layout()` con tale matrice come argomento si produce una divisione dell'area in celle 'indicizzate' tali che valori uguali individuano lo spazio riservato ad uno stesso grafico mentre celle con valori nulli individuano spazi vuoti. Gli spazi così creati sono riempiti da `max(m)` (in questo caso 3) grafici prodotti. Nell'esempio, piuttosto che ripetere tre volte la funzione `plot()` si è utilizzata `replicate()` che in generale eseguirà `n` volte una assegnata espressione `expr`. La Figura 2 illustra il risultato dei codici riportati sopra.

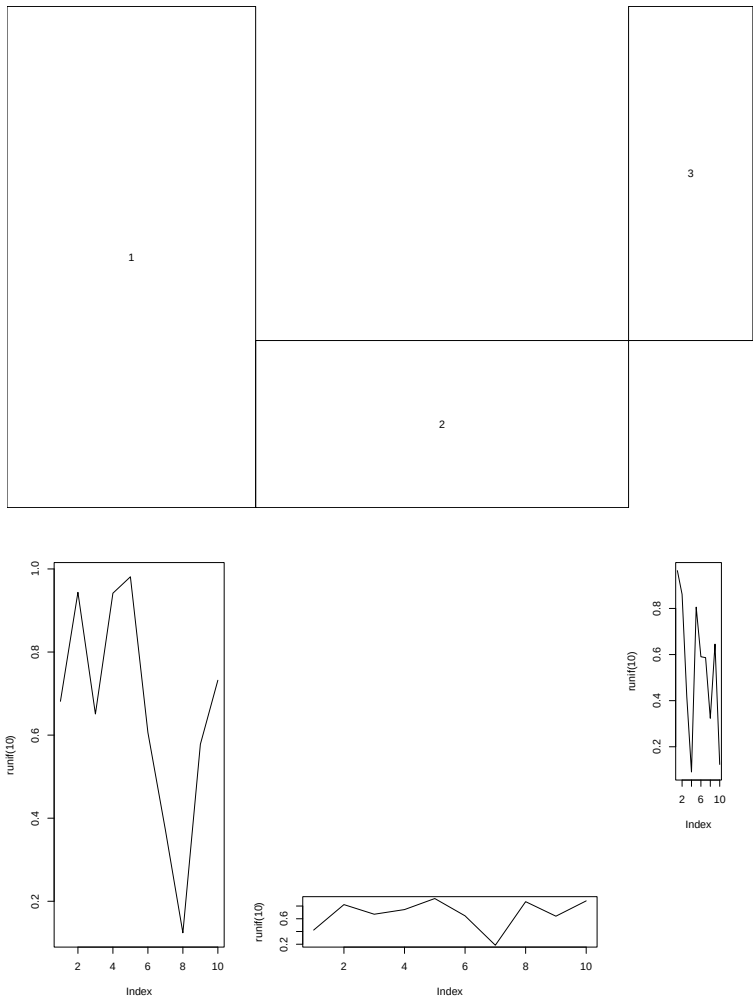

Figura 2: Divisione dell'area ottenuta con `layout()` e grafici.

Si noti la divisione dell'area che rispecchia la matrice `m` con dimensioni che invece dipendono dagli argomenti `widths` e `heights` di `layout()`. Tali argomenti definiscono fattori di espansione (default è il valore unitario per tutti) per aumentare/diminuire (con valori maggiori/minori di uno) la dimensione.

La Figura 2 evidenzia lo spazio che viene lasciato tra l'area riservata al

grafico e gli assi del grafico stesso. Tale spazio è usualmente necessario in quanto riservato ai nomi degli assi, le etichette, il titolo, ma spesso il nome può essere omesso oppure può essere utilizzato un font più piccolo; in ogni caso ed in qualche occasione tale spazio risulta essere eccessivo. La funzione `par()` può essere utilizzata per modificare i valori di `mai` (o `mar`) che regolano la distanza nei quattro lati della regione e che si differenziano soltanto per il modo in cui tale distanza può essere espressa; in `mai` la distanza viene specificata in pollici (*inch*, 1in=2,54cm), mentre con `mar` il numero di linee. In ogni caso dovrà essere considerato un vettore di dimensione quattro relativo al lato in basso, lato sinistro, lato in alto e lato destro.

Si consideri il seguente esempio, dove l'area viene divisa in quattro parti con margini differenti; i risultati sono riportati in Figura 3

```
> par()$mar #valori di default (in linee)
[1] 5.1 4.1 4.1 2.1
> par()$mai #valori di default (in pollici)
[1] 0.95625 0.76875 0.76875 0.39375
> par(mfrow=c(2,2))
> plot(1:10) #primo grafico
> par(mar=c(5.1,4.1,2.1,2.1)) #modifica i margini
> plot(1:10) #secondo grafico
> par(mar=c(5.1,3.1,4.1,2.1))
> plot(1:10)
> par(mar=c(2.1,2.1,1.1,1.5))
> plot(1:10,xlab="")
```

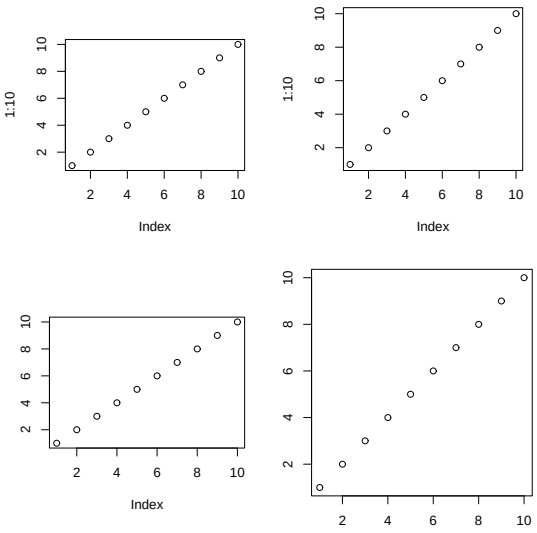

Figura 3: Divisione dell'area modificando le distanze dai margini.

Si noti come ridurre i margini possa portare ad un notevole guadagno di spazio sebbene sia necessario fare attenzione se si vogliono riportare i nomi e/o le etichette degli assi; ad esempio nel terzo e quarto grafico i valori ridotti per il

margine sinistro non hanno lasciato sufficiente spazio per il nome dell'asse delle ordinate.

Per comprendere meglio cosa rappresenti 'una linea', come specifica unità di misura in R, introduciamo altre funzioni che sono molto utili per personalizzare i grafici e dove l'argomento `line` riveste un ruolo fondamentale.

La Figura 4 è il risultato finale dei codici riportati sotto. Per semplicità non verranno discusse tutte le linee, ma verrà fatto soltanto qualche commento per illustrare i risultati sul grafico prodotto; per agevolare e migliorare la comprensione, l'utente potrebbe eseguirli di volta in volta per apprezzare l''effetto' dei diversi argomenti utilizzati.

Dopo aver creato un grafico senza etichette per gli assi ox e oy (attraverso `xaxt="n"` e `yaxt="n"`, rispettivamente) la funzione `axis()` viene ripetutamente chiamata per aggiungere etichette; si noti il ruolo di `side` per specificare a quale asse riferirsi (1=asse in basso, 2=asse a sinistra e così via) e `line` che regola la distanza dall'asse. `at` ed eventualmente `labels` servono a definire le posizioni (`at`) e le eventuali corrispondenti etichette (`labels`), con grandezza ed orientamento del testo rispetto all'asse dipendenti dagli argomenti `cex.axis` e `las` rispettivamente; infine `tick` è utile per voler marcare con una trattino tali valori sull'asse. La funzione `mtext()` è simile ad `axis()`, ma con un utilizzo più ampio: infatti oltre ai nomi degli assi, tale funzione può essere anche utilizzata per disegnare etichette in una specifica posizione ed anche per aggiungere un titolo così come è stato fatto con la funzione `title()` vista in precedenza.

