Introduzione

Questa parte del corso `Statistica Computazionale` riguarda l'uso del linguaggio statistico R. Considereremo varie possibilità di R per applicazioni a problemi statistici. Inoltre, introdurremo aspetti di statistica il cui studio può essere realizzato usando tecniche computazionali moderne.

Il pacchetto R è uno sviluppo dal linguaggio S. Esiste uno sviluppo simile che si chiama S-Plus. Però, mentre S-Plus è un prodotto commerciale (e costoso) R è gratis. Per la maggiore parte delle applicazioni, R funziona in un modo simile a S-Plus, e per alcune applicazioni il suo funzionamento è migliore.

Si può ottenere una copia del programma R da internet all'indirizzo:

`http://cran.r-project.org`

Sono disponibili allo stesso indirizzo vari documenti e guide all'uso di R.

Il sistema di `help` compreso in R è molto esteso e utile. Inoltre, ci sono molti libri che danno introduzioni a R. Questi includono Venables e Ripley, e Bortot, Ventura e Salvan.

1 Concetti di Base

1.1 Descrizione

In questo capitolo consideriamo le varie possibilità in R per l'esplorazione o la sintesi di dati. Useremo esempi che sono già inclusi in R. Le cose da notare sono:

1. Ogni oggetto in R ha una `class` (tipo). Molte funzioni (ad esempio `plot`) si comportano in modi diversi che dipendono dalla `class` dell'oggetto al quale sono applicate.

2. Si usi la funzione `help` per una descrizione di qualsiasi oggetto.

1. **Analisi del vettore** `precip` (si usi `help(precip)` per una spiegazione).

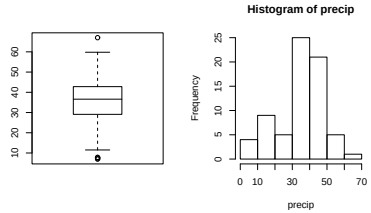

Figure 1:

```
# carica i dati

> data(precip)

# controllare il class

> class(precip)
NULL
> data.class(precip)
[1] "numeric"


# si vedono i dati

> precip
```

Mobile	Juneau	Phoenix	Little Rock
67.0	54.7	7.0	48.5

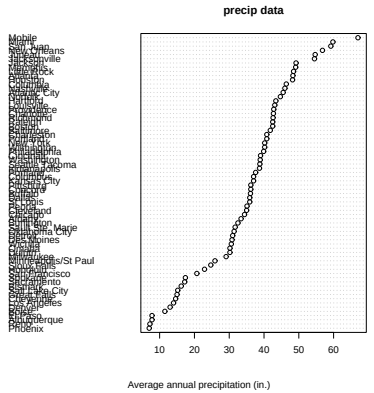

Figure 2:

```
       Los Angeles          Sacramento       San Francisco              Denver

              14.0                17.2               20.7                13.0
```

```
...

# statistiche empiriche

> mean(precip)
[1] 34.88571
> var(precip)
[1] 187.8723
> sqrt(var(precip))
[1] 13.70665
> summary(precip)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   7.00   29.38   36.60   34.89   42.78   67.00

#grafici

> par(mfrow=c(1,2),pty='s')

> boxplot(precip)
> hist(precip)

> par(mfrow=c(1,1),pty='m')

> dotchart(precip[order(precip)], main = "precip data")
> title(sub = "Average annual precipitation (in.)")
```


2. **Analisi della serie storica** `airmiles`.

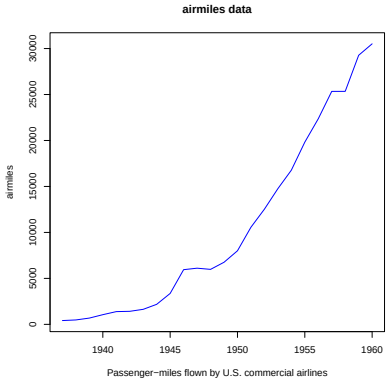

Figure 3:

```
> data(airmiles)

> class(airmiles)
[1] "ts"

> airmiles
Time Series:
Start = 1937
End = 1960
Frequency = 1
 [1]   412   480   683  1052  1385  1418  1634  2178  3362  5948  6109  5981
[13]  6753  8003 10566 12528 14760 16769 19819 22362 25340 25343 29269 30514

> summary(airmiles)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    412    1580    6431   10530   17530   30510

> plot(airmiles, main = "airmiles data",
         xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

3. **Analisi del data.frame** `quakes`.

```
> data(quakes)

> class(quakes)
[1] "data.frame"
```

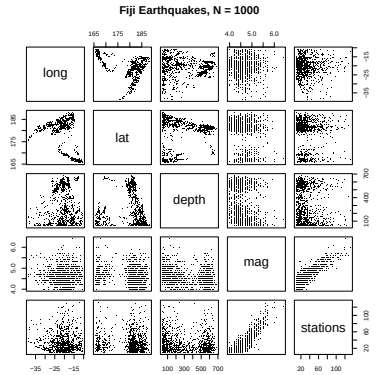

Figure 4:

```
> quakes
      long    lat depth mag stations
1   -20.42 181.62   562 4.8       41
2   -20.62 181.03   650 4.2       15
3   -26.00 184.10    42 5.4       43
4   -17.97 181.66   626 4.1       19
.....

> summary(quakes)
      long             lat            depth            mag
 Min.   :-38.59   Min.   :165.7   Min.   : 40.0   Min.   :4.00
 1st Qu.:-23.47   1st Qu.:179.6   1st Qu.: 99.0   1st Qu.:4.30
 Median :-20.30   Median :181.4   Median :247.0   Median :4.60
 Mean   :-20.64   Mean   :179.5   Mean   :311.4   Mean   :4.62
 3rd Qu.:-17.64   3rd Qu.:183.2   3rd Qu.:543.0   3rd Qu.:4.90
 Max.   :-10.72   Max.   :188.1   Max.   :680.0   Max.   :6.40
    stations
 Min.   : 10.00
 1st Qu.: 18.00
 Median : 27.00
 Mean   : 33.42
 3rd Qu.: 42.00
 Max.   :132.00

> pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main=1.2, pch=".")

# nota che plot(quakes) da' un  grafico simile a pairs
# riconosce la class di quakes come una matrice
```


Un'altra funzione molto utile è `apply`. Questa permette l'esecuzione di una funzione su tutte le righe o colonne di una matrice. Per esempio, per calcolare la media di ogni colonna:

```
> apply(quakes,2,mean)
     long       lat     depth       mag  stations
-20.64275 179.46202 311.37100   4.62040  33.41800
```

Il due significa la seconda dimensione della matrice, cioè, le colonne. Se, invece, avessimo voluto le medie delle righe (sebbene non abbia molto senso in questo esempio), sostituiremmo 2 con 1.

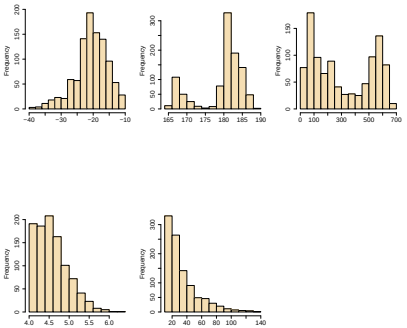

Figure 5:

Qualsiasi funzione può venire usata invece di `mean`. Per esempio, per produrre un istogramma di ciascuna colonna:

```
> par(mfrow=c(2,3))
> apply(quakes,2,hist,main='',xlab='',col='wheat')
```

Si noti in questo caso la possibilità di passare altri argomenti alla funzione `hist` dentro la funzione `apply`.

4. **Analisi della tabella** `HairEyeColor`

```
> data(HairEyeColor)

> class(HairEyeColor)
[1] "table"

> HairEyeColor
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
```

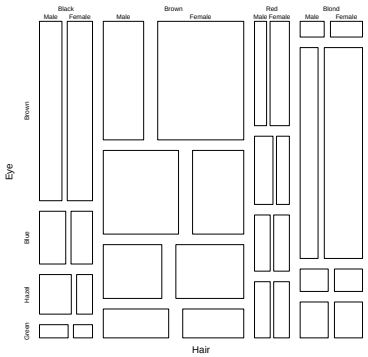

Figure 6:

```
  Black    32   11    10     3
  Brown    38   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    81   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8


> summary(HairEyeColor)
Number of cases in table: 592
Number of factors: 3
Test for independence of all factors:
        Chisq = 171.81, df = 9, p-value = 2.584e-32
        Chi-squared approximation may be incorrect



> plot(HairEyeColor)
```

5. **Analisi del data.frame** `faithful`

```
> data(faithful)

> class(faithful)
[1] "data.frame"
```


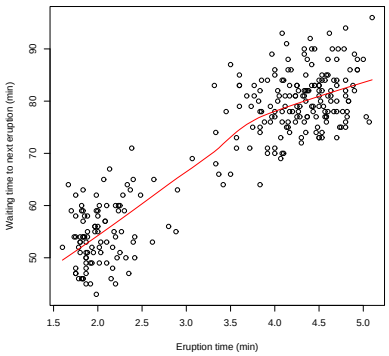

Figure 7:

```
> faithful
    eruptions waiting
1       3.600      79
2       1.800      54
3       3.333      74
4       2.283      62
.....

> summary(faithful)
   eruptions        waiting
 Min.   :1.600   Min.   :43.0
 1st Qu.:2.163   1st Qu.:58.0
 Median :4.000   Median :76.0
 Mean   :3.488   Mean   :70.9
 3rd Qu.:4.454   3rd Qu.:82.0
 Max.   :5.100   Max.   :96.0

> plot(faithful,xlab = "Eruption time (min)",
          ylab = "Waiting time to next eruption (min)")
     lines(lowess(faithful),col = "red")
```

6. **Analisi del data.frame** `pressure`

```
> data(pressure)

> class(pressure)
[1] "data.frame"

> pressure
   temperature pressure
```

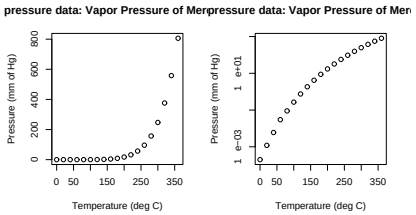

Figure 8:

```
1            0   0.0002
2           20   0.0012
3           40   0.0060
4           60   0.0300
5           80   0.0900
6          100   0.2700
....

> par(mfrow=c(1,2),pty='s')
> plot(pressure, xlab = "Temperature (deg C)",
          ylab = "Pressure (mm of Hg)",
          main = "pressure data: Vapor Pressure of Mercury")

# qualchevolte un rapporto e' rappresentato meglio usando
# una scala non lineare

> plot(pressure, xlab = "Temperature (deg C)",  log = "y",
          ylab = "Pressure (mm of Hg)",
          main = "pressure data: Vapor Pressure of Mercury")

> par(mfrow=c(1,1),pty='m')
```

7. **Analisi della matrice** `phones`

```
> data(phones)

> class(phones)
NULL
> data.class(phones)
```


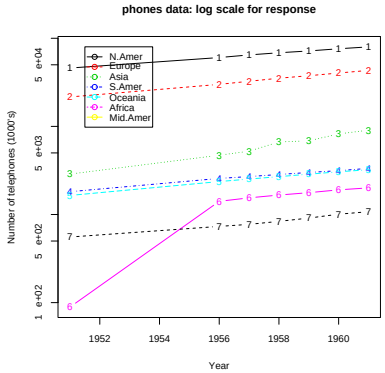

Figure 9:

```
[1] "matrix"



> phones
     N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
1951  45939  21574 2876   1815    1646     89      555
1956  60423  29990 4708   2568    2366   1411      733
1957  64721  32510 5230   2695    2526   1546      773
1958  68484  35218 6662   2845    2691   1663      836
1959  71799  37598 6856   3000    2868   1769      911
1960  76036  40341 8220   3145    3054   1905     1008
1961  79831  43173 9053   3338    3224   2005     1076

>  matplot(rownames(phones), phones, type = "b", log = "y",
             xlab = "Year", ylab = "Number of telephones (1000's)")
> legend(1951.5, 80000, colnames(phones), col = 1:7, lty = 1:7,
        pch = rep(21, 7))
>  title(main = "phones data: log scale for response")
```

8. **Analisi del data.frame InsectSprays**

```
> data(InsectSprays)

> class(InsectSprays)
[1] "data.frame"


> summary(InsectSprays)
     count       spray
 Min.   : 0.00   A:12
```

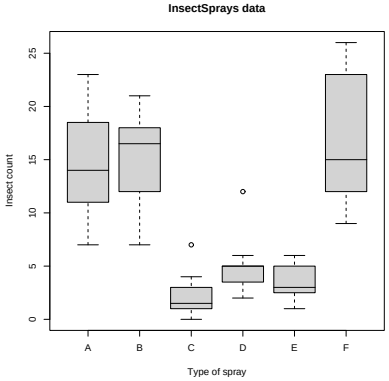

Figure 10:

```
 1st Qu.: 3.00   B:12
 Median : 7.00   C:12
 Mean   : 9.50   D:12
 3rd Qu.:14.25   E:12
 Max.   :26.00   F:12


> plot(InsectSprays)


> boxplot(count ~ spray, data = InsectSprays,
            xlab = "Type of spray", ylab = "Insect count",
            main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
```

9. **Analisi del data.frame** `mtcars`

```
> data(mtcars)

> class(mtcars)
[1] "data.frame"


> mtcars
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
....

> coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
           panel = panel.smooth, rows = 1)
```


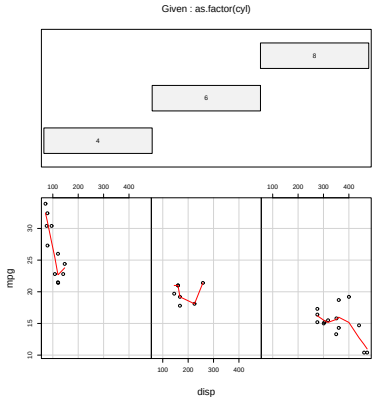

Figure 11:

10. **Analisi della matrice** `volcano`

```
> data(volcano)

> class(volcano)
NULL
> data.class(volcano)
[1] "matrix"
>


> par(mfrow=c(1,2),pty='s')
> contour(volcano)
> filled.contour(volcano, color = terrain.colors, asp = 1)
```

1.2 Esercizi

1. Leggere e compilare gli esercizi dell'appendice.
2. Analizzare tutti i seguenti insiemi di dati:
   (a) `airquality`
   (b) `sunspots`
   (c) `cars`
   (d) `OrchardSprays`
   (e) `warpbreaks`
   (f) `iris`
   (g) `Titanic`
   (h) `chickwts`
   (i) `trees`

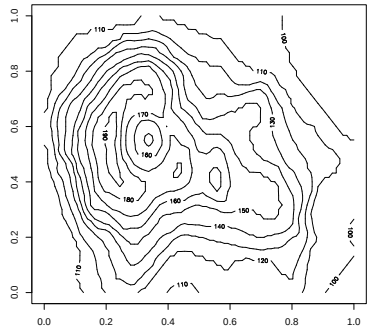

Figure 12:

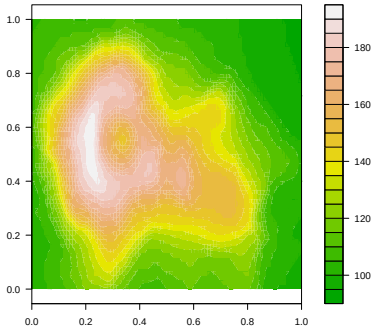

Figure 13:

In ciascuno caso usare la funzione `help` per ottenere una descrizione dei dati; stabilire la `class` dell'oggetto che contiene i dati; produrre una sintesi, sia empirica, sia grafica, dei dati (usare l'informazione data da `help` per un suggerimento, ma non limitarsi a questo); scrivere un breve paragrafo sugli aspetti principali che vengono evidenzati dall'analasi.

Facendo questi esercizi, provate ad usare le opzioni che sono disponibili nelle funzioni `plot` e `par` (per esempio).

2 Simulazione

2.1 Introduzione

Le techniche di simulazione sono diventate molto importanti per un'ampia gamma di usi statistici. Soprattutto, in situazioni in cui la distribuzione di una stimatore, per esempio, è troppo complessa per essere valutato, le techniche di simulazione possono fornire un'alternativa. Inoltre, la disponibilità di tali techniche ha cambiato sostanzialmente l'insieme di problemi che può essere affrontato dalla statistica.

In questo capitolo consideriamo alcune delle tecniche più semplici per simulare dati da una distribuzione specificata. Nei capitoli successivi, svilupperemo metodi simili per la simulazione di processi stocastici, e sfruteremo la capacità di simulare per fare inferenza.

In R (come in qualsiasi pachetto statistico) ci sono funzioni per generare dati simulati da una distribuzione specificata. Considereremo tali funzioni e svilupperemo altre possibilità di R.

2.2 Simulazione per inversione

Se una variabile casuale X ha funzione di ripartizione F continua, allora la variabile $U = F(X)$ ha distribuzione uniforme su $[0,1]$. Segue che, se U ha distribuzione uniforme su $[0,1]$, dunque la variabile definita come

$$X = F^{-1}(U)$$

ha funzione di ripartizione F. Assumando che si possa simulare dati dalla distribuzione uniforme, questo risultato fornisce un metodo semplice per simulare da F.

1. Simulare dati $u_1, \ldots, u_n$ dalla distribuzione uniforme;
2. Trasformare: $x_i = F^{-1}(u_i),\ i = 1, \ldots, n$;
3. I dati $x_1, \ldots, x_n$ avranno la distribuzione F come richiesto.

In R la funzione `runif` è usata per simulare dati uniformemente distribuiti. Ad esempio

```
u <- runif(10)
```

simula 10 valori dalla distribuzione uniforme[1] ed assegna i valori ad un vettore `u`. Il simbolo `<-` è usato per un assegnazione: il simbolo `_` è sinonimo, mentre il simbolo `=` è vietato per questo obiettivo.

Nota

In realtà si possono passare altri argomenti alla funzione `runif`. Per esempio,

```
u2 <- runif(10,3,6)
```

simula 10 valori dalla distribuzione uniforme su $[3,6]$. Questo illustra un aspetto di R: in genere una funzione ha valori di default per alcuni dei sui argomenti. Se questi argomenti non vengono specificati in altro modo, vengono usati i valori di default; se vengono specificati, i valori di default sono ignorati.

In questo esempio, se scriviamo `help(runif)`, troviamo (in parte)

[1] Non discuteremo come R simula da una distribuzione uniforme. In verità non lo fa: genera valori secondo un algoritmo deterministico i quali assomigliano a valori casuali sull'intervallo $[0,1]$


```
Usage:

     runif(n, min=0, max=1)

Arguments:


       n: number of observations. If 'length(n) > 1', the length is
          taken to be the number required.

 min,max: lower and upper limits of the distribution.
```

Cioè, viene assunta una distribuzione uniforme su $[0,1]$ a meno che non vengono specificati i valori di `min` e `max`.

Si noti anche che nell'esempio precedente abbiamo scritto `runif(10,3,6)` piuttosto che `runif(n=10,min=3,max=6)`. I due hanno lo stesso significato purchè i valori nel primo siano dati in ordine, mentre il secondo permette uno scambio dell'ordine: per esempio, `runif(max=6,min=3,n=10)` avrebbe lo stesso significato.

Si possono anche accorciare i nomi, purchè si evitono ambiguità. Per esempio `runif(n=10,mi=3,ma=6)` funziona, ma non (ovviamente) `runif(n=10,m=3,6)`. Con il secondo otteniamo l'errore

```
Error in runif(n = 10, m = 3, 6) :
        argument 2 matches multiple formal arguments
```

In fine, si nota che non è mai necessario usare gli argomenti `min` e `max` in `runif`: se $X \sim \mathrm{U}[0,1]$ allora $a + (b-a)X \sim \mathrm{U}[a,b]$. Quindi, nell'esempio precedente,

```
u2 <- 3+3*runif(10)
```

ha lo stesso effetto.

2.3 Altre possibilità di R

Un aspetto importante di R è la capacità di scrivere funzioni nuove. Per esempio se volessimo simulare dati dalla variabile $X = 1/U$, dove U ha la distribuzione uniforme, potremmo definire

```
recrunif <- function(n,min=0,max=1) 1/runif(n,min,max)
```

Questa istruzione definisce una funzione con argomenti `n,min,max`, tra i quali `min` e `max` hanno un valore di default. La funzione è stata assegnata all'oggetto con nome `recrunif`. Viene usata nel solito modo: `recrunif(10)` o `recrunif(10,3,6)` ad esempio.

Ci sono diversi modi per modificare una funzione:

1. Si può usare un editor esterno a R per scrivere la funzione, modificare la funzione tramite l'editor, e copiare e incollare la funzione nella sessione di R; o
2. Scrivendo `fix(recrunif)` si apre un editor che contiene l'oggetto `recrunif`, che può essere modificato.

Spesso, vogliamo che una funzione esigua varie istruzioni. Le varie istuzioni devono essere scritte come righe incluse tra parentesi. Per esempio, per modificare

`recrunif` in modo tale che dopo aver simulato i dati produca un istogramma, si scrive:[2]

```
recrunif <- function(n,min=0,max=1) {
                  u <- 1/runif(n,min,max)
                  hist(u)
                  }
```

Una funzione può includere un numero qualsiasi di istruzioni.

Ritorniamo al tema della simulazione dalla distribuzione uniforme. Chiaramente, ogni volta che usiamo `runif` otteniamo diversi valori (altrimenti i valori non sarebbero casuali).[3] Supponiamo di essere interessati a vedere i campioni ottenuti in prove ripetute. Usiamo questo obiettivo per illustrare un altro aspetto di R: l'uso di cicli. Si consideri:

```
simrep <- function(nrep,ncamp,umin=0,umax=1){
              nrow <- ceiling(nrep/3)
              par(mfrow=c(nrow,3),pty='s')
              for(i in 1:nrep){
                  hist(runif(ncamp,min=umin,max=umax),main='',xlab='x',
                        xlim=c(0,1),col='lightblue')
              }
         }
```

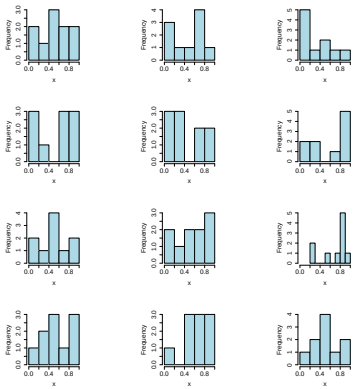

Figure 14:

Questa istruzione definisce una nuova funzione che viene assegnata all'oggetto `simrep` e che viene usata come una funzione qualsiasi. Per esempio:

```
> simrep(12,10)
```

Commenti:

[2] In realtà, basterebbe una riga in questo caso: `recrunif <- function(n,min=0,max=1) hist(1/runif(n,min,max))`

[3] Ci sono situazioni in cui potremmo volere ripetere un esercizio usando gli stessi valori simulati. Si può farlo usando la funzione `.set.seed`: non avremo bisogno di tale funzione in questo corso.

1. Come prima, gli argomenti `min` e `max` hanno valori di default;

2. La funzione `ceiling` è una funzione che arrotonda all'intero più grande;

3. Abbiamo usato la forma: `for(condizione)`. Le istruzioni successive a `for`, incluse tra parentesi, vengono eseguite per ogni valore di `condizione`. In questo esempio `1:ncamp` sarà la serie $1, 2, \ldots,$`ncamp`. Quindi, le istruzioni fra parentesi sono eseguite prima con $i = 1$, poi con $i = 2$, e cosi via fino a $i =$`ncamp`. In questo particolare esempio, il valore di `i` non viene usato, e il ruolo di `for` è soltanto di permettere una repetizione delle istruzioni `ncamp` volte. Se, sostituissimo il comando `hist` con

   ```
   hist(runif(ncamp,min=umin,max=umax),main=paste('campione ',i),xlab='x',
                                 xlim=c(0,1),col='lightblue')
   ```

   il valore di `i` verrebbe usato come parte del titolo di ogni istogramma.

4. Abbiamo usato vari argomenti opzionali nella funzione `hist`. In particolare, `xlim=c(0,1)` assicura che tutti gli istogrammi abbiano la stessa scala per l'asse delle x. Avremmo potuto fare la stessa cosa per l'asse delle y specificando `ylim`, però sarebbe meno facile scegliere un opportuno valore per il valore massimo in questo esempio.

Allora, se volessimo produrre un campione dalla distribuzione Exp(λ), la funzione di ripartizione è

$$F(x) = 1 - \exp(-\lambda x), \quad x > 0,$$

dalla quale

$$F^{-1}(u) = -\log(1-u)/\lambda, \quad 0 < u < 1.$$

Dunque, la funzione

```
expsim <- function(n,lambda) -log(1-runif(n))/lambda
```

produce un campione di dimensione `n` dalla distribuzione Exp(λ). Se volessimo che $\lambda = 1$ fosse un valore di default, potremmo definire invece

