Introducción a R

Klaus Langohr

Departament d'Estadística i Investigació Operativa
Universitat Politècnica de Catalunya

Barcelona, septiembre 2016

Índice general

Índice de tablas

Índice de figuras

Actividad 1

Los primeros pasos

Contenido

Básicamente, R es un lenguaje que permite manipular objetos estadísticos y crear gráficos de alta calidad. Es a la vez un entorno interactivo y un lenguaje de programación interpretado con funciones orientadas a objetos. En esta primera actividad trataremos los principales elementos del lenguaje R contestando las siguientes preguntas:

- ¿Cómo obtener e instalar R?
- ¿Cuáles son las características de R?
- ¿De qué forma podemos trabajar usando R?

Nota: Buena parte del contenido de las secciones 1.2 y 1.3 se entenderán mejor una vez se tenga algo de experiencia con R. Así que, para comenzar a trabajar con el *software*, se recomienda pasar directamente del Apartado 4 de la sección 1.2.1 a la Actividad 2.

1.1. Instalación de R

R es un *software* libre para el análisis estadístico de datos que utiliza el mismo lenguaje de programación, el lenguaje S, que anteriormente el programa de análisis estadístico comercial S-Plus (SolutionMetrics Pty Ltd; http://www.solutionmetrics.com.au/products/splus/default.html). Desde su creación en la segunda mitad de los años noventa ha ganado cada vez más popularidad debido a que a) su adquisición es gratuita, b) se pueden llevar a cabo los mismos análisis estadísticos que con S+, y c) estadísticos de todo el mundo contribuyen con paquetes que permiten realizar análisis cada vez más específicos y sofisticados.
Para instalar R, seguid los siguientes pasos:

- Abrid la página web de R: `http://www.r-project.org/`.
- Haced clic en 'CRAN' y a continuación escoged uno de los servidores (*mirrors*) de CRAN (*Comprehensive R Archive Network*).

- Según vuestro sistema operativo, haced clic en Linux, MacOS X o Windows y seguid las instrucciones correspondientes.
- Si usáis Windows, haced clic en 'base' y a continuación en 'Download R 3.x.y for Windows', en donde `x` e `y` indican la versión actual de R. Al escribir el presente documento, ésta es la versión R-3.3.1.
- Ejecutad el fichero desde la carpeta en la cual fue guardado y seguid las instrucciones de instalación.

De esta manera se instala la versión básica de R que contiene una serie de paquetes. Si en algún momento queréis instalar otros de las más de 6800 paquetes contribuidos, haced lo siguiente (después de abrir R): haced clic en 'Paquetes' en la barra de herramientas y a continuación en 'Instalar paquete(s)...'. En la ventana que se abre, se ha de escoger primero uno de los servidores de CRAN y después el (los) paquete(s) deseado(s). De la misma manera se puede ejecutar la función `install.packages` y especificar el paquete a instalar como argumento de la misma, por ejemplo:

```
> install.packages("doBy")                  # Instala el paquete doBy
> install.packages(c("catspec", "FHtest")) # Instala 2 paquetes: catspec y FHtest
```

1.2. Generalidades de R

1.2.1. Miscelánea

1. Abriendo R, en el modo por defecto, se abre una sola ventana, la consola o ventana de comandos de R, en la cual se pueden entrar los comandos y donde se verán los resultados de los análisis.

   El indicador o *prompt* del sistema es el signo >. En un principio, cada instrucción acaba con un *Enter* que indica su ejecución. Si no utilizamos el punto y coma e intentamos ejecutar la orden con *Enter*, el intérprete de comandos probará de traducir la instrucción y, si es correcta, la ejecutará; si no es correcta, mostrará un mensaje de error, y si es incompleta, quedará a la espera de completar la orden en la línea siguiente mostrando como indicador el signo `+`. En la práctica, se utiliza *Enter* para acabar la instrucción o para dividir la línea, si la instrucción es larga.

   El signo # indica la introducción de comentarios. Por ejemplo, la siguiente instrucción sumará 3 y 4 e ignorará el comentario:

   ```
   > 3 + 4     # es igual a 7

   [1] 7
   ```

   La tecla `Esc` permite interrumpir la edición o ejecución en curso y para recuperar instrucciones ejecutadas anteriormente en la misma sesión se puede utilizar la tecla de movimiento del cursor ↑. Para más ayuda sobre el uso de la consola:
   Ayuda ▶ Consola

2. El nombre de un objeto de R, sea un vector, una lista, una función, etc., puede ser cualquier cadena alfanumérica formada por letras (teniendo en cuenta que **R distingue entre mayúsculas y minúsculas**), dígitos del 0 al 9 (no como primer carácter) y el signo . (punto), sin limitación de número de caracteres. Por ejemplo, `Exp1289, muestra.ini` o `muestra.ini.ajuste` son nombres válidos.

   R tiene palabras reservadas como son los nombres de las instrucciones utilizadas en el lenguaje de programación (`break, for, function, if, in, next, repeat, return, while`) y los de las funciones incorporadas en el propio entorno del programa, que no se pueden usar como identificador de objetos. En el caso de intentar redefinir una función ya utilizada por el programa, R advierte de la duplicidad de definiciones.

3. R es un lenguaje a través de funciones. Las instrucciones básicas son expresiones o asignaciones. Para realizar una asignación se pueden utilizar los signos `<-` , `->` y `=`.

```
> n <- 5 * 2 + sqrt(144)
> m = 4^-0.5
> n + m -> p
```

   Para visualizar el contenido de un objeto sólo es necesario escribir su nombre. Si el objeto es una función se mostrará en pantalla el programa que la función ejecuta.

```
> n

[1] 22

> m; p

[1] 0.5

[1] 22.5

> (x <- log(7))     # El uso de (...) hace que se muestre el valor de x

[1] 1.94591

> log

function (x, base = exp(1))  .Primitive("log")
```

4. Los comandos `objects()` y `ls()` visualizan el listado de objetos presentes en el actual espacio o área de trabajo (*workspace*).

```
> objects()

[1] "i"   "m"   "n"   "out" "p"   "x"

> ls()
```

```
[1] "i"   "m"   "n"   "out" "p"   "x"
```

5. La mayoría de los paquetes disponibles en la versión local de R han de ser cargados antes de que se los pueda utilizar. Por ejemplo, para cargar el paquete `survival`, que contiene funciones para el análisis de supervivencia, se puede ejecutar la siguiente instrucción

   ```
   > library(survival)
   ```

   o escoger `survival` desde la barra de herramientas:

   Paquetes ▶ Cargar paquete...

6. La instrucción `search()` devuelve la ruta de búsqueda de R:

   ```
   > search()
   ```

   ```
    [1] ".GlobalEnv"        "package:grDevices" "package:datasets"
    [4] "package:LEpack"    "package:Hmisc"     "package:Formula"
    [7] "package:survival"  "package:splines"   "package:graphics"
   [10] "package:utils"     "package:stats"     "package:lattice"
   [13] "package:grid"      "package:methods"   "Autoloads"
   [16] "package:base"
   ```

   en donde `.GlobalEnv` se refiere al área de trabajo actual y `package:xxx` a los paquetes cargados. Es decir, R busca cualquier objeto primero en el área de trabajo actual y después en los paquetes cargados según el orden de esta lista.

7. Cuando se hace referencia a algún fichero de disco debe utilizarse la dirección entre comillas y usar barra (/) entre subcarpetas. Por ejemplo:

   ```
   > save.image("C:/Archivos de programa/R/nombredearchivo.RData")
   ```

   La alternativa es el uso de la doble barra inversa:

   ```
   > save.image("C:\\Archivos de programa\\R\\nombredearchivo.RData")
   ```

8. Es posible abrir varias sesiones de R y trabajar simultáneamente en ellas. Esto puede ser útil por ejemplo, si estamos ejecutando un programa de simulación que dura bastante tiempo. Se puede ejecutar la simulación en una ventana de R y estar trabajando al mismo tiempo en otra.

9. Se pueden cambiar distintos aspectos de la consola de R, por ejemplo su formato, color o fuente. Para ello hay que ir a la barra de herramientas:

   Editar ▶ Preferencias de la interface gráfica

   Los posibles cambios pueden ser guardados para futuras sesiones guardando el fichero `Rconsole` en la carpeta 'etc' dentro de las carpetas locales del programa.

10. Puede ser de utilidad etiquetar los objetos de una sesión de R para acordarnos posteriormente de su contenido. Esto es posible con la función `comment`:

```
> x <- 1:4
> comment(x)

NULL

> comment(x) <- "Los números naturales de 1 a 4"
> x

[1] 1 2 3 4

> comment(x)

[1] "Los números naturales de 1 a 4"
```

1.2.2. Buscando ayuda

1. R dispone de una ayuda muy completa sobre todas las funciones, procedimientos y elementos que configuran el lenguaje. También dispone de manuales a los cuales se puede acceder vía la barra de herramientas de R:

   Ayuda ▶ Manuales (en PDF) ▶ ...

2. Además de las opciones de menú propias de R, desde la ventana de comandos se puede acceder a información específica sobre las funciones de R con el comando `help` o mediante '?':

   ```
   > help(objects)
   > help(log)
   > ?ls
   ```

3. El comando `library()` abre una ventana con información sobre los paquetes (paquetes) instaladas en R. Para obtener más información sobre estos paquetes, se puede utilizar las funciones `library` y `help` conjuntamente:

   ```
   > library(help = "foreign")
   ```

   Otra posibilidad para obtener la misma información es accediendo a ella desde la barra de herramientas

   Ayuda ▶ Ayuda Html

   y después, en la página web que se abre, hacer clic en 'Packages'. A continuación hacer clic en el paquete correspondiente.

4. También es posible obtener ayuda sobre diferentes temas mediante la función `help.search`. R buscará ayuda sobre el tema escogido en todos los paquetes instalados. Por ejemplo:

   ```
   > help.search("logistic regression")
   > help.search("R help")
   ```

5. La función `RSiteSearch` permite buscar palabras de interés en todas las páginas de ayuda existentes, es decir tanto entre los paquetes instalados en el ordenador como entre los paquetes disponibles en el *CRAN*. Por ejemplo, si estamos buscando información sobre la prueba de Hosmer Lemeshow en R, podemos ejecutar la siguiente instrucción

   ```
   > RSiteSearch("Hosmer Lemeshow test")
   ```

6. Además existen muchas listas de ayuda de correo para R. Se recomiendan, por un lado, las diferentes listas que se presentan en la página web de R (hacer clic sobre *Mailing lists*) en http://www.r-project.org. Por otro lado puede ser muy útil apuntarse a la lista de ayuda en castellano en la página web https://stat.ethz.ch/mailman/listinfo/r-help-es.

1.2.3. Cerrando la sesión de R

1. Durante una sesión de R se puede guardar el histórico de todas las instrucciones ejecutadas hasta el momento desde la barra de herramientas:

   Archivo ▶ Guardar Histórico...

   El fichero guardado es un fichero de formato ASCII que puede ser editado con otro *software* si interesa. Además es posible cargar el histórico en otra sesión de R mediante (en la barra de herramientas):

   Archivo ▶ Cargar Histórico...

   De esta manera se pueden volver a ejecutar los comandos de la sesión anterior.

2. En cualquier momento de una sesión de R se puede guardar su contenido, es así llamado área de trabajo o *workspace*. Esto es muy recomendable si queremos volver a utilizar los objetos de R en uso. La función para guardar el área de trabajo es `save.image()`. Otra posibilidad es usar el cuadro de dialogo correspondiente accesible vía la barra de herramientas:

   Archivo ▶ Guardar área de trabajo...

   Si queremos guardar solamente algunos de los elementos del área de trabajo, por ejemplo los objetos `x` e `y`, tenemos dos posibilidades: o eliminar primero los demás objetos con la función `rm()` y después usar la función `save.image()`, o usar la función `save`:

   ```
   > save(x, y, file="nombredearchivo.RData")
   ```

   Podemos abrir un espacio de trabajo con la función `load()` o yendo a la barra de herramientas:

   Archivo ▶ Cargar área de trabajo...

   Notad que durante una sesión se pueden cargar diferentes áreas de trabajo, pero que R no avisa si contienen objetos con el mismo nombre.

3. Se puede salir de R ejecutando la orden `q()`. Antes de cerrarse, R le pregunta al usuario si quiere guardar el actual área de trabajo. Si hemos guardado el espacio de trabajo anteriormente, no hace falta volverlo a hacer. En cambio, si contestamos con 'Sí', se guardará

una copia de seguridad del área de trabajo actual bajo el nombre `.RData` en la carpeta de trabajo actual conjuntamente con el histórico de la sesión.

1.3. Formas de trabajar en R

1. Hemos visto anteriormente que, trabajando en la consola de R, es posible navegar entre los comandos ejecutados anteriormente mediante las teclas ↑ y también ↓. Sin embargo, si el usuario quiere (volver a) ejecutar una serie de comandos, es más práctico y eficiente ejecutarlos desde una ventana *script* que se puede abrir desde la barra de herramientas mediante:

   Archivo ▶ Nuevo script

   En las ventanas *script* se pueden entrar varios comandos, separados o por ';' o por líneas, que se pueden ejecutar conjuntamente yendo a la barra de herramientas:

   Editar ▶ Ejecutar todo

   Si se desea ejecutar solamente una selección de los comandos de la ventana *script*, hay que marcar los mismos y ejecutarlos mediante 'Ctrl-R' o la tecla F5. Las instrucciones no se borrarán y los resultados aparecerán en la ventana de comandos. Los *scripts* se pueden guardar y utilizar en cualquier otro momento siendo '.R' el postfix por defecto.

2. Mediante la función `source()` se puede cargar un *script* de R entero, por ejemplo:

   ```
   > source("C:/Archivos de programa/R/script.R")
   ```

   Lo mismo se consigue desde la barra de herramientas:

   Archivo ▶ Interpretar código fuente R...

3. Por defecto, todos los resultados aparecen en la consola de R. Existe, sin embargo, la posibilidad de enviar los resultados directamente a un fichero externo (de formato ASCII) utilizando la función `sink()`. Veamos un ejemplo:

   ```
   > sink("C:/Mis documentos/prueba.txt")    # Crea o sobrescribe el fichero prueba.txt
   > n <- 5 * 2 + sqrt(144)
   > n
   > sink()

   > n

   [1] 22
   ```

   En la pantalla aparecen solamente las instrucciones, mientras se escriben los resultados en el fichero `prueba.txt`. Una vez ejecutado la instrucción `sink()`, los resultados aparecen de nuevo en la consola de R.

4. Mientras la función `getwd` devuelve el directorio de trabajo actual, la función `setwd` puede ser usada para cambiarlo:

```
> getwd()

"C:/Documents and Settings/klangohr/Mis documentos"

> setwd("G:/Rfiles")
> getwd()

"G:/Rfiles"
```

También es posible cambiar la carpeta de trabajo en el cuadro de dialogo que se abre mediante (barra de herramientas)

Archivo ▶ Cambiar dir...

Para ver el contenido de la carpeta de trabajo se puede usar la función dir: `dir()`.

5. Existen diferentes editores que pueden facilitar el trabajo con R. Uno de ellos es el Tinn-R. Se trata de un *software* libre que se puede bajar desde la página web http://www.sciviews.org/Tinn-R/. En este editor se pueden abrir y editar diferentes *scripts* y se les puede enviar a ejecución en R. La ventaja de este editor sobre la ventana de los *scripts* en R es que ofrece una serie de opciones no existentes en R. Por ejemplo, se puede comprobar rápidamente si existen paréntesis sin cerrarse.

   Otra plataforma gratuita para R cada vez más popular y con características similares a las del Tinn-R es el RStudio (http://rstudio.org/). La Figura 1.1 a continuación muestra una imagen donde se puede ver que se pueden organizar fácilmente las ventanas para los scripts, las salidas de R y los gráficos.

6. El paquete `Rcmdr`, el *R Commander*, también puede facilitar el trabajo con R, ya que usa un sistema de ventanas (parecido al SPSS) que lo hace más amigable para el usuario. No obstante, es mucho menos flexible ya que limita el uso de opciones de muchas funciones. Para activarlo el *R Commander* hay que instalar el paquete `Rcmdr` y después cargarlo mediante `library(Rcmdr)` [1]. Una buena referencia para aprender a usar el *R Commander* es el libro de diferentes profesores de la Universidad de Cádiz [2].

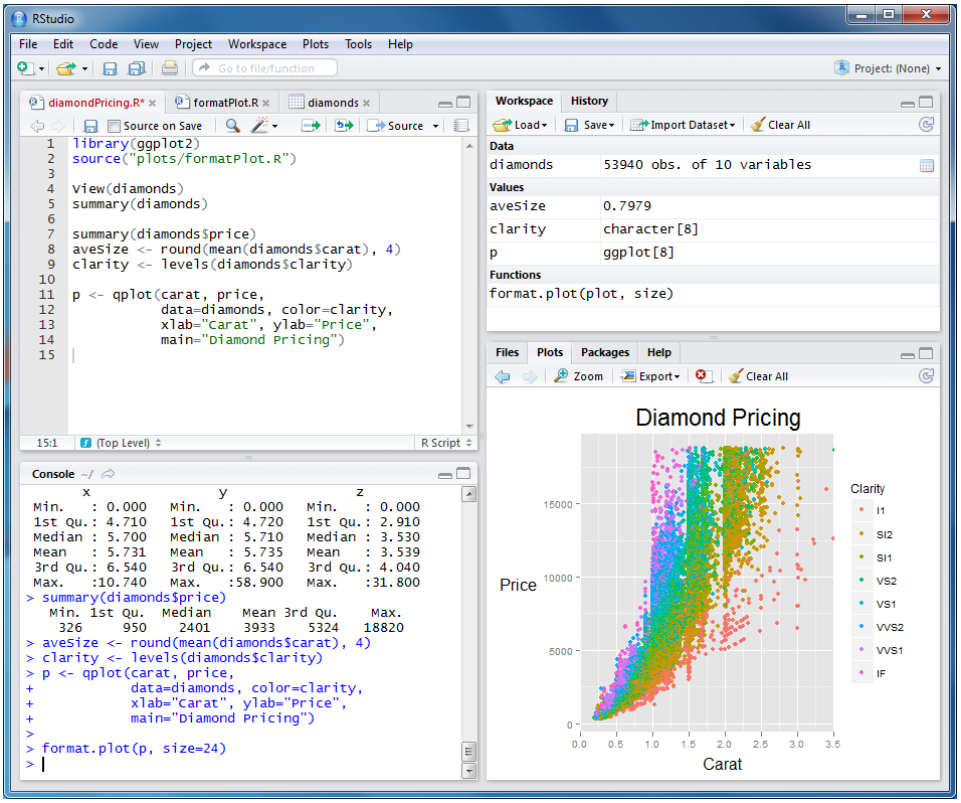

Figura 1.1: Imagen de la plataforma RStudio (Fuente: http://rstudio.org/)

Actividad 2

Vectores, matrices, listas y *data frames*

Contenido

En esta actividad se presentan diferentes objetos básicos de R incluyendo vectores, listas y *data frames*. Además se considera la importación de datos desde ficheros externos y de distintos formatos.