Differente da `axis()` e `mtext()` è la funzione `text()` che consente di inserire del testo all'interno del grafico; a differenza delle prime due funzioni, quest'ultima richiede proprio le coordinate dove il testo deve essere ancorato, mentre nell'argomento `labels` deve essere passato il testo da visualizzare nel grafico. In un caso sono state utilizzate le prime due componenti di `letters`, (un oggetto di R contenente le lettere in minuscolo), negli altri un'unica etichetta dove il simbolo `\n` serve per incominciare una nuova riga; tale simbolo non è specifico a `text()`, ma può essere utilizzato con ogni funzione che richiede una stringa come argomento (ad esempio `xlab`). Si noti che per ottenere utili riferimento per la posizione del testo nel grafico, può risultare conveniente dare uno sguardo alla componente `usr` restituita da `par()` contenente gli estremi delle coordinate del grafico corrente. Infine l'argomento `pos` regola la posizione del testo; ad esempio `pos=3` e `pos=2` specificano il testo rispettivamente sopra e a sinistra delle coordinate di ancoraggio (vedi i codici con relativo risultato); default è al centro.

```
> plot(1:10,type="s",yaxt="n",xaxt="n",
+     xlab="linea di default per xlab",
+     ylab="linea di default per ylab") #frame.plot=FALSE)
> axis(side=1,at=c(2,3))
> axis(1,at=4,line=-.5)
> axis(1,at=5,line=-.5,tick=FALSE,las=2,cex.axis=.5)
> axis(1,at=10,line=.5)
> axis(1,at=c(6,8),labels=c("sei","otto"),las=1)
> axis(1,at=7,labels="sette",las=2,cex.axis=1.2)
> axis(2,at=9:10)
```

```
> axis(2,at=8,las=2)
> mtext(text="linea 0",side=2,line=0)
> mtext(text="linea 1",side=2,line=1,cex=1.2)
> mtext(text="linea 2",side=2,line=2,cex=.6)
> mtext(text=1:3,at=1:3,side=2,cex=.8)
> mtext(text=1:3,at=1:3,cex=.8) #default è side=3 (sopra)
> mtext(text="Palestina libera",side=4,line=-2,cex=1.8)
> par()$usr #le coordinate degli estremi..
[1]  0.64 10.36  0.64 10.36
> text(x=c(3.2,8),y=c(sqrt(25),3),labels=letters[1:2])
> text(3,8,pos=3,"great minds discuss ideas,
+     \n middle minds discuss events,
+     \n small minds discuss people.",cex=.7)
> text(3,8,"(Anonimo, Web)",pos=2,cex=.6)
> points(3,8,pch=3)
> legend(7,5.5,legend=c("Boycott","Israeli","Goods"),pch=1:3,
+     lty=1:3,col=1:3,cex=.7)
```

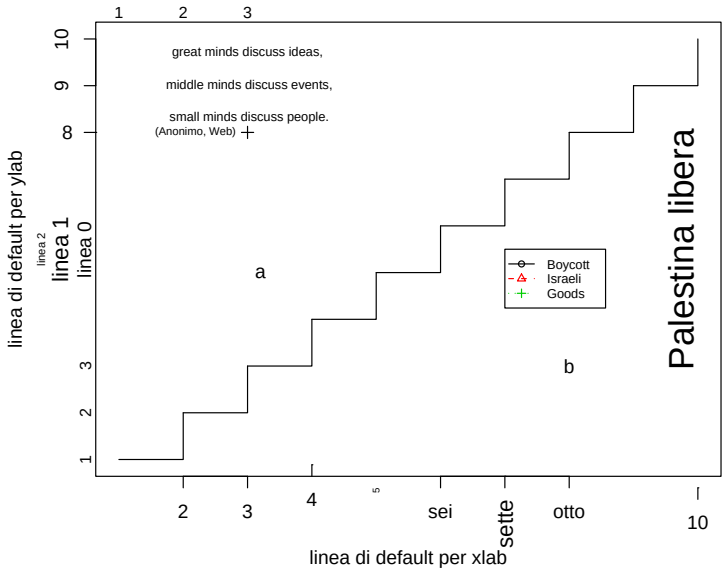

Figura 4: `axis()`, `mtext()` e `text()` in azione.

`legend()` è un'altra funzione particolarmente utile per aggiungere del testo all'interno delle rappresentazioni grafiche; tuttavia essa è specificatamente disegnata per inserire una legenda al grafico per cui, non sorprendentemente, le componenti corrispondenti dei diversi vettori suoi argomenti (`legend`, `pch`, `col`) vengono accostate nel momento della creazione del testo.

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, si veda `?plotmath` per dettagli; comunque gli utenti di LaTeX potrebbero trovare il pacchetto psfrag molto più produttivo.

5 Introduzione all'analisi dei dati

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, alcuni dei 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.

Nel prosieguo verrà dato qualche cenno all'analisi dei dati in R, illustrando l'analisi delle componenti principali e i modelli di regressione. Qualche basilare nozione di Statistica dell'argomento viene data anche se, coerentemente con quanto fatto fino ad ora, lo scopo è solo fornire un'ulteriore opportunità per evidenziare e discutere (alcune del)le potenzialità di R in questo contesto.

5.1 Analisi delle componenti principali (ACP)

Dato un insieme di K variabili statistiche numeriche ($\{X_k\}_{k=1,\ldots,K}$) rilevate su n unità, lo scopo della ACP è quello di ottenere $L \leq K$ nuove pseudo-variabili ($\{U_l\}_{l=1,\ldots,L}$) che siano fra loro non-correlate, tali che la 'varianza complessiva' (intesa come 'inerzia totale', ovvero somma delle varianze di ciascuna variabile) delle variabili originali X_k sia pari a quella delle nuove pseudo-variabili U_l.

R è già dotato di alcune funzioni 'pronte' per poter svolgere una ACP in qualche linea di comando, tuttavia di seguito verranno illustrate le istruzioni per effettuare l'analisi 'passo-per-passo' evitando l'utilizzo delle funzioni già esistenti che verranno accennate alla fine del paragrafo.

I dati di cui ci si servirà sono quelli del dataset `USArrests` che racchiude alcune informazioni sui crimini violenti registrati in ognuno dei 50 stati degli Stati Uniti d'America nel 1973. R è anche dotato di molti dataset, prevalentemente dati provenienti da studi e articoli già pubblicati e per lo più noti nella letteratura; scrivendo `data()` si apre una finestra con l'elenco (e relativa breve descrizione) dei dataset, mentre informazioni più dettagliate possono essere ottenute invocando il file di aiuto del dataset (ad esempio `?USArrest`); infine i dati possono essere caricati scrivendo il nome del dataset come argomento di `data()`, ammesso che il luogo dove tali dati sono salvati sia nel *search-path*.

```
> data(USArrests) #carica il dataset
> pairs(x=USArrests,panel=panel.smooth)
```

Dopo aver reso i dati disponibili nell'ambiente, la funzione `pairs()` viene utilizzata per ottenere una matrice di diagrammi di dispersione tra le coppie delle variabili del dataframe `x` passato come argomento: questo risulta un passo essenziale nell'analisi per verificare, almeno graficamente la non-indipendenza tra le variabili. Tra i diversi possibili argomenti, `panel=panel.smooth` consente di visualizzare una stima lisciata non-parametrica (*smoothed*) delle diverse relazioni.

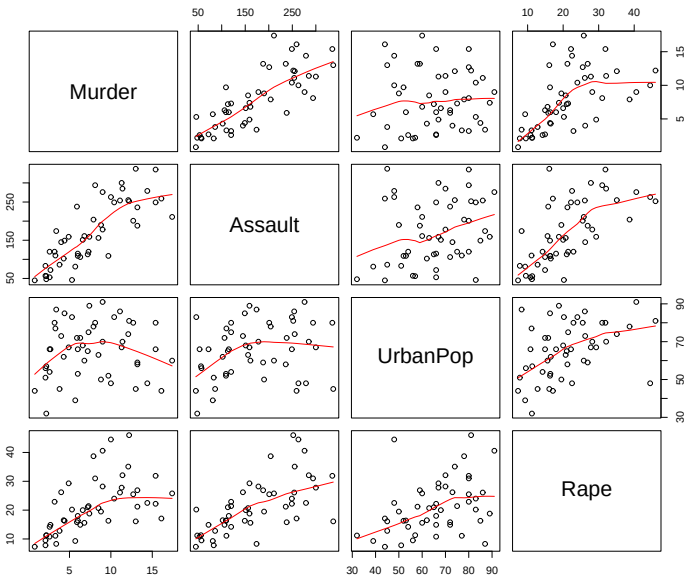

Figura 5: Matrice di diagrammi di dispersione per le variabili del dataset `USArrests`

L'analisi si sviluppa attraverso lo studio degli autovalori e autovettori della matrice di varianze e covarianze o, in alternativa, della matrice di correlazione. Va ricordato che poichè la ACP non gode della proprietà di invarianza per trasformazioni di scala, i risultati a cui si perviene dipendono dalla matrice utilizzata; invero la matrice di correlazione in questo caso risulta preferibile in quanto le unità di misura delle variabili sono molto diverse fra loro e conseguentemente l'ordine di grandezza delle varianze potrebbe non essere indicativo della reale variabilità.