```
expsim <- function(n,lambda=1) -log(1-runif(n))/lambda
```

In questo caso, ad esempio, o `expsim(10,1)` o `expsim(10)` darebbe un campione di dimensione 10 dalla distribuzione Exp(1), mentre `expsim(10,2)` darebbe un campione di dimensione 10 dalla distribuzione Exp(2).

2.4 Altre distribuzioni in R

Come abbiamo visto, `runif` viene usata per simulare dalla distribuzione uniforme. Inoltre, si possono trasformare tali dati per simulare da altre distribuzioni (purchè sia disponibile e facile da calcolare l'inversa della funzione di ripartizione). Tuttavia, per semplicità, R ha funzioni specifiche per simulare dalle distribuzioni usate più spesso. Per esempio `rpois` per la distribuzione di Poisson; `rnorm` per la distribuzione gaussiana (normale); `rbinom` per la distribuzione binomiale; `rexp` per la distribuzione esponenziale. In ogni caso la struttura è uguale. Per esempio,

```
rnorm(10)
```

genera 10 dati dalla distribuzione gaussiana standard (media = 0, deviazione standard = 1), mentre

```
rnorm(10,4,7)
```

fa la stessa cosa, ma per una distribuzione con media 4 e deviazione standard 7. Si usi `help` per capire esattamente la forma del comando.

La funzione `sample` è particolarmente utile. Come si vede da `help(sample)` viene usata tramite

```
sample(x, size, replace = FALSE, prob = NULL)
```

Supponiamo che `x` sia un vettore che contiene i valori $1, 2, \ldots, 6$ (che rappresenta, per esempio, i lati di un dado). A tale proposito, potremmo costruire `x` in vari modi:

1. `x <- scan()` seguito dai numeri 1,2,...,6.
2. `x <- 1:6`
3. `x <- seq(1,6,by=1)`. (La funzione `seq` è un modo generale per costruire serie di numeri: si vedi `help(seq)` per una descrizione completa.)

Illustriamo diversi usi della funzione `sample`:

1. `sample(10,x,replace=TRUE)`

   campiona 10 valori da `x` assegnando ad ogni valore di `x` uguale probabilità di essere campionato. Il valore di default di `replace` è `FALSE` (falso). Sostituendo il valore di default, in questo caso con `replace=TRUE`, si ha che dopo che un valore di `x` è stato campionato, rimane disponibile per le estrazioni successivi. (Nell'esempio, simulo l'esperimento di lanciare un dado bilanciato 10 volte).

2. `sample(10,x,replace=TRUE,prob=c(rep(.1,5),.5))`

   La forma `c(rep(.1,5),.5)` richiede spiegazioni. Innanzitutto, la funzione `rep` ripete un'espressione: in questo esempio semplice, il valore .1 viene ripetuto 5 volte. Si usi `help(rep)` per maggiori dettagli. La funzione `c` esegue una concatenazione. Infine, `c(rep(.1,5),.5)` restituisce il vettore (.1, .1, .1, .1, .1, .5). Dando `prob=c(rep(.1,5),.5)` dentro la chiamata alla funzione `sample` i valori di `x` sono campionati con le probabilità specificata in `prob`. Ossia, i valori da 1 a 5 sono scelti con probabilità 0.1, mentre il valore 6 è scelto con probabilità 0.5. (Nell'esempio si simula 10 volte l'esperimento del lancio di un dado non bilanciato, nel senso che il valore 6 è 5 volte più probabile che ogni altro valore).

3. `sample(3,x)`

   In questo caso viene usato il valore di default di `replace`. Cioè, dopo la scelta di un valore di `x`,questo diviene non disponibile. (Nell'esempio simulo l'esperimento di scegliere, senza reinsertimento, carte con i valori da 1 a 6. Si noti che questa volta la chiamata `sample(10,x)` non ha senso.)

2.5 Simulazione per il Calcolo dei Integrali

Un'applicazione comune è il calcolo degli integrali tramite simulazione. Innanzitutto si consideri l'integrale

$$\theta = \int_{-\infty}^{\infty} g(x) f(x) dx,$$

dove f è una funzione di densità di probabilità. Cioè,

$$\theta = \mathrm{E}_f[g(x)].$$

Siano $x_1, x_2, \ldots, x_n$ valori simulati da una variabile con la funzione di densità di probabilità f. Segue dalla legge di numeri grandi che il valore di

$$\hat{\theta} = \frac{1}{n} \sum_{i=1}^{n} g(x_i),$$

cioè la media campionaria di $g(X)$, fornisce una stima consistente di θ.

Ad esempio, per calcolare

$$\theta = \int_0^1 \sin x dx$$

nella quale $f(x) = 1$ sull'intervallo $[0, 1]$, che corrisponde alla distribuzione uniforme U$[0, 1]$, potremmo scrivere la seguente funzione.

```
integrale <- function(n){
   camp<-runif(n)
   g<-sin(camp)
   mean(g)
}
```

Due applicazioni di questa funzione conducono ai risultati:

```
> integrale(1000)
[1] 0.4677327

> integrale(1000)
[1] 0.449604
```

Da questi vediamo che anche con la simulazione di 1000 valori il risultato potrebbe essere non preciso. (Il risultato esatto è conosciuto per questo esempio: $\theta = 1 - \cos 1 = 0.4597$). Dunque, la semplicità di applicazione deve essere equilibrata con l'assenza di precisione. Però ci sono modificazioni a questa procedura che possono migliorare la precisione.

Si noti che qualisiasi integrale può venire espresso nella forma precedente. Ad esempio:

$$\begin{aligned} \theta &= \int_{-\infty}^{\infty} h(x) dx \\ &= \int_{-\infty}^{\infty} g(x) f(x) dx \end{aligned}$$

dove $g(x) = h(x)/f(x)$, purchè $f(.)$ non sia zero nell'insieme di punti dove $h(.)$ non è zero. Dunque, qualsiasi integrale può venire scritto nella forma precedente con qualsiasi scelta di f. In realtà, la scelta di f determina la precisione dell'approssimazione: la sostituzione di una scelta con un'altra per migliorare la precisione del calcolo del integrale è chiamata 'importance sampling'.

Ad esempio, potremmo modificare l'esempio precedente, simulando dalla distribuzione normale anzichè la distribuzione uniforme:

$$\theta = \int_0^1 \frac{h(x)}{\phi(x)} \phi(x) dx$$

dove ϕ è la densità della distribuzione standard normale e

$$h(x) = \begin{cases} \sin x & \text{se } 0 < x < 1 \\ 0 & \text{altrimenti} \end{cases}$$

In **R** si scrive:

```
> integrale2 <- function(n){
   camp<-rnorm(n)
   id<-(1:n)[camp>0&camp<1]
   g<-rep(0,n)
   g[id]<-sin(camp[id])/dnorm(camp[id])
   mean(g)
}
```

Questa volta si ottengono i seguenti risultati

```
> integrale2(1000)
[1] 0.4315168
> integrale2(1000)
[1] 0.4873704
```

che sembrano più variabili dei risultati precedenti. Infatti, la scelta di una distribuzione normale in questo caso non è buona: i valori vengono simulati in una regione per cui la funzione h è costante (zero). L'idea dell' 'importance sampling' è quella di simulare più realizzazioni in regioni per cui la funzione h cambia molto.

2.6 Esercizi

1. Si scriva una funzione che simula campioni dalla distribuzione normale, con media, varianza e numerosità del campione arbitrarie, e che per ogni campione produce un istogramma e un grafico quantile-quantile. (si veda `qqnorm`).

2. La distribuzione di Pareto ha funzione di ripartizione

$$F(x) = 1 - x^{-a}, \quad x > 1,$$

per un parametro $a > 0$. Si usa il metodo di inversione per scrivere una funzione che simula un campione di dimensione `n` dalla distribuzione di Pareto con parametro `a`.

3. Si scriva una funzione che simula campioni di larghezza n dalla distribuzione uniforme, calcola la media del campione, ripete questa operazione per tanti campioni, e costruisce un istogramma delle medie. Usando questa funzione, cosa si osserva nella forma della istogramma? In quale modo il risultato dipende da n?

4. Si ripeta 3. sostituendo la distribuzione uniforme con la distribuzione esponenziale. I risultati sono simili o no?

5. Si usi la funzione 'sample' per simulare un gioco in cui un dado bilanciato è lanciato 100 volte con un punteggio che consiste nella somma dei singoli punti. Si ripeti questa procedura 1000 volte e si produca una tabella e un istogramma dei punteggi ottenuti. Si commenti sulla forma della distribuzione di valori.

6. Si scrivano le istruzioni in 5. nella forma di una funzione che permette un cambio dei valori di 100 e 1000 come argomenti della funzione (i valori di 100 e 1000 possono essere usati come valori di default).

7. Si modifichi la funzione in 6. per permettere la simulazione dei lanci di un dado non bilanciato, per cui le probabilità di uscito di ogni faccia sono contenuti in un vettore che viene usato come argomento della funzione.

8. Usare campioni dalla distribuzione uniforme (con domini appropriati) per calcolare approssimazioni dei seguenti integrali

(a)

$$\int_0^1 x^3 dx$$

(b)

$$\int_{-1}^1 \exp(x) dx$$

(c)

$$\int_{-\pi}^{\pi} \sin x dx$$

Ripetere questi calcoli diverse volte e notare quanto variabili sono i risultati. Si confronti i valori anche con i risultati esatti.

9. Si noti che se $X \sim \mathrm{N}(0,1)$, la probabilità $P(X > 1)$ può essere scritta come l'integrale

$$\theta = \int_{-\infty}^{\infty} I(x)\phi(x)dx$$

dove ϕ è la densità della distribuzione standard normale e

$$I(x) = \begin{cases} 1 & \text{se } x > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Usare questo per ottenere tramite simulazione un'approssimazione per $P(X > 1)$.

10. Ottenere approssimazioni per la probabilità nell'esempio precedente tramite simulazione dalla distribuzione esponenziale. I calcoli sembrano migliorati o no? Migliorano con un cambio del parametro della distribuzione esponenziale?

3 Tecniche Non Parametriche

Un vantaggio dell'uso di pachetti come R, che hanno alta potenza computazionale, è che forniscono l'opportunità di sviluppare techniche che, altrimenti, non sarebbero disponibili. Questo è il caso con techniche moderne per metodi non parametrici.

3.1 Il metodo del nucleo

Questa è una tecnica per la stima di una densità. Siano $x_1, \ldots, x_n$ realizzazioni da una distribuzione con densità ignota f.

L'idea di questo metodo è disperdere una unità di massa attorno ad ognuno dei dati. Allora, la stima della funzione di densità è la somma di tutte le unità di massa. Formalmente:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

dove K è una funzione di densità (il nucleo), centrata su zero, e h è un parametro di lisciamento. Per $h \downarrow 0$ $\hat{f}$ diviene concentrata sui dati $x_1, \ldots, x_n$. Per $h \uparrow$, $\hat{f}$ diviene sempre più piatta. Cosi, h determina l'equilibrio fra lisciamento e vicinanze ai dati.

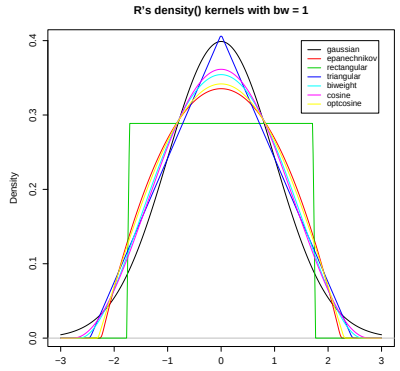

Figure 15:

La scelta della funzione K ha meno importanza: succede che tutte le scelte danno risultati simili. Le varie opzioni disponibili in R sono dimostrate di sopra. Il nucleo 'rectangular' è chiaramente il più semplice, ma risulta in stime con salti. Il nucleo 'epanechnikov' ha, in un certo senso, proprietà ottimali, però il nucleo gaussiano viene usato più spesso siccome la sua forma è ben conosciuta.

Illustriamo con il dataset `faithful`. In particolare, il nostro obiettivo è stimare la densità dei tempi di eruzione. Questi sono contenuti nella colonna del dataframe `faithful` con nome `eruptions`. Perciò, si chiamano `faithful$eruptions`.

La funzione che calcola la stima della funzione di densità si chiama `density`. Nella sua forma più semplice funziona cosi:

```
data(faithful)
d <- density(faithful$eruptions)
plot(d)
```


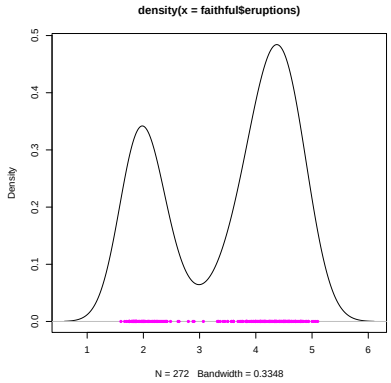

Figure 16:

```
points(faithful$eruptions,rep(0,length(faithful$eruptions)),
          col=6,cex=.5)
```

In questa maniera, la densità è stimata:

1. Su un intervallo che contiene tutti i dati;
2. A 512 punti di referenza su questo intervallo;
3. Tramite il nucleo gaussiano;
4. Usando un metodo empirico e semplice per la scelta del parametro di lisciamento.

Tutti questi valori di default possono essere modificati con opportuni argomenti: si veda `help(density)`.

Per esaminare l'effetto del parametro di lisciamento:

```
h <- seq(.01,1,le=9)
par(mfrow=c(3,3))
for(i in 1:9){
       d <- density(faithful$eruptions,bw=h[i])
       plot(d,main='')
       }
```

Come previsto, la scelta del parametro di lisciamento ha un effetto profondo sulla forma della stima: valori che sono troppo piccoli risultano in stime irregolari; valori che sono troppo grandi conducono a una stima per cui i dettagli suggeriti dai dati sono persi in conseguenza ad un lisciamento troppo forte.

Per esaminare l'effetto del parametro di lisciamento:

```
kernels <- eval(formals(density)$kernel)
par(mfrow=c(3,3))
 for(i in 1:7){
         d <- density(faithful$eruptions,kern=kernels[i])
         plot(d,main=kernels[i])
         }
```

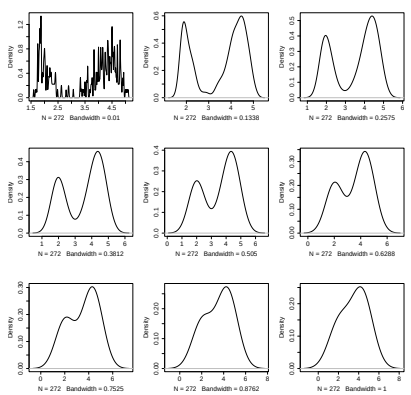

Figure 17:

Si noti che il primo comando, `kernels <- eval(formals(density)$kernel)` è soltanto un trucco per ottenere un vettore che contiene i nomi dei nuclei. (Si può usare la funzione `help` per capire le funzioni `formals` e `eval`.) Chiaramente, tutte le scelte sono quasi uguali, e l'unica scelta con una differenza osservabile è il nucleo 'rectangular'.

3.2 Regressione non parametrica

L'idea dell'uso di un modulo si estende al concetto di regressione. Accade spesso che l'uso di polinomi per una regressione risulta in modelli stimati che non seguono bene i dati. Come alternativa si può assumere che i dati sono legati linearmente, ma solo localmente. Questa idea conduce all'algoritmo di lowess (locally weighted regression). In sostanze l'idea è stimare $E[Y|x]$ come

$$\hat{E}[Y|x] = \sum_{x_i \in \delta x} w_i y_i$$

dove δx è un intorno di x e i w_i sono pesi che vengono determinati da una regressione ai minimi quadrati pesati dei punti (x_i, y_i) per i dati dentro la vicinanza δx. I pesi della regressione sono determinati dalla specificazione di un nucleo nell'intorno δx. Il liscamento del modello stimato è determinato dalla larghezza degli intorni.

In R ci sono due opportune funzioni `lowess` e `loess`. La più facile è `lowess`. Illustriamo con un esempio:

```
data(cars)
par(pch=16)
plot(cars, main = "lowess(cars)")
f <- c(2/3,.1,.5,1)
for(i in 1:4){
     lines(lowess(cars,f=f[i]), col = i+1,lty=i)
     }
legend(5, 120, c(paste("f = ", f)), lty = 1:4, col = 2:5)
```


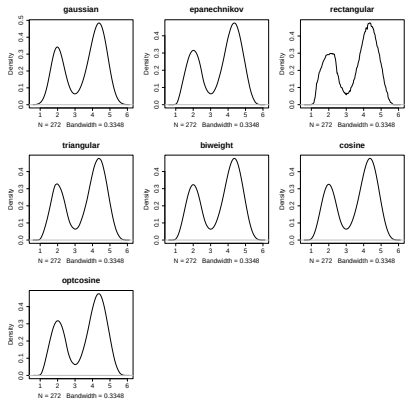

Figure 18:

In questo esempio assumiamo che le colonne di `cars` contengono i valori di x e y respettivamente. Più esplicitamente, potremmo scrivere

```
lowess(cars$speed,cars$dist)
```

o

```
lowess(cars[,1],cars[,2])
```

per ottenere lo stesso effetto.

Il parametro f determina la proporzione dei dati che viene usata nella specificazione degli intorni. Il valore di default è $f = 2/3$. Si vede, come previsto, che f determina il grado di lisciamento del modello.

La funzione `loess` offre più flessibilità nella forma del modello (che può essere localmente polinomiale anzichè lineare). Per esempio:

```
> cars.lo <- loess(dist ~ speed, cars)
```

Dunque

```
> summary(cars.lo)
Call:
loess(formula = dist ~ speed, data = cars)

Number of Observations: 50
Equivalent Number of Parameters: 4.77
Residual Standard Error: 15.29
Trace of smoother matrix: 5.23

Control settings:
  normalize:  TRUE
  span      :  0.75
  degree    :  2
  family    :  gaussian
  surface   :  interpolate         cell = 0.2
```

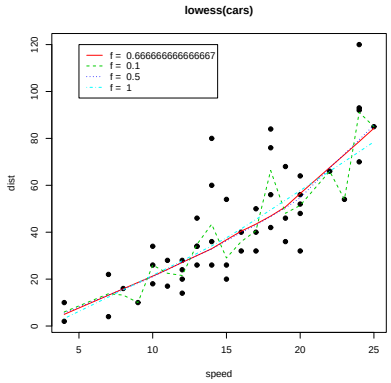

Figure 19:

dà vari dettagli del modello stimato. Per ottenere previsioni:

```
>  predict(cars.lo, data.frame(speed = 5:30), se = TRUE)
$fit
 [1]  7.810489 10.041808 12.567960 15.369183 18.425712 21.828039 25.539675
 [8] 29.350386 33.230660 37.167935 41.205226 45.055736 48.355889 49.824812
[15] 51.986702 56.445263 62.008703 68.529340 76.193111 85.142467 95.323096
[22]         NA         NA         NA         NA         NA

$se.fit
 [1] 7.568539 5.943649 4.976453 4.515801 4.316362 4.030120 3.750561 3.715593
 [9] 3.776298 4.091044 4.708759 4.244697 4.035236 3.752765 4.004017 4.056945
[17] 4.005540 4.065234 4.579053 5.948757 8.300416       NA       NA       NA
[25]       NA       NA

$residual.scale
[1] 15.29233

$df
[1] 44.62733
```

dà valori di previsone ed errori standard per valori di `speed` da 5 a 30. Si noti che il risultato è `NA` per valori di `speed` oltre 25; si veda `help(loess)` per un modo che permette l'estrapolazione, ed anche per una spiegazione di altri aspetti e argomenti di `loess`.

Per esaminare il modello stimato:

```
>  plot(cars, main = "loess(cars)")
> lines(5:30,predict(cars.lo,data.frame(5:30)),col=2)
```

3.3 Esercizi

1. Si usi il metodo del nucleo per stimare la densità delle due variabili contenute nel dataset `women`. Per ciascuna, esaminare l'effetto di variazioni nel

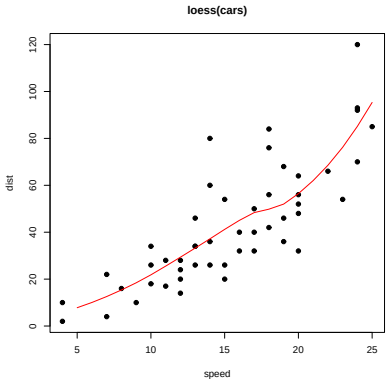

Figure 20:

parametro di lisciamento.

2. Scrivere una funzione che produce un grafico della stima delle densità per ciascuna colonna di una matrice. Si applichi questa funzione al dataset `women`. La si applichi anche al dataset `swiss`.

3. Si scriva una funzione che ha la stessa effetto di `density`: cioè, la calcola della stima della densità di un dataset tramite il metodo del nucleo.

4. Si usi la funzione `lowess` per stimare la relazione fra le variabili nel dataset `women`. Si esamini l'effetto della scelta del parametro di lisciamento `f`.

5. La funzione `loess` viene usata anche per modelli più complessi. Si esplori questo aspetto sul dataset `trees`. Per esempio:

   ```
   loess(Volume~Girth+Height,trees)
   ```

4 Verosimiglianza

In questo capitolo usiamo R per esaminare vari aspetti della verosimiglianza.

4.1 Verosimiglianza esponenziale

Siano $x_1, \ldots, x_n$ realizzazioni dalla distribuzione esponenziale Exp(λ). La log-verosimiglianza è

$$\ell(\lambda) = n \log \lambda - \lambda \sum_{i=1}^{n} x_i$$

dalla quale è facile dimostrare che la stimatore di massima verosimiglianza è $\hat{\lambda} = 1/\bar{x}$.

Per fare un grafico della log-verosimiglianza:

```
function(dati,llim,ulim){
            l <- length(dati)
            llik <- function(x)l*log(x)-x*sum(dati)
            curve(llik,llim,ulim,col=2,xlab='lambda',
               ylab='log-verosimiglianza')
            abline(v=1/mean(dati),col=3,lty=2)
         }
```

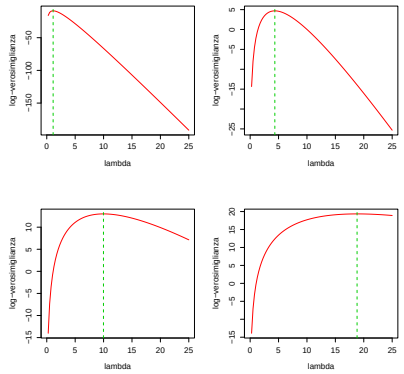

Figure 21:

Possiamo usare questa funzione innanzitutto per esaminare l'effetto del valore del parametro λ.

```
> dati <- matrix(rexp(40,rep(c(1,5,10,20),rep(10,4))),byrow=T,nrow=4)
> par(mfrow=c(2,2))
> apply(dati,1,expll,llim=0,ulim=25)
```

Il risultato è la simulazione di 4 campioni, ciascuno di 10 valori, con valori di λ uguale a 1, 5, 10 e 20 rispettivamente. Viene poi fatto un grafico della log-verosimiglianza e tracciata una reta in corrispondenza della stima di massima verosimiglianza. Si noti l'uso delle funzioni `rep` e, per evitare cicli, `apply`.

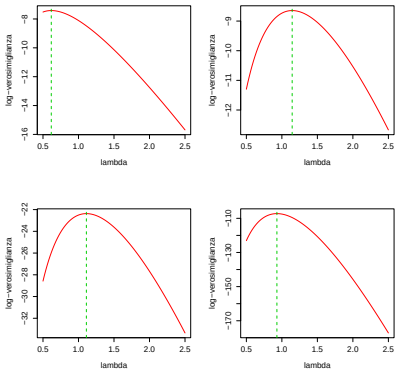

Figure 22:

È più interessante ripetere questo esercizio con λ fissato, ma con campioni di dimensioni diverse:

```
> dati <- rexp(140,1)
> datil <- list(dati[1:5],dati[6:15],dati[16:40],dati[41:140])
> lapply(datil,expll,llim=0.5,ulim=2.5)
```

In questo caso, siccome ogni campione ha dimensione diversa, è stato più comodo mettere i valori in un oggetto della 'class' `list`. Un `list` in R può essere qualsiasi collezione di oggetti, una mistura di vettori, matrici, tabelle, ecc. Per questo esempio, il nostro `list`, che si chiama `xl`, consiste nei 4 vettori che contengono i 4 campioni. (Non si può costruire da questi una matrice dato che le dimensioni sone diverse.) Per eseguire una funzione su ogni elemento di un `list`, si usa la funzione `lapply` in modo simile alla funzione `apply` (senza dover specificare la dimensione). L'applicazione ai campioni simulati di dimensioni $n = 5, 10, 25$ e 100 dalla distribuzione Exp(λ) con $\lambda = 1$ illustra la tendenza della funzione di log-verosimiglianza ad essere parabolica attorno al vero valore di λ per n che aumenta.

Un aspetto collegato è il risultato che, asintoticamente, la distribuzione della stimatore di massima verosimiglianza è normale:

$$\hat{\theta} \sim \mathrm{N}(\theta, 1/I(\theta))$$

dove $I(\theta)$ è l'informazione di Fisher per il parametro θ. Per il modello esponenziale:

$$\hat{\lambda} \sim \mathrm{N}(\lambda, \lambda^2/n)$$

Creiamo una funzione che simula campioni dalla distribuzione esponenziale, calcola la stima di massima verosimiglianza ($\hat{\lambda} = 1/\bar{x}$), produce un istogramma e un grafico quantile-quantile per fare un confronto con la distribuzione asintotica.