2.1. Vectores y matrices

2.1.1. Creación y manipulación de vectores

1. R está diseñado de forma que la mayoría de operaciones y de funciones están definidas con carácter vectorial. Es conveniente pues, en la medida de lo posible, explotar dicha posibilidad a fin de agilizar el tiempo de computación. La función principal para definir un vector es a través de sus componentes, con la función `c()`. Para referirnos a la componente enésima del vector `v` escribiremos `v[n]`, por ejemplo:

```
> v <- c(2, 1, 3, 4)
> v

[1] 2 1 3 4

> (w <- c(0, 2, -2, 1))

[1]  0  2 -2  1

> w[3]

[1] -2
```

2. Se pueden modificar los elementos de vectores, ampliar éstos o borrarlos; ver el siguiente ejemplo, en donde la función `length(vector)` muestra el número de componentes de `vector` y `NA` identifica elementos faltantes (*missings*):

```
> w[4] <- 7
> w[6]

[1] NA

> w[6] <- -4
> w

[1]  0  2 -2  7 NA -4

> length(w)

[1] 6

> length(w) <- 4
> w

[1]  0  2 -2  7

> w[-3]

[1] 0 2 7
```

3. Las operaciones básicas $+, -, *, /, \hat{}$ o funciones matemáticas como `log()`, `exp()`, `sqrt()`, etc. están definidas para operar vectorialmente, componente a componente. Ejecutad las siguientes operaciones, analizad los resultados obtenidos y observad los mensajes de advertencia:

```
> 2*v-3*w+2
> v*w
> w/v
> v/w
> v^3
> sqrt(w)
> log(w)
> # Operaciones con vectores de distintas longitudes
> vw <- c(v, w)
> vw
> vw/v
> vwa <- c(vw, 6)
> vwa/v
```

4. En cambio, funciones estadísticas como la suma, el producto o la media aritmética devuelven un solo valor para cada vector:

```
> sum(v)

[1] 10

> prod(w)

[1] 0

> prod(c(sum(v), sum(w)))

[1] 70

> mean(v)

[1] 2.5
```

5. Las instrucciones `seq(inicio, fin, paso)`, `rep(vector, num_veces)` e `inicio:fin` permiten generar sucesiones de valores. Por ejemplo,

```
> seq(1, 29, 2)

 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29

> -3:5

[1] -3 -2 -1  0  1  2  3  4  5

> rep(c(1, -2, 0), 5)

 [1]  1 -2  0  1 -2  0  1 -2  0  1 -2  0  1 -2  0

> rep(c(1, -2, 0), each = 5)

 [1]  1  1  1  1  1 -2 -2 -2 -2 -2  0  0  0  0  0

> rep(c(1, -2, 0), c(4, 1, 3))

[1]  1  1  1  1 -2  0  0  0
```

6. En determinadas situaciones, por ejemplo al comienzo de una simulación o programando nuevas funciones, es importante crear un nuevo vector sin especificar sus elementos. Lo podemos hacer utilizando la función `numeric()`:

```
> x <- numeric()
> x

numeric(0)
```

```
> x[1:3] <- c(1, 2, 4)
> x

[1] 1 2 4

> (y <- numeric(4))

[1] 0 0 0 0

> y[x] <- 3
> y

[1] 3 3 0 3
```

7. También se pueden utilizar operadores lógicos como subíndice. La expresión lógica será evaluada, componente a componente, como un 0 (*FALSE*) o un 1 (*TRUE*), y se considerarán aquellos componentes para los cuales la expresión sea verdadera.

```
> vwa <= 0

[1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE

> as.numeric(vwa <= 0)

[1] 0 0 0 0 1 0 1 0 0

> vwa[vwa > 1]

[1] 2 3 4 2 7 6

> # Número de elementos de vwa superiores a 1
> sum(vwa > 1)

[1] 6

> # Posiciones de dichos elementos
> which(vwa > 1)

[1] 1 3 4 6 8 9
```

8. Hasta aquí, los vectores utilizados han sido de tipo numérico. Otra posibilidad es que sean de tipo cadena (*string*). En este contexto, números se interpretan como caracteres, ya que los elementos de un vector no pueden ser de dos tipos diferentes:

```
> z <- c("Barcelona", "Lleida", "Barcelona", "Girona")
> z

[1] "Barcelona" "Lleida"    "Barcelona" "Girona"
```

```
> c(z, 9)

[1] "Barcelona" "Lleida"    "Barcelona" "Girona"    "9"
```

9. Si queremos analizar una variable categórica o usarla en modelos de regresión es recomendable guardarla como un factor. Aparentemente parece no haber diferencias entre una variable de tipo cadena y otra de tipo factor, pero internamente R codifica los distintos niveles de un factor como enteros.

```
> (zf <- factor(z))

[1] Barcelona Lleida    Barcelona Girona
Levels: Barcelona Girona Lleida

> is.character(z)

[1] TRUE

> is.character(zf)

[1] FALSE

> as.numeric(z)

[1] NA NA NA NA

> as.numeric(zf)

[1] 1 3 1 2

> levels(z)

NULL

> levels(zf)

[1] "Barcelona" "Girona"    "Lleida"
```

10. También es posible ponerles una etiqueta a los diferentes elementos de un vector:

```
> msd <- c(Mean = mean(vw), SD = sd(vw))
> msd

    Mean       SD
2.125000 2.695896

> names(msd)

[1] "Mean" "SD"
```

```
> names(msd) <- c("Media", "Desv. est.")
> msd

     Media Desv. est.
  2.125000   2.695896
```

11. La función `paste()` permite concatenar elementos de diferentes tipos:

```
> nombres <- paste("Var", 1:5, sep = "-")
> nombres

[1] "Var-1" "Var-2" "Var-3" "Var-4" "Var-5"

> length(nombres)

[1] 5

> paste("Now it's", date())

[1] "Now it's Fri Sep 19 20:10:17 2014"

> paste("Raíz de", w, "es", round(sqrt(w), 3))

[1] "Raíz de 0 es 0"     "Raíz de 2 es 1.414" "Raíz de -2 es NaN"
[4] "Raíz de 7 es 2.646"
```

2.1.2. Creación y manipulación de matrices

1. La función `matrix(vector, ...)` organiza los componentes de un vector (`vector`) en forma de matriz de tantas filas (columnas) como especificado mediante la opción `nrow` (`ncol`). El número de columnas (filas) se determina mediante el redondeo por exceso de la longitud del vector entre el número de columnas. Si faltan elementos, R replica el vector a partir de la primera componente. La función `dim(matriz)` muestra el número de filas y columnas de `matriz`.

```
> A <- matrix(1:12, ncol = 4)
> A

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> log(A)

          [,1]     [,2]     [,3]     [,4]
[1,] 0.0000000 1.386294 1.945910 2.302585
[2,] 0.6931472 1.609438 2.079442 2.397895
[3,] 1.0986123 1.791759 2.197225 2.484907
```

```
> sum(A)

[1] 78

> dim(A)

[1] 3 4

> dim(A)[2]

[1] 4

> nrow(A)

[1] 3

> ncol(A)

[1] 4

> (AA <- matrix(1:12, nrow = 3, byrow = T))

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> (B <- matrix(-5:4, nc = 3))

     [,1] [,2] [,3]
[1,]   -5   -1    3
[2,]   -4    0    4
[3,]   -3    1   -5
[4,]   -2    2   -4
```

2. Los operadores $+, -, *$ y / se pueden aplicar a dos o más matrices de la misma dimensión. Serán ejecutadas componente a componente con lo cual el resultado es otra matriz de la misma dimensión.

```
> A+AA

     [,1] [,2] [,3] [,4]
[1,]    2    6   10   14
[2,]    7   11   15   19
[3,]   12   16   20   24
```

```
> A*AA

     [,1] [,2] [,3] [,4]
[1,]    1    8   21   40
[2,]   10   30   56   88
[3,]   27   60   99  144

> A*B                       #causa mensaje de error
```

3. El producto matricial se denota por `%*%` y la matriz traspuesta por `t(matriz)`.

```
> (C <- A%*%B)

     [,1] [,2] [,3]
[1,]  -62   26  -56
[2,]  -76   28  -58
[3,]  -90   30  -60

> t(A)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12

> t(B)%*%t(A)

     [,1] [,2] [,3]
[1,]  -62  -76  -90
[2,]   26   28   30
[3,]  -56  -58  -60
```

4. Con `A[i, j]`, `A[i, ]` y `A[, j]` nos referimos a un elemento, a una fila o a una columna de la matriz `A`, respectivamente. Si se utiliza un vector como subíndice obtenemos la submatriz correspondiente. Las funciones `cbind` y `rbind` permiten combinar matrices por columnas y por filas, respectivamente.

```
> A[2, 3]

[1] 8

> A[1, ]

[1]  1  4  7 10

> B[, 1:2]
```

```
     [,1] [,2]
[1,]   -5   -1
[2,]   -4    0
[3,]   -3    1
[4,]   -2    2
```

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

```
     [,1] [,2]
[1,]   26  -56
[2,]   30  -60
```

```
> A[2:3, ]
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
```

```
> t(B[, c(1, 3)])
```

```
     [,1] [,2] [,3] [,4]
[1,]   -5   -4   -3   -2
[2,]    3    4   -5   -4
```

```
> cbind(A[2:3, ], AA[1:2, ])
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    2    5    8   11    1    2    3    4
[2,]    3    6    9   12    5    6    7    8
```

```
> rbind(A[2:3, ], t(B[, c(1, 3)]))
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
[3,]   -5   -4   -3   -2
[4,]    3    4   -5   -4
```

5. Es posible poner nombres tanto a las filas como a las columnas de una matriz y utilizar los mismos para extraer elementos de la matriz.

```
> rownames(A)
```

```
NULL
```

```
> colnames(A)
```

```
NULL
```

```
> rownames(A) <- paste("Fila", 1:3)
> colnames(A) <- paste0("Col.", 1:ncol(A)) # Same as paste("Col.", 1:ncol(A), sep = "")
> A

       Col.1 Col.2 Col.3 Col.4
Fila 1     1     4     7    10
Fila 2     2     5     8    11
Fila 3     3     6     9    12

> A["Fila 2", ]

Col.1 Col.2 Col.3 Col.4
    2     5     8    11

> A["Fila 1", "Col.3"]

[1] 7
```

2.2. Listas y *data frames*

2.2.1. Creación y manipulación de listas

1. A menudo resulta conveniente organizar la información en forma de listas. Una lista es un objeto de tipo vectorial en el que, a diferencia de los elementos de vectores o matrices, cada componente puede ser de un tipo distinto. Mediante '[[ ]]' nos podemos referir a los componentes de un objeto de tipo lista. R organiza gran parte de sus variables en forma de listas. Veamos algunos ejemplos:

```
> list(c("Joan", "Rosa", "Miguel"), c(30, 29, 2))

[[1]]
[1] "Joan"   "Rosa"   "Miguel"

[[2]]
[1] 30 29  2

> lista1 <- list(Nombres = c("Joan", "Rosa", "Miguel"), Edades = c(30, 29, 2))
> lista1

$Nombres
[1] "Joan"   "Rosa"   "Miguel"

$Edades
[1] 30 29  2

> lista1[[1]]

[1] "Joan"   "Rosa"   "Miguel"
```

```
> lista1$Nombres

[1] "Joan"   "Rosa"   "Miguel"

> lista1[[1]][2]

[1] "Rosa"

> lista1$E[2]

[1] 29
```

2. Fijaos en la diferencia entre `lista1[2]` y `lista1[[2]]`: utilizando un solo par de corchetes, nos referimos a una (sub)lista, con dos (en este caso) a un vector numérico:

```
> lista1[2]

$Edades
[1] 30 29  2

> lista1[[2]]

[1] 30 29  2

> lista1[2]*2     # Error

> lista1[[2]]*2

[1] 60 58  4
```

3. Ejecutad también los siguientes comandos y estudiad los resultados:

```
> lista2 <- list(Valores = matrix(seq(100, 900, 100), ncol = 3), Estado = c(T, F),
+ Elementos = lista1)
> lista2
> lista2$Valores
> lista2$E
> lista2$Es
> lista2$Es[1]
> lista2$Elementos$Nombres[1]
> lista2$Elementos[2]
```

4. De la siguiente manera se pueden crear listas vacías, por ejemplo para su uso posterior en simulaciones o la creación de funciones:

```
> newlist <- vector("list", 2)
> newlist
```

```
[[1]]
NULL

[[2]]
NULL
```

```
> names(newlist) <- c("New names", "New ages")
> newlist[[1]] <- c("Carles", "Luisa")
> newlist
```

```
$`New names`
[1] "Carles" "Luisa"

$`New ages`
NULL
```

2.2.2. Creación de *data frames*

1. En R hay un tipo particular de objetos pensado para contener datos estructurados en forma de variables (columnas) para los distintos individuos (filas), los *data frames*. Las variables de un *data frame* pueden ser de distintos tipos y para crear un *data frame* se puede utilizar la función `data.frame`.

```
> datfram <- data.frame(Nom = c("Marta", "Jordi", "Pol"), Edat = c(34, 43, 13))
> datfram
```

```
    Nom Edat
1 Marta   34
2 Jordi   43
3   Pol   13
```

```
> datfram$E
```

```
[1] 34 43 13
```

```
> B
```

```
     [,1] [,2] [,3]
[1,]   -5   -1    3
[2,]   -4    0    4
[3,]   -3    1   -5
[4,]   -2    2   -4
```

```
> data.frame(B)
```

```
  X1 X2 X3
1 -5 -1  3
2 -4  0  4
3 -3  1 -5
4 -2  2 -4
```

```
> data.frame(lista1)

  Nombres Edades
1    Joan     30
2    Rosa     29
3  Miguel      2
```

2. Con el siguiente comando se puede abrir el editor de datos y crear un nuevo *data frame* entrando los datos uno por uno:

   `datfram2 <- edit(data.frame())`

   Notad que en el editor de datos de R se pueden modificar también los nombres de las variables y definir su tipo (variable numérica o alfanumérica).

2.2.3. Lectura de un fichero ASCII en un *data frame*

1. Para leer datos de un fichero en formato ASCII existen dos funciones: `scan(dirección)` y `read.table(dirección, cabecera)`. La primera permite leer un fichero de datos numéricos para después organizarlo en forma matricial. La segunda permite recuperar un fichero, con identificación de variables y de individuos, directamente hacia un formato de *data frame*. Notad que, para ejecutar los siguientes ejemplos, los ficheros tipo ASCII indicados, `valores.txt`, `valores2.txt` y `tabla.txt` (véase página 85) han de estar guardados en el directorio de trabajo actual.

```
> scan("valores.txt")

 [1]  25 167  65  21 160  57  23 178  83  29 170  69  23 163  65  19 185  90

> datos1 <- matrix(scan("valores.txt"), ncol = 3, byrow = T)
> datos1

     [,1] [,2] [,3]
[1,]   25  167   65
[2,]   21  160   57
[3,]   23  178   83
[4,]   29  170   69
[5,]   23  163   65
[6,]   19  185   90

> is.matrix(datos1)

[1] TRUE

> is.data.frame(datos1)

[1] FALSE
```

En cambio, la ejecución de `scan("valores2.txt")` causaría un mensaje de error ya que el fichero `valores2.txt` contiene una variable alfanumérica.