Sia D la matrice dei dati originari e X la matrice dei dati scartati dalle proprie medie data da $X = D - M$, essendo M la matrice la cui k-esima colonna è la media della k-esima variabile originaria replicata n volte. La matrice di varianze e covarianze V altro non è quindi che $V = \frac{(D-M)'(D-M)}{n} = \frac{X'X}{n}$, infine la matrice di correlazione R può essere calcolata come matrice di varianze dei dati standardizzati Z. Per costruire tali matrici è sufficiente:

```
> Da<-data.matrix(USArrests)
> n<-nrow(Da) #n. osservazioni
> Un<-matrix(1,n,n)
```

```
> Me<-Un%*%Da/n
> dim(Me)
[1] 50  4
> fix(Me)
> Me[1:2,]
     Murder Assault UrbanPop   Rape
[1,]  7.788  170.76    65.54 21.232
[2,]  7.788  170.76    65.54 21.232
> colMeans(Me) #uno (stupido) controllo..
  Murder  Assault UrbanPop     Rape
   7.788  170.760   65.540   21.232
> X<-Da-Me #matrice degli scarti
> V<-crossprod(X)/(n-1) #matrice di varianze; anche var(USArrests)
> Sd<-diag(sqrt(diag(V)))
> Z<-X%*%solve(Sd) #matrice dei dati standardizzati
> ##matrice di correlazione; anche cor(USArrests)
> R<-crossprod(Z)/(n-1)
```

Così come `as.matrix()`, anche `data.matrix()` trasforma il suo argomento in una matrice, ma quest'ultima funzione ha la caratteristica di convertire le etichette delle eventuali variabili categoriali presenti nel dataframe in numeri interi. Ottenuta R, è possibile utilizzare la funzione `eigen()` che restituisce come risultato una lista di due elementi: gli autovalori e i corrispondenti autovettori.

```
> A<-eigen(R)
> A
$values
[1] 2.4802416 0.9897652 0.3565632 0.1734301

$vectors
           [,1]       [,2]       [,3]        [,4]
[1,] -0.5358995  0.4181809 -0.3412327  0.64922780
[2,] -0.5831836  0.1879856 -0.2681484 -0.74340748
[3,] -0.2781909 -0.8728062 -0.3780158  0.13387773
[4,] -0.5434321 -0.1673186  0.8177779  0.08902432
```

Gli autovettori sono normalizzati (norma unitaria), infatti ricordando la funzione `apply()` si ottiene

```
> apply(A[[2]],2,crossprod)
[1] 1 1 1 1
```

Le CP sono ottenute come combinazioni lineari delle X_k secondo coefficienti degli autovettori e risultano essere a media nulla, incorrelate e con varianze uguali ai corrispondenti autovalori:

```
> U<-Z%*%A[[2]] #calcola CP
> colMeans(U) #le medie
```

```
[1] -5.240253e-16  3.319567e-16  7.827072e-16  1.720846e-17
>
> round(var(U),digits=3) #matrice di var-cov
     [,1] [,2]  [,3]  [,4]
[1,] 2.48 0.00 0.000 0.000
[2,] 0.00 0.99 0.000 0.000
[3,] 0.00 0.00 0.357 0.000
[4,] 0.00 0.00 0.000 0.173
> sum(diag(var(U))) #inerzia totale, uguale a sum(diag(R))
[1] 4
```

Nei risultati ottenuti con `colMeans()` che calcola le medie di colonna (come `apply(U,2,mean)`, ma in modo più efficiente), si noti che i valori non sono esattamente zero, ma piuttosto 'zero-macchina'; in queste e simili circostanze, `round()` può essere utilizzata per arrotondare e visualizzare soltanto alcune cifre decimali.

Poiché la somma delle varianze delle componenti principali è pari alla somma delle varianze delle variabili iniziali (inerzia totale), per decidere il numero di componenti principali sufficienti a sintetizzare l'informazione contenuta nelle variabili iniziali, è possibile utilizzare come regola di decisione quella basata sul calcolo della percentuale cumulata di varianza, ottenibile attraverso la funzione `cumsum()`:

```
> cumsum((A$values)/sum(A$values))*100
[1]  62.00604  86.75017  95.66425 100.00000
```

e scegliendo un numero di componenti tali che la percentuale cumulata raggiunge una certa soglia prefissata. Nel caso in esame potrebbe essere sufficiente scegliere solamente le prime due componenti principali che spiegano il 87% circa della variabilità delle variabili iniziali. Per completezza, si osservi che con opportune assunzioni sulle X_k è possibile condurre test sul numero di CP da scegliere.

La rappresentazione grafica delle proiezioni delle osservazioni originarie nel piano individuato dalle prime due componenti principali può essere considerata una mappa a dimensione ridotta delle osservazioni multivariate; di fatto ogni punto K-dimensionale viene ridotto ed individuato da un vettore bidimensionale avente come coordinate i rispettivi valori delle due componenti. Un grafico 'ad-hoc' come quello in Figura 6 può essere ottenuto nel seguente modo:

```
> par()$pty #il default
[1] "m"
> par(pty="s") #per ottenere un grafico quadrato
> plot(U[,1:2],xlab="CP 1",ylab="CP 2",type="n")
> etich=abbreviate(row.names(USArrests),minlength=2)
> text(U[,1], U[,2],labels=etich)
> abline(h=0,v=0,lty=2,lwd=1.5)
NULL
```

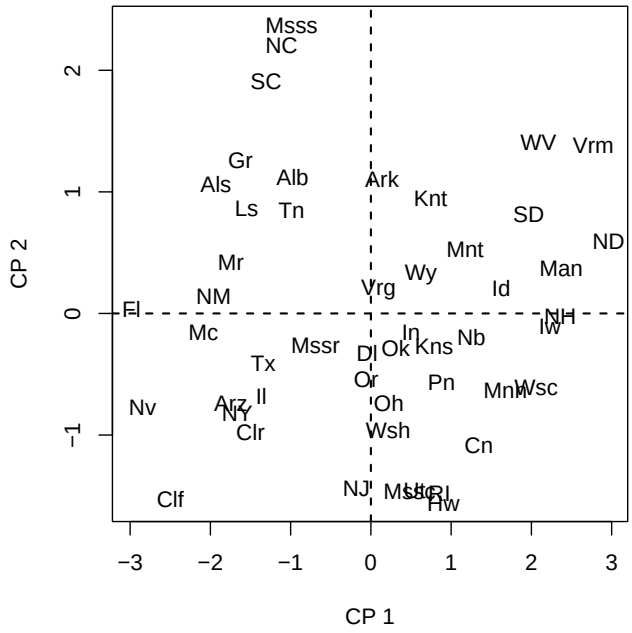

Figura 6: Diagramma di dispersione per le prime due componenti principali

Prima `par()` viene utilizzata per modificare il valore di `pty` ed ottenere un grafico con assi di uguale lunghezza. Poi il grafico viene creato ma i punti vengono visualizzati attraverso la funzione `text()` che consente di impostare le etichette (piuttosto che i simboli) da rappresentare. Queste etichette non sono i semplici nomi delle righe del dataframe (si noti `row.names()` che estrae i nomi di riga di un dataframe, mentre `rownames()` lavora per matrici), ma sono delle non-banali abbreviazioni ottenute da `abbreviate()` attraverso un opportuno algoritmo anche descritto nel relativo file di aiuto. L'uso di etichette facilita l'individuazione nel piano dei punti.

```
> etich[1:5]
   Alabama     Alaska    Arizona   Arkansas California
     "Alb"      "Als"      "Arz"      "Ark"      "Clf"
```

Il procedimento 'inverso', ovvero quello per risalire dalle abbreviazioni ai nomi completi, viene garantito dalla funzione `match()`. Ad esempio per visualizzare i nomi completi di `"Msss"`,`"SC"` e `"NC"` che nella Figura 6 risultano essere leggermente distaccate dagli altri

```
> match(x=c("Msss","SC","NC"),table=etich) #restituice gli indici
[1] 24 40 33
> etich[c(24,40,33)] #..a cui corrispondono
   Mississippi South Carolina North Carolina
        "Msss"           "SC"           "NC"
```

`match()` restituisce gli indici del vettore `table` corrispondenti a ciascuna delle componenti del vettore di stringhe `x`; dagli indici è poi facile risalire ai nomi.