```
mvsim <- function(n,nsim,lambda){
    par(mfrow=c(length(n),2),pty='s')
    for(i in 1:length(n)){
        dati <- matrix(rexp(n[i]*nsim,lambda),nrow=nsim)
```

```
        mv <- 1/apply(dati,1,mean)
        hist(mv,prob=T,main='',xlab='MLE')
        qqplot(mv,qnorm((1:nsim)/(nsim+1),
             lambda,lambda/sqrt(n[i])),xlab='Empirici',ylab='Asintotici')
        abline(0,1,col=2)
        }
}
```

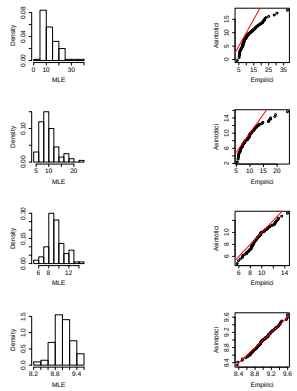

Figure 23:

Applicando questa funzione con

```
> mvsim(c(5,10,25,1000),100,9)
```

che corrisponde a campioni di dimensione $n = 5, 10, 25$ e 100 dalla distribuzione Exp(9), si vede chiaramente il miglioramento dell'approssimazione asintotica all'aumentare di n.

4.2 Modelli Multiparametrici

Techniche simili possono venire usate per lo studio di verosimiglianze con due parametri. Consideriamo il modello Gamma(a, b). Con dati $x_1, \ldots, x_n$ indipendenti la log-verosimiglianza è

$$\ell(a, b) = na \log b + (a - 1) \sum_{i=1}^{n} \log x_i - b \sum_{i=1}^{n} x_i - n \log \Gamma(a)$$

La funzione seguente permette la costruzione di un grafico della log-verosimiglianza su una regione scelta.

```
mvgam <- function(x,alim,blim){
     a <- seq(alim[1],alim[2],length=100)
     b <- seq(blim[1],blim[2],length=100)
     mvfun <- function(aval,bval,x){
        n <- length(x)
        mv <- n*aval*log(bval) + (aval-1)*sum(log(x))-
```



```
              bval*sum(x)-n*lgamma(aval)
              mv
        }
     mvf <- outer(a,b,mvfun,x=x)
     image(a,b,mvf,col=terrain.colors(100))
}
```

Si noti:

1. La capacità di definire una funzione all'interno di una funzione;
2. L'uso della funzione `outer` per evitare un doppio ciclo;
3. L'uso della funzione `image`.

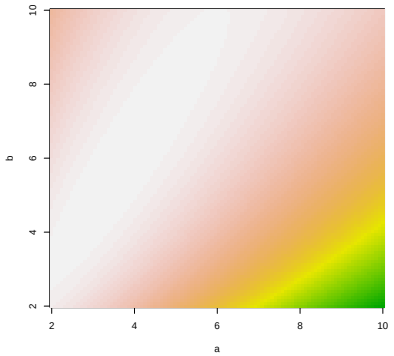

Figure 24:

Ad esempio:

```
> x <- rgamma(50,4,6)
> mvgam(x,c(2,10),c(2,10))
```

4.3 Esercizi

1. Scrivere una funzione per fare un grafico della log-verosimiglianza basata sul modello $x_1, \ldots, x_n \sim$ Poisson(λ).
2. Scrivere una funzione che simula campioni dalla distribuzione Poisson(λ) per vari valori di λ e fa un grafico della log-verosimiglianza in ciascun caso. La forma cambia con λ?
3. Scrivere una funzione che simula campioni di dimensioni n diverse dalla distribuzione Poisson(λ) per un valore fisso di λ e costruisce un grafico della log-verosimiglianza in ciascun caso. La forma cambia con n?
4. Scrivere una funzione che fornisce una valutazione della bontà della distribuzione normale per la stima di massima verosimiglianza del modello Poisson.

5. Siano $x_1, \ldots, x_n$ osservazioni dalla distribuzione normale $\mathrm{N}(\mu, \sigma^2)$. Allora, la log-verosimiglianza ha la forma

$$\ell(\mu, \sigma) = -n \log \sigma - \frac{1}{2\sigma^2} \left\{ S_2 - 2\mu S_1 + n\mu^2 \right\}$$

dove $S_1 = \sum x_i$ e $S_2 = \sum x_i^2$. Le stime di massima verosimiglianza sono $\hat{\mu} = S_1/n$, $\hat{\sigma} = \sqrt{n^{-1} \sum (x_i - \hat{\mu})^2}$. Scrivere una funzione che assume come argomenti i valori di S_1 e S_2 e produce, con un dominio appropriato, un grafico tramite la funzione `image` della log-verosimiglianza per (μ, σ)

6. Nel modello multinomiale $\mathcal{M}(\theta_1, \theta_2; n)$, le variabili x_1, x_2 soddisfanno

$$\Pr(x_1, x_2) \propto x_1^{\theta_1} x_2^{\theta_2} (n - x_1 - x_2)^{1-\theta_1-\theta_2}$$

dove $0 \leq x_1, x_2 \leq n$ per parametri $0 \leq \theta_1, \theta_2 \leq 1$. Le stime di massima verosimiglianza sono $\hat{\theta}_1 = x_1/n$, $\hat{\theta}_2 = x_2/n$. Scrivere una funzione che produce un grafico tramite la funzione `image` della log-verosimiglianza per (θ_1, θ_2) sul triangolo $0 \leq \theta_1, \theta_2 \leq 1$ basata su osservazioni x_1 e x_2 dal modello $\mathcal{M}(\theta_1, \theta_2; n)$. Come cambia la forma del grafico con x_1, x_2 e n?

7. Usare la funzione `sample` per simulare dal modello multinomiale. Fare un grafico delle stime di massima verosimiglianza. Come cambia la distribuzione campionaria con i valori veri dei parametri o con n?

5 Ottimizzazione

In questo capitolo esaminiamo funzioni per:

1. La soluzione di equazioni;
2. La minimizzazione (o massimizzazione) di funzioni;

ed alcuni applicazioni statistiche.

5.1 Soluzione di equazioni

La funzione `polyroot` fornisce un metodo semplice per la soluzione di polinomi. Ad esempio, per trovare le radici del polinomio

$$x^2 - 1 = 0$$

si scriva:

```
> polyroot(c(-1,0,1))
[1]  1+0i -1+0i
```

Si noti che le radici sono espresse in forma complessa. Però, c'è una tendenza per l'introduzione di errori di arrondamento. Per esempio:

```
>  polyroot(c(1, 2, 1))
[1] -1-1.110223e-16i -1+1.110223e-16i
```

mentre le radici vere sono tutte e due uguali a -1.

La funzione vale anche per polinomi di alto ordine:

```
> polyroot(c(3,5,-2,1,.5,.75,-3))
[1] -0.4799190+0.000000e+00i -0.8030183+8.126505e-01i -0.8030183-8.126505e-01i
[4]  0.5410649-9.901882e-01i  0.5410649+9.901882e-01i  1.2538257+5.921189e-16i
```

Ancora, ci vuole un minimo di senso comune per eliminare gli effetti di arrondamenti.

La funzione `uniroot` viene usata per la soluzione di funzioni che non sono polinomi. Per esempio, supponiamo di aver la funzione

$$e^x + ax = 0$$

che richiede una soluzione per a fisso.

Innanzitutto, definiamo la funzione

```
f <- function (x,a) exp(x) + a*x
```

Dunque, per trovare una soluzione dentro l'intervallo $[-1, 1]$ con, per esempio, $a = 1$,

```
> uniroot(f,c(-1,1),a=1)
$root
[1] -0.5671256

$f.root
[1] 2.764964e-05

$iter
[1] 4

$estim.prec
[1] 6.103516e-05
```

dalle quale la soluzione è -0.567. Gli altri valori del list contengono dettagli dell'algoritmo; questi possono essere cambiati tramite altri argomenti nella chiamata di `uniroot`. (Si veda `help(uniroot)`).

Ovviamente è importante che l'intervallo specificato contenga una radice:

```
> uniroot(f,c(0,1),a=1)
Error in uniroot(f, c(0, 1), a = 1) :
   f() values at end points not of opposite sign
```

Per un'applicazione statistica, consideriamo il problema di calcolare un intervallo di confidenza per la stima di massima verosimiglianza. Per un modello con un parametro unico, un risultato asintotico è che la devianza,

$$D(\theta) = 2\{\ell(\hat{\theta}) - \ell(\theta)\}$$

ha distribuzione χ^2_1 quando $\theta = \theta_0$, il vero valore del parametro. Perciò, le soluzioni dell'equazione

$$D(\theta) - q_{1-\alpha} = 0$$

dove $q_{1-\alpha}$ è il $(1-\alpha)$ quantile della distribuzione χ^2_1 danno i limiti di un intervallo di confidenza $100(1-\alpha)\%$ per θ.

Per esempio, siano $x_1, \ldots, x_n$ osservazioni indipendenti dalla distribuzione Exp(λ). Perciò, $\hat{\lambda} = 1/\bar{x}$ e

$$D(\lambda) = 2n\{-\log(\lambda\bar{x}) - 1 + \lambda\bar{x}\}$$

Scriviamo una funzione per la devianza:

```
expdev <- function(lambda,x){
          n <- length(x)
         2*n*(-log(lambda*mean(x))-1+lambda*mean(x))
}
```

e, per un'applicazione, simuliamo un campione esponenziale (con vero valore di $\lambda = 2$):

```
x <- rexp(10,2)
```

Per fare un grafico della devianza sull'intervallo $[0.1, 10]$:

```
lambda <- seq(0.1,10,length=100)
plot(lambda,expdev(lambda,x),type='l',xlab='lambda',ylab='deviance')
```

Allora, c'è un modo interattivo con cui si può trovare un intervallo di confidenza. Ad esempio, per l'intervallo di confidenza 95% si scriva

```
abline(h=qchisq(.95,1),col=2)
```

quindi si usi la funzione `locator` per trovare i punti di intersezione. (Si scriva `locator(2)` e si clicchi sulle posizioni richieste).

Per ottenere un intervallo con maggiore precisione, richiediamo i radici dell'equazione

$$D(\lambda) - q_{.95} = 0$$

Definiamo

```
expdevroot <- function(lambda,x,q){
        expdev(lambda,x)-qchisq(q,1)
    }
```

Ci vogliono due chiamate di `uniroot` per trovare le due radici:

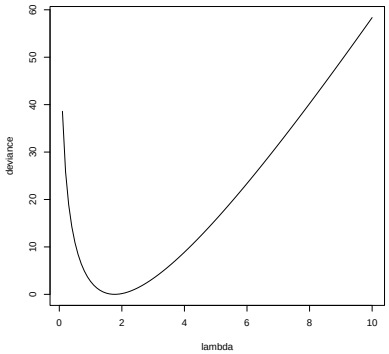

Figure 25:

```
> uniroot(expdevroot,c(.1,1/mean(x)),x=x,q=.95)
$root
[1] 0.8869633

$f.root
[1] 5.853147e-06

$iter
[1] 6

$estim.prec
[1] 6.103516e-05
```

e

```
> uniroot(expdevroot,c(1/mean(x),10),x=x,q=.95)
$root
[1] 3.104697

$f.root
[1] 0.0001405638

$iter
[1] 8

$estim.prec
[1] 6.103516e-05
```

dalle quali troviamo $[0.887, 3.105]$ come un intervallo di confidenza 95% per λ.

Possiamo aggiungere le rette al grafico con:

```
abline(v=c(0.887,3.105),col=3)
```

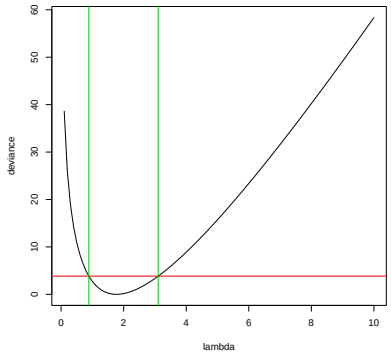

Figure 26:

5.2 Massimi e minimi

Il problema, in genere, è la ricerca del valore minimo (o massimo) di una funzione $f(x)$ dove x può essere o uno scalare o un vettore. Non è neccessario avere sia funzioni per minimizzazione che funzioni per la massimizzazione, dato che

$$\max(f(x)) = -\min(-f(x))$$

La situazione più semplice è quando x è uno scalare. In questo caso, si usa la funzione `optimize`. Ad esempio, per trovare il valore minimo sull'intervallo $[0, 1]$ di

$$f(x) = x^2 - x - a$$

con $a = 1$:

```
> f <- function (x,a) x^2-x-a
>      xmin <- optimize(f, c(0, 1), a = 1)
> xmin
$minimum
[1] 0.5

$objective
[1] -1.25
```

da cui il valore minimo è -1.25, che si verifica per $x = 0.5$

Per funzioni di un vettore x viene usata la funzione `optim`. Ad esempio, consideriamo la funzione

$$f(x_1, x_2) = (x_1 - a_1)^2 + (x_2 - a_2)^2$$

Con $a_1 = 3, a_2 = 5$ per esempio,

```
>  f <- function(x, a) sum((x-a)^2)
>      optim(c(10,10), f, a=c(3,5))
$par
[1] 3.000375 4.999701
```



```
$value
[1] 2.299783e-07

$counts
function gradient
      65       NA

$convergence
[1] 0

$message
NULL
```

che conferma la soluzione come $x_1 = 3, x_2 = 5$.

La possibilità di minimizzare una funzione viene usata spesso in statistica, ad esempio per ottenere la stima di massima verosimiglianza. Consideriamo il problema di trovare la stima di massima verosimiglianza del modello $x_1, \ldots, x_n \sim$ Gamma(a, b). Innanzitutto, scriviamo una funzione per la log-verosimglianza negata

$$-\ell(a, b) = -na \log b - (a - 1) \sum_{i=1}^{n} \log x_i + b \sum_{i=1}^{n} x_i + n \log \Gamma(a)$$

```
gamneglik <- function(x,p){
     n <- length(x)
     a <- p[1]
     b <- p[2]
     mv <- -n*a*log(b) - (a-1)*sum(log(x)) +b*sum(x)+n*lgamma(a)
     mv
     }
```

Simuliamo un campione dalla distribuzione Gamma:

```
> x <- rgamma(100,7,.1)
```

da cui otteniamo

```
> optim(c(1,1),gamneglik,x=x)
$par
[1] 6.6376287 0.1010564

$value
[1] 460.5628

$counts
function gradient
      95       NA

$convergence
[1] 0

$message
NULL

Warning messages:
1: NaNs produced in: log(x)
```

```
2: NaNs produced in: log(x)
3: NaNs produced in: log(x)
4: NaNs produced in: log(x)
5: NaNs produced in: log(x)
6: NaNs produced in: log(x)
7: NaNs produced in: log(x)
```

Ossia, la stima di massima verosimiglianza è $\hat{a} = 6.64, \hat{b} = 0.101$, rispetto ai valori veri, $a = 7, b = 0.1$. I 'warnings' sono a causa della valutazione della verosimglianza al di fuori dello spazio parametrico; sarebbe meglio evitare questa difficoltà.

Ci sono altri argomenti disponibili per la funzione `optim`. In particolare, si può fornire il gradiente della funzione con l'effetto di aumentare la velocità del algoritmo. (Si veda `help(optim)` per una spiegazione, ed un esempio).

Possiamo esaminare la distribuzione campionaria della stima di massima verosimiglianza:

```
multneglik <- function(x,p){
          optim(p,gamneglik,x=x)$par
}
```

e

```
mvgamsim <- function(n,nsim,a,b){
par(mfrow=c(ceiling(length(n)/2),2),pch='.')
for(i in 1:length(n)){
         x <- matrix(rgamma(n[i]*nsim,a,b),nrow=nsim)
         mv <- apply(x,1,multneglik,p=c(1,1))
         plot(mv[1,],mv[2,],xlab='a',ylab='b')
       }
}
```

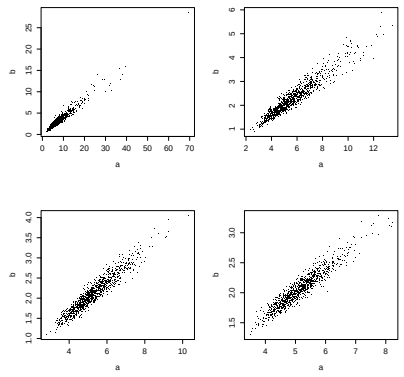

Figure 27:

Per esempio,

```
> mvgamsim(c(10,25,50,100),1000,5,2)
```


dimostra la tendenza della stima di massima verosimiglianza ad avere una distribuzione normale bivariata, con media uguale al valore vero del modello, all'aumentare delle dimensione del campione.

5.3 Esercizi

1. Trovare le radici del polinomio
$$5x^3 - x^2 + 1 = 0$$

2. (a) Scrivere una funzione che calcola il valore di $f(x) = \cos x - x$.
   (b) Fare un grafico di questa funzione per valori di x nell'intervallo $[-5, 5]$.
   (c) Verificare con questo grafico che c'è una soluzione unica dell'equazione
   $$\cos x = x$$
   nell'intervallo $[-5, 5]$.
   (d) Usare la funzione `locator` per ottenere dal grafico un'approsimazione della soluzione dell'equazione del punto c.
   (e) Trovare una soluzione con maggiore precisione tramite la funzione `uniroot`.

3. Usare la funzione `uniroot` per ottenere un intervallo di confidenza $100(1-\alpha)\%$ per λ nel modello Poisson(λ) basato sulle osservazioni $x_1, \ldots, x_n$ usando le proprietà della devianza.

4. Scrivere una funzione, basata sulla funzione svillupata in 3., che produce un insieme di campioni dalla distribuzione di Poisson(λ) e per ciascuno calcola un intervallo di confidenza $100(1-\alpha)\%$ per λ. In media, una proporzione $(1-\alpha)$ degli intervalli dovrebbe contenere il vero valore del parametro λ. Il comportamento empirico è simile? Dipende dalla dimensione dei campioni?

5. Usare la funzione `optim` per trovare il valore minimo della funzione $f(x, y) = (x^2 - xy)\exp(x - xy)$.

6. Scrivere una funzione che calcola la stima di massima verosimiglianza basata sulle osservazioni $x_1, \ldots, x_n$ dalla distribuzione Beta(a, b) con funzione di densità
$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$
per $0 \le x \le 1$ e parametri $a > 0,\ b > 0$.

7. Scrivere una funzione che simula campioni dalla distribuzione Beta (si usi `rbeta`), trova per ciascuno la stima di massima verosimiglianza (tramite la funzione scritta in 6.) e produce un grafico dei risultati. Come è la forma della distribuzione campionaria? Cambia con i valori veri del modello o con la dimensione dei campioni estratti dalla distribuzione Beta?

8. Scrivere una funzione (basata su una chiamata alla funzione `polyroot`) che ottiene le radici di un polinomio, e converte qualsiasi radice la cui parte immaginaria è zero (o, diciamo, meno di 10^{-6} distante da zero per risolvere il problema degli errori di arrondamento) a un valore reale (le funzioni `Re` e `Im` vengono usate per ottenere la parte reale e immaginaria rispettivamente. Anche la funzione `round` può essere utile.)

Appendice

R è un linguaggio statistico per effetuare analisi statistiche. Sono comprese funzioni per la sintesi e l'esplorazione dei dati, per la rappresentazione grafica, e la modellazione dei dati. Questo capitolo introduce i concetti di base.

Quando si lavora in **R** si creano oggetti nel 'workspace' corrente (a cui ci si referisce anche con 'image'). Ogni oggetto creato rimane nell'image fino a quando viene esplicitamente tolto o sostituito. Alla fine della sessione il workspace è perso a meno che non sia salvato. Si può salvare il workspace in qualsiasi momento facendo 'clic' sull'icona in alto nel panello di controllo.

I comandi scritti in **R** sono tenuti in memoria durante la sessione. Si può tornare ai comandi precedenti con la freccia 'su' (e 'giù per ritornare). Si possono usare i bottoni per copiare e incollare come al solito. Se in qualsiasi momento si vuole salvare il trascritto della sessione si fa 'clic' su 'File' e poi 'Save History'. Questo fornisce la possibiltà di salvare i comandi già scritti per farne uso più avanti. Come alternativa, si può copiare e incollare comandi tramite un editor di notepad (o qualunque editor).

Una sessione di **R** viene chiusa con il comando

```
> q()
```

Dopo questo viene richiesto se si vuole salvare l'image corrente. Se si risponde di no, l'image viene persa.

Oggetti e Aritmetica

R trattiene informazione e effettua comandi tramite oggetti. Gli oggetti più semplici sono *scalari*, *vettori* e *matrici*. Però ce ne sono tanti altri: ad esempio *elenchi* e *dataframe*. In usi avanzati di **R** può essere utile definire nuovi tipi di oggetti specifici per applicazioni particolari. Iniziamo con gli oggetti che vengono più utilizzati.

Un aspetto importante di **R** è che le operazioni tra oggetti di diversi tipi si comportano in modi diversi. Per esempio, il risultato di:[4]

```
> 4+6
```

dovrebbe essere

```
[1] 10
```

Quindi, **R** usa l'addizione tra scalari fa aritmetico scalare, ritornando il valore scalare di 10. (In realtà, **R** ritorna un vettore di lunghezza 1 - perciò il simbolo [1] denota il primo elemento del vettore).

Possiamo assegnare valori agli oggetti per farne uso dopo. Per esempio:

```
x<-6
y<-4
z<-x+y
```

Il calcolo è uguale al precedente, e il risultato è memorizzato in un oggetto con il nome `z`. Controlliamo i contenuti di un oggetto scrivendo il suo nome:

```
z
[1] 10
```

In qualsiasi momento possiamo vedere un elenco degli oggetti che esistono:

[4] Usiamo la convenzione di usare 'type font' per denotare comandi scritti in **R**. Però il segno > non viene scritto; esso denota il simbolo di 'prompt'.


```
> ls()
[1] "x"             "y"             "z"
```

Si noti che `ls` stesso è un oggetto. Il risultato che si ottiene scrivendo `ls` è una descrizione dei sui contenuti, in questo caso i comandi della funzione. L'uso di parentesi, `ls()`, assicura che la funzione viene effetuata e il suo risultato - in questo caso un elenco degli oggetti nell'image - mostrato.

Di solito una funzione si effettua su un oggetto. Per esempio,

```
> sqrt(16)
[1]  4
```

calcola la radice quadrata di 16. Per togliere oggetti dall'image si usa la funzione `rm`:

```
> rm(x,y)
```

per esempio.

I vettori vengono creati in diversi modi. Potremmo descrivere tutti gli elementi:

```
> z<-c(5,9,1,0)
```

Si noti l'uso della funzione `c` per *concatenare* diversi elementi. Questa funzione ha un uso più ampio: ad esempio i comandi

```
> x<-c(5,9)
> y<-c(1,0)
> z<-c(x,y)
```

conducono allo stesso risultato.

Le serie sono create in questa maniera:

```
> x<-1:10
```

oppure con l'uso della funzione `seq`, che permette una maggiore flessibilità. Per esempio:

```
> seq(1,9,by=2)
[1] 1 3 5 7 9
```

e

```
> seq(8,20,length=6)
[1]  8.0 10.4 12.8 15.2 17.6 20.0
```

Questi esempi dimostrano che tante funzioni in **R** hanno argomenti opzionali, in questo caso la lunghezza del passo o la dimensione totale della serie (non ha senso di usare ambedue.) Se queste opzioni sono omesse, **R** fa una scelta di default, in questo caso assumendo una lunghezza di passo uguale a 1. Dunque, ad esempio, anche

```
> x<-seq(1,10)
```

crea la serie da 1 a 10.

A questo punto facciamo il comento che esiste in **R** la risorsa *help*. Se non si sa usare una funzione, o non si sa le opzioni disponibili o i valori di default, si scrive `help(`**nome**`)` dove **nome** è il nome della funzione di interesse. Il risultato sarà una descrizione della funzione, i suoi argomenti, e (di solito) un esempio.

Un'altra funzione utile per la costruzione dei vettori è `rep` che permette la repetizione di oggetti. Per esempio

```
> rep(0,100)
  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

o

```
> rep(1:3,6)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

C'è anche una variazione nell'uso di questa funzione:

```
> rep(1:3,c(6,6,6))
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

che viene semplificata con

```
> rep(1:3,rep(6,3))
 [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

Come spiegato **R** si adatta agli oggetti sui quali opera. Per esempio:

```
> x<-c(6,8,9)
> y<-c(1,2,4)
> x+y
[1]  7 10 13
```

e

```
> x*y
[1]  6 16 36
```

dimostrano che **R** usa l'aritmetica componente per componente sui vettori. Se elementi sono misti, **R** prova a comportarsi nella maniera più logica. Per esempio,

