Introducción a R

Klaus Langohr

Departament d'Estadística i Investigació Operativa
Universitat Politècnica de Catalunya

Barcelona, Octubre 2007

Índice general

Actividad 1

Los primeros pasos

Contenido

R es, básicamente, un lenguaje que permite la manipulación de objetos estadísticos y la creación de 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 programa contestando las siguientes preguntas:

- ¿Cómo conseguir e instalar R?
- ¿Cuáles son las características de R?
- ¿Qué formas de trabajar hay en R?
- ¿Cómo importar datos de otros paquetes estadísticos?

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 el programa de análisis estadístico comercial S-PLUS (MathSoft, Inc., Seattle, EE UU). 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-PLUS, y c) estadísticos en todo el mundo están contribuyendo librerías para 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' para después bajar el fichero `R-x.y.z-win32.exe`, en donde `x`, `y` y `z` indican la versión actual de R. Al escribir el presente documento, ésta es la versión R-2.6.0.

- Ejecutad el fichero desde la carpeta en la cual fue guardado y seguid las instrucciones de instalación.

Junto con la versión básica de R, se instalan varias librerías (paquetes). No obstante, la gran mayoría de las más de mil librerías contribuidas no se habrán instalado. Para bajar los paquetes de interés, una vez abierto R, hay que hacer en 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 la(s) librería(s) deseada(s).

1.2. Generalidades de R

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

   El indicador o *prompt* del sistema es el signo >. En principio, cada instrucción acaba con un signo de punto y coma (;) y *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 un comentario. Por ejemplo, la siguiente instrucción sumará 3 y 4 e ignorará el comentario:

   ```
   3+4      #es un ejemplo
   ```

   La tecla `Esc` permite reiniciar la actual línea en edición y la combinación `Ctrl-C` interrumpe la edición o ejecución en curso. Para recuperar y volver a ejecutar instrucciones utilizadas en la misma sesión se puede utilizar la tecla de movimiento del cursor ↑.

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
m
p
(x <- log(7))
x
log
```

4. Los comandos `objects()` y `ls()` visualizan el listado de objetos presentes en el actual espacio de trabajo del directorio de trabajo. El objeto `last.warning` (si existe) recoge el último mensaje de advertencia (`Warning message`).

```
objects()
ls()
last.warning
```

Con el comando `ls(n)` se puede visualizar el contenido del enésimo elemento de la lista de búsqueda de R. Si se trata de un paquete, podemos ver el listado de todas las funciones del mismo.

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

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

6. El actual espacio o área de trabajo (*workspace*), que contiene todos los objetos de R en uso, es el primer elemento de la lista de búsqueda de R para encontrar un objeto. Por ejemplo, podemos haber creado un vector $x = (1, 3, 2)'$ y además tener abierto un *data frame* (véase Actividad 2.2) con nombre `datos1` conteniendo una variable con nombre x. Si entonces hacemos referencia a x, R va a identificar el vector $(1, 3, 2)'$.

Para visualizar la lista de búsqueda tenemos la instrucción `search()`. Por ejemplo:

```
search()
 [1]  ".GlobalEnv"        "datos1"            "package:stats"
 [4]  "package:graphics"  "package:grDevices" "package:utils"
 [7]  "package:datasets"  "package:methods"   "Autoloads"
 [10] "package:base"
```

en donde `.GlobalEnv` se refiere al área de trabajo actual y `package:xxx` a diferentes librerías cargadas.

En cambio, si eliminamos el vector x, entonces R identificará la variable x de `datos1` cuando se hace referencia a x.

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

   Ayuda ▶ Manuales (en PDF) ▶ ...

   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
   ```

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

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

   Otra posibilidad para obtener esta información es accederla 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 la librería deseada.

9. También es posible obtener ayuda sobre diferentes temas mediante la función `help.search`. R buscará ayuda sobre el tema escogido en todas las librerías instaladas. Por ejemplo, para obtener información sobre regresión logística:

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

10. La mayoría de las librerías disponibles en la versión local de R han de ser cargadas antes de que se las pueda utilizar. Por ejemplo, para cargar la librería `survival` se puede ejecutar la siguiente instrucción

    ```
    library(survival)
    ```

    o escoger `survival` desde la barra de herramientas:

    Paquetes ▶ Cargar paquete...