Come è noto, dall'analisi dei coefficienti degli autovettori è possibile illustrare il peso che ciascuna variabile possiede all'interno di ogni componente principale. Nell'esempio qui riportato è possibile notare che le variabili 1 (Murder) 2 (Assault) e 4 (Rape) hanno un peso maggiore all'interno della prima componente principale mentre la variabile 3 (UrbanPop) è quella che ha il peso maggiore nella determinazione della seconda componente; ciò significa, come era lecito aspettarsi, che la prima componente principale racchiude "principalmente" l'informazione sulle tre variabili che riguardano i tipi di crimini violenti mentre la seconda componente racchiude l'informazione sulla dimensione della popolazione. La stessa conclusione può essere confermata dall'analisi delle correlazioni tra le variabili originarie X_k e i le CP, calcolate empiricamente con:

```
> cor(Da,U[,1:2])
               [,1]       [,2]
Murder   -0.8439764  0.4160354
Assault  -0.9184432  0.1870211
UrbanPop -0.4381168 -0.8683282
Rape     -0.8558394 -0.1664602
```

Come detto inizialmente, sono già presenti in R alcune funzioni che permettono di effettuare la ACP senza far ricorso alla procedura 'manuale' qui implementata; tra queste si possono citare le funzioni `prcomp()` e `princomp()`, il cui utilizzo è pressoché identico.

`prcomp()` può essere utilizzata specificando una formula del tipo `~x1+x2+..` in cui vengono esplicitate le variabili (x1, x2,..) da cui si vogliono estrarre le componenti principali; alternativamente le variabili originarie possono essere inserite attraverso un dataframe o una matrice. Ad esempio

```
> o<-prcomp(USArrests, scale.=TRUE)
```

dove `scale.=TRUE` specifica che i calcoli vanno effettuati sulla matrice di correlazione. I risultati sono salvati nell'oggetto `o`:

```
> o
Standard deviations:
[1] 1.5748783 0.9948694 0.5971291 0.4164494

Rotation:
                PC1        PC2        PC3         PC4
Murder   -0.5358995  0.4181809 -0.3412327  0.64922780
Assault  -0.5831836  0.1879856 -0.2681484 -0.74340748
UrbanPop -0.2781909 -0.8728062 -0.3780158  0.13387773
Rape     -0.5434321 -0.1673186  0.8177779  0.08902432
```

dove le 'standard deviations' sono le radici quadrate degli autovalori. Infine la funzione `screeplot()` (`screeplot(o)`) produce un banale e (forse migliorabile) grafico degli autovalori.

5.2 I modelli di dipendenza

I modelli di regressione (univariati) hanno lo scopo di studiare la relazione tra una variabile risposta Y e una o più variabili esplicative $X_1, X_2, \ldots, X_p$. Forse il più semplice modo di esprimere una dipendenza tra variabili è assumere una relazione lineare, per cui per l'osservazione $i = 1, 2, \ldots, n$ risulta:

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \ldots + \beta_p x_{pi} + \epsilon_i \tag{1}$$

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 attribuibile 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

$$\mu_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \ldots + \beta_p x_{pi}$$

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

5.3 Cenni ai Modelli Lineari Generalizzati

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 nostra 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.

5.3.1 Cenni teorici

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 coerenti ed inammissibili, ad esempio negativi. Conseguentemente il modello più generico deve avere come equazione

$$g(E[Y]) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p \tag{2}$$

dove $g(\cdot)$ è la funzione che 'lega' la parte sistematica $E[Y] = \mu$ della variabile risposta Y al *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 $\hat{\mu} = g^{-1}(\hat{\eta})$.

5.3.2 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

`glm(formula,family=`distribuzione`(link=`link`))`

Il primo argomento è la formula dell'equazione del modello, in cui variabile risposta e variabili esplicative sono separate dal simbolo `~` mentre le variabili esplicative sono tipicamente separate da un `+`, mentre `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.

Per illustrare l'impiego di `glm()`, utilizziamo il dataset di R `Titanic` che contiene i dati sui passeggeri del Titanic al momento dell'incidente, aggregati secondo le seguenti variabili categoriali: sopravvivenza all'incidente, classe (stato economico-sociale), sesso ed età; i dati sono salvati in forma di un array a quattro dimensioni ($4 \times 2 \times 2 \times 2$):

```
> data(Titanic)
> dim(Titanic)
[1] 4 2 2 2
> dimnames(Titanic)
$Class
```

```
[1] "1st"  "2nd"  "3rd"  "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"
```

Si è interessati a investigare quali siano stati i fattori principalmente responsabili della sopravvivenza o meno a seguito dell'incidente. La natura della variabile risposta suggerisce che un modello logit potrebbe essere appropriato, e a tale scopo è necessario ristrutturare i dati in tabella perché `glm()` possa accettarli come *input*. La sintassi di un modello logit non è unica e quindi per ragioni che saranno più chiare in seguito conviene convertire l'array in due modi: una tabella di contingenza 'ad hoc' con la variabile risposta in colonna ed una tabella espressa in ordine lessicografico.

```
> datiTit<-as.data.frame(Titanic) #ordine lessicografico
> datiTit[1:5,] #visualizza solo alcune righe
  Class    Sex   Age Survived Freq
1   1st   Male Child       No    0
2   2nd   Male Child       No    0
3   3rd   Male Child       No   35
4  Crew   Male Child       No    0
5   1st Female Child       No    0
> dim(datiTit)
[1] 32  5
>
> datiTit1<-ftable(Titanic,row.vars=1:3,col.vars=4) #tabella 'ad hoc'
>
> datiTit1
                  Survived  No Yes
Class Sex    Age
1st   Male   Child           0   5
             Adult         118  57
      Female Child           0   1
             Adult           4 140
2nd   Male   Child           0  11
             Adult         154  14
      Female Child           0  13
             Adult          13  80
3rd   Male   Child          35  13
             Adult         387  75
      Female Child          17  14
             Adult          89  76
Crew  Male   Child           0   0
             Adult         670 192
      Female Child           0   0
             Adult           3  20
```

```
>
> dim(datiTit1)
[1] 16  2
```

La tabella prodotta da `ftable()`, `datiTit1`, è probabilmente più leggibile in quanto consente di valutare prontamente le percentuali di sopravvivenza per ogni combinazione di modalità di covariabili. Tuttavia visualizzando tale tabella sullo schermo e confrontando le dimensioni della tabella stessa, si osserva che i risultati sono contraddittori riguardo al numero delle colonne (ne risultano due ma ne sono visualizzate cinque). Non è questa la sede per approfondire il perché di tale discordanza tra l'oggetto stesso ed il modo in cui viene visualizzato; soltanto si tenga presente che tali differenze sono dovute ad un particolare 'attributo' dell'oggetto che talvolta determina che ciò che viene riprodotto sullo schermo non è contenuto nell'oggetto stesso e quindi non può essere utilizzato prontamente. Questo implica che per usufruire di una tabella come quella visualizzata è necessario crearsi un dataframe appositamente. A tal fine potremmo pensare di utilizzare più volte `rep()`, ma un modo più semplice per ottenere tutte le possibili combinazioni delle categorie delle tre variabili è con la funzione `expand.grid()`:

```
> nomi<-dimnames(Titanic) #salva le etichette
> datiTit2<-expand.grid(age=nomi[[3]], sesso=nomi[[2]],
+   classe=nomi[[1]])
> datiTit2<-cbind(datiTit2,datiTit1[,1:2]) #aggiungi le freq
> dim(datiTit2) #OK
[1] 16  5
> names(datiTit2)
[1] "age"    "sesso"  "classe" "1"      "2"
> names(datiTit2)[4:5]<-nomi[[4]]
```

Dopo aver creato la matrice del disegno utilizzando i nomi e le etichette dovute, le due colonne di frequenze contenute in `datiTit1` sono state aggiunte: si noti che è necessario esplicitare le colonne per evitare di incorrere in problemi legati all''attributo' di `datiTit1`.

Organizzati i dati, uno stesso modello logistico quale

$$\text{logit}(Pr(Y = 1|x)) = \eta = \sum \beta_J X_J$$

può essere stimato in diversi modi: i) utilizzando i dati in `datiTit2` organizzati nella forma 'y_i/n_i', ovvero di distribuzioni binomiali condizionate (alle esplicative); ii)utilizzando le frequenze contenute in `datiTit`

```
> #usa le 'distribuzioni condizionate':
> o1<-glm(cbind(Yes,No)~age+sesso+classe+age:sesso,
+   family=binomial(link=logit),data=datiTit2)
> o2<-glm(Yes/I(Yes+No)~age+sesso+classe+age:sesso,weight=I(Yes+No),
+   family=binomial,data=datiTit2)
>
> #usa l'ordine lessicografico:
```

```
> o3<-glm(Survived~Age+Sex+Class+Age:Sex,weight=Freq,
+   family=binomial,data=datiTit)
```

dove la specificazione della funzione link può essere omessa essendo il *logit* il *link* canonico per la distribuzione binomiale.