No obstante es posible usar la función `scan()` para leer datos externos y crear una lista con ellos independientemente del tipo de variables:

```
> scan("valores.txt", list(0, 0, 0))

[[1]]
[1] 25 21 23 29 23 19

[[2]]
[1] 167 160 178 170 163 185

[[3]]
[1] 65 57 83 69 65 90

> scan("valores2.txt", list(Var1 = 0, Var2 = 0, Var3 = 0, Sexo = ""))

$Var1
[1] 25 21 23 29 23 19

$Var2
[1] 167 160 178 170 163 185

$Var3
[1] 65 57 83 69 65 90

$Sexo
[1] "M" "M" "H" "H" "M" "H"
```

2. Veamos ahora el uso de la función `read.table()` para importar los datos en `tabla.txt`:

```
> read.table("tabla.txt")

      V1   V2     V3   V4   V5
1 Nombre Edad Altura Peso Sexo
2  Laura   25    167   65    M
3  Maria   21    160   57    M
4  Pedro   23    178   83    H
5  Josep   29    170   69    H
6 Martha   23    163   65    M
7  Jordi   19    185   90    H

> datos2 <- read.table("tabla.txt", header = T)
> datos2

  Nombre Edad Altura Peso Sexo
1  Laura   25    167   65    M
2  Maria   21    160   57    M
```

```
3  Pedro   23    178   83    H
4  Josep   29    170   69    H
5 Martha   23    163   65    M
6  Jordi   19    185   90    H
```

3. A continuación algunos funciones para estudiar diferentes propiedades de `datos2`:

```
> is.matrix(datos2)

[1] FALSE

> is.data.frame(datos2)

[1] TRUE

> length(datos2)

[1] 5

> dimnames(datos2)

[[1]]
[1] "1" "2" "3" "4" "5" "6"

[[2]]
[1] "Nombre" "Edad"   "Altura" "Peso"   "Sexo"

> colnames(datos2)

[1] "Nombre" "Edad"   "Altura" "Peso"   "Sexo"

> datos2[, 4]

[1] 65 57 83 69 65 90

> datos2[, "Altura"]

[1] 167 160 178 170 163 185

> datos2$A

[1] 167 160 178 170 163 185

> datos2["Altura"]

  Altura
1    167
2    160
3    178
4    170
5    163
6    185
```

```
> is.data.frame(datos2[, "Altura"])

[1] FALSE

> is.vector(datos2[, "Altura"])

[1] TRUE

> is.data.frame(datos2["Altura"])

[1] TRUE
```

2.2.4. Edición y manipulación de *data frames*

1. Podemos convertir el objeto `datos1` en *data frame* utilizando la función `data.frame()`

```
> datos3 <- data.frame(datos1)
> datos1

     [,1] [,2] [,3]
[1,]   25  167   65
[2,]   21  160   57
[3,]   23  178   83
[4,]   29  170   69
[5,]   23  163   65
[6,]   19  185   90

> datos3

  X1  X2 X3
1 25 167 65
2 21 160 57
3 23 178 83
4 29 170 69
5 23 163 65
6 19 185 90
```

2. Una posibilidad para editar los identificadores de columna de `datos3` es mediante la función `colnames`:

```
> colnames(datos3)

[1] "X1" "X2" "X3"

> colnames(datos3) <- c("Edad", "Altura", "Peso")
> datos3
```

```
  Edad Altura Peso
1   25    167   65
2   21    160   57
3   23    178   83
4   29    170   69
5   23    163   65
6   19    185   90
```

Y también los identificadores de fila:

```
> rownames(datos3) <- c("Laura", "Maria", "Pedro", "Josep", "Martha", "Jordi")
> datos3

       Edad Altura Peso
Laura    25    167   65
Maria    21    160   57
Pedro    23    178   83
Josep    29    170   69
Martha   23    163   65
Jordi    19    185   90

> datos3["Pedro", ]

      Edad Altura Peso
Pedro   23    178   83

> datos3[c("Maria", "Martha"), 2:3]

       Altura Peso
Maria     160   57
Martha    163   65
```

3. Es posible editar los objetos de una sesión de R con las funciones `edit()` y `fix()`. Hay que resaltar que la función `edit()` sólo permite editar objetos existentes y que **no** modifica el contenido del objeto editado; por consiguiente, el resultado de la edición se debe asignar a otro objeto. En cambio, la función `fix()` permite definir un nuevo objeto y modificar los existentes. Para entenderlo mejor, ejecutad los siguientes ejemplos:

```
> edit(datos2)          # Modificad un valor de los datos
> datos2                # Comprobad que los datos son los iniciales
> datos4 <- edit(datos2) # Modificad de nuevo un valor de los datos
> datos4                # Comprobad que los datos han cambiado
> fix(datos2)           # Modificad un valor de los datos
> datos2                # Comprobad que los datos han cambiado
```

4. Mediante las funciones `subset()` y `transform()` se pueden crear subconjuntos y nuevas variables de un *data frame*, respectivamente. Por ejemplo:

```
> subset(datos2, Altura>170)

  Nombre Edad Altura Peso Sexo
3  Pedro   23    178   83    H
6  Jordi   19    185   90    H

> datos23 <- subset(datos2, Edad == 23)
> datos23

  Nombre Edad Altura Peso Sexo
3  Pedro   23    178   83    H
5 Martha   23    163   65    M

> transform(datos2, Altura = Altura/100, BMI = round(Peso/(Altura/100)^2, 2))

  Nombre Edad Altura Peso Sexo   BMI
1  Laura   25   1.67   65    M 23.31
2  Maria   21   1.60   57    M 22.27
3  Pedro   23   1.78   83    H 26.20
4  Josep   29   1.70   69    H 23.88
5 Martha   23   1.63   65    M 24.46
6  Jordi   19   1.85   90    H 26.30
```

Para crear (o sobreescribir) una sola variable existe una alternativa al uso de la función `transform()`:

```
> datos23

  Nombre Edad Altura Peso Sexo
3  Pedro   23    178   83    H
5 Martha   23    163   65    M

> datos23$BMI = round(datos23$Peso/(datos23$Altura/100)^2, 2)
```

5. Para poder usar directamente cada una de las variables de un *data frame* es necesario identificarlas temporalmente como objetos propiamente dichos. Esta acción se consigue con la función `attach(df)` que añade el *data frame* `df` al camino de búsqueda de R. Para quitar `df` de allí se usa la función `detach(df)`. Algunos ejemplos:

```
> Altura                  # Notad que no existe dicho objeto

> attach(datos2)
> search()

 [1] ".GlobalEnv"        "datos2"            "package:grDevices"
 [4] "package:datasets"  "package:LEpack"    "package:Hmisc"
 [7] "package:Formula"   "package:survival"  "package:splines"
[10] "package:graphics"  "package:utils"     "package:stats"
[13] "package:lattice"   "package:grid"      "package:methods"
[16] "Autoloads"         "package:base"
```

```
> Altura

[1] 167 160 178 170 163 185

> detach(datos2)
> search()

 [1] ".GlobalEnv"        "package:grDevices" "package:datasets"
 [4] "package:LEpack"    "package:Hmisc"     "package:Formula"
 [7] "package:survival"  "package:splines"   "package:graphics"
[10] "package:utils"     "package:stats"     "package:lattice"
[13] "package:grid"      "package:methods"   "Autoloads"
[16] "package:base"
```

6. Si añadimos una nueva columna a un *data frame* que está en la lista de búsqueda de R, hemos de volver a aplicar la función `attach()` antes de que podamos referirnos a la columna por su nombre:

```
> attach(datos2)
> datos2$Coche <- factor(rep(c("Sí", "No"), 3))

> Coche                          # El vector Coche no existe

> detach(datos2)
> attach(datos2)
> Coche

[1] Sí No Sí No Sí No
Levels: No Sí

> detach(datos2)
```

7. Una alternativa muy recomendable al uso de la función `attach` es la función `with`, cuyo uso se muestra a continuación:

```
> BMI <- round(Peso/(Altura/100)^2, 2)   # No funciona

> BMI <- with(datos2, round(Peso/(Altura/100)^2, 2))
> BMI

[1] 23.31 22.27 26.20 23.88 24.46 26.30
```

8. La función para fundir dos *data frames* es `merge`. Para ilustrar su uso creamos otro *data frame*, `datos4`, que contiene un subconjunto de los casos y variables de `datos2` y además una nueva variable y un caso más (véase el fichero `tabla2.txt` en el apéndice A).

```
> datos4 <- read.table("tabla2.txt", header = T)
> datos4
```

```
  Nombre Edad Altura Peso Ciudad
1  Laura   25    167   65    BCN
2  Josep   29    170   69    BCN
3  Jordi   19    185   90 Lleida
4  Adela   30    162   62    BCN
```

```
> merge(datos2, datos4)
```

```
  Nombre Edad Altura Peso Sexo Coche Ciudad
1  Jordi   19    185   90    H    No Lleida
2  Josep   29    170   69    H    No    BCN
3  Laura   25    167   65    M    Sí    BCN
```

```
> merge(datos2, datos4, all = T)
```

```
  Nombre Edad Altura Peso Sexo Coche Ciudad
1  Jordi   19    185   90    H    No Lleida
2  Josep   29    170   69    H    No    BCN
3  Laura   25    167   65    M    Sí    BCN
4  Maria   21    160   57    M    No   <NA>
5 Martha   23    163   65    M    Sí   <NA>
6  Pedro   23    178   83    H    Sí   <NA>
7  Adela   30    162   62 <NA>  <NA>    BCN
```

```
> merge(datos2, datos4, all.x = T)
```

```
  Nombre Edad Altura Peso Sexo Coche Ciudad
1  Jordi   19    185   90    H    No Lleida
2  Josep   29    170   69    H    No    BCN
3  Laura   25    167   65    M    Sí    BCN
4  Maria   21    160   57    M    No   <NA>
5 Martha   23    163   65    M    Sí   <NA>
6  Pedro   23    178   83    H    Sí   <NA>
```

```
> merge(datos2, datos4, all.y = T)
```

```
  Nombre Edad Altura Peso Sexo Coche Ciudad
1  Jordi   19    185   90    H    No Lleida
2  Josep   29    170   69    H    No    BCN
3  Laura   25    167   65    M    Sí    BCN
4  Adela   30    162   62 <NA>  <NA>    BCN
```

9. Mediante función `order()` se pueden ordenar las filas de un *data frame* según una o más variables:

```
> with(datos2, datos2[order(Sexo, Edad), ])
```

```
  Nombre Edad Altura Peso Sexo Coche
6  Jordi   19    185   90    H    No
3  Pedro   23    178   83    H    Sí
4  Josep   29    170   69    H    No
2  Maria   21    160   57    M    No
5 Martha   23    163   65    M    Sí
1  Laura   25    167   65    M    Sí
```

Otra función muy útil para este fin es la función `orderBy` del paquete `doBy`[3]. Una vez instalado este paquete, se puede utilizar esa función después de haberlo cargado:

```
> install.packages("doBy")       # Para instalar el paquete en el ordenador

> library(doBy)
> orderBy(~Sexo+Edad, datos2)

  Nombre Edad Altura Peso Sexo Coche
6  Jordi   19    185   90    H    No
3  Pedro   23    178   83    H    Sí
4  Josep   29    170   69    H    No
2  Maria   21    160   57    M    No
5 Martha   23    163   65    M    Sí
1  Laura   25    167   65    M    Sí
```

2.3. Importar y exportar datos

1. Si queremos importar datos de otro *software* estadístico, podemos usar funciones de la librería Hmisc [4] una vez que ésta esté instalada. Las funciones para datos procedentes de SPSS, SAS y STATA son `spss.get`, `sas.get` y `stata.get`, respectivamente. A continuación se muestra un ejemplo para el uso de `spss.get`. Notad que para que éste funcione un fichero con nombre `DatosSPSS.sav` ha de estar guardado en la carpeta de trabajo actual.

```
> install.packages("Hmisc")      # Para instalar el paquete en el ordenador

> library(Hmisc)
> newdata <- spss.get("DatosSPSS.sav", lowernames = T, datevars = "bday")
```

La opción `lowernames = T` convierte los nombres de las variables de `newdata` en minúscula mientras la opción `datevars` identifica las variables de tipo de fecha del fichero original. Aunque aparezca un mensaje de advertencia parecido a

```
Warning message:
DatosSPSS.sav: File-indicated character representation code (1252)...
```

la importación del fichero suele haber funcionado sin ningún problema.

2. Para importar datos desde un fichero EXCEL se recomienda guardar los datos (dentro de EXCEL) en formato csv y posteriormente aplicar la función `csv.get` en R. Otra posibilidad, que no requiere la conversión del fichero en otro de formato csv, existe si está en uso el *R Commander* (véase Sección 1.3). En la barra de herramientas del mismo hay que ir a:

   Datos ▶ Importar datos ▶ Desde Excel, Access o dBase...

3. Al mismo tiempo es posible exportar datos desde R a ficheros de formato ASCII utilizando las funciones `write` and `write.table`. Mientras la primera permite exportar vectores y matrices, con la segunda se pueden exportar *data frames*. Más información se puede encontrar en el documento de ayuda 'R Data Import/Export' [5] accesible vía la barra de herramientas:

   Ayuda ▶ Manuales (en PDF) ▶ R Data Import/Export

Actividad 3

Análisis descriptivo y generación de números aleatorios

Contenido

R ofrece muchas posibilidades para realizar análisis descriptivos y exploratorios de un conjunto de datos. En la presente actividad se presentan las principales instrucciones y funciones necesarias para realizar estas tareas. Además se explicará brevemente cómo generar datos aleatorios de distintas distribuciones.

3.1. Análisis descriptivo

3.1.1. Análisis descriptivo de una variable

1. Con la función `mean(x)` se puede calcular la media de un vector o de una matriz. La función `var(x)` calcula la varianza de un vector, la matriz de covarianzas entre las columnas de una matriz de datos o la matriz de covarianzas entre las columnas de dos matrices. Veamos algunos ejemplos utilizando los datos de dos vectores con datos aleatorios de dos distribuciones normales (para más información acerca de la generación de datos aleatorios véase Sección 3.2):

```
> x <- rnorm(100, 5, 3)
> mean(x)

[1] 4.875726

> xna <- c(x, rep(NA, 5))
> mean(xna)

[1] NA

> mean(xna, na.rm = T)
```

```
[1] 4.875726

> var(x)

[1] 8.835228

> y <- x+rnorm(100, 0, 2)
> var(cbind(x, y))

          x         y
x 8.835228  8.648971
y 8.648971 12.177274

> sd(x)                    # La desviación estándar

[1] 2.972411
```

Como se ha podido ver, por defecto la función `mean` no calcula la media si hay datos omitidos. Para descartar éstos se ha de usar la opción `na.rm = T`. Lo mismo también es válido para otras funciones estadísticas como `sum`, `var` o `sd`.

2. Otros indicadores numéricos de interés son la mediana o el rango intercuartílico:

```
> median(x)

[1] 4.798625

> median(datos2$Peso)

[1] 67

> min(datos2$Altura)

[1] 160

> max(datos2$Altura)

[1] 185

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -4.517   3.146   4.799   4.876   6.360  13.950

> IQR(x)

[1] 3.214702
```

3. La función `quantile(x, p)` calcula los cuantiles empíricos de `x` para un vector de probabilidades `p`. Por ejemplo:

```
> quantile(x, 0.3)

    30%
3.54187

> quantile(x, c(0.1, 0.3, 0.6, 0.8))

     10%      30%      60%      80%
1.328582 3.541870 5.578533 6.933785

> quantile(x, seq(.2, .9, .1))

     20%      30%      40%      50%      60%      70%      80%      90%
2.329090 3.541870 4.135441 4.798625 5.578533 6.002360 6.933785 8.652313
```

Nota: Esta función ofrece nueve algoritmos distintos para el cálculo de los cuantiles. Para ver las diferencias, mirad la ayuda:

```
> ?quantile
```

4. La función `summary` proporciona un resumen estadístico de un objeto. Si se trata de un vector numérico, `summary` nos devuelve el mínimo, el máximo, la media y los tres cuartiles de los datos. Si además hay datos omitidos, indica el número de éstos.

```
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -4.517   3.146   4.799   4.876   6.360  13.950

> xna <- c(x, rep(NA, 5))
> summary(xna)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 -4.517   3.146   4.799   4.876   6.360  13.950       5

> summary(x)[3:4]

Median   Mean
 4.799  4.876

> summary(x)[5]-summary(x)[2]     # =IQR(x)

3rd Qu.
  3.214
```

5. Las funciones `colMeans` y `colSums` devuelven, respectivamente, las medias y sumas de las columnas de una matriz o un *data frame*.

```
> colMeans(datos2[2:4])
```

```
     Edad    Altura      Peso
 23.33333 170.50000  71.50000
```

```
> colSums(cbind(x, y))
```

```
       x        y
487.5726 480.5035
```

Si se trata de una matriz numérica pueden ser de interés también las funciones `rowMeans` y `rowSums` que calculan los mismos estadísticos para las filas de la matriz.

6. En caso de una variable categórica nos suele interesar la representación de los datos en forma de una tabla de frecuencia. Con la función `table` se puede construir una tabla sencilla. Por ejemplo, utilicemos para este objetivo un vector tipo factor generado mediante la función `sample` (véase Sección 3.2, página 43).

```
> prov <- factor(sample(c("BCN", "Girona", "Lleida", "Tarragona"), size = 80,
+                       replace = T))
> prov[1:7]
```

```
[1] BCN       Girona    Tarragona Tarragona BCN       Tarragona Girona
Levels: BCN Girona Lleida Tarragona
```

```
> table(prov)
```

```
prov
      BCN    Girona    Lleida Tarragona
       14        26        16        24
```

```
> with(datos2, table(Coche))
```

```
Coche
No Sí
 3  3
```

El mismo resultado obtenemos con la función `summary` siempre y cuando se trate de un factor. En cambio, si es una variable alfanumérica no guardada como factor, entonces obtenemos tal tabla solamente con la función `table`:

```
> summary(prov)
```

```
      BCN    Girona    Lleida Tarragona
       14        26        16        24
```

```
> prov2 <- sample(c("BCN", "Girona", "Lleida", "Tarragona"), size = 80, replace = T)
> table(prov2)
```

```
prov2
      BCN    Girona    Lleida Tarragona
       20        15        22        23
```

```
> summary(prov2)

   Length     Class      Mode
       80 character character
```

7. Si queremos que añadir además las frecuencias relativas podemos usar, por ejemplo, las funciones `describe` (del paquete `Hmisc`) o `ctab` (`catspec` [6]).

```
> library(Hmisc)
> describe(prov)

prov
      n missing  unique
     80       0       4

BCN (14, 18%), Girona (26, 32%), Lleida (16, 20%)
Tarragona (24, 30%)
```

```
> describe(datos2[, c(5, 6)])

datos2[, c(5, 6)]

 2  Variables      6  Observations
---------------------------------------------------------------------------
Sexo
      n missing  unique
      6       0       2

H (3, 50%), M (3, 50%)
---------------------------------------------------------------------------
Coche
      n missing  unique
      6       0       2

No (3, 50%), Sí (3, 50%)
---------------------------------------------------------------------------
```

```
> library(catspec)
> with(datos2, ctab(Sexo))

      Count Total %
Sexo
H         3      50
M         3      50
```

```
> ctab(prov)

           Count Total %
prov
BCN         14.0    17.5
Girona      26.0    32.5
Lleida      16.0    20.0
Tarragona   24.0    30.0
```

8. Si se aplica la función `summary` a una matriz o a un *data frame*, se obtiene un resumen de cada una de sus variables:

```
> summary(datos2)

    Nombre       Edad           Altura          Peso      Sexo  Coche
 Jordi :1   Min.   :19.00   Min.   :160.0   Min.   :57.0   H:3   No:3
 Josep :1   1st Qu.:21.50   1st Qu.:164.0   1st Qu.:65.0   M:3   Sí:3
 Laura :1   Median :23.00   Median :168.5   Median :67.0
 Maria :1   Mean   :23.33   Mean   :170.5   Mean   :71.5
 Martha:1   3rd Qu.:24.50   3rd Qu.:176.0   3rd Qu.:79.5
 Pedro :1   Max.   :29.00   Max.   :185.0   Max.   :90.0
```

Nota: La función `summary` es una de las funciones genéricas de R, ya que se puede aplicar a tipos de objetos muy distintos. El resumen que devuelve esta función difiere de un tipo a otro.