```
> x<-c(6,8,9)
> x+2
[1]  8 10 11
```

sebbene richieda attenzione per assicurarsi che **R** fa quello che è inteso.

Due funzioni utili sono `length` che ritorna la dimensione di un vettore (cioè i numeri di elementi contenuti) e `sum` che calcola la somma degli elementi del vettore.

Esercizi

1. Definire

   ```
   > x<-c(4,2,6)
   > y<-c(1,0,-1)
   ```

   Decidere il risultato di:

   (a) `length(x)`
   (b) `sum(x)`
   (c) `sum(x^2)`
   (d) `x+y`
   (e) `x*y`
   (f) `x-2`

   (g) `x^2`

   Si usi **R** per controllare le risposte.

2. Determinare le serie seguenti e usare **R** per controllare le risposte:

   (a) `7:11`
   (b) `seq(2,9)`
   (c) `seq(4,10,by=2)`
   (d) `seq(3,30,length=10)`
   (e) `seq(6,-4,by=-2)`

3. Determinare il risultato dei seguenti comandi e farne un controllo in **R**:

   (a) `rep(2,4)`
   (b) `rep(c(1,2),4)`
   (c) `rep(c(1,2),c(4,4))`
   (d) `rep(1:4,4)`
   (e) `rep(1:4,rep(3,4))`

4. Usare la funzione `rep` per creare questi vettori:

   (a) 6,6,6,6,6,6
   (b) 5,8,5,8,5,8,5,8
   (c) 5,5,5,5,8,8,8,8

Strumenti di Sintesi ed Estrazioni di Sottoinsiemi

Supponiamo di aver raccolto dati da un esperimento e che sono stati messi nel vettore `x`:

```
> x<-c(7.5,8.2,3.1,5.6,8.2,9.3,6.5,7.0,9.3,1.2,14.5,6.2)
```

Statistiche che sintetizzano i dati possono essere le seguenti:

```
> mean(x)
[1] 7.216667
> var(x)
[1] 11.00879
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   6.050   7.250   7.217   8.475  14.500
```

che non richiedono spiegazione. Però, può essere che conosciamo che i primi 6 dati corrispondano a misure fatte su una macchina, mentre gli atri 6 vengano da un'altra macchina. Dunque, avrebbe senso sintetizzare i due insiemi di dati separatamente. Tale operazione richiede che estraiamo da `x` i due sottovettori rilevanti. Si usano le parentesi quadre nel seguente modo:

```
> x[1:6]
```

e

```
> x[7:12]
```

risultano dei sottovettori. Dunque,

```
> summary(x[1:6])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.100   6.075   7.850   6.983   8.200   9.300
> summary(x[7:12])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   6.275   6.750   7.450   8.725  14.500
```

Altri sottoinsiemi sono creati in maniera simile. Per esempio:

```
> x[c(2,4,9)]
[1] 8.2 5.6 9.3
```

I numeri negativi sono usati per *escludere* elementi. Per esempio

```
x[-(1:6)]
```

ha lo stesso effetto di `x[7:12]`.

Esercizi

1. Se `x<- c(5,9,2,3,4,6,7,0,8,12,2,9)` decidere il risultato dei seguenti. Usare **R** per controllare le risposte.
   (a) `x[2]`
   (b) `x[2:4]`
   (c) `x[c(2,3,6)]`
   (d) `x[c(1:5,10:12)]`
   (e) `x[-(10:12)]`
2. I dati `y<-c(33,44,29,16,25,45,33,19,54,22,21,49,11,24,56)` corrispondono alle vendite di latte in litri per 5 giorni in 3 negozi (i primi 3 valori corrispondono ai negozi 1,2 e 3 per lunedi, ecc.) Produrre una sintesi delle vendite per ciascun giorno della settimana e per ciascun negozio.

Matrici

Ci sono diversi modi per creare matrici in **R**. Forse quello più semplice è creare le colonne in vettori e poi incollarli per fare la matrice con il comando `cbind`. Ad esempio,

```
> x<-c(5,7,9)
> y<-c(6,3,4)
> z<-cbind(x,y)
> z
     x y
[1,] 5 6
[2,] 7 3
[3,] 9 4
```

La dimensione di una matrice viene controllata con il comando `dim`:

```
> dim(z)
[1] 3 2
```

cioè, tre righe e due colonne. Esiste anche un comando simile, `rbind`, che costruisce matrici incollando righe.

Le funzioni `cbind` e `rbind` si effettuano anche su matrici (purchè le dimensioni corrispondono) per creare matrici più grandi. Per esempio,


```
> rbind(z,z)
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
[4,]    5    7
[5,]    9    6
[6,]    3    4
```

Le matrici vengono costruite esplicitamente anche con la funzione `matrix`. Per esempio,

```
z<-matrix(c(5,7,9,6,3,4),nrow=3)
```

risulta identica alla matrice z nell'esempio precedente. Si noti che la dimensionse della matrice è determinata dalla dimensione del vettore e dalla richiesta che ci siano tre righe (`nrow=3`). Come alternativa avremmo potuto specificare il numero di colonne con l'argomento `ncol=2` (ovviamente non è necessario dare tutti e due). Si noti che la matrice viene riempita colonna per colonna. Per riempire la matrice riga per riga, si aggiunga l'opzione `byrow=T`. Ad esempio,

```
> z<-matrix(c(5,7,9,6,3,4),nr=3,byrow=T)
> z
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
```

Si noti che l'argomento `nrow` è stato accorciato a `nr`. Tali accorciamenti sono sempre disponibili purchè non siano introdotte ambiguità. Nel dubbio è meglio usare sempre il nome completo.

Come sempre, **R** cerca di interpretare operazioni su matrici nel modo più naturale. Per esempio, con il vettore z precedente, e

```
> y<-matrix(c(1,3,0,9,5,-1),nrow=3,byrow=T)
> y
     [,1] [,2]
[1,]    1    3
[2,]    0    9
[3,]    5   -1
```

otteniamo

```
> y+z
     [,1] [,2]
[1,]    6   10
[2,]    9   15
[3,]    8    3
```

e

```
> y*z
     [,1] [,2]
[1,]    5   21
[2,]    0   54
[3,]   15   -4
```

Si noti che la moltiplicazione è componente per componente anzichè essere la consueta moltiplicazione per matrici. Infatti, tale moltiplicazione convenzionale non esiste per y e z dato che le loro dimensioni non corrispondono. Definiamo

```
> x<-matrix(c(3,4,-2,6),nrow=2,byrow=T)
> x
     [,1] [,2]
[1,]    3    4
[2,]   -2    6
```

La moltiplicazione per matrici viene espressa tramite la notazione `%*%`:

```
> y%*%x
     [,1] [,2]
[1,]   -3   22
[2,]  -18   54
[3,]   17   14
```

Altre funzioni utili per matrici sono `t` per calcolare la trasposta e `solve` per calcolare l'inversa:

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

e

```
> solve(x)
           [,1]       [,2]
[1,] 0.23076923 -0.1538462
[2,] 0.07692308  0.1153846
```

Come nel caso di vettori è utile estrarre componenti di matrici. In questo caso potremmo volere estrarre elementi singoli, righe, colonne, o sottomatrici. Ancora il comando `[ ]` viene usato per estrarre elementi. Si considerino questi esempi:

```
> z[1,1]
[1] 5

> z[c(2,3),2]
[1] 6 4

> z[,2]
[1] 7 6 4

> z[1:2,]
     [,1] [,2]
[1,]    5    7
[2,]    9    6
```

Dunque, in particolare, è necessario specificare le righe e colonne richieste. Omettendo il numero per qualsiasi dimensione implica che ogni elemento di tale dimensione viene incluso.

Esercizi

1. Creare in **R** le matrici

$$x = \begin{bmatrix} 3 & 2 \\ -1 & 1 \end{bmatrix}$$

e

$$y = \begin{bmatrix} 1 & 4 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Calcolare le seguenti e usare **R** per controllare le risposte:

(a) `2*x`
(b) `x*x`
(c) `x%*%x`
(d) `x%*%y`
(e) `t(y)`
(f) `solve(x)`

2. Con `x` e `y` come nell'esempio precedente, calcolare l'effetto delle operazioni di estrazioni e usare **R** per controllare le risposte:

(a) `x[1,]`
(b) `x[2,]`
(c) `x[,2]`
(d) `y[1,2]`
(e) `y[,2:3]`

Il comando `attach`

R include dataset che servono come esempi. Per ottenere un elenco basta scrivere

```
> data()
```

Per ottenere un dataset particolare scrivere `data(`**dataset**`)` dove **dataset** è il nome del dataset che vuoi usare. Per esempio,

```
> data(trees)
```

Scrivendo

```
> trees[1:5,]
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
```

vediamo le prime 5 righe di questi dati, e vediamo che le colonne corrispondo a misure di 'girth', 'height' e 'volume' di alberi (in realtà ciliegi: si veda `help(trees)`).

Allora, se vogliamo lavorare sulle colonne di questi dati, potremmo usare la tecnica di estrazione: per esempio, `trees[,2]` ci dá tutti i valori di altezza. Questo è un pò noioso però, e sarebbe più facile se potessimo riferire alle altezze più esplicitamente. Otteniamo tale possibilità con il comando `attach`:

```
> attach(trees)
```

Effettivamente, tale comando fa un directory per i contenuti dell'oggetto. Allora, scrivendo il nome di un oggetto, per esempio `Height`, rende **R** a cercare questo oggetto anche dentro l'oggetto `trees`. Dunque, per esempio,

```
> mean(Height)
[1] 76
```

e

```
> mean(trees[,2])
[1] 76
```

sono sinonomi, sebbene sia più facile capire l'effetto dalla prima versione. In realtà `trees` è un oggetto di tipo `dataframe`. Tale oggetto è essenzialmente una matrice con colonne che hanno un nome. Però, un dataframe può anche contenere variabili non numeriche. A causa di questo c'è un terzo modo per estrarre il vettore di altezze:

```
> trees$Height
```

che non richiede di avere già usato il comando `attach`.

Esercizi

1. Fare `attach` al dataset `quakes` e produrre una sintesi statistica delle variabili `depth` e `mag`.
2. Fare `attach` al dataset `mtcars` e scoprire le medie dei pesi e del consumo di benzina per questi veicoli (scrivere `help(mtcars)` per una spiegazione delle variabili disponibili).

La funzione `apply`

È possibile scrivere cicli in R, ma è meglio evitare tale uso se possibile. Una situazione comune accade quando vogliamo effettuare una funzione ad ogni riga o colonna di una matrice. Ad esempio, potremmo voler calcolare la media di ciascuna variabile del dataset `trees`. Ovviamenete, una possibilità sarebbe effetuare la funzione su ogni colonna individuamente, ma questa è noiosa, sopratutto se ci sono tante colonne. Invece, la funzione `apply` rende le cose più semplici. Ad esempio

```
> apply(trees,2,mean)
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

ha l'effetto di calcolare la media di ciascuna colonna (dimensione 2) dell'oggetto `trees`. Sostituendo 2 con 1 si avrebbe come risultato il calcolo della media per ciascuna riga.

Questa tecnica può essere effettuata su qualsiasi funzione. Argomenti supplementari per la funzione devono essere specificati come argomenti nell'uso di `apply`.

Esercizi

1. Ripetere le analisi dei dataset `quakes` e `mtcars` utilizzando la funzione `apply` per semplificare i calcoli.
2. Se
$$y = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & -1 \end{bmatrix}$$
quale è il risultato di `apply(y[,2:3],1,mean)`? Controllare il risultato in **R**.

Computazione e Simulazione

Tanti calcoli noiosi che una volta avrebbero richiesti l'uso di tabelle statistiche sono effettuati semplicimente in **R**. Questo è utile nei calcoli di intervalli di confidenza ecc. Facciamo un esempio con la distribuzione normale. Esistono funzioni in **R** per calcolare le funzioni di densità, di ripartizione e dei quantili (l'inversa della funzione di ripartizione). Questi sono, rispettivamente, `dnorm`, `pnorm` e `qnorm`. a differenza di quando si usano tabelle non c'è bisogno di standardizzare le variabili. Per esempio, se $X \sim N(3, 2^2)$, allora

```
> dnorm(x,3,2)
```

calcola la funzione di densità nei valori contenuti nel vettore `x` (si noti, `dnorm` assume media 0 e deviazione standard 1 a meno che questi sono specificati). Si noti anche che la funzione richiede la deviazione standard invece della varianza. Ad esempio

```
> dnorm(5,3,2)
[1] 0.1209854
```

calcola la densità della distribuzione $N(3, 4)$ in $x = 5$. Inoltre

```
> x<-seq(-5,10,by=.1)
> dnorm(x,3,2)
```

calcola la funzione di densità in passi di 0.1 sul dominio $[-5, 10]$. Le funzioni `pnorm` e `qnorm` funzionano in un modo identico - usare `help` per maggiore informazione.