Ogni chiamata a `glm()` prevede una formula, con a destra i termini del predittore lineare dove il simbolo ":" indica l'interazione tra due esplicative. Tuttavia il modo in cui viene scritta la variabile risposta è diverso: in `o1` la risposta è una matrice con le colonne di 'successi' e 'insuccessi', mentre in `o2` è una variabile di proporzione di successi y_i/n_i, con l'accortezza di specificare anche la variabile di peso (cioè la colonna di n_i) nell'argomento `weight`. Si noti la funzione `I()` che funge da protezione per l'oggetto che figura come suo argomento. Ad esempio, in una formula i segni "+" e ""- hanno un'opportuno significato: aggiungere o eliminare un termine dal predittore lineare; così se si vuole inserire un termine ottenuto come reale somma algebrica tra due variabili, `x1` e `x2` diciamo, allora è necessario 'proteggere' la scrittura `x1+x2` semplicemente inserendo `I(x1+x2)` ed evitare che R la interpreti come un predittore lineare con due variabili.

Il terzo modello `o3` utilizza le osservazioni disposte in ordine lessicografico. Come è noto questo comunque produce stime ed errori standard dei parametri ma non permette di fare conclusioni su qualche altra quantità talvolta di interesse; ad esempio sulla devianza residua (il valore ottenuto ed i rispettivi gradi di libertà in `o3` sono sbagliati).

Ogni modello ottenuto con `glm()` è una lista contenente diverse componenti, fra cui stime dei parametri, e relativa matrice di varianze, residui, valori attesi oltre ad alcune informazioni relative al processo di convergenza dell'algoritmo. Queste e altre quantità (si usi `names()` sull'oggetto) possono essere estratte con le opportune funzioni `coef()`, `vcov()`, `residuals()`, `fitted()` e `logLik()` che restituisce il valore della log-verosimiglianza. Ancora utilizzando la funzione `summary()` è possibile salvare ulteriori informazioni in un altro oggetto (qualcosa come `ogg<-summary(o1)`) oppure ottenere direttamente una sintesi sul modello stimato:

```
> ogg<-summary(o1) #salva informazioni in ogg
> summary(o1) #stampa direttamente una sintesi

Call:
glm(formula = cbind(Yes, No) ~ age + sesso + classe + age:sesso,
    family = binomial, data = datiTit2)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.7871  -1.3773   0.2436   2.5168   4.4755

Coefficients:
                    Estimate Std. Error z value Pr(>|z|)
(Intercept)           1.3565     0.3033   4.473 7.71e-06 ***
ageAdult             -1.7921     0.2829  -6.334 2.38e-10 ***
sessoFemale           0.7151     0.4062   1.760   0.0784 .
```

```
classe2nd             -1.0338     0.1998  -5.174 2.29e-07 ***
classe3rd             -1.8105     0.1759 -10.290  < 2e-16 ***
classeCrew            -0.8033     0.1598  -5.027 4.99e-07 ***
ageAdult:sessoFemale   1.9021     0.4331   4.392 1.12e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 671.962  on 13  degrees of freedom
Residual deviance:  94.548  on  7  degrees of freedom
AIC: 155.17

Number of Fisher Scoring iterations: 5
```

Naturalmente il risultato ottenuto con ‘`summary(o1)`’ è del tutto equivalente a quello che si sarebbe ottenuto digitando direttamente ‘`ogg`’. L’eventuale vantaggio nel salvare il sommario in un oggetto è che questo poi è sempre disponibile e le sue componenti estraibili ed utilizzabili, si veda `names(summary(ogg))`

Per ogni termine del predittore lineare, figurano le stime dei coefficienti beta, i loro errori standard ed i valori empirici (e relativi p-valori) della statistica di Wald che, sotto certe condizioni, può essere utilizzata per saggiare la significatività della stima. Si noti che i nomi dei termini che compaiono sono costituiti dal nome della variabile e dalla modalità a cui il parametro si riferisce. Per variabili categoriali, questo è essenziale per interpretare correttamente i parametri che, come è noto, dipendono dal modo in cui le stesse variabili sono state parametrizzate. I vincoli più comuni sono a somma nulla (tipo ‘anova’) oppure ‘corner-point’, che è il metodo usato in R per default: qui una categoria viene assunta come riferimento (*baseline*) ed i coefficienti delle altre categorie misurano la differenza da quella *baseline*: in particolare usando quest’ultima parametrizzazione i coefficienti in un modello logistico possono essere interpretati come log-*odds ratio* di ciascuna categoria rispetto a quella *baseline* (che non compare nell’*output*). Perciò poiché `'Child'`, `'Male'` e `'1st'` sono le categorie di riferimento delle tre variabili, una possibile lettura dei risultati potrebbe essere che la sopravvivenza a seguito dell’incidente dipende da tutte le variabili rilevate; in particolare il ‘rischio’ di sopravvivere per gli adulti è $\exp\{-1.792\} = 0.166$ volte quello dei bambini tra i maschi e $\exp\{-1.792+1.902\} = 1.116$ tra le donne. Per ottenere altri contrasti e differenti *odds ratio*, le categorie di riferimento possono essere cambiate con `relevel(x,ref)` dove `ref` individua la nuova categoria *baseline* di `x`.

Le altre parametrizzazioni per variabili categoriali possono essere conseguite utilizzando `C()` direttamente nella chiamata di `glm()`, come `C(class,"contr.sum")` oppure mediante `contrasts()` che viene utilizzata anche soltanto per visualizzare i contrasti.

```
> contrasts(datiTit2$classe)
     2nd 3rd Crew
1st    0   0    0
2nd    1   0    0
```

```
3rd    0   1    0
Crew   0   0    1
> contrasts(datiTit2$classe)<-"contr.poly" #modifica
> contrasts(datiTit2$classe) #ecco i nuovi
             .L   .Q         .C
[1,] -0.6708204  0.5 -0.2236068
[2,] -0.2236068 -0.5  0.6708204
[3,]  0.2236068 -0.5 -0.6708204
[4,]  0.6708204  0.5  0.2236068
```

Infine una modifica permanente riguardante tutte le variabili categoriali può essere effettuata cambiando la 'variabile' `contrasts` attraverso la funzione `options()`

```
> options("contrasts") #default (stesso che options()$contrasts)
$contrasts
        unordered           ordered
"contr.treatment"      "contr.poly"
> options(contrasts=c("contr.sum", "contr.poly")) #un cambiamento
```

dove un vettore con due componenti è richiesto in quanto i contrasti devono essere stabiliti per le variabili categoriali sia nominali che ordinali.

5.3.3 Ancora sui GLM: il ciclo while

Nel paragrafo precedente è stato illustrato l'uso della funzione `glm()` per la stima di un modello binomiale. È ben noto che la stima dei GLM può avvenire attraverso una stima iterativa di un classico modello gaussiano pesato con opportuna variabile risposta e pesi che dipendono dalla funzione link e dalla funzione varianza del modello di interesse. Quindi piuttosto che utilizzare `glm()` con uno specificato argomento `family`, avremmo potuto costruirci un'opportuna (pseudo) variabile risposta con rispettivi pesi e stimare un classico modello gaussiano 'iterativamente'.

Iterare un procedimento significa ripetere quel procedimento fino ad un certo punto e a tal fine possiamo utilizzare il ciclo *while* la cui sintassi è

`while(` *condizione* `) {` *operazioni* `}`

che significa "ripeti tutte le operazioni *operazioni* fino a quando la condizione *condizione* è verificata". Un uso molto elementare potrebbe essere il seguente:

```
> i<-1 #condizione iniziale
> while(i<3){
+     print(2)
+     i<-i+1
+     }
[1] 2
[1] 2
```

che significa 'stampa 2 fino a quando l'oggetto i è inferiore a 3'. A questo punto alcune osservazioni sono necessarie:

1. innanzitutto la scrittura avviene su più righe per agevolarne la comprensione, ma avremmo potuto utilizzare un'unica linea con i comandi separati da un `;`, cioè `while(i<3){print(2);i<-i+1}`;
2. È necessario definire prima l'oggetto (in questo caso `i`) che `while` deve valutare per poter decidere se procedere o fermarsi, altrimenti il ciclo non partirebbe;
3. Naturalmente tra le espressioni che `while()` deve eseguire è stata messa quella che consente al ciclo stesso di fermarsi: ovvero incrementare la variabile `i` finché la condizione `(i<3)` non si mantenga vera; appena *condizione* diventa falsa, in questo caso `i` non è più minore di 3, il ciclo si arresta. Perciò la semplice scrittura `while(i<3){print(2)}` stamperebbe sullo schermo il `2` indefinitivamente.