3.1.2. Análisis descriptivo de dos variables

1. En caso de un par de variables numéricas, es de interés cuantificar la asociación entre ellas mediante la correlación, sea ésta la correlación según el coeficiente de Pearson o el de Spearman. Si aplicamos la función `cor` a más de dos variables numéricas, el resultado es una matriz con las correlaciones de cada par de variables.

```
> cor(x, y)

[1] 0.8338358

> round(cor(datos2[2:4]), 3)

         Edad Altura   Peso
Edad    1.000 -0.289 -0.340
Altura -0.289  1.000  0.986
Peso   -0.340  0.986  1.000

> round(cor(datos2[2:4], method = "spearman"), 3)

         Edad Altura   Peso
Edad    1.000 -0.116 -0.162
Altura -0.116  1.000  0.986
Peso   -0.162  0.986  1.000
```

2. Si una de las dos variables es categórica se pueden calcular distintos indicadores numéricos de interés para la variable numérica en función de la variable categórica con la función `tapply`:

```
> with(datos2, tapply(Peso, Sexo, mean))

       H        M
80.66667 62.33333

> with(datos2, tapply(Peso, Sexo, summary))

$H
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  69.00   76.00   83.00   80.67   86.50   90.00

$M
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  57.00   61.00   65.00   62.33   65.00   65.00

> with(datos2, tapply(Peso, Sexo, summary)[1])

$H
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  69.00   76.00   83.00   80.67   86.50   90.00
```

La función `by` permite realizar los mismos cálculos y además para varias variables numéricas a la vez:

```
> with(datos2, by(Peso, Sexo, summary))

Sexo: H
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  69.00   76.00   83.00   80.67   86.50   90.00
------------------------------------------------------------
Sexo: M
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  57.00   61.00   65.00   62.33   65.00   65.00

> by(datos2[, 2:4], datos2$Sexo, summary)

datos2$Sexo: H
      Edad           Altura           Peso
 Min.   :19.00   Min.   :170.0   Min.   :69.00
 1st Qu.:21.00   1st Qu.:174.0   1st Qu.:76.00
 Median :23.00   Median :178.0   Median :83.00
 Mean   :23.67   Mean   :177.7   Mean   :80.67
 3rd Qu.:26.00   3rd Qu.:181.5   3rd Qu.:86.50
 Max.   :29.00   Max.   :185.0   Max.   :90.00
------------------------------------------------------------
```

```
datos2$Sexo: M
      Edad         Altura           Peso
 Min.   :21   Min.   :160.0   Min.   :57.00
 1st Qu.:22   1st Qu.:161.5   1st Qu.:61.00
 Median :23   Median :163.0   Median :65.00
 Mean   :23   Mean   :163.3   Mean   :62.33
 3rd Qu.:24   3rd Qu.:165.0   3rd Qu.:65.00
 Max.   :25   Max.   :167.0   Max.   :65.00
```

Otras funciones que se pueden utilizar y que ofrecen más opciones son las funciones `aggregate` y `summaryBy`, ésta última del paquete `doBy` [3].

3. La presentación de la relación entre dos variables categóricas se hace mediante una tabla de contingencia. Para este objetivo se pueden utilizar las mismas funciones como en el caso univariante: `table` y `ctab`.

```
> with(datos2, table(Sexo, Coche))

    Coche
Sexo No Sí
   H  2  1
   M  1  2

> with(datos2, ctab(Sexo, Coche))

     Coche No Sí
Sexo
H           2  1
M           1  2

> sexo <- factor(sample(c("Mujer", "Hombre"), size = 80, replace = T))
> sexo[1:7]

[1] Hombre Mujer  Mujer  Hombre Hombre Hombre Hombre
Levels: Hombre Mujer

> table(prov, sexo)

           sexo
prov        Hombre Mujer
  BCN            6     8
  Girona        12    14
  Lleida         8     8
  Tarragona     11    13
```

```
> ctab(prov, sexo)

          sexo Hombre Mujer
prov
BCN                6     8
Girona            12    14
Lleida             8     8
Tarragona         11    13
```

Utilizando la opción `type` de la función `ctab`, se pueden añadir los porcentajes por filas (`type = 'r'`) y/o por columnas (`type = 'c'`).

```
> ctab(prov, sexo, type = c("n", "r"))

                sexo Hombre Mujer
prov
BCN       Count        6.00  8.00
          Row %       42.86 57.14
Girona    Count       12.00 14.00
          Row %       46.15 53.85
Lleida    Count        8.00  8.00
          Row %       50.00 50.00
Tarragona Count       11.00 13.00
          Row %       45.83 54.17
```

```
> ctab(prov, sexo, type = c("n", "c"), addmargins = T)

                  sexo Hombre  Mujer    Sum
prov
BCN       Count          6.00   8.00  14.00
          Column %      16.22  18.60  34.82
Girona    Count         12.00  14.00  26.00
          Column %      32.43  32.56  64.99
Lleida    Count          8.00   8.00  16.00
          Column %      21.62  18.60  40.23
Tarragona Count         11.00  13.00  24.00
          Column %      29.73  30.23  59.96
Sum       Count         37.00  43.00  80.00
          Column %     100.00 100.00 200.00
```

Otras funciones con aún más opciones son las funciones `stat.table` (del paquete `Epi` [7]) y `CrossTable` (`gmodels` [8]).

3.2. Generación de datos aleatorios

1. La generación de datos aleatorios se realiza con la función `r` seguida del nombre de la distribución de la cuál se desee generar. Por ejemplo, para generar un vector de 7 valores de una distribución normal de media 5 y desviación estándar 2 y otro de 20 valores proveniente de una distribución de Poisson con media 17 es suficiente escribir las siguientes instrucciones:

```
> rnorm(7, 5, 2)

[1] 4.237133 2.226432 1.402858 9.167782 4.899811 4.737949 6.609087

> rpois(20, 17)

 [1] 13 17 12 19 17 23 19 17 13  9 17 23 15 15 17 21 11 18 23 18
```

2. En la Tabla 3.1[1] se muestran los nombres en R de distintas distribuciones. En la mayoría de ellas existen valores por defecto de los parámetros que se pueden conocer mediante

   `> help(rdistribution)`.

   Las funciones `ddistribution`, `pdistribution` y `qdistribution` devuelven los valores de las funciones de densidad y de distribución y los cuantiles, respectivamente, según la ley *distribution*.

Tabla 3.1: Nombre de distribuciones en R

Distribución	**Nombre en R**	**Parámetros**
Beta	`beta`	`shape1, shape2, ncp`
Binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
Chi cuadrado	`chisq`	`df, ncp`
Exponencial	`exp`	`rate`
F	`f`	`df1, df2`
Gamma	`gamma`	`shape, scale`
Geométrica	`geom`	`prob`
Hipergeométrica	`hyper`	`m, n, k`
Lognormal	`lnorm`	`meanlog, sdlog`
Logística	`logis`	`location, scale`
Binomial negativa	`nbinom`	`size, prob`
Normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
T	`t`	`df, ncp`
Uniforme	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

3. La función `set.seed()` permite fijar la semilla que utiliza R cuando inicia un algoritmo para la generación de datos aleatorios. Esta función es de mucha utilidad cuando llevamos a cabo simulaciones que requieran la generación de datos aleatorios y queramos asegurarnos de poder reproducir los resultados obtenidos:

[1] de Venables *et al.* [11]

```
> rbinom(15, 20, 0.8)

 [1] 13 17 17 17 15 14 15 16 18 16 17 17 15 16 18

> set.seed(123)
> rbinom(15, 20, 0.8)

 [1] 17 15 17 14 13 19 16 14 16 16 13 16 15 16 18

> set.seed(123)
> rbinom(15, 20, 0.8)

 [1] 17 15 17 14 13 19 16 14 16 16 13 16 15 16 18
```

Muestreo con y sin reemplazo

1. La función `sample` realiza un muestreo (con o sin reemplazo) de un determinado tamaño de los elementos de un vector. En el primer ejemplo a continuación se realiza un muestreo de tamaño 15 de los números del 1 al 15 sin reemplazo. Para hacer muestreo con reemplazo, se utiliza la opción `replace`.

   ```
   > sample(1:15)

    [1]  4  6  8 11  3  9 10 14  5  1  2 13 12 15  7

   > x <- sample(1:15, replace = T)
   > table(x)

   x
    1  2  4  5  6  8 10 11 12 14 15
    1  1  1  1  4  1  1  1  1  1  2
   ```

2. Como ya se ha visto en la Sección 3.1, se ha de especificar el tamaño de la muestra con la opción `size` si éste es distinto a la longitud del vector:

   ```
   > treat <- factor(sample(c("Treatment A", "Treatment B", "Placebo"), size = 90,
   +                        replace = T))
   > table(treat)

   treat
       Placebo Treatment A Treatment B
            30          24          36
   ```

3. Si se desea generar una muestra de los elementos de un vector dándole probabilidades distintas a estos elementos, se puede usar la opción `prob`:

   ```
   > treat <- factor(sample(c("Treat. A", "Treat. B", "Placebo"), size = 90,
   +                        replace = T, prob = c(0.4, 0.4, 0.2)))
   > describe(treat)
   ```

```
treat
      n missing  unique
     90       0       3

Placebo (15, 17%), Treat. A (41, 46%), Treat. B (34, 38%)
```

Actividad 4

Gráficos

Contenido

R ofrece muchas posibilidades para realizar salidas gráficas muy variadas y de alta calidad. En la presente actividad presentamos las principales instrucciones y funciones necesarias para realizar estas tareas.

4.1. Generalidades

1. La instrucción genérica para obtener un gráfico en R es `plot()`. Esta función admite un gran número de parámetros con el fin de configurar el gráfico según nuestras necesidades. Veremos a continuación varios ejemplos. Para información más detallada consultad la ayuda sobre la función `plot`.

2. Antes de generar un gráfico conviene abrir una ventana gráfica. La ventana se puede abrir con la orden `windows()`. En el primer ejemplo (Figura 4.1) se hace un gráfico de dispersión (*scatterplot*) simple de peso versus altura utilizando el *data frame* `datos2` de la Actividad 2:

   ```
   > windows()
   > plot(Peso~Altura, data = datos2)
   > title("Peso vs. Altura")
   ```

   Varias notas:

   - El mismo gráfico se puede dibujar ejecutando la instrucción:

     ```
     > with(datos2, plot(Altura, Peso, main = "Peso vs. Altura"))
     ```

   - Si ejecutamos un comando para dibujar un gráfico, éste será dibujado aunque no hayamos ejecutado `windows()` previamente. Sin embargo, en este caso, si ya existe un gráfico en la ventana de gráficos, éste será sobrescrito.

   - En el sistema operativo Windows se puede usar también el comando `windows()`, las funciones en Linux y MacOS X son `x11` y `quartz`, respectivamente. Todas éstas per-

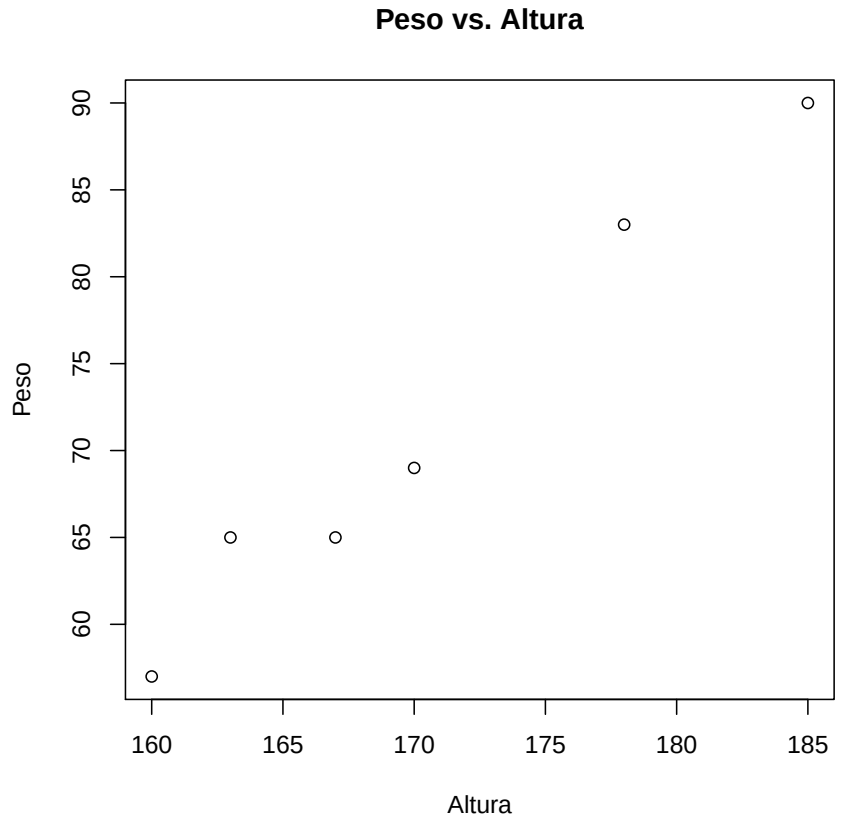

Figura 4.1: Un gráfico de dispersión sencillo

miten las opciones `width` y `height` mediante las cuales se puede cambiar el aspecto de la ventana; por defecto, éste es de seis por seis pulgadas.

3. El comando `par()` permite modificar distintos parámetros de la ventana gráfica como, por ejemplo, el tipo de fuente (opción `font`), la orientación de la numeración de los ejes (opción `las`) y la organización de los gráficos si se quiere dibujar más de un gráfico por ventana (opción `mfrow`). Por ejemplo, en la Figura 4.2 se dibujan dos gráficos de dispersión que se organizan en una fila y dos columnas (`mfrow = c(1, 2)`):

```
> # Código de la Figura 4.2
> windows(width = 7, height = 4)
> par(mfrow = c(1, 2), font = 2, font.lab = 3, font.axis = 4, las = 1)
> plot(Peso~Altura, data = datos2)
> title("Peso vs. Altura")
> plot(Peso~Edad, data = datos2)
> title("Peso vs. Edad")
```

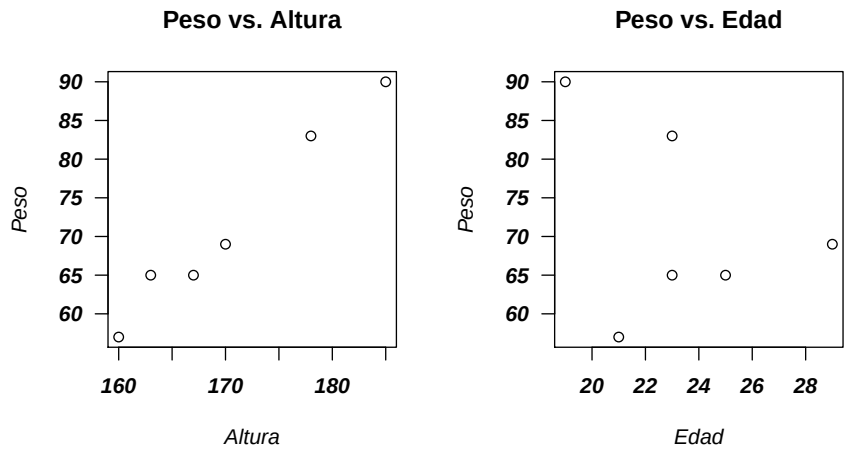

Figura 4.2: Ejemplo para dos gráficos en una ventana gráfica

Nota: Se recomienda estudiar con detenimiento la ayuda de la función `par` para tener una visión de la gran variedad de opciones que ofrece.