Funzioni simili esistono per altre distribuzioni. Per esempio, `dt`, `pt` e `qt` per la distribuzione t, sebbene in questo caso sia necessario specificare i gradi di libertà invece della media e deviazione standard. Altre distribuzioni disponibili includono la binomiale, l'exponenziale, la Poisson e la gamma.

Una tecnica importante per molte applicazioni statistiche è la simulazione di dati da una distribuzione specificata. **R** permette la simulazione da un'ampia collezione di distribuzioni, usando una forma simile a quella precedente. Ad esempio, per simulare 100 osservazioni dalla distribuzione $N(3, 4)$ scriviamo

```
> rnorm(100,3,2)
```

In maniera simile, `rt`, `rpois` vengono utilizzati per simulare dalle distribuzioni t e Poisson ecc.

Esercizi

1. Sia $X \sim N(2, 0.25)$. Denotare con f and F le funzioni di densità e di ripartizione di X rispettivamente. Usare **R** per calcolare
   (a) $f(0.5)$
   (b) $F(2.5)$
   (c) $F^{-1}(0.95)$ (si ricordi che F^{-1} è la funzione dei quantili)
   (d) $\Pr(1 \leq X \leq 3)$
2. Ripetere l'esercizio precedente nel caso che X ha la distribuzione t con 5 gradi di libertà.
3. Usare la funzione `rpois` per simulare 100 valori da una distribuzione Poisson con parametro a scelta. Produrre una sintesi statistica dei valori simulati e controllare che la media e varianza siano coerenti con i valori veri della popolazione.
4. Ripetere l'esercizio precedente sostituendo `rpois` con `rexp`.

Grafici

R include tante opportunità per produrre grafici di alta qualità. Una di queste, prima di creare i grafici, è di dividere la pagina in sottoparti per permettere la costruzione di un numero di grafici nella stessa pagina. Per esempio:

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

crea una finestra di grafici con 2 righe e 2 colonne. Con questa scelta le sottofinestre sono riempite riga per riga. Si usa `mfcol` invece di `mfrow` per riempire colonna per colonna. La funzione `par` è generica per la specificazione di parametri grafici. Esistono tante opzioni: si veda `help(par)`.

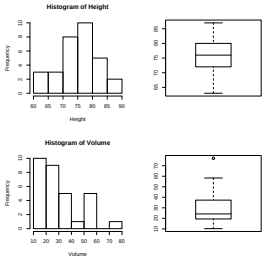

Figure 28: Altezze e volumi di alberi

Dunque, per esempio,

```
> par(mfrow=c(2,2))
> hist(Height)
> boxplot(Height)
> hist(Volume)
> boxplot(Volume)
> par(mfrow=c(1,1))
```

produce Fig. 28. Si noti l'ultimo uso di `par` per ritornare alla dimensione standard della finestra grafica.

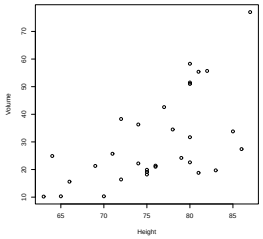

Figure 29: Scatterplot di altezze e volumi di alberi

Possiamo anche fare un grafico di una variabile contro un altra tramite la funzione `plot`:


```
> plot(Height,Volume)
```

Si veda Fig. 29.

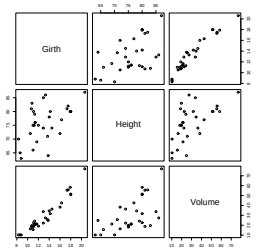

Figure 30: Scatterplot matrice dei dati `trees`

R può anche produrre un scatterplot di matrici attraverso la funzione `pairs`:

```
> pairs(trees)
```

Si veda 30. Come altre funzioni, `plot` è 'object-specific': si comporta in una maniera che dipende dall'oggetto al quale viene effettuato. Ad esempio, se l'oggetto è una matrice, `plot` è identico a `pairs`: provare `plot(trees)`. Per alcune possibilità diverse provare:

```
> data(nhtemp)
> plot(nhtemp)

> data(faithful)
> plot(faithful)

> data(HairEyeColor)
> plot(HairEyeColor)
```

Tutte le funzioni dei grafici permettono di usare vari argomenti opzionali che controllano, per esempio, colori, caratteri, nomi degli assi, titoli ecc. Le funzioni `points`, `lines` e `curve` sono utili per aggiungere punti e rette rispettivamente a un grafico corrente. La funzione `abline` è utile per aggiungere una retta con intercetta e gradiente scelti.

Per stampare un grafico basta mettere il cursore sopra la finestra grafica e premere il tasto destro del mouse. Questo dovrebbe aprire un menú che include l'opzione 'print'. C'è anche l'opzione per salvare ('save') la figura in diversi formati, per esempio nella forma di un file postscript.

Esercizi

1. Usare

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

   o qualcosa di simile, per generare dei dati. Produrre un grafico singolo che include un istogramma e un boxplot dei dati. Modificare i nomi degli assi e il titolo del grafico in una maniera appropriata.

2. Scrivere il seguente

```
> x<- (-10):10
> n<-length(x)
> y<-rnorm(n,x,4)
> plot(x,y)
> abline(0,1)
```

Individuare l'effetto di ciascun comando e interpretare il grafico che ne risulta.

3. Scrivere il seguente:

```
> data(nhtemp)
> plot(nhtemp)
```

Questo produce un grafico della serie storica di misure della temperature media annuale a New Hampshire, U.S.A.

4. L'esempio precedente ha dimostrato che la funzione `plot` ha un risultato che dipende dal tipo di oggetto al quale viene applicato (l'oggetto `nhtemp` è di classe `time series`.) Più in generale, potremmo avere le osservazioni annuali in un vettore, ma avremmo bisogno di creare il grafico della serie storica noi stessi. Infatti, si scriva

```
> temp<-as.vector(nhtemp)
```

che crea il vettore `temp` che contiene solo le temperature. Facciamo un grafico che è simile al grafico della serie storica scrivendo

```
> plot(1912:1971,temp)
```

sebbene il grafico comprenda punti anzichè rette. Invece, per unire i dati con rette, si scriva

```
> plot(1912:1971,temp,type='l')
```

Per ottenere sia punti che rette si usi l'argomento `type='b'`.

Funzioni

Un aspetto importante di **R** è la facilità di estendere il linguaggio scrivendo nuove funzioni. Per esempio, non esiste una funzione che calcola esplicitamente la deviazione standard di un vettore di valori. Ma possiamo definire tale funzione nel seguente modo:

```
> sd <- function(x) sqrt(var(x))
```

Allora, per esempio,

```
x<-c(9,5,2,3,7)

> sd(x)
[1] 2.863564
```

Per modificare la funzione il comando `fix(sd)` apre un editor dal quale la funzione può essere cambiata. Tale comando può essere usato anche per la definizione della funzione: questo modo è preferito per la costruzione di funzioni complesse. Se la funzione non esiste di già l'editor apre lo scheletro


```
function ()
{
}
```

L'editor permette una costruzione e modificazione facile della funzione. Le parentesi permettono che vengano inclusi tutti i comandi necessari per la funzione. Ad esempio, potremmo scrivere

```
fix(several.plots)
```

e definire

```
several.plots<-function(x){
par(mfrow=c(3,1))
hist(x[,1])
hist(x[,2])
plot(x[,1],x[,2])
par(mfrow=c(1,1))
apply(x,2,summary)
}
```

L'ultima riga della funzione specifica il risultato della funzione. Le righe precedenti sono passi intermedi, e non sono incluse nel risultato della funzione (sebbene i grafici, se inclusi, siano creati nella finestra grafica). La funzione `several.plots` - che ha senso solo se `x` è una matrice con due colonne - produce un istogramma di ciascuna colonna ed un scatterplot dei valori delle due colonne. Infine, ritorna una sintesi dei valori.

Per creare una funzione che ritorna due oggetti (forse di diversi tipi - vettori, matrici, ecc.) è utile usare il comando `list`. Per esempio, la funzione

```
sdvar<-function(x){
list(sd=sd(x),var=var(x))
}
```

ritorna sia la deviazione standard che la varianza degli elementi del vettore `x`. Il risultato è un oggetto del tipo `list`. In questo esempio gli elementi del 'list' sono vettori con nomi `sd` e `var`. Con il valore precedente di `x`:

```
> sdvar(x)
$sd
[1] 2.863564

$var
[1] 8.2
```

I componenti del `list` possono essere estratti individuamente. Per esempio

```
> ris<-sdvar(x)
```

mette il risultato in un oggetto (che prende il tipo `list`). Allora, per estrarre la componente che contiene la deviazione standard:

```
> ris$sd
[1] 2.863564
```

Per stampare i contenuti di una funzione, usare `fix` per aprire un editor che contiene i comandi della funzione e fare clic su 'file' e 'print'.

Dunque, per esempio

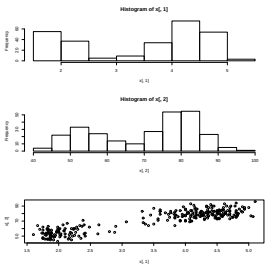

Figure 31: Output of `several.plots`

```
> several.plots(faithful)
       eruptions waiting
Min.       1.600    43.0
1st Qu.    2.163    58.0
Median     4.000    76.0
Mean       3.488    70.9
3rd Qu.    4.454    82.0
Max.       5.100    96.0
```

che produce anche Fig. 31.

Esercizi

1. Scrivere una funzione che prenda come argomento due vettori, `x` e `y`, produca un scatterplot, e calcoli il coefficiente di correlazione (tramite la funzione `cor(x,y)`).

2. Scrivere una funzione che si effettui con un vettore $(x_1, \ldots, x_n)$ e calcola sia $\sum x_i$ che $\sum x_i^2$. (Si ricordi l'uso della funzione `sum`).

Altre cose

Ci sono tante altre possibilità in **R**. Queste comprendono:

1. Funzioni per la stima di modelli statistici come modelli lineari e modelli lineari generalizzati.

2. Funzioni per il lisciamento dei dati.

3. Funzioni per l'ottimizzazione e la soluzione di equazioni.

4. Facilità nel creare programmi con cicli usando comandi del tipo `if` e `while`.

5. Tecniche per visualizzare dati tridimensionale.

Esiste anche la possibilità di aggiungere librerie di funzioni con utilità specifiche. Scrivendo

```
> library()
```

si ottiene un elenco ed una descrizione delle librerie disponibili. Scrivendo


```
> library(libraryname)
```

dove `libraryname` è il nome della libreria richiesta permette accesso alle sue funzioni.

Ottenendo aiuto

Questo capitolo è di base. Maggiore aiuto viene trovato da:

1. Il sistema di `help` incluso nel pachetto;

2. I manuali inclusi nel pachetto: fai clic su 'help' e poi su 'manuals'. Il manuale 'Introduction to R' è utilissimo.

3. Libri: esistono tanti che includono l'uso di **R** (e/o il linguaggio simile di S-Plus).

Esercizi Generali

1. Si scriva una funzione che calcola il coefficiente di variazione degli elementi di un vettore (definito come il rapporto fra deviazione standard e media).

2. Si scriva una funzione della forma `rplot(n,lambda)` che simula un campione di dimensione `n` dalla distribuzione di Poisson con media `lambda` e produce sia una tabella dei valori simulati che un'istogramma. (La funzione `table` è utile per il calcolo della tabella).

3. La seguente funzione produce un grafico della funzione di probabilità della distribuzione di Poisson.

   ```
   pgraf <- function(x,lambda){
      prob <- dpois(x,lambda)
      plot(x,prob,type='h',ylab='prob')
   }
   ```

   Si scriva questa funzione e si la usi per produrre grafici della distribuzione di Poisson con vari valori della media `lambda`.

4. Si ripeta l'esercizio 2., con la distribuzione binomiale, con parametri n e p anzichè la distribuzione di Poisson. (Si noti: la funzione `dbinom` calcola la probabilità in questo caso).

5. Si scriva una funzione come `pgraf`, ma con un ciclo, che risulta in un grafico della distribuzione di Poisson per ciascun valore della media incluso nel vettore `lambda`.

6. La stima naturale del parametro λ della distribuzione Poisson(λ) basata su un campione $x_1, \ldots, x_n$ è la media della campione $\bar{x}$, il cui errore standard è $\sqrt{\bar{x}/n}$. Scrivere una funzione che prende come argomento il vettore di valori, $x_1, \ldots, x_n$, e produce un oggetto di tipo 'list' che contiene sia la stima che l'errore standard.

7. Si modifichi la funzione in 6. in modo che produca anche un grafico che contiene:

   (a) Un istogramma dei dati;

   (b) Un grafico delle probabilità stimate.