Piuttosto che stampare sullo schermo i risultati del ciclo, potrebbe essere utile salvarli in un opportuno oggetto da essere utilizzato in seguito. Un'idea potrebbe essere quella di creare tale oggetto, `x` diciamo, prima del ciclo e salvare i risultati di ogni iterazione 'incrementando' l'oggetto iniziale. Ad esempio se il risultato di ogni iterazione è un numero, allora l'oggetto `x` potrebbe essere un vettore con componenti corrispondenti ai risultati delle diverse iterazioni.

```
> i<-1 #condizione iniziale
> x<-0 #oggetto iniziale
> while(i<5){
+      x[length(x)+1]<-2+i^2
+      i<-i+1
+      }
> x
[1]  0  3  6 11 18
```

È facile verificare che, ad esempio, `x[4]` è proprio il risultato dell'espressione $2+i^2$ alla terza iterazione ($i = 3$). Si noti l'espressione `x[length(x)+1]` che consente di aggiungere una nuova componente ad `x` incrementando la sua dimensione e lasciando inalterate le componenti precedenti. Inoltre `x[1]` potrebbe essere cancellata in quanto essa è servita solo a definire l'oggetto iniziale; una alternativa è quella di creare `x` di dimensione 0 con `x<-vector(length=0)`. Potrebbe sembrare un po' strano creare un vettore di dimensione nulla, ma consente di ottenere alla fine del ciclo un vettore solo con componenti desiderate.

Ciò premesso è molto istruttivo impostare un ciclo *while* per la stima iterativa di un GLM, attraverso l'algoritmo noto come IWLS *(Iterative Weighted Least Square)*. Per illustrare tale procedura consideriamo un modello di Poisson per il dataset `Titanic` introdotto in precedenza. La teoria dice che il modello lineare pesato iterativo alla m-esima iterazione ha come matrice del disegno quella delle esplicative $\boldsymbol{X}$ e come variabile risposta e vettore dei pesi:

$$\boldsymbol{z}^{(m)} = \boldsymbol{X}\hat{\boldsymbol{\beta}}^{(m)} + (\boldsymbol{W}^{(m)})^{-1}(\boldsymbol{y} - \hat{\boldsymbol{\mu}}^{(m)}) \qquad \boldsymbol{w}^{(m)} = \boldsymbol{W}^{(m)}\mathbf{1}_n$$

in cui $\mathbf{1}_n$ è un vettore n-dimensionale di 1, $(\boldsymbol{W}^{(m)})^{-1}$ è l'inversa della matrice diagonale avente sulla diagonale principale il vettore dei valori attesi $\boldsymbol{\mu}^{(m)} = \exp\{\boldsymbol{X}\boldsymbol{\beta}^{(m)}\}$ al passo m. Quindi la stima dei parametri al passo successivo è data da:

$$\hat{\boldsymbol{\beta}}^{(m+1)} = (\boldsymbol{X'WX})^{-1}\boldsymbol{X'W}\boldsymbol{z}^{(m)}$$

che è la ben-nota formula per la stima dei parametri con il metodo dei minimi quadrati ponderati; da qui il nome "metodo dei Minimi Quadrati Ponderati Iterativi". Quando l'algoritmo converge, ovvero la differenza tra due consecutive iterazioni è piccola (in termini di log verosimiglianza oppure di stime dei parametri), allora $\hat{\boldsymbol{\beta}}^{(m+1)}$ costituisce la stima di massima verosimiglianza dei parametri $\boldsymbol{\beta}$ nel GLM iniziale.

Per il dataset Titanic organizzato nel dataframe `datiTit` definito in precedenza, si supponga di voler studiare la struttura di relazione fra le variabili Class, Sex, Age e Survived attraverso un modello log-lineare del tipo:

$$\log(m_{ijkl}) = \lambda_0 + \lambda_i^{Age} + \lambda_j^{Class} + \lambda_k^{Sex} + \lambda_l^{Surv} + \lambda_{il}^{Age*Surv}$$

con ovvio significato dei parametri. Allora evitando di usare `glm()` con argomento `family=poisson`, implementiamo il ciclo *while* descritto sopra:

```
> #la matrice del modello:
> X<-model.matrix(~Sex+Class+Age*Survived,data=datiTit)
> #la risposta:
> y<-datiTit$Freq
>
> beta.nuovo<-rep(1,ncol(X)) #valori iniziali di beta
> #log-verosimiglianza:
> ll.nuovo<-sum(dpois(x=y,lambda=as.vector(exp(X%*%bbeta)),
+   log=TRUE))
> iter<-0
> toll<-1
> while(abs(toll)>0.00001){
+          iter<-iter+1
+          eta<-X%*%beta.nuovo
+          mu<-as.vector(exp(eta))
+          W<-diag(mu)
+          z<-eta+solve(W)%*%(y-mu) #vettore di lavoro
+          a<-crossprod(X,W)
+          beta.nuovo<-solve(a%*%X)%*%(a%*%z)#stima aggiornata
+          eta.nuovo<-drop(X%*%beta.nuovo)
+          ll.vecchio<-ll.nuovo
+          ll.nuovo<-sum(dpois(y,drop(exp(eta.nuovo)),log=TRUE))
+          toll<-(ll.nuovo-ll.vecchio)/ll.vecchio
+          }
>
> iter #il numero delle iterazioni
[1] 8
```

```
> ll.nuovo #la logLik del modello
[1] -675.7524
```

Per avviare l'algoritmo sono state fornite 'stime' iniziali salvate nell'oggetto `beta.nuovo` in corrispondenza delle quali è stata calcolata la log verosimiglianza; qui si noti l'uso di `dpois()` con l'argomento `log=TRUE` che direttamente calcola il logaritmo della densità poissoniana. `drop()` e `as.vector()` vengono utilizzate per ottenere vettori da matrici con una dimensione pari ad uno. Durante le iterazioni si osservi come i valori delle stime e quello della log verosimiglianza vengano 'aggiornati', come anche il valore della tolleranza `toll` che rappresenta il cambio (relativo) nella funzione di verosimiglianza. Quando tale differenza diventa in valore assoluto (la funzione `abs()` è stata utilizzata per ignorare il segno di `toll`) $\leq$ .00001, allora il ciclo si ferma e le quantità di interesse sono prontamente disponibili: ad esempio le iterazioni necessarie (`iter`), la verosimiglianza (`ll.nuovo`), stime dei parametri (`beta.nuovo`) e relativa matrice di varianze-covarianze (`solve(a%*%X)`). Si confrontino i risultati (a meno di quantità trascurabili dovute ad aspetti computazionali) con quelli ottenuti con `glm()`:

```
> mod<-glm(Freq~Sex+Class+Age*Survived,data=datiTit,family=poisson)
```

Naturalmente la domanda sorge spontanea: "perché si dovrebbe scrivere un ciclo *while* se la funzione `glm()` è in grado di stimare qualsiasi GLM"? È ovvio che se l'interesse è in uno standard GLM, `glm()` è davvero adeguata; comunque se l'utente si trova ad affrontare un problema specifico (come l'introduzione di termini autoregressivi nel predittore), allora può essere necessario scrivere il proprio ciclo per ottenere il modello desiderato.

6 Le simulazioni in R: il ciclo for

Le simulazioni costituiscono uno strumento molto valido per valutare il comportamento di statistiche (intese come quantità funzioni di dati campionari) nell'universo campionario. Esse trovano largo impiego nello studio delle distribuzioni campionarie di stimatori e statistiche test quando la loro forma è ignota oppure quando si vogliono accertare, anche solo per curiosità, i risultati teorici.