4. Veamos algunos ejemplos más. En el gráfico de dispersión del panel izquierdo de la Figura 4.3 se incluye una recta mediante la función `lines`. En cambio, en el panel derecho se muestran tres funciones de densidad de distribuciones χ^2 distintas, en donde la sobreposición de las curvas ha sido posible gracias a la opción `add = T`. Éstas se distinguen por su color (opción `col`) y su grosor (opción `lwd`). La leyenda en la esquina superior derecha (`"topright"`) se ha incluido con el comando `legend`.

```
> # Código de la Figura 4.3
> windows(width = 7, height = 4)
> par(mfrow = c(1, 2), font = 2, font.lab = 4, font.axis = 2, las = 1)
> y <- seq(0, 10, 0.2)
> plot(2*y+rnorm(length(y))~y)
> lines(2*y~y)
> title("Highly correlated data")
> curve(dchisq(x, 1), from = 0, to = 10, xlab = "", ylab = "Chi-square density")
> curve(dchisq(x, 2), from = 0, to = 10, type = "l", col = 2, lwd = 2, add = T)
> curve(dchisq(x, 3), from = 0, to = 10, type = "l", col = 3, lwd = 3, add = T)
> title("Chi-square distributions")
> legend("topright", c("n=1", "n=2", "n=3"), lty = 1, col = 1:3, lwd = 3)
```

5. En el ejemplo que se presenta en la Figura 4.4 se utilizan los datos `state.x77` del paquete `datasets`. Contiene información sobre los distintos estados de los Estados Unidos como la población, superficie, nivel de ingresos o la esperanza de vida correspondiente al año 1977. Usamos estos datos para ilustrar la relación entre salario medio y la tasa de analfabetismo

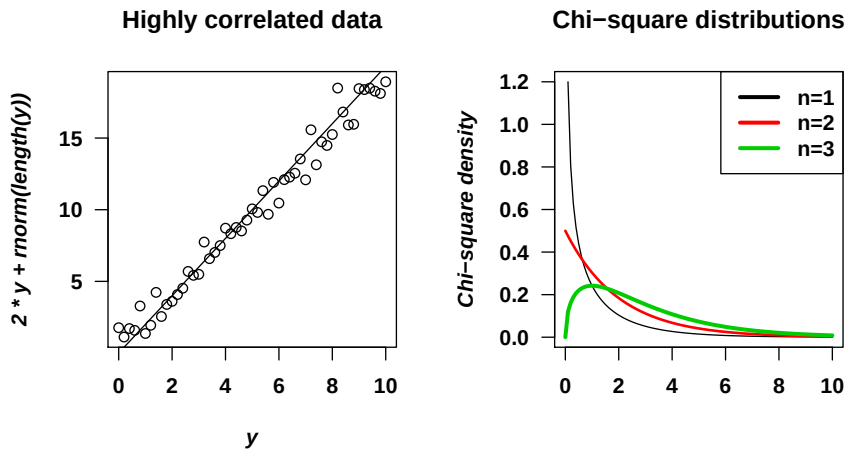

Figura 4.3: Datos altamente correlacionados y densidades de la distribución χ^2

de cada estado (Figura 4.4). Además se incluyen los nombres de algunos estados en el gráfico mediante la función `text`.

```
> # Código de la Figura 4.4
> windows(width = 7, height = 5)
> par(font = 2, font.lab = 4, font.axis = 2, las = 1)
> plot(Illiteracy~Income, data = as.data.frame(state.x77), xlab = "Income (Dollars)",
+      ylab = "Percentage of illiterates", pch = 16)
> states.lab <- c("Connecticut", "Florida", "Hawaii", "Louisiana", "Mississippi",
+                 "New York", "Texas")
> with(state.x77, text(Income[which(states%in%states.lab)],
+      Illiteracy[which(states%in%states.lab)], paste0(" ", states.lab), adj = 0))
```

Nota: Para que se pueda reproducir el gráfico se ha de convertir la matriz `state.x77` en un *data frame* y añadir al mismo la variable `states` con los nombres de los estados. Además es importante que el orden de los estados del vector `states.lab` sea en orden alfabético.

6. Desde la barra de herramientas los gráficos pueden ser guardados en distintos formatos (pdf, png, postscript, etc.) para su uso posterior. Esto les permite ser insertados en un documento WORD o incluidos en un documento de LaTeX:

   Archivo ▶ Guardar como ▶ ...

   Como alternativa se puede ejecutar el comando `savePlot(file, type)` que requiere un nombre (`file`) y el formato (`type`) en el cual se quiere guardar el gráfico.

7. R permite realizar sofisticados análisis gráficos mediante múltiples comandos gráficos. La Tabla 4.1 muestra algunas funciones para diferentes tipos de gráficos[1].

[1] El contenido de la tabla se ha copiado del *User's Guide* de S-Plus

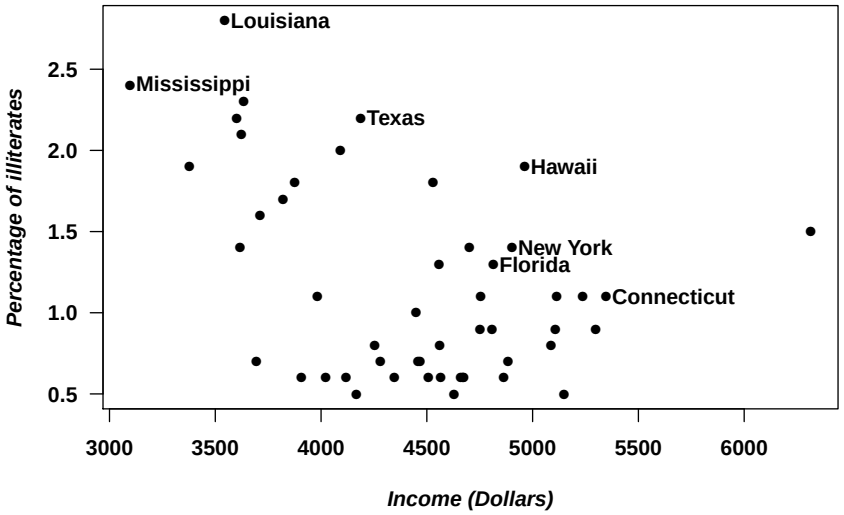

Figura 4.4: Tasa de analfabetismo versus ingreso medio en los estados de EE UU en 1977

8. Otros comandos de interés son los siguientes:

   - La instrucción `dev.off()` cierra la ventana activa actual.
   - En cambio, `graphics.off()` cierra todas las ventanas gráficas abiertas.
   - Se pueden dibujar gráficos directamente en un archivo externo sin tener que abrir ninguna ventana gráfica en R. El comando depende del tipo de archivo: `pdf`, `png`, `postscript`, etc. En este caso, hay que especificar el nombre del archivo a crear como opción de la función y para cerrarlo hay que ejecutar `dev.off()`.

4.2. Expresiones gráficas de las distribuciones

1. La función `hist(x)` proporciona un histograma convencional. Entre otros, admite parámetros para fijar el número de intervalos (opción `breaks`) o los puntos de corte de los intervalos. Dos ejemplos sencillos utilizando datos aleatorios provenientes de una distribución normal se presentan en la Figura 4.5.

```
> # Código de la Figura 4.5
> windows(width = 8, height = 5)
> par(mfrow = c(1, 2), font = 2, font.lab = 4, font.axis = 2, las = 1)
> z <- rnorm(10000, 6, 2)
> hist(z)
> hist(z, breaks = 50, col = "steelblue")
```

Tabla 4.1: Instrucciones para diferentes tipos y elementos de gráficos

Función	Gráfico/ Elemento de gráfico
`barplot, hist`	Bar graph, histogram
`boxplot`	Boxplot
`brush`	Brush pair-wise scatter plots; spin 3D axes
`contour, image`	3D plots
`coplot`	Conditioning plot
`dotchart`	Dotchart
`faces, stars`	Display multivariate data
`pairs`	Plot all pair-wise scatter plots
`pie`	Pie chart
`plot`	Generic plotting
`qqnorm, qqplot`	Normal and general QQ-plots
`scatter.smooth`	Scatter plot with a smooth curve
`tsplot`	Plot a time series
`abline`	Add line in intercept-slope form
`axis`	Add axis
`box`	Add a box around plot
`identify`	Use mouse to identify points on a graph
`legend`	Add a legend to the plot
`lines, points`	Add lines or points to a plot
`mtext, text`	Add text in the margin or in the plot
`stamp`	Add date and time information to the plot
`title`	Add title, x-axis labels, y-axis labels, and/or subtitle to plot

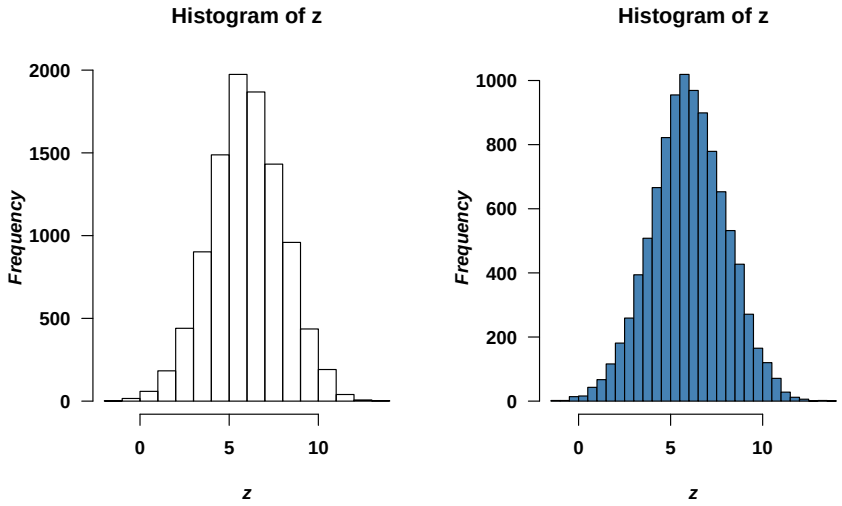

Figura 4.5: Dos histogramas de datos aleatorios de una distribución normal

2. A un histograma le podemos sobreponer la función de densidad teórica de una distribución. Para ello se ha de cambiar la opción `freq` para que la ordenada del gráfico muestre la densidad y no la frecuencia de los datos. En el ejemplo a continuación (Figura 4.6) se utiliza la función `dnorm` para la función de densidad de la distribución normal.

```
> # Código de la Figura 4.6
> windows(width = 7, height = 5)
> par(font = 2, font.lab = 4, font.axis = 2, las = 1)
> hist(z, breaks = 50, col = "tomato", freq = F)
> dz <- seq(min(z), max(z), 0.001)
> lines(dnorm(dz, 6, 2)~dz, type = "l", lwd = 3)
```

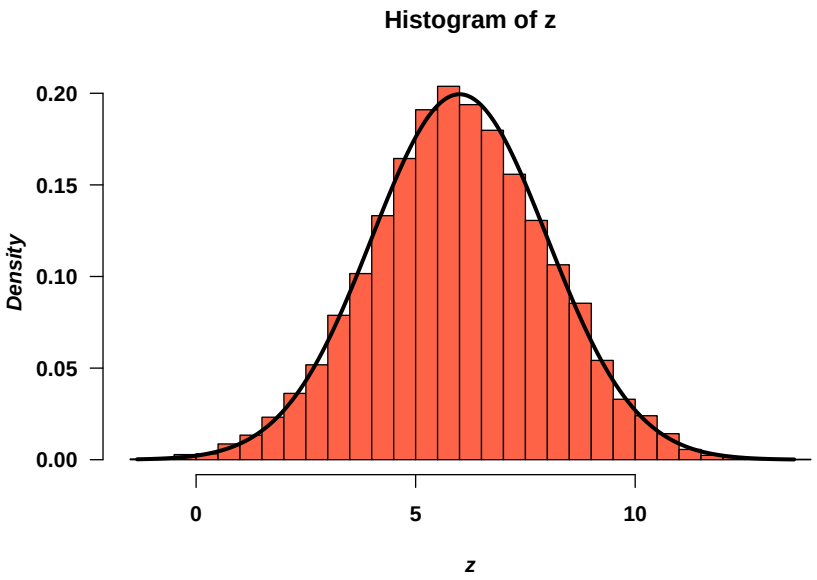

Figura 4.6: Histograma con la función de densidad de la distribución $\mathcal{N}(6, 2)$

3. También puede resultar ilustrativo complementar los histogramas con gráficos de densidad estimada, es decir histogramas suavizados. La función `density()` permite obtener los valores para dibujar estos histogramas suavizados. Dibujemos con trazo continuo la función de densidad de datos aleatorios de una distribución χ^2_4 (Figura 4.7):

```
> # Código de la Figura 4.7
> windows(width = 7, height = 5)
> par(font = 2, font.lab = 4, font.axis = 2, las = 1)
> z <- rchisq(1500, 4)
> hist(z, breaks = 50, col = "wheat2", freq = F)
> lines(density(z), type = "l", lwd = 3)
```

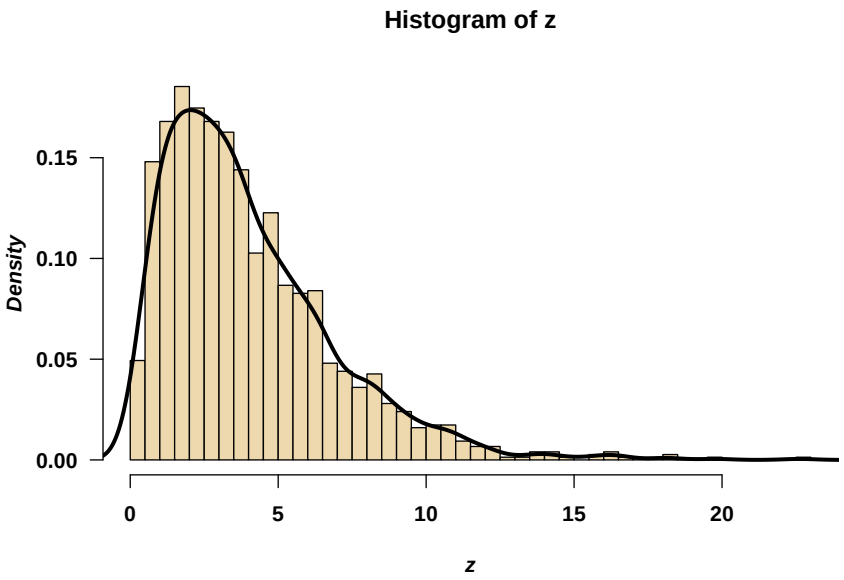

Figura 4.7: Histograma con función de densidad estimada

4. Para ver de qué colores disponemos en R para hacer los gráficos más vistosos, se puede ejecutar el comando `colours()`. ¡Hay más de 650 colores!

5. La función `boxplot(x)` dibuja un diagrama de caja (*boxplot*) de los datos del vector `x`. Dos gráficos de este tipo se muestran en la Figura 4.8.

```
> # Código de la Figura 4.8
> windows(width = 9, height = 5)
> par(mfrow = c(1, 2), font = 2, font.lab = 4, font.axis = 2, las = 1)
> x <- rnorm(1000, 15, 3)
> fac <- gl(4, 250)       # factor con 4 niveles de longitud 250
> boxplot(x)
> boxplot(x~fac, col = 2:5, lwd = 2)
```

6. R permite guardar los indicadores característicos de un *boxplot* en una lista (utilizando la opción `plot = F`). Éstos se pueden aprovechar para añadir más elementos al mismo gráfico, por ejemplo el valor de la mediana (Figura 4.9) que se guarda en la matriz `stats` de esa lista.

```
> bx <- boxplot(x~fac, plot = F)
> bx$stats[3, ]               # Las medianas
[1] 15.14980 14.68217 15.02205 15.01323

> # Código de la Figura 4.9
> windows(width = 8, height = 6)
```

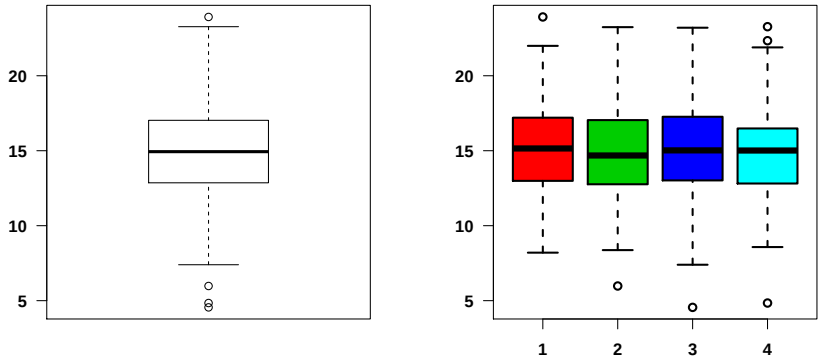

Figura 4.8: Dos diagramas de caja para 1000 datos aleatorios de una distribución normal

```
> par(font = 2, font.lab = 4, font.axis = 2, las = 1)
> boxplot(x~fac, col = gray(c(.8, .85, .9, .95)), lwd = 2, pch = 16)
> text(1:4, bx$st[3, ]-0.1, paste("Median: ", round(bx$st[3, ], 2)),
+      adj = c(0.5, 1), font = 2)
```

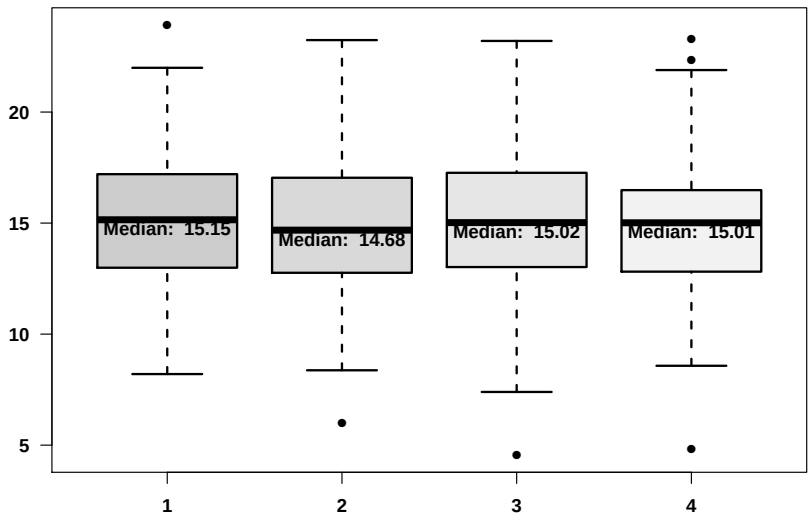

Figura 4.9: Diagrama de caja con texto

7. Una de las mejores formas de comparar la distribución de una muestra con la de una ley dada es mediante un *Q-Q plot*. Un *Q-Q plot* es un gráfico de los cuantiles de la distribución

empírica versus los cuantiles de la distribución teórica. En particular, cuando la distribución teórica es la normal podemos utilizar la función `qqnorm`. Para ver más claramente la calidad del ajuste se le puede añadir al gráfico una recta mediante la función `qqline`. Veamos dos ejemplos con datos provenientes de una distribución normal y otros de una distribución χ^2 (Figura 4.10).

```
> # Código de la Figura 4.10
> x <- rnorm(1000, 10, 3)
> y <- rchisq(1000, 5)
> windows(width = 9, height = 5)
> par(mfrow = c(1, 2), font = 2, font.lab = 4, font.axis = 2, las = 1)
> qqnorm(x, pch = 19)
> qqline(x, lwd = 2)
> qqnorm(y, pch = 19)
> qqline(y, lwd = 2)
```