11. Durante una sesión de R se puede guardar el histórico de todos los comandos ejecutados 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.

12. Se puede cambiar el aspecto de la consola de R, por ejemplo su altura y/o anchura por defecto. 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.

13. Es posible abrir varias sesiones de R y trabajar simultáneamente en ellas.

14. Para salir de R se puede ejecutar la orden `q()`. En este momento, R preguntará al usuario si quiere guardar el actual espacio de trabajo. Si hemos guardado el espacio de trabajo anteriormente, no hace falta volverlo a hacer. No obstante, si contestamos con 'Sí', se guardará la actual sesión en el fichero `.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'. Los comandos no se borrarán y los resultados aparecerán en la ventana de comandos. Los *scripts* se pueden guardar y utilizar en cualquier otro momento. El postfix por defecto es '.R'.

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:/Archivos de programa/R/prueba.txt")
n<-5*2+sqrt(144)
n
sink()
n
```

A partir de la instrucción `sink()`, los resultados aparecen de nuevo en la consola de R.

4. Por defecto, el directorio de trabajo de R es '...\R\R-2.5.1' (siendo R-2.5.1 la versión de R en uso). Éste se puede cambiar en el cuadro que se abre mediante (barra de herramientas)

Archivo ▶ Cambiar dir...

5. En cualquier momento de una sesión de R se puede guardar su contenido. 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`*`")`

Como mencionado anteriormente, aunque ya se haya guardado el área de trabajo de R, al salir de R el programa siempre pregunta si se quiere guardar el área de trabajo. Evidentemente, al haberlo hecho ya, no hace falta volver a hacerlo.

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.

6. Existen diferentes editores que pueden facilitar el trabajo con R. Uno de éstos es el RWinEdt disponible para usuarios de R en el sistema de operativo Windows [1]. Éste requiere que el editor WinEdt (http://www.winedt.com) esté instalado en el ordenador. Entonces es posible instalar la librería RWinEdt y abrir el editor mediante `library(RWinEdt)`.

   En RWinEdt se pueden abrir y editar diferentes *scripts* y rápidamente enviarlos a ejecución en R. La ventaja de este editor sobre 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.

7. La instalación del *R Commander* permite la ejecución de muchas funciones, incluyendo el ajuste de diferentes modelos de regresión, desde cuadros de dialogo, lo que puede facilitar el trabajo en R. Para activarlo hay que instalar el paquete `Rcmdr` y después de cargarlo mediante `library(Rcmdr)` [2].

1.4. Importar y exportar datos

1. Si queremos importar datos otro *software* estadístico, podemos usar funciones de la librería Hmisc [3] una vez que esté instalada. Las funciones para datos procedentes de SPSS, SAS y STATA son `spss.get`, `sas.get` y `stata.get`, respectivamente. Un ejemplo para el uso de `spss.get`:

   ```
   library(Hmisc)
   newdata <- spss.get("DatosSPSS.sav",lowernames=T,datevars='bday')
   ```

   La opción `lowernames=T` hace que los nombres de las variables de `newdata` sean en minúscula mientras la opción `datevars` identifica las variables en formato de fecha del fichero original. Aunque aparezca un mensaje de advertencia como

   ```
   Warning message:
   DatosSPSS.sav: Unrecognized record type 7, subtype 16 encountered ...
   ```

   la importación del fichero suele haber funcionado sin ningún problema.

2. Para importar datos desde ficheros de formato ASCII su pueden usar las funciones `scan()` y `read.table()`. Su uso será ilustrado en la Sección 2.2.

3. 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...

4. 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 (véase Sección 2.1), con la segunda se pueden exportar *data frames* (Sección 2.2). Más información se puede encontrar en el documento de ayuda 'R Data Import/Export' [4] accesible vía la barra de herramientas:

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

Actividad 2

Vectores, matrices, listas y *data frames*

Contenido

Presentamos en esta actividad diferentes objetos básicos de R incluyendo vectores, listas y *data frames*. Además veremos cómo importar datos de un fichero ASCII a formato de matrices y *data frames* en R.

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
   (w<-c(0,2,-2,1))
   w[3]
   ```

   Se pueden cambiar elementos de vectores, ampliarlos o borrarlos siguiendo 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]
   w[6]<--4
   ```

```
w
length(w)
length(w)<-4
w
```

2. Las operaciones básicas $+, -, *, /, \hat{}$ o funciones como `log()`, `exp()`, `sqrt()`, etc. están definidas para operar vectorialmente, componente a componente. Calculad 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
v^w
sqrt(w)
log(w)
vw<-c(v,w)
vw
vw/v
vwa<-c(vw,6)
vwa/v
```

3. Otras funciones como, por ejemplo, `sum` o `prod` ejecutan un solo cálculo aplicado a todos los elementos de un vector:

```
sum(v)
prod(w)
sqrt(sum(v*w))
```

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

```
seq(1,50,2)
rep(c(1,-1,0),5)
rep(c(1,-1,0),each=5)
rep(c(1,-1,0),1:3)
-3:5
vwa[3:6]
```

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

```
x<-numeric()
y<-numeric(4)
x
y
x[1:3]<-2:4
y[x]<-3
x
y
```

6. 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
as.numeric(vwa <=0)
vwa[vwa>1]
```

7. 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 tipos diferentes:

```
z<-c('Hola','Adeu')
z
z[3]<-9
z
character(3)
```

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

```
paste("Raíz de",w,"es",sqrt(w))
nombres<-paste("Var",1:5,sep="-")
nombres
length(nombres)
paste("Now it's",date())
```

2.1.2. Creación y manipulación de matrices

1. La función `matrix(vector,...)` organiza las 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
log(A)
sum(A)
dim(A)
dim(A)[2]
nrow(A)
ncol(A)
AA<-matrix(1:12,nrow=3,byrow=T)
B<-matrix(-5:4,nc=3)
B
dim(B)
```

2. Las operaciones $+, -, *$ y / se pueden aplicar a dos o más matrices de la misma dimensión. Serán ejecutadas a componentes en la misma posición con lo cual el resultado es otra matriz de esta dimensión. En cambio, el producto de matrices se denota por `%*%` y la matriz traspuesta por `t(matriz)`.

```
A+AA
A*B                     #causa mensaje de error
C<-A%*%B
C
t(B)%*%t(A)
diag(ncol(C))
C-3*diag(ncol(C))
```

3. 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]
A[1,]
B[,1:2]
C[c(1,3),2:3]
A[2:3,]
A[A<5]
t(B[,c(1,3)])
cbind(A[2:3,],t(B[,c(1,3)]))
rbind(A[2:3,],t(B[,c(1,3)]))
```

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 una variable de tipo vectorial en la que, a diferencia de los elementos de vectores o matrices, cada componente puede ser de un tipo distinto. Por referencia mediante '[[ ]]' se puede acceder a cada componente de un objeto de tipo lista. R organiza gran parte de sus variables en forma de listas. Veamos algunos ejemplos:

```
list(c("Juan","Rosa","Miguel"),c(30,29,2))
lista1<-list(nombres=c("Juan","Rosa","Miguel"),edades=c(30,29,2))
lista1
lista1[[1]]
lista1[[1]][2]
lista1$nombres
lista1$edades[3]
```

2. Mientras `lista1[2]` es una lista (los componentes de una lista son listas), `lista1[[2]]` es un vector:

```
lista1[2]
lista1[[2]]
lista1[2]*2     # Error
lista1[[2]]*2
```

3. Otros ejemplos:

```
lista2<-list(valores=matrix(seq(100,900,100),ncol=3),estado=c(T,F),
             elementos=lista1)
lista2
lista2$valores
lista2$estado[1]
lista2$elementos$nombres[1]
lista2$elementos[2]
```

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

```
newlist<-vector("list",2)
newlist
names(newlist)<-c('New names','New ages')
newlist[[1]]<-c('Carlos','Luisa')
newlist
```

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). Son los llamados *data frames*. Las variables de un *data frame* pueden ser de tipos diferentes. 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
datfram$E
datfram$Edat[1:2]
data.frame(A)
data.frame(lista1)
data.frame(lista2)            #causa un error
```

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 puede editar también los nombres y definir el tipo de las variables (numérico o no).

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*. Para ejecutar los siguientes ejemplos, los ficheros tipo ASCII indicados, `valores.txt`, `valores2.txt` y `tabla.txt` (véase página 41) han de estar guardados en el directorio de trabajo.

```
scan("valores.txt")
datos1<-matrix(scan("valores.txt"),ncol=3,byrow=T)
datos1
is.matrix(datos1)
is.data.frame(datos1)
dim(datos1)
length(datos1)
```

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

   Notad que se puede usar la función `scan()` también para leer datos externos en una lista, incluso si contiene variables de tipo cadena:

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

2. Veamos ahora el uso de la función `read.table()` y algunos ejemplos para comprobar de qué tipo son las columnas de un *data frame*:

```
read.table("tabla.txt")
datos2<-read.table("tabla.txt",header=T)
datos2
is.matrix(datos2)
is.data.frame(datos2)
dim(datos2)
length(datos2)
dimnames(datos2)
datos2[,4]
datos2[,"Altura"]
datos2["Altura"]
datos2$Altura
is.data.frame(datos2[,"Altura"])
is.vector(datos2[,"Altura"])
is.data.frame(datos2["Altura"])
class(datos1)
class(datos2[,"Altura"])
class(datos2$Altura)
```

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
datos3
is.data.frame(datos3)
dimnames(datos3)
datos3["X3"]
```

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

```
dimnames(datos3)
dimnames(datos3)[[2]][2]<-'Peso'
dimnames(datos3)
dimnames(datos3)[[2]][c(1,3)]<-c('Edad','Altura')
datos3
```

Y también los identificadores de fila:

```
dimnames(datos3)[[1]]<-c('Laura','Maria','Pedro','Josep','Martha','Jordi')
datos3['Pedro',]
datos3['Maria',2:3]
```

3. Todos los objetos creados se pueden editar 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. Veamos algunos ejemplos a los *data frames* creados:

```
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. Las funciones `subset()` y `transform()` permiten crear un subconjunto y nuevas variables de un *data frame*, respectivamente. Por ejemplo:

```
datos23<-subset(datos2,Edad==23)
datos23
datos2.new<-transform(datos2, Altura=Altura/100,
                              BMI=round(Peso/(Altura/100)^2,2))
datos2.new
```

5. El uso de la función `order()` puede servir para ordenar las filas de un *data frame* según una o más variables:

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

6. 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 volver a quitar `df` de allí se usa la función `detach(df)`. Algunos ejemplos:

```
Altura                  # Notad que no existe dicho objeto
attach(datos2)
search()
Altura
detach(datos2)
```

7. 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<-rep(c('Sí','No'),3)
Coche                    # El vector coche no existe
detach()
attach(datos2)
Coche
```

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 `tabla.txt` en el apéndice A).

```
datos4<-read.table("tabla2.txt",header=T)
datos4
merge(datos2,datos4)
merge(datos2,datos4,all=T)
merge(datos2,datos4,all.x=T)
merge(datos2,datos4,all.y=T)
```

Actividad 3

Análisis descriptivo y gráficos

Contenido

R ofrece muchas posibilidades para realizar un análisis descriptivo y exploratorio de datos. Permite, asimismo, producir salidas gráficas muy variadas y de alta calidad. En la presente actividad presentamos las principales instrucciones y funciones necesarias para realizar estas tareas.

3.1. Análisis descriptivo y exploratorio

1. La función `summary` proporciona un resumen estadístico de un objeto. Si se trata de un vector numérico, `summary` nos devuelve el mínimo, el primer cuartíl, la mediana, la media, el tercer cuartíl y el máximo de los datos. Para ilustrar esta función, generamos un vector de 100 datos aleatorios de una distribución normal con media 5 y desviación estándar 3.

```
?rnorm
x<-rnorm(100,5,3)
summary(x)
xna<-c(x,rep(NA,5))
summary(xna)
summary(x)[3:4]
```

2. El *data frame* `state.x77` es uno de los conjuntos de datos de la librería `datasets`. Contiene información sobre los Estados Unidos como es población, superficie, nivel de ingresos o la esperanza de vida correspondiente al año 1977. Mediante la función `summary` podemos obtener un resumen de esta matriz de datos:

```
state.x77
?state.x77
is.matrix(state.x77)
is.data.frame(state.x77)
```

```
dimnames(state.x77)
summary(state.x77)
is.matrix(summary(state.x77))
summary(state.x77)[,3]
summary(datos2)
```

3. La función `mean(x)` calcula la media de un vector o matriz `x`. La función `var(x)` calcula los estimadores de la varianza de un vector `x`, de la matriz de covarianzas entre las columnas de una matriz de datos `x`, de la covarianza entre los vectores `x` e `y` (`var(x,y)`) o de la matriz de covarianzas entre las columnas de las matrices `x` e `y` (`var(x,y)`). Veamos algunos ejemplos:

```
mean(x)
mean(datos1)
mean(xna)               # NA
mean(xna,na.rm=T)
var(x)
sd(x)
var(datos1)
var(datos2)             # Mensaje de error
var(datos2[,2:4])
var(datos1[,1],datos1[,3])
var(datos1[,1:2],datos1[,2:3])
```

4. Otros indicadores de interés son la mediana, el rango intercuartílico o la correlación entre dos variables numéricas:

```
median(datos2$Peso)
min(datos2$Altura)
max(datos1)
IQR(x)
cor(datos1)
round(cor(datos2[2:4]),3)
```

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

```
quantile(x,0.3)
quantile(x,c(0.1,0.3,0.6,0.8))
quantile(x,seq(.2,.9,.1))
```

   La función tiene la opción `type` que permite calcular los cuantiles según nueve algoritmos diferentes.

```
help(quantile)
```

6. Con la función `tapply` podemos calcular estos indicadores para diferentes niveles de un factor. Por ejemplo:

```
attach(datos2)
tapply(Peso,Sexo,mean)
tapply(Peso,Sexo,summary)
tapply(Peso,Sexo,summary)[1]
detach(datos2)
fac<-gl(4,25)
fac
tapply(x,fac,quantile)
tapply(x,fac,quantile,c(0.4,0.6))
```

La función `by` permite realizar estos cálculos para varias variables numéricas:

```
by(datos2,datos2$Sexo,mean)              # Mensaje de advertencia
by(datos2[,2:4],datos2$Sexo,summary)
```

7. En caso de una variable categórica nos suele interesar la representación de los datos en forma de una tabla con las frecuencias absolutas y/o relativas. En caso de un par de variables categóricas, se usa una tabla de contingencia para representar la distribución conjunta de ambas.

```
attach(datos2)
table(Sexo)
table(Coche)
table(Sexo,Coche)
detach(datos2)
table(datos2[,5:6])
```

Otras funciones para crear tablas de contingencia son, por ejemplo, `ftable` o `xtabs`.

8. Mediante la función `ctab` del paquete `catspec` [5] es posible añadir las frecuencias relativas a estas tablas.

```
?ctab                    # no existe
library(catspec)
?ctab
attach(datos2)
ctab(Sexo)
ctab(Coche)
ctab(as.factor(Coche))
ctab(Sexo,Coche)
ctab(Sexo,Coche,type=c("n","r"),addmargins=T)
ctab(Sexo,Coche,Nombre)
detach(datos2)
```

3.2. Salidas Gráficas

3.2.1. 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. Para información detallada consultad la ayuda sobre la función `plot`.

2. Antes de generar un gráfico conviene abrir una ventana gráfica. La ventana se genera con la orden `win.graph()`. Por ejemplo:

```
attach(datos2)
win.graph()
plot(Altura,Peso)
title("Altura vs Peso")
detach(datos2)
```

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

3. El comando `par()` permite definir diferentes parámetros de la ventana gráfica, como por ejemplo, el tipo de fuente o el número de gráficos en la ventana.

```
?par
attach(datos2)
win.graph()
par(mfrow=c(1,2),font=2,font.lab=3,font.axis=4)
plot(Altura,Peso)
title("Peso vs. Altura")
plot(Edad,Peso)
title("Peso vs. Edad")
detach(datos2)
```

4. R permite realizar sofisticados análisis gráficos mediante múltiples comandos gráficos. La Tabla 3.1 muestra algunas funciones para diferentes tipos de gráficos[1].

5. Veamos algunos ejemplos más:

```
win.graph()
par(mfrow=1:2,font=2)
y<-seq(0,10,0.2)
plot(y,2*y+rnorm(length(y)))
lines(y,2*y)
```

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

Tabla 3.1: Instrucciones para diferentes tipos y elementos de gráficos

Función	Gráfico/ Elemento de gráfico
`abline`	Add line in intercept-slope form
`axis`	Add axis
`barplot, hist`	Bar graph, histogram
`box`	Add a box around plot
`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
`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
`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
`stamp`	Add date and time information to the plot
`title`	Add title, x-axis labels, y-axis labels, and/or subtitle to plot
`tsplot`	Plot a time series

```
plot(y,dchisq(y,1),xlab="",ylab="Chi-square density",type="l")
points(y,dchisq(y,2),type="l",col=2,lwd=2)
lines(y,dchisq(y,3),type="l",col=3,lwd=3)
title("Chi-square distributions")
legend(4,0.6,c("n=1","n=2","n=3"),lty=1,col=1:3,lwd=3)
```

6. El siguiente ejemplo utiliza los datos del *data frame* `state.x77`:

```
win.graph()
par(font=2,font.axis=2,font.lab=2)
population <- state.x77[,"Population"]
area <- state.x77[,"Area"]
plot(area, population, log="xy", xlab="Area in square miles",
     ylab="Population in thousands")
states.lab <- c("Alaska", "California", "Florida", "Hawaii",
     "New Jersey", "New York", "Rhode Island", "Texas", "Wyoming")
text(area[states.lab], population[states.lab],
     paste(" ", states.lab, sep=""), adj=0)
```

7. Desde la barra de herramientas los gráficos pueden ser guardados en formatos diferentes para su uso posterior. Esto les permite ser insertados en un documento WORD o incluidos en un documento de LaTeX:

   Archivo ▶ Guardar como ▶ ...

3.2.2. Expresiones gráficas de las distribuciones

1. La función `hist(x)` proporciona un histograma convencional. Entre otros, admite parámetros para fijar el número de intervalos o los puntos de corte de los intervalos.

```
z<-rnorm(1000,6,2)
hist(z)
hist(z,nclass=50,col="steelblue")
colours()
```

   A un histograma le podemos sobreponer una densidad teórica:

```
win.graph()
par(mfrow=1:2,las=1)
hist(z,nclass=35,col="plum",freq=F)
dz<-seq(min(z),max(z),0.001)
lines(dz,dnorm(dz,6,2),type="l",lwd=2)
```

2. La función `boxplot(x)` construye el *boxplot* de los datos del vector `x`. Veamos varios ejemplos:

```
boxplot(x)
boxplot(x~fac,col=2:5)
bx<-boxplot(x~fac,col=2:5,plot=F)
bx
bx$stats
boxplot(x~fac,col=2:5)
text(1:4,bx$st[3,]-0.1,paste("Med: ",round(bx$st[3,],2)),adj=c(0.5,1))
```

3. Puede resultar ilustrativo complementar los histogramas y *boxplots* con gráficos de densidad, o sea 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:

```
z<-rchisq(500,4)
density(z)
win.graph()
par(mfrow=1:2)
hist(z,nclass=20)
plot(density(z), type="l")
```

4. 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 podemos añadir al gráfico anterior la recta de regresión de la nube de puntos:

```
win.graph()
par(mfrow=1:2)
qqnorm(x)
qqline(x)
qqnorm(z)
qqline(z)
```

3.2.3. Representación de datos categóricos

1. Una manera sencilla de presentar gráficamente la distribución de una variable categórica es el diagrama de pastel (*pie chart*).

```
income<-cut(state.x77[,"Income"],breaks=seq(3000,7000,1000),dig.lab=4)
income
table(income)
pie(table(income),col=1:4,main="Average Income in 50 States")
```

2. Otro tipo de gráfico, que además permite la presentación de distribución conjunta de un par de variables categóricas, es el diagrama de barras (*bar chart*) mediante la función `barplot`.

```
barplot(table(income))
bi<-rbinom(1000,25,0.3)
barplot(table(bi),col=rainbow(25))
VADeaths                                