Di seguito utilizzeremo le simulazioni per illustrare alcuni noti risultati teorici riguardanti la distribuzione campionaria della media aritmetica e dei coefficienti di regressione e di qualche statistica test in un modello di dipendenza. Le simulazioni hanno un risvolto anche dal punto di vista pratico in quanto spesso vengono utilizzate per fare inferenza su quantità per cui i risultati teorici non sono noti o non sono applicabili per una qualche 'particolare' configurazione dei dati (ad esempio ridotta numerosità campionaria o dati sparsi in tabelle di contingenza). La distribuzione della statistica di interesse a rigore dovrebbe essere valutata sull'intero universo campionario la cui completa enumerazione è quasi sempre difficile se non impossibile. Per questa ragione ciò che si fa di solito è esaminarla soltanto su un numero finito di campioni, usualmente 1000, ed ottenere in questo modo una stima, detta di 'Monte Carlo', di tale distribuzione.

Come è noto, la distribuzione campionaria dello stimatore media aritmetica $\bar{Y} = \sum Y_i/n$ per un campione di n v.c. Y_i è normale con opportune media μ e varianza σ^2/n; questo risultato è esatto se la distribuzione della variabile originaria è normale (cioè se $Y_i \sim \mathcal{N}(\mu, \sigma)$) e approssimato per altre situazioni in virtù del teorema centrale del limite. Si consideri il caso molto semplice in cui si vuole ottenere la distribuzione di campionamento di $\bar{Y}$ quando $Y_i \sim \mathcal{N}(0, 1)$ e $Y_i \sim \mathcal{U}(0, 1)$ per $n = 50$.

Per effettuare una simulazione è necessario generare un campione da una data densità, calcolarne la media e ripetere tali operazioni 1000 volte. Le 1000 medie così ottenute costituiscono una stima della distribuzione campionaria della statistica di interesse (in questo caso lo stimatore media aritmetica).

In generale per ripetere un insieme di operazioni, è possibile utilizzare un *ciclo for* la cui sintassi in R è

`for(`*indice* `in` *valoriIndice*`) {` *operazioni* `}`

che significa "esegui le operazioni *operazioni* per i diversi valori di *indice* compresi nel vettore *valoriIndice*". Nel suo utilizzo elementare si potrebbe scrivere

```
> for(j in 1:3){
+     print(j)
+     }
[1] 1
[1] 2
[1] 3
```

Si noti che non è necessario definire prima la variabile `j` (come è per *while*) che se dovesse esistere verrebbe sovrascritta.

Ciò premesso, in R possiamo procedere nel seguente modo avendo cura di salvare opportunamente i risultati ottenuti in ogni passo:

```
> B<-1000 #numero di repliche
> n<-30 #dimensione del campione
> #n<-80
> medie<-vector(length=B) # predisponi vettore dove salvare i risultati
> for(i in 1:B){
+         x<-rnorm(n) #genera n valori da una N(0,1)
+         #x<-runif(n,0,1)
+         medie[i]<-mean(x) #calcolo la media del campione i-esimo
+         }
> medie.N.30<-medie
> mean(medie.N.30) #confronta con 0
[1] -0.001444174
> var(medie.N.30) #confronta con 1/n
[1] 0.03406269
```

I valori ottenuti per media e varianza confermano (per fortuna, `;-)`) i risultati teorici. Aumentando `B` e modificando altre variabili (quali `n` e la densità di

provenienza delle osservazioni) si possono ottenere altri risultati. La Figura 6 è stata ottenuta con i codici sotto e sintetizza le distribuzioni campionarie per tre differenti scenari. `medie.U.30` e `medie.U.80` contengono i valori per le simulazioni relative ad osservazioni provenienti da una variabile uniforme per $n = 30$ e $n = 80$ rispettivamente.

```
> o30<-hist(medie.U.30,breaks=30,freq=FALSE,plot=FALSE)
> o80<-hist(medie.U.80,breaks=30,freq=FALSE,plot=FALSE)
>
> par(mfrow=c(1,2))
> hist(medie.N.30,breaks=30,freq=FALSE,xlab="",ylab="Densità",
+   main="")
> lines(seq(-.6,.6,length=500),dnorm(seq(-.6,.6,length=500),
+   mean=0,sd=sqrt(1/30)),col=2,lwd=2)
> #secondo grafico:
> plot(o30$mids,o30$density,xlim=c(.3,.7),ylim=c(0,14),type="s",
+   lwd=1.5,xlab="",ylab="Densità",frame.plot=FALSE)
> points(o80$mids,o80$density,lty=2,lwd=1.8,type="s")
```

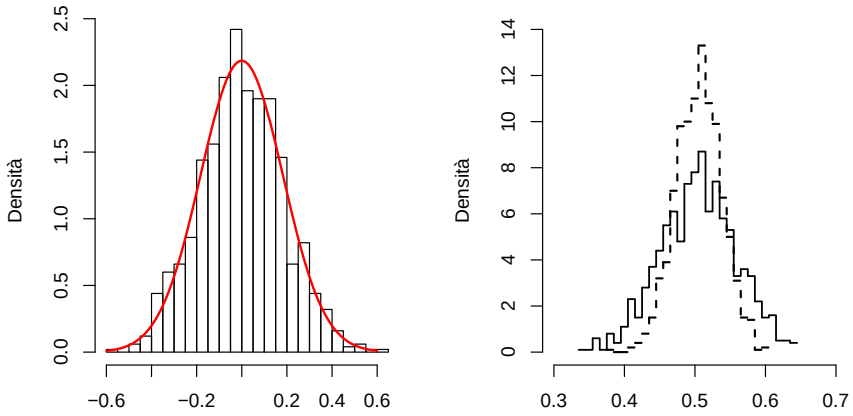

Figura 7: Rappresentazione grafica dei risultati delle simulazioni.

`hist()` è stata utilizzata per ottenere un istogramma della distribuzione di campionamento e successivamente `lines()` è stata impiegata per sovrapporre i valori della curva normale teorica; a tale riguardo si noti che le ordinate sono state calcolate con `dnorm()`. La rappresentazione grafica per le distribuzioni campionarie di $\bar{Y}$ nel caso di $Y_i \sim \mathcal{U}(0, 1)$ è stata leggermente personalizzata: per ottenere grafici sovrapponibili si è deciso di disegnare soltanto i bordi dell'istogramma. Per cui `hist()` è stata invocata con l'argomento `plot=FALSE` (nessun grafico prodotto) e dagli oggetti risultanti (`o30` e `o80`) sono state estratte le quantità (ascisse ed ordinate) utili per ottenere gli istogrammi, disegnati

successivamente attraverso le funzioni `plot()` e `points()`. Tra gli argomenti utilizzati si notino `type="s"`, già vista in precedenza e `frame.plot` che ha impedito che venisse disegnato il contorno del grafico.
Da un punto di vista sostantivo, gli istogrammi sovrapposti evidenziano come la distribuzione campionaria per $n = 80$ risulti meno dispersa ed apparentemente più vicina alla 'normalità' di quella ottenuta per $n = 30$.

Come secondo esempio, impieghiamo le simulazioni per valutare il livello di significatività empirico del test di *Wald* per saggiare la significatività delle stime dei coefficienti di regressione in un modello lineare semplice:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \qquad \epsilon \sim \mathcal{N}(0, \sigma) \tag{3}$$

Allora per sviluppare la simulazione occorre:

1. fissare un livello di significatività α che definisce l'ampiezza nominale del test;

2. generare un numero finito di volte n osservazioni attraverso il modello (3) sotto l'ipotesi nulla (qui poniamo $H_0 : \beta_1 = 0$) ed estrarre ogni volta il valore della statistica di Wald;

3. Sul totale delle repliche effettuate, il livello reale di significatività del test è dato dalla proporzione dei campioni per i quali H_0 non viene accettata, ovvero per i quali il valore empirico della statistica di Wald supera il valore critico dipendente da α fissato al punto 1.

Sviluppiamo i passi di cui sopra in codici R assumendo $\alpha = 0.05$ e $(\beta_0, \beta_1)' = (1.5, 0)'$:

```
> B<-1000 #numero dei campioni
> n<-80 #dimensione campionaria
> x<-runif(n) #variabile esplicativa
> A<-matrix(,B,8) #predisponi la matrice per salvare i risultati
> colnames(A)<-c("interc","beta","se.interc","se.beta",
+ "t.interc","t.beta","p1","p2")
> for(i in 1:B){
+     lp<-1.5+0*x
+     y<-rnorm(n,mean=lp,sd=.8) #genera i dati
+     o<-lm(y~x) #stima il modello
+     A[i,]=as.vector(summary(o)$coef[,1:4]) #estrai i risultati
+ }
```

Come si può notare per ogni replica vengono salvati anche altre quantità a cui inizialmente non eravamo interessati. Salvare anche grandezze non di reale, ma di probabile, interesse è solitamente buona norma; infatti quando si eseguono simulazioni (naturalmente non tanto semplici come quelle presentate qui) il lavoro oneroso per R è la stima del modello e forse anche la generazione dei dati in qualche caso particolare. Una volta che questi 'sforzi' sono stati compiuti, salvare una matrice 1000×3 o 1000×15, diciamo, non fa molta differenza per R

soprattutto se questa poi viene direttamente esportata in un file esterno. Infatti il problema alla mano potrebbe essere tale da suscitare *ex-post* qualche idea o curiosità e sollevare dubbi che potrebbero essere risolti soltanto guardando i risultati delle altre grandezze campionarie. Se questi non sono disponibili l'utente dovrebbe fare girare nuove simulazioni con conseguente spreco di risorse.