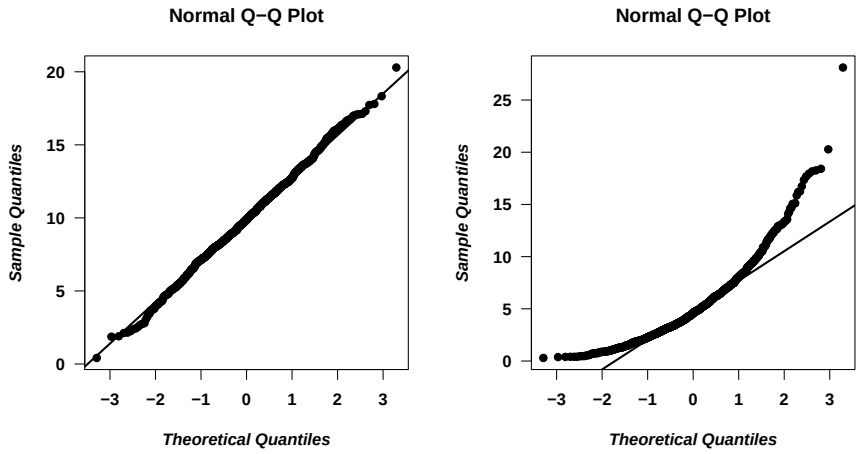

Figura 4.10: *Q-Q plots* para datos de una distribución normal (izquierda) y χ^2

4.3. Representación de datos categóricos

1. Una manera sencilla de presentar gráficamente la distribución de una variable categórica es el diagrama de pastel (*pie chart*) que muestra las frecuencias de cada una de las categorías. Usamos para ilustrar el uso de la función `pie` la variable 'Income' de *data frame* `state.x77`, la que convertimos primero en una variable ordinal mediante la función `cut`.

```
> state.x77$inco2 <- cut(state.x77$Income, breaks = c(3000, 4000, 4500, 5000, 6500),
+                        dig.lab = 4)
> summary(state.x77$inco2)
```

```
(3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
          13           11           18           8
```

A continuación aplicamos la función `pie` para dibujar el gráfico que se muestra en la Figura 4.11. Notad que se ha de pasar la variable en forma de una tabla; sin usar la función `table`, la función no dibujaría el gráfico.

```
> # Código de la Figura 4.11
> windows()
> par(font = 2, font.lab = 4, font.axis = 2)
> pie(table(state.x77$inco2), col = 2:5, main = "Average Income in 50 States",
+     clockwise = T)
```

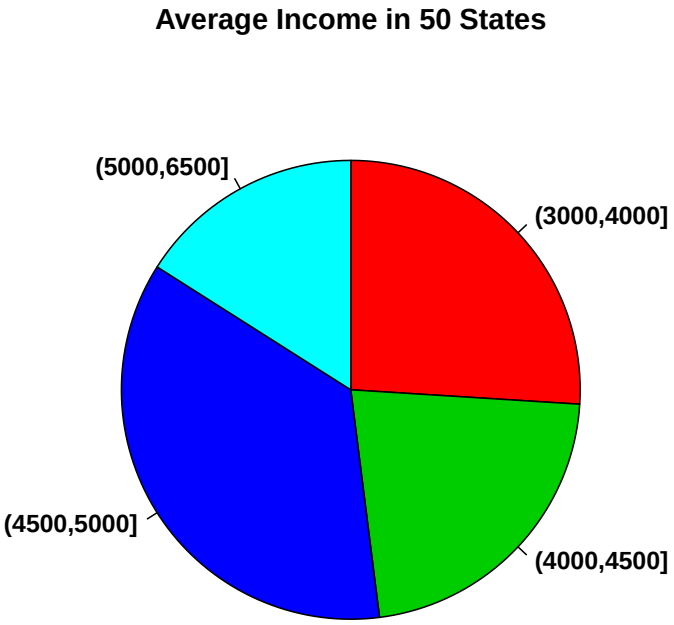

Figura 4.11: Diagrama de pastel del salario medio en los estados de EE UU en 1977

2. Si queremos representar gráficamente la relación entre dos variables categóricas, podemos hacerlo mediante un diagrama de barras. Nos podría interesar por ejemplo, cuál era la distribución de la variable salario medio del ejemplo anterior en cada una de las cuatro regiones de EE UU en el año 1977. Añadamos para ello primero la variable 'Región' al

data frame `state.x77` que está guardada en la matriz `state.region` y luego construimos el diagrama de barra respectivo.

```
> state.x77$region <- state.region
> summary(state.x77$region)

    Northeast         South North Central          West
            9            16            12            13

> with(state.x77, table(region, inco2))

               inco2
region          (3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
  Northeast                2           2           3           2
  South                   10           2           3           1
  North Central            0           4           6           2
  West                     1           3           6           3
```

La función `barplot` permite dibujar el diagrama de barra de interés. Como en caso de la función `pie`, se le han de pasar las variables a `barplot` en forma de una tabla. El diagrama de barra se muestra en la Figura 4.12 (página 57).

```
> # Código de la Figura 4.12
> windows(width = 8, height = 6)
> par(font = 2, font.lab = 4, font.axis = 2)
> with(state.x77, barplot(table(inco2, region), legend.text = T, col = 2:5,
+                         ylim = c(0, 18)))
> title("Average income per region")
```

Nota: Con la opción `ylim` se ha ampliado el rango de la ordenada para que la leyenda no se solape con la barra de la region *West*.

3. Como alternativa a un diagrama de barras se recomienda el uso de la función `mosaicplot` que permite la representación gráfica de una tabla de contingencia. Por ejemplo, el gráfico en la Figura 4.13 representa la distribución condicional de la variable salario en cada una de las cuatro regiones y además la distribución marginal de éstas.

```
> # Código de la Figura 4.13
> windows(width = 7, height = 6)
> par(font = 2, font.lab = 4, font.axis = 2, las = 1)
> with(state.x77, mosaicplot(region~inco2, col = 2:5, xlab = "Region",
+                            ylab = "Average income",
+                            main = "Average income per region", cex.axis = 1))
```

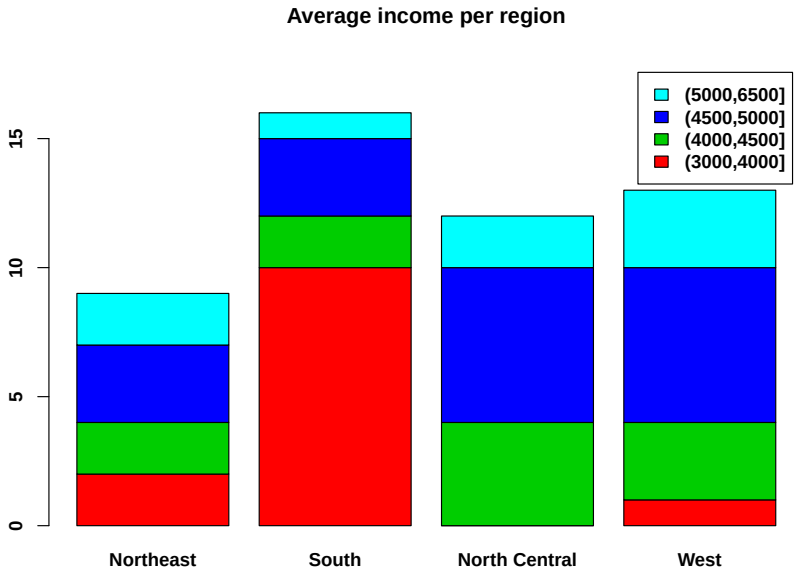

Figura 4.12: Diagrama de barra de las variables salario medio y región en los estados de EE UU en 1977

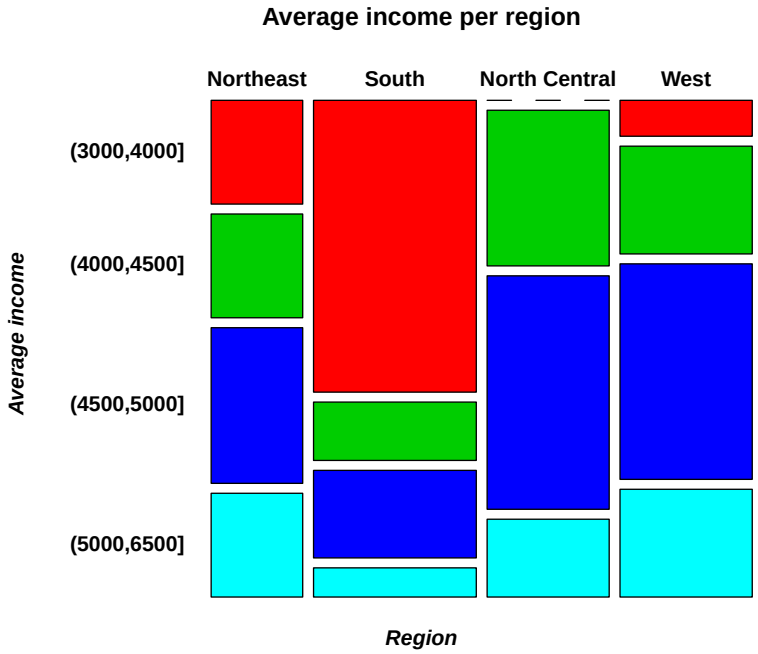

Figura 4.13: Gráfico de mosaicos de las variables salario medio y región en los estados de EE UU en 1977

Existen muchos paquetes de R para dibujar gráficos de muy buena calidad. Se recomiendan, por ejemplo, los paquetes `lattice` [9] o `ggplot2` [10]. Ambos paquetes contienen funciones para la representación de datos longitudinales.
Además es recomendable explorar los paquetes presentados en la página web *R Graph Gallery* (`http://rgraphgallery.blogspot.com.es/`) para obtener una visión de las posibilidades que existen con R.

Actividad 5

Creación de funciones propias y programación básica en R

Contenido

Una de las ventajas de R es la posibilidad de poder crear funciones que se adapten a la necesidad del usuario. Veremos en esta actividad cómo realizarlo y qué opciones se le ofrecen al usuario. Además se presentarán algunas de las instrucciones existentes para el control de flujos de programas.

5.1. Creación de funciones

1. R permite al usuario el personalizar algunas de las funciones incorporadas en el lenguaje así como el crear nuevas funciones. En general, una función espera los valores de unos argumentos y, en un principio, devuelve al usuario el resultado de la última acción. Para ver el contenido de una función es suficiente escribir su nombre (sin paréntesis). Por ejemplo, el código de la función `matrix` es el siguiente:

```
> matrix

function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
{
    if (is.object(data) || !is.atomic(data))
        data <- as.vector(data)
    .Internal(matrix(data, nrow, ncol, byrow, dimnames, missing(nrow),
        missing(ncol)))
}
<bytecode: 0x06c307b0>
<environment: namespace:base>
```

2. Veamos ahora cómo definir una función nueva en R. Por ejemplo, como la función `var` no ofrece opción alguna para calcular el estimador de máxima verosimilitud de la varianza, es decir

$$\mathrm{Var}_{ML}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2, \tag{5.1}$$

podemos crear una función, `varML`, que lo realice. Para ello, abrid el editor mediante

```
> fix(varML)
```

y en él escribid (omitiendo los comentarios):

```
function(x)            # x: vector que se le pasa a la función varML
{
  n <- length(x)         # Número de datos
  val <- var(x)*(n-1)/n  # La variable val es temporal
  val                  # La función devuelve el contenido de val
}
```

Después de cerrar el editor, comprobemos que la función `varML` existe pero ningún objeto identificado como `n` o `val` y apliquemos la nueva función al vector de los datos de 1 a 10:

```
> varML

function(x)
{
  n <- length(x)
  val <- var(x)*(n-1)/n
  val
}

> val             # causa mensaje de error

> x <- 1:10
> mean(x)

[1] 5.5

> var(x)

[1] 9.166667

> varML(x)

[1] 8.25
```

3. Para crear una función con un código más largo, es recomendable programarla en una ventana *script* y guardar el mismo. En este caso, en vez de ejecutar `fix(varML)`, se ha de usar el comando `function` de la siguiente manera:

```
> varML <-  function(x)
+ {
+   n <- length(x)
+   val <- var(x)*(n-1)/n
+   val
+ }
```

4. Para que una función devuelva los resultados de más de un cálculo, hay que guardar los valores en una lista o utilizar las funciones `print` y/o `cat`. En caso contrario, la función devolverá solamente el resultado del último cálculo. Para ilustrarlo creamos tres funciones, `vars1`, `vars2` y `vars3`, que calculan tanto la varianza empírica como la varianza según la fórmula (5.1):

```
> vars1 <-  function(x){
+   val1 <- var(x)
+   val2 <- varML(x)
+   val1
+   val2
+ }

> vars2 <-  function(x){
+   val1 <- var(x)
+   val2 <- varML(x)
+   print(val1)
+   cat("Estimador de MV:", val2, "\n")
+ }

> vars3 <-  function(x){
+   val1 <- var(x)
+   val2 <- varML(x)
+   list("Varianza" = val1, "Varianza según MV" = val2)
+ }
```

Veamos qué resultados devuelven las tres funciones aplicadas al vector `x`:

```
> vars1(x)

[1] 8.25

> vars2(x)

[1] 9.166667
Estimador de MV: 8.25

> vars3(x)
```

```
$Varianza
[1] 9.166667

$`Varianza según MV`
[1] 8.25
```

5. Otro ejemplo: puede ser útil crear una función que genere los siguientes cuatro gráficos:

   a) Histograma
   b) Boxplot
   c) Gráfico de densidad
   d) *Q-Q plot*

   Esta función, llamémosla `cuatfigs`, la podemos programar con el siguiente código:

```
> cuatfigs <- function(x)
+ {
+    windows()
+    par(mfrow = c(2, 2))
+    hist(x, main = "Histogram of the data")
+    boxplot(x, main = "Boxplot of the data")
+    iqd <- summary(x)[5]-summary(x)[2]
+    plot(density(x, width = 2*iqd), main = "Kernel estimate of the density",
+         xlab = "x", ylab = "", type = "l")
+    qqnorm(x)
+    qqline(x)
+ }
```

   Para comprobar que funciona apliquemos la función a un vector de datos provenientes de una distribución normal. El resultado se muestra en la Figura 5.1.

```
> # Código del gráfico 5.1
> x <- rnorm(1000, 12, 3)
> cuatfigs(x)
```

6. Se les puede asignar valores por defecto a los argumentos de una función. Esto implica que, si no se les pasa el parámetro a la función, éste tomará el valor por defecto. Por ejemplo, la función `log` calcula por defecto el logaritmo natural; para utilizar otra base hay que modificar este valor:

```
> log

function (x, base = exp(1))  .Primitive("log")

> log(100)
```

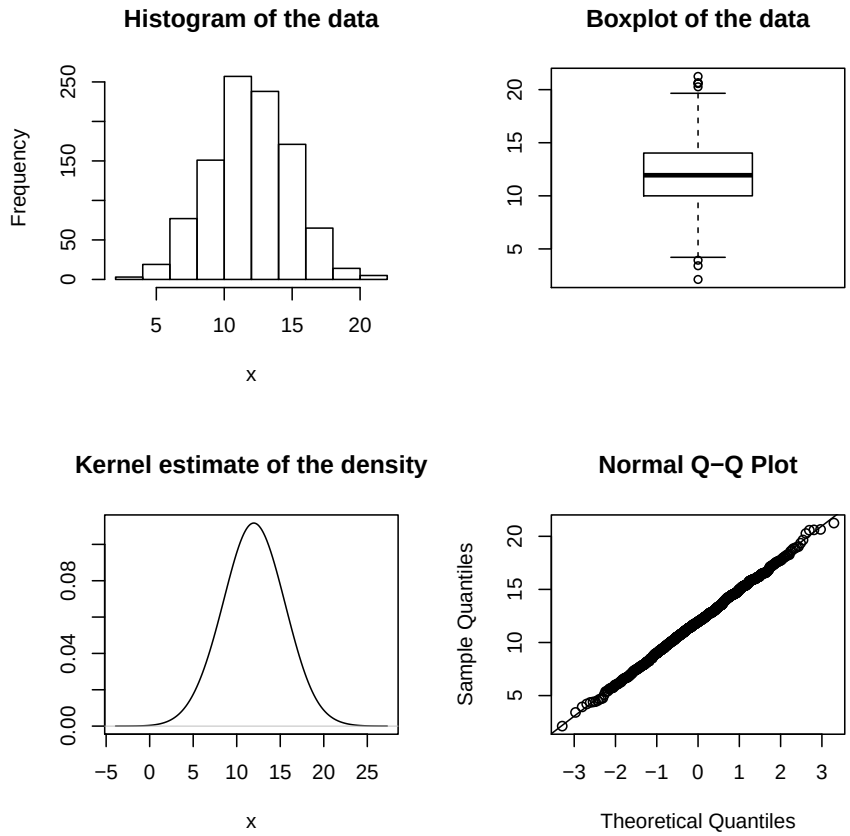

Figura 5.1: Salida de la función `cuatfigs`

```
[1] 4.60517
```

```
> log(100, exp(1))
```

```
[1] 4.60517
```

```
> log(100, 10)
```

```
[1] 2
```

7. Si creamos una función que internamente llama a otra ya existente, podemos definir opciones de ésta como argumentos de aquella. Para conseguirlo, se le añade a la lista de argumentos una secuencia de tres puntos (`...`). Lo ilustramos con una mejora de la función `cuatfigs` (que llamamos `cuatbetter`). Ésta, aparte de tener dos nuevas opciones, `clr` y `lwi`, para colorear el diagrama de caja y la recta del *Q-Q plot* y cambiar el grosor de algunas líneas, respectivamente, nos permite especificar opciones de la función `par`.

```
> cuatbetter <- function(x, clr = 2, lwi = 2, ...)
+ {
```

```
+    windows()
+    par(mfrow = c(2, 2), ...)
+    hist(x, main = "Histogram of the data")
+    boxplot(x, main = "Boxplot of the data", col = clr)
+    iqd <- summary(x)[5]-summary(x)[2]
+    plot(density(x, width = 2*iqd), main = "Kernel estimate of the density",
+         xlab = "x", ylab = "", type = "l", lwd = lwi)
+    qqnorm(x)
+    qqline(x, col = clr, lwd = lwi)
+ }
```

En la Figura 5.2 mostramos un ejemplo, en el cual, entre otras cosas, cambiamos la orientación de la numeración de la ordenada y las fuentes de los ejes:

```
> # Código del gráfico 5.2
> x <- rchisq(1000, 3)
> cuatbetter(x, clr = "steelblue", las = 1, font.axis = 2, font.lab = 4, pch = 19)
```

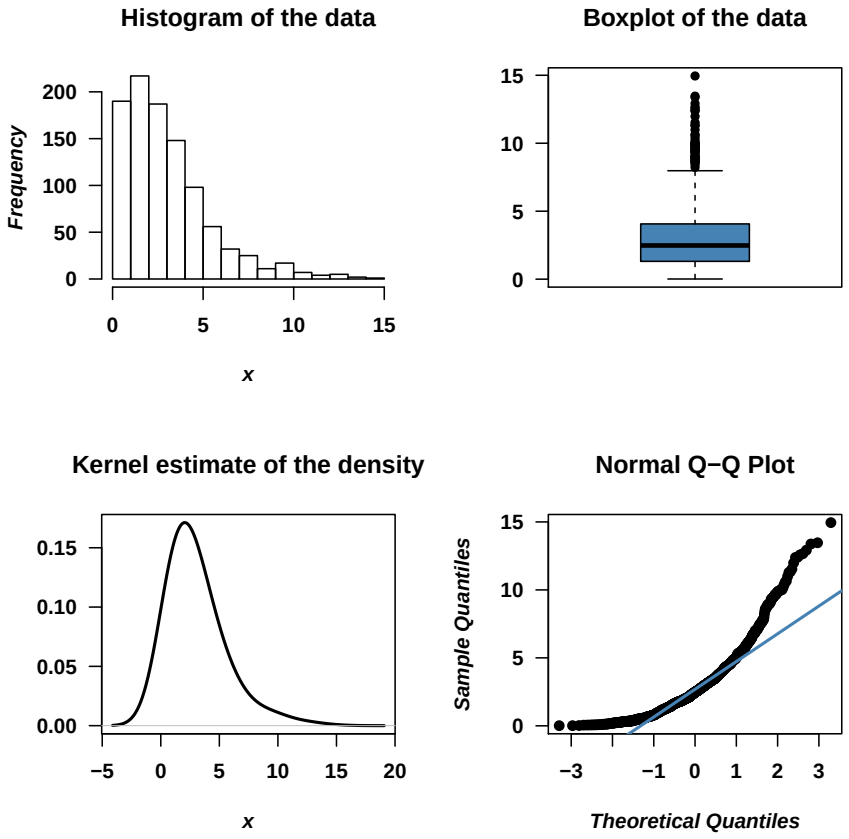

Figura 5.2: Salida de la función `cuatbetter`

8. Las funciones se les puede guardar como elementos de un espacio de trabajo de R. Así, al volver a abrirlo, las funciones están disponibles en seguida. No obstante, se recomienda guardar los *scripts* (con las definiciones de funciones) y cargarlos en cada sesión que sea necesario mediante la función `source()` o desde la barra de herramientas:

   Archivo ▶ Interpretar código fuente R...

5.2. Programación básica en R

1. Como en cualquier otro lenguaje de programación, en R existen instrucciones para controlar el flujo de un programa: `for`, `if else`, `while`, `repeat`, etc. Veremos a continuación algunos ejemplos, empezando por la función `for` que permite automatizar la ejecución de ciertos comandos tantas veces que se especifique:

```
> for (i in 1:5)
+     print(i^2)

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25

> for (i in c(1, 4, 10)){
+         print("¡Buenos días!")
+         cat("La raíz de", i, "es", sqrt(i), fill = T)
+ }

[1] "¡Buenos días!"
La raíz de 1 es 1
[1] "¡Buenos días!"
La raíz de 4 es 2
[1] "¡Buenos días!"
La raíz de 10 es 3.162278
```

   Nota: Es imprescindible usar las funciones `print` o `cat` si queremos que se devuelva un resultado u objeto en cada iteración del bucle.

2. Usando la función `while(`*condición*`)` los comandos del bucle se ejecutan repetidamente mientras se cumpla *condición*:

```
> x <- 1:6
> i <- 1
> while (x[i] < 4){
+   cat("x[", i, "] es igual a ", x[i], " e inferior a 4", sep = "", fill = T)
+   i = i+1
+ }
```

```
x[1] es igual a 1 e inferior a 4
x[2] es igual a 2 e inferior a 4
x[3] es igual a 3 e inferior a 4
```

3. Los comandos `if(`*`condición`*`) else` se pueden usar para ejecutar ciertas instrucciones en caso de que se cumpla *`condición`* u otras en caso contrario:

```
> (x <- rbinom(6, 20, 0.5))

[1]  9 12  9 11  6 10

> for (i in 1:6){
+   if (x[i] < 10){
+     cat("x[", i, "] es inferior a 10", sep = "", fill = T)
+   } else{
+     cat("x[", i, "] es superior o igual a 10", sep = "", fill = T)
+   }
+ }

x[1] es inferior a 10
x[2] es superior o igual a 10
x[3] es inferior a 10
x[4] es superior o igual a 10
x[5] es inferior a 10
x[6] es superior o igual a 10
```

   Para ver más ejemplos y otros comandos para el control del flujo, estudiad la ayuda:

```
> help(Control)
```

4. Se recomienda evitar, siempre que sea posible, los bucles de la función `for` debido a que ralentizan bastante el trabajo en R. Comparad la siguiente adición de dos vectores `x` e `y` de forma directa con la de la aplicación de la función `for`:

```
> x <- rnorm(20000)
> y <- rnorm(20000)

> z <- x + y                # muy rápido
> z2 <- numeric()
> for (i in 1:20000) {       # muy lento
+         z2[i] <- x[i]+y[i]
+ }
```

   Utilizando la función `system.time` podemos medir el tiempo de ejecución de las dos maneras de sumar vectores:

```
> system.time(z <- x + y)
```

```
   user  system elapsed
      0       0       0
```

```
> z2 <- numeric()
> system.time(
+         for(i in 1:20000)
+         z2[i] <- x[i]+y[i]
+ )
```

```
   user  system elapsed
   1.41    0.03    1.44
```

5. Una función muy práctica que nos permite prescindir del uso de los bucles es la función `ifelse(`*`condición`*`, A, B)`. Comprueba si se cumple *`condición`* y en caso afirmativo devuelve `A`, en caso contrario `B`. Veamos dos ejemplos:

```
> (x <- rbinom(10, 1, .5))
```

```
 [1] 1 1 0 0 0 1 1 0 0 1
```

```
> sexo <- factor(ifelse(x ==0, "Hombre", "Mujer"))
> sexo
```

```
 [1] Mujer  Mujer  Hombre Hombre Hombre Mujer  Mujer  Hombre Hombre Mujer
Levels: Hombre Mujer
```

```
> (x <- rpois(10, 22))
```

```
 [1] 28 18 23 24 19 21 17 17 26 17
```

```
> factor(ifelse(x <= 20, "Joven", ifelse(x <= 25, "No tan joven", "Mayor")))
```

```
 [1] Mayor        Joven        No tan joven No tan joven Joven
 [6] No tan joven Joven        Joven        Mayor        Joven
Levels: Joven Mayor No tan joven
```

Para ver cómo comprobar dos o más condiciones simultáneamente, mirad los ejemplos de los operadores lógicos de la librería `base`:

```
> help(Logic)
```

Actividad 6

Pruebas estadísticas para dos poblaciones y modelos de regresión

Contenido

R ofrece funciones tanto para realizar las pruebas estadísticas estándares (y muchas más) como para el ajuste de modelos de regresión. En este capítulo se presentarán funciones centrándonos en pruebas de dos poblaciones y en ajustar un modelo lineal sencillo. Funciones para aplicar otras pruebas u otros modelos estadísticos suelen tener las mismas características como las aquí presentadas.

6.1. Pruebas estadísticas para dos poblaciones

6.1.1. Pruebas de independencia para dos variables categóricas

1. Dadas las muestras de dos variables categóricas suele interesar si existe independencia entre ellas. Las pruebas estadísticas más comunes para contrastar la hipótesis de independencia son la prueba de χ^2 y la prueba exacta de Fisher. Las funciones de R que llevan a cabo ambas pruebas son `chisq.test` y `fisher.test`, respectivamente. Veamos su aplicación a los datos `state.x77` de la Sección 4.1 examinando si existe una asociación entre la región de los estados y el salario medio (tratando esta variable como una variable categórica).

```
> with(state.x77, table(region, inco2))

               inco2
region          (3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
  Northeast                2           2           3           2
  South                   10           2           3           1
  North Central            0           4           6           2
  West                     1           3           6           3

> with(state.x77, chisq.test(region, inco2))
```

```
        Pearson's Chi-squared test

data:  region and inco2
X-squared = 18.3523, df = 9, p-value = 0.0313

> with(state.x77, fisher.test(region, inco2))

        Fisher's Exact Test for Count Data

data:  region and inco2
p-value = 0.02531
alternative hypothesis: two.sided
```

Nota: Como `state.x77` contiene todos los estados de EE UU, de hecho, disponemos de datos poblacionales y no muestrales y no haría falta aplicar pruebas estadísticas para estudiar la relación entre dos variables a nivel de EE UU. No obstante, aplicamos estas pruebas con el fin de ilustrar el uso de las funciones de las pruebas estadísticas más relevantes.

2. R ofrece la posibilidad de guardar la aplicación de cada una de estas funciones como un objeto; se trata en cada caso de una lista. Posteriormente podemos referirnos a cada uno de sus elementos.

```
> ct <- with(state.x77, chisq.test(region, inco2))
> names(ct)

[1] "statistic" "parameter" "p.value"   "method"    "data.name" "observed"
[7] "expected"  "residuals" "stdres"

> is.list(ct)

[1] TRUE

> ct[1]

$statistic
X-squared
 18.35232

> ct$p.value

[1] 0.03129913

> ct$observed

               inco2
region          (3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
  Northeast                2            2            3            2
  South                   10            2            3            1
  North Central            0            4            6            2
  West                     1            3            6            3
```

```
> ct$expected

               inco2
region          (3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
  Northeast             2.34        1.98        3.24        1.44
  South                 4.16        3.52        5.76        2.56
  North Central         3.12        2.64        4.32        1.92
  West                  3.38        2.86        4.68        2.08

> ft <- with(state.x77, fisher.test(region, inco2))
> names(ft)

[1] "p.value"     "alternative" "method"      "data.name"

> ft$p.value

[1] 0.02530768
```

6.1.2. Comparación de medias, medianas y varianzas de dos poblaciones

1. Si se quieren comparar dos poblaciones con respecto a la media disponiendo de datos muestrales, se puede aplicar la prueba t. Para ilustrar la función `t.test` se generan dos vectores de datos proviniendo de distribuciones normales:

```
> x <- rnorm(50, 3, 2)
> y <- rnorm(75, 4, 1.5)
> t.test(x, y)

        Welch Two Sample t-test

data:  x and y
t = -4.2765, df = 82.975, p-value = 5.048e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.2490455 -0.8211379
sample estimates:
mean of x mean of y
 2.665440  4.200531
```

   Como se puede ver, la función devuelve no solamente un valor p, sino también la estimación puntual y el intervalo de confianza (al 95 %) de la diferencia de las dos medias poblacionales. El intervalo de confianza se puede extraer también de la siguiente manera:

```
> names(t.test(x, y))

[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
[6] "null.value"  "alternative" "method"      "data.name"

> t.test(x, y)$estimate
```

```
mean of x mean of y
 2.665440  4.200531
```

```
> t.test(x, y)$conf.int
```

```
[1] -2.2490455 -0.8211379
attr(, "conf.level")
[1] 0.95
```

Nota: La misma función se puede aplicar también a un solo vector. En este caso, se lleva a cabo la prueba t para una sola muestra (contrastando por defecto la hipótesis $\mu = 0$):

```
> t.test(x)
```

```
        One Sample t-test

data:  x
t = 8.6414, df = 49, p-value = 2.031e-11
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 2.045586 3.285293
sample estimates:
mean of x
  2.66544
```

2. En muchas ocasiones, los datos están organizados en dos vectores: un vector numérico y otro factor indicando a qué grupo pertenecen los datos. Veamos un ejemplo utilizando uno de los conjuntos de datos (`sleep`) del paquete `datasets`.

```
> ?sleep
```

Se trata de los datos de un estudio sobre los efectos de dos soporíferos. Los datos miden el incremento de sueño (en horas) comparado con un placebo.

```
> summary(sleep)
```

```
     extra        group        ID
 Min.   :-1.600   1:10   1      :2
 1st Qu.:-0.025   2:10   2      :2
 Median : 0.950          3      :2
 Mean   : 1.540          4      :2
 3rd Qu.: 3.400          5      :2
 Max.   : 5.500          6      :2
                         (Other):8
```

```
> with(sleep, by(extra, group, summary))
```

```
group: 1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.600  -0.175   0.350   0.750   1.700   3.700
------------------------------------------------------------
group: 2
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -0.100   0.875   1.750   2.330   4.150   5.500
```

La Figura 6.1 muestra los datos.

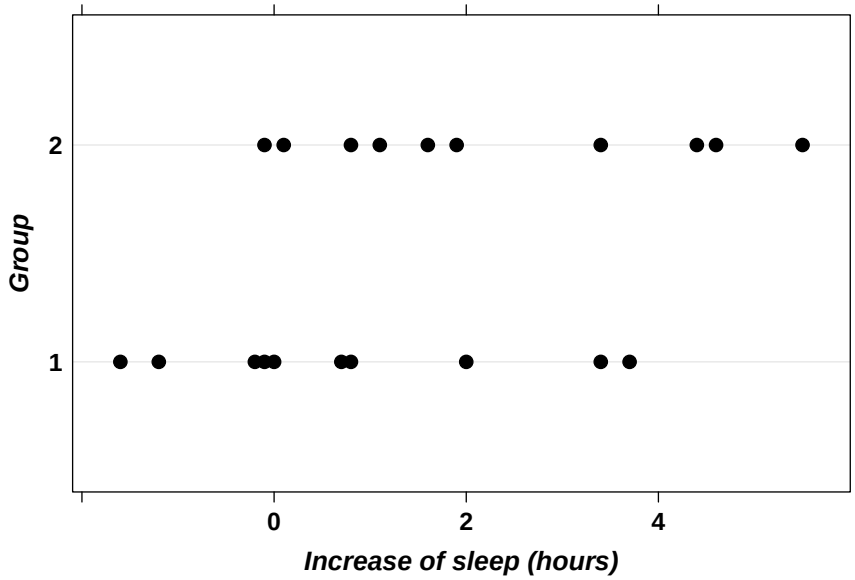

Figura 6.1: Datos del *data frame* `sleep`: incremento de horas de sueño en dos grupos de soporíferos comparados con un placebo

Apliquemos ahora la prueba t para la comparación de ambos medicamentos respecto al incremento medio de sueño:

```
> tt <- t.test(extra~group, data = sleep)
> tt

        Welch Two Sample t-test

data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.3654832  0.2054832
sample estimates:
mean in group 1 mean in group 2
           0.75            2.33
```

```
> tt$estimate

mean in group 1 mean in group 2
           0.75            2.33
```

3. En caso de que los datos de interés provengan de dos muestras apareadas, hay que usar la opción `paired` de `t.test`:

```
> x <- rnorm(50, 3, 2)
> y <- x+rnorm(50, 1, 1.5)
> t.test(x, y, paired = T)

        Paired t-test

data:  x and y
t = -5.2532, df = 49, p-value = 3.229e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.336060 -0.596695
sample estimates:
mean of the differences
             -0.9663773
```

4. Por defecto, la función `t.test` supone varianzas desiguales en ambas poblaciones. Para comprobar si no podemos suponer lo contrario, es decir homocedasticidad, podemos usar las funciones `var.test` y `leveneTest`. Ésta última, que no supone distribución normal de los datos, está disponible en el paquete `car` [12].

```
> var.test(extra~group, data = sleep)

        F test to compare two variances

data:  extra by group
F = 0.7983, num df = 9, denom df = 9, p-value = 0.7427
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.198297 3.214123
sample estimates:
ratio of variances
         0.7983426

> library(car)
> with(sleep, leveneTest(extra, group))

Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  1  0.2482 0.6244
      18
```

Basándonos en el resultado de ambas pruebas, podemos suponer homocedasticidad y volver a aplicar la prueba t:

```
> t.test(extra~group, data = sleep, var.equal = T)

        Two Sample t-test

data:  extra by group
t = -1.8608, df = 18, p-value = 0.07919
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.363874  0.203874
sample estimates:
mean in group 1 mean in group 2
           0.75            2.33
```

Nota: Mientras la prueba F (función `var.test`) sirve para comprobar la homocedasticidad en dos poblaciones, la prueba de Levene la podemos aplicar también si disponemos de muestras de más de dos poblaciones.

5. Una alternativa a las pruebas paramétricas son las pruebas no-paramétricas. En este caso, la hipótesis a contrastar es la de la igualdad de medianas de dos poblaciones y la prueba que se aplica es la de Wilcoxon. En R la ejecutamos mediante la función `wilcox.test`

```
> wt <- wilcox.test(extra~group, data = sleep, exact = T)
> wt

        Wilcoxon rank sum test with continuity correction

data:  extra by group
W = 25.5, p-value = 0.06933
alternative hypothesis: true location shift is not equal to 0
```

Nota: Esta función aplica un cálculo exacto del valor p solamente en caso de que no haya empates. Si éste es el caso, se puede usar la función `wilcox_test` del paquete `coin` [13] para obtener el valor p exacto:

```
> library(coin)
> wilcox_test(extra~group, data = sleep, distribution = "exact")

        Exact Wilcoxon Mann-Whitney Rank Sum Test

data:  extra by group (1, 2)
Z = -1.8541, p-value = 0.06582
alternative hypothesis: true mu is not equal to 0
```

6.2. Construcción de modelos lineales

1. La función `lm` se usa para ajustar un modelo lineal, sea un modelo de regresión lineal, de análisis de varianza o de análisis de covarianza (ANCOVA). A continuación volvemos a utilizar el *data frame* state.x77 con el fin de ajustar un modelo lineal para la variable esperanza de vida. Calcularemos primero las correlaciones entre las variables numéricas para después ajustar un primer modelo de ANCOVA mediante la función `lm` que guardamos bajo el nombre `lmod`:

```
> round(cor(state.x77[1:8]), 2)

           Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area
Population       1.00   0.21       0.11    -0.07   0.34   -0.10 -0.33  0.02
Income           0.21   1.00      -0.44     0.34  -0.23    0.62  0.23  0.36
Illiteracy       0.11  -0.44       1.00    -0.59   0.70   -0.66 -0.67  0.08
Life.Exp        -0.07   0.34      -0.59     1.00  -0.78    0.58  0.26 -0.11
Murder           0.34  -0.23       0.70    -0.78   1.00   -0.49 -0.54  0.23
HS.Grad         -0.10   0.62      -0.66     0.58  -0.49    1.00  0.37  0.33
Frost           -0.33   0.23      -0.67     0.26  -0.54    0.37  1.00  0.06
Area             0.02   0.36       0.08    -0.11   0.23    0.33  0.06  1.00

> lmod <- lm(Life.Exp~Illiteracy+Murder+inco2, data = state.x77)
> lmod

Call:
lm(formula = Life.Exp ~ Illiteracy + Murder + inco2, data = state.x77)

Coefficients:
     (Intercept)        Illiteracy            Murder  inco2(4000, 4500]
         72.1104            0.1069           -0.2588            0.6102
inco2(4500, 5000]  inco2(5000, 6500]
          0.8946            0.6018
```

2. Si aplicamos la función `summary` a un objeto `lm`, obtenemos una salida más detallada e informativa. Guardamos a continuación el contenido de esta salida y veremos que en parte se guardan aspectos distintos del modelo ajustado:

```
> summary(lmod)

Call:
lm(formula = Life.Exp ~ Illiteracy + Murder + inco2, data = state.x77)

Residuals:
     Min       1Q   Median       3Q      Max
-1.43659 -0.62726  0.04705  0.48580  1.99648

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)       72.11040    0.47436 152.018  < 2e-16 ***
Illiteracy         0.10694    0.30484   0.351   0.7274
Murder            -0.25881    0.04566  -5.669 1.03e-06 ***
inco2(4000, 4500]  0.61023    0.38389   1.590   0.1191
inco2(4500, 5000]  0.89457    0.34772   2.573   0.0135 *
inco2(5000, 6500]  0.60185    0.41231   1.460   0.1515
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8218 on 44 degrees of freedom
Multiple R-squared:  0.6635,         Adjusted R-squared:  0.6252
F-statistic: 17.35 on 5 and 44 DF,  p-value: 1.853e-09
```

```
> sumod <- summary(lmod)
> names(lmod)

 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"
```

```
> names(sumod)

 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```
> lmod$coef

     (Intercept)       Illiteracy           Murder inco2(4000, 4500]
      72.1104034        0.1069391       -0.2588120        0.6102336
inco2(4500, 5000] inco2(5000, 6500]
       0.8945712        0.6018462
```

```
> round(sumod$coef, 3)

                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         72.110      0.474 152.018    0.000
Illiteracy           0.107      0.305   0.351    0.727
Murder              -0.259      0.046  -5.669    0.000
inco2(4000, 4500]    0.610      0.384   1.590    0.119
inco2(4500, 5000]    0.895      0.348   2.573    0.014
inco2(5000, 6500]    0.602      0.412   1.460    0.151
```

3. En combinación con la función `abline`, la función `lm` permite sobreponer una recta de regresión a un gráfico de dispersión. El siguiente código genera la Figura 6.2:

```
> # Código de la Figura 6.2
> windows(width = 7, height = 6)
> par(cex = 1.25, font = 2, font.lab = 4, font.axis = 2, las = 1)
> plot(Life.Exp~Murder, state.x77,
+      xlab = "Tasa de asesinatos (por 100000 habitantes)",
+      ylab = "Esperanza de vida (Años)", pch = 19)
> abline(lm(Life.Exp~Murder, state.x77), lwd = 3)
```

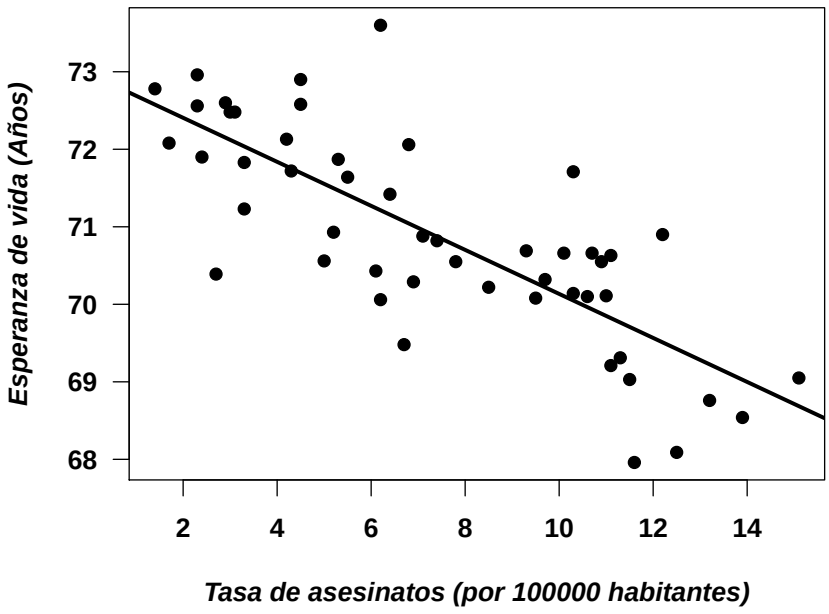

Figura 6.2: Esperanza de vida versus la tasa de asesinatos en los estados de EE UU en 1977

4. El uso de la función `by` (Sección 3.1) permite ajustar un modelo de regresión para cada uno de los niveles de un factor. Por ejemplo, se puede estudiar la relación entre la esperanza de vida y la tasa de asesinatos en cada región:

```
> bystat <- with(state.x77, by(state.x77, region, function(x)
+ summary(lm(Life.Exp~Murder, data = x))))
> is.list(bystat)

[1] TRUE

> names(bystat)

[1] "Northeast"     "South"         "North Central" "West"
```

```
> names(bystat$Northeast)

 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"

> round(coef(bystat$Northeast), 3)

            Estimate Std. Error t value Pr(>|t|)
(Intercept)   71.915      0.490 146.897    0.000
Murder        -0.138      0.091  -1.507    0.176

> round(coef(bystat$South), 3)

            Estimate Std. Error t value Pr(>|t|)
(Intercept)   71.954      0.948  75.907    0.000
Murder        -0.212      0.087  -2.439    0.029

> round(coef(bystat$"North Central"), 3)

            Estimate Std. Error t value Pr(>|t|)
(Intercept)   73.195      0.209 350.481        0
Murder        -0.271      0.033  -8.149        0

> round(coef(bystat$West), 3)

            Estimate Std. Error t value Pr(>|t|)
(Intercept)   73.697      0.861  85.642    0.000
Murder        -0.341      0.112  -3.038    0.011
```

5. Si tenemos un factor en el modelo, como en nuestro caso la variable `inco2`, R escoge el primer nivel como categoría de referencia para el ajuste del modelo. Si queremos cambiarla por otra, podemos aplicar la función `relevel` a esta variable:

```
> with(state.x77, summary(inco2))

(3000, 4000] (4000, 4500] (4500, 5000] (5000, 6500]
          13           11           18           8

> state.x77$inco2 <- relevel(state.x77$inco2, ref = "(4000, 4500]")
> with(state.x77, summary(inco2))

(4000, 4500] (3000, 4000] (4500, 5000] (5000, 6500]
          11           13           18           8
```

```
> summary(lm(Life.Exp~Illiteracy+Murder+inco2, state.x77))

Call:
lm(formula = Life.Exp ~ Illiteracy + Murder + inco2, data = state.x77)

Residuals:
     Min       1Q   Median       3Q      Max
-1.43659 -0.62726  0.04705  0.48580  1.99648

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       72.720637   0.348797 208.490  < 2e-16 ***
Illiteracy         0.106939   0.304843   0.351    0.727
Murder            -0.258812   0.045657  -5.669 1.03e-06 ***
inco2(3000, 4000] -0.610234   0.383892  -1.590    0.119
inco2(4500, 5000]  0.284338   0.315555   0.901    0.372
inco2(5000, 6500] -0.008387   0.384858  -0.022    0.983
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8218 on 44 degrees of freedom
Multiple R-squared:  0.6635,         Adjusted R-squared:  0.6252
F-statistic: 17.35 on 5 and 44 DF,  p-value: 1.853e-09
```

6. Para comprobar si el ajuste del modelo es satisfactorio se recomienda el uso de la función `plot` aplicada al objeto `lm`. Dibujará varios gráficos usando los residuos del modelo que nos pueden dar una idea si se cumplen las presuposiciones del modelo (véase Figura 6.3).

```
> # Instrucciones para Figura 5.3
> par(mfrow = c(2, 2), font.lab = 2, las = 1, font.axis = 2)
> plot(lmod, ask = F)
```

Además existe la función `residuals` que nos devuelve los residuos del modelo.

```
> residuals(lmod)[1:5]

   Alabama     Alaska    Arizona   Arkansas California
 0.6230849 -0.6380833 -0.6287318  0.9604130  1.5458804

> summary(residuals(lmod))

    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.43700 -0.62730  0.04705  0.00000  0.48580  1.99600
```

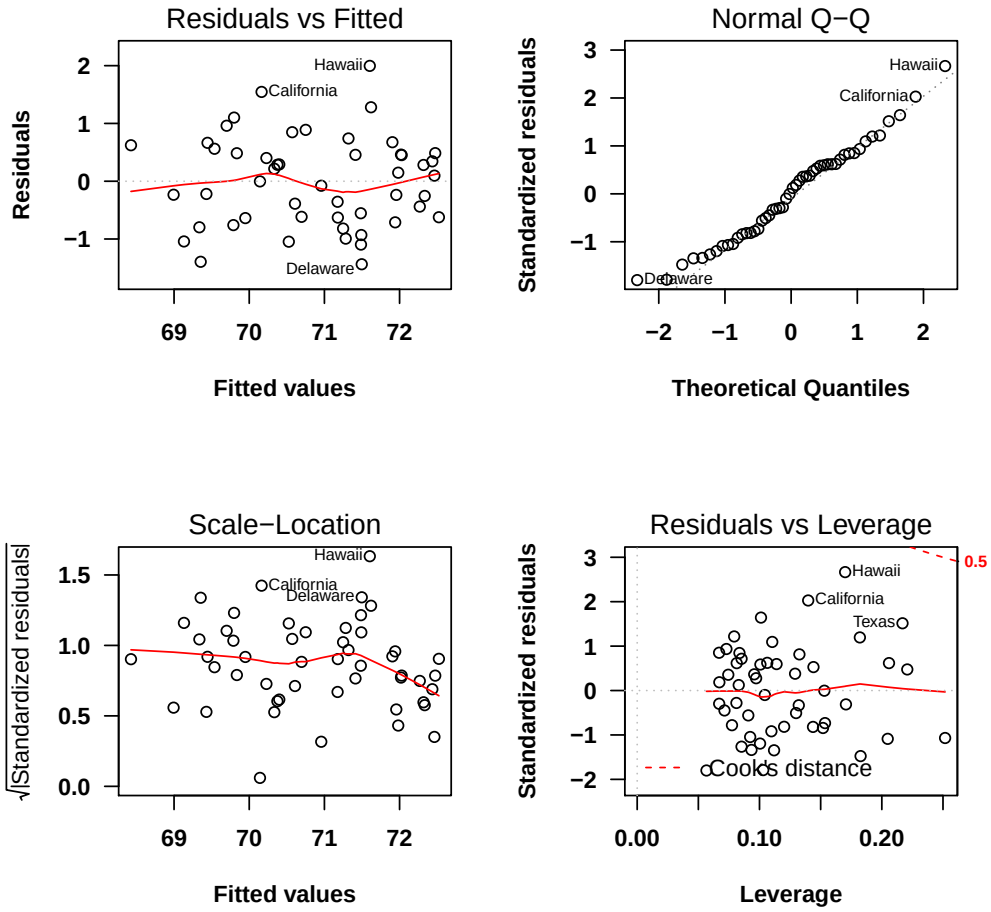

Figura 6.3: Gráficos de diagnósticos para un modelo de lineal

7. Existe la posibilidad de escoger las variables de un modelo de regresión según el procedimiento de la selección llamada *stepwise forward/backward*. Es un método muy controvertido y quizás no recomendable; aún así es muy popular y no está de más saber cómo realizarlo en R. Para ello se utiliza la función `step`:

```
> slm <- step(lm(Life.Exp~Illiteracy+Murder+HS.Grad+Frost+Area+region+inco2,
+                data = state.x77))

> summary(slm)

Call:
lm(formula = Life.Exp ~ Murder + HS.Grad + Frost + region, data = state.x77)

Residuals:
     Min       1Q   Median       3Q      Max
-1.18495 -0.45301 -0.02013  0.47848  1.19088

Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)         70.732287   1.437650  49.200  < 2e-16 ***
Murder              -0.259015   0.038693  -6.694 3.58e-08 ***
HS.Grad              0.054610   0.024701   2.211 0.032411 *
Frost               -0.008976   0.002499  -3.592 0.000837 ***
regionSouth         -0.126868   0.409159  -0.310 0.758006
regionNorth Central  0.669721   0.312011   2.146 0.037520 *
regionWest          -0.097658   0.397556  -0.246 0.807126
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7034 on 43 degrees of freedom
Multiple R-squared:  0.7591,         Adjusted R-squared:  0.7254
F-statistic: 22.58 on 6 and 43 DF,  p-value: 8.055e-12
```

Bibliografía

[1] Fox, J. (2005). The R Commander: A Basic Statistics Graphical User Interface to R. Journal of Statistical Software, 14(9): 1–42.

[2] Arriaza Gómez, A. J., F. Fernández Palacín, M. A. López Sánchez, M. Muñoz Márquez, S. Pérez Plaza y A. Sánchez Navas (2008). *Estadística Básica con R y R–Commander*. Servicio de Publicaciones de la Universidad de Cádiz. ISBN: 978-84-9828-186-6. http://knuth.uca.es/ebrcmdr

[3] Søren Højsgaard and Ulrich Halekoh (2016). doBy: Groupwise Statistics, LSmeans, Linear Contrasts, Utilities. R package version 4.5-15. https://CRAN.R-project.org/package=doBy

[4] Frank E Harrell Jr, with contributions from Charles Dupont and many others. (2016). Hmisc: Harrell Miscellaneous. R package version 3.17-4. https://CRAN.R-project.org/package=Hmisc

[5] R Development Core Team (2016). R Data Import/Export. Version 3.3.1 (2016-06-21).

[6] John Hendrickx (2013). catspec: Special models for categorical variables. R package version 0.97. https://CRAN.R-project.org/package=catspec

[7] Bendix Carstensen, Martyn Plummer, Esa Laara, Michael Hills (2016). Epi: A Package for Statistical Analysis in Epidemiology. R package version 2.0. http://CRAN.R-project.org/package=Epi

[8] Gregory R. Warnes, Ben Bolker, Thomas Lumley, Randall C Johnson. Contributions from Randall C. Johnson are Copyright SAIC-Frederick, Inc. Funded by the Intramural Research Program, of the NIH, National Cancer Institute and Center for Cancer Research under NCI Contract NO1-CO-12400. (2015). gmodels: Various R Programming Tools for Model Fitting. R package version 2.16.2. https://CRAN.R-project.org/package=gmodels

[9] Sarkar, Deepayan (2008) Lattice: Multivariate Data Visualization with R. Springer, New York. ISBN 978-0-387-75968-5

[10] H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2009.

[11] W. N. Venables, D. M. Smith and the R Development Core Team (2016). An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics. Version 3.3.1 (2016-06-21).

[12] John Fox and Sanford Weisberg (2011). An R Companion to Applied Regression, Second Edition. Thousand Oaks CA: Sage. URL: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[13] Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel, Achim Zeileis (2008). Implementing a Class of Permutation Tests: The coin Package. Journal of Statistical Software 28(8), 1-23. http://www.jstatsoft.org/v28/i08/

Apéndice A

Ficheros de formato ASCII

Los ficheros de formato ASCII utilizados en la Sección 2.2.3 tienen el siguiente contenido:

valores.txt

```
25   167   65
21   160   57
22   178   83
29   170   69
28   163   65
19   185   90
```

valores2.txt

```
25   167   65  M
21   160   57  M
22   178   83  H
29   170   69  H
28   163   65  M
19   185   90  H
```

tabla.txt

```
Nombre    Edad   Altura  Peso  Sexo
Laura     25     167      65   M
Maria     21     160      57   M
Pedro     23     178      83   H
Josep     29     170      69   H
Martha    23     163      65   M
Jordi     19     185      90   H
```

tabla2.txt

```
Nombre    Edad   Altura  Peso  Ciudad
Laura     25     167      65   BCN
Josep     29     170      69   BCN
Jordi     19     185      90   Lleida
Adela     30     162      62   BCN
```