Per l'esempio discusso in questa sede è di rilievo guardare ai risultanti concernenti il parametro β_1:

```
> #valore atteso dello stimatore di beta1 (confronta con 0)
> mean(A[,2])
[1] 0.01490738
#ampiezza del test (confronta con 0.05)
> sum(abs(A[,"t.beta"])>1.96)/B
[1] 0.056
```

dove `A[,"t.beta"]` contiene i valori empirici del test di Wald per le diverse repliche che vengono ogni volta confrontati con il valore soglia di 1.96; `abs()` serve a considerare test a due code ed il confronto con 1.96 produce un valore logico (`TRUE` o `FALSE`) a seconda se nel campione H_0 è stata accettata o meno; infine sommando le componenti di tale vettore si ottiene il numero dei `TRUE`, ovvero l'ampiezza del test.

Avendo salvato extra-informazioni è possibile stimare il livello di copertura empirico dell'intervallo di confidenza per β_1; a tale fine è sufficiente seguire i seguenti passi:

```
> inf<-(A[,2]-1.96*A[,4])#gli estremi inferiori dell'interv.conf.
> sup<-(A[,2]+1.96*A[,4])#gli estremi superiori dell'interv.conf.
> #la proporzione di intervalli contenenti lo 0 (vero parametro):
> sum(ifelse(0>=inf&0<=sup,1,0))/B
[1] 0.944
```

Il risultato è, non sorprendentemente, lo 0.95. In quest'ultima parte di codici è interessante osservare la funzione `ifelse()` la cui sintassi è

`ifelse(`*condizione*`,` *valoreA*`,` *valoreB* `)`

che significa "se *condizione* è vera allora restituisci *valoreA* altrimenti *valoreB*". La funzione lavora, naturalmente, con scalari

```
> ifelse(5>2,"sicuro","insicuro")
[1] "sicuro"
>
> ifelse(5<2,"leggero","pesante")
[1] "pesante"
```

ma può manipolare vettori in modo efficiente:

```
> a<-1:10
> ifelse(a>2&a<5,0,1)
 [1] 1 1 0 0 1 1 1 1 1 1
```

dove il simbolo `&`, già visto in precedenza, è stato utilizzato per definire una condizione come *intersezione* di due altre condizioni; questo è quello che si è fatto sopra per calcolare il livello di copertura degli intervalli di confidenza, dove vengono contate quante volte il vero valore del parametro è simultaneamente maggiore e minore degli estremi. D'altro canto il simbolo `|` consente di specificare l'*unione* di condizioni:

```
> ifelse(a<2|a>=8,0,1)
 [1] 0 1 1 1 1 1 1 0 0 0
```

La non-distorsione del test può anche essere apprezzata attraverso il p-valore che ha una sua distribuzione campionaria alla stregua della statistica test; se il test è non distorto, sotto H_0 deve risultare che p-valore $\sim \mathcal{U}(0,1)$ e quindi per ogni $\alpha \in]0,1[$ deve verificarsi $Pr(p\text{-valore} \leq \alpha) = \alpha$.

Per i risultati ottenuti, poiché `A[,"p2"]` individua la distribuzione del p-valore, ulteriori verifiche possono essere realizzate attraverso:

```
> sum(A[,"p2"]<=0.05)/B
[1] 0.05
> #grafico:
> hist(A[,"p2"],breaks=30,freq=FALSE,main="Istogramma",
+   xlab="p-valore",ylab="Densita'",col=grey(.8))
> rect(0,0,1,1,border=2,lwd=1.5)
```

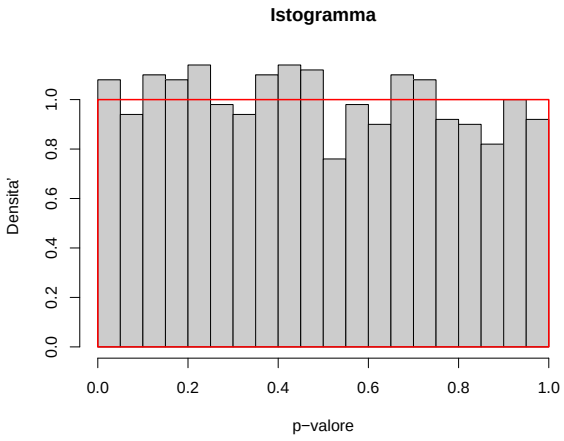

Figura 8: Istogramma p-valore

Nei codici sopra la funzione `grey()` è stata inserita come argomento di `col` per ottenere una gradazione di grigio (`grey(1)` è bianco) e `rect()` per aggiungere (la periferica deve essere già aperta) la distribuzione teorica di riferimento, ovvero un rettangolo con specificati vertici; l'argomento `border` determina i colori (del bordo) di tale rettangolo.

Seguendo le linee guida di sopra, può essere istruttivo valutare il comportamento del test esatto di Fisher usualmente impiegato per tabelle di contingenza

con dati sparsi e/o piccoli campioni. Naturalmente in questo caso si ha a disposizione soltanto il p-valore:

```
> B=1000
> pval<-vector(length=B)
> for(i in 1:1000){
+     x<-rmultinom(n=1,size=30,prob=c(.125,.125,.375,.375))
+     tabella<-matrix(x,2,2,byrow=TRUE)
+     pval[i]=fisher.test(tabella)$p.value
+     }
>
> for(k in c(.01,.05,.10)) print(sum(pval<=k)/B)
[1] 0.004
[1] 0.031
[1] 0.066
```

La funzione `rmultinom()` viene utilizzata per creare i conteggi multinomiali per un totale fissato di $n = 30$; poiché viene fissato soltanto il totale e attraverso l'argomento `prob` di `rmultinom()` vengono specificate le probabilità congiunte, tale generazione può associarsi ad un campionamento multinomiale; il lettore può provare a simulare dati secondo gli altri piani di campionamento, poissoniano e 'prodotto di multinomiali'. I conteggi generati vengono poi organizzati in una matrice su cui viene applicata la funzione `fisher.test()`. Il risultato di tale funzione è una lista in cui la componente `p.value` contiene il risultato del test. Infine un piccolo ciclo *for* viene impiegato per visualizzare i percentili di tre specificati valori.

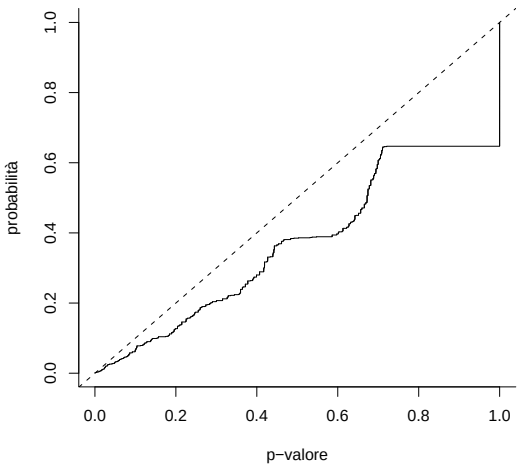

Figura 9: Funzione di ripartizione del p-valore ottenuto con il test condizionato di Fisher

Piuttosto che disegnare un istogramma con la densità, la Figura 9 rap-

presenta la funzione di ripartizione (f.r.) del p-valore che mette in risalto le caratteristiche del test:

```
> plot(sort(pval),(1:length(pval))/length(pval),bty="l",type="s",
+   xlab="p-valore",ylab="probabilità")
> abline(0,1,lty=2) #la f.r. teorica di riferimento
```

All'interno di `plot()` si noti l'argomento `type="s"`, già visto in precedenza, e quello `bty="l"` che permette di disegnare soltanto due assi che si intersecano all'origine.

I risultati delle simulazioni consolidano, se mai ce ne fosse stato bisogno, le proprietà teoriche del test di Wald e di Fisher per i casi considerati.

Indice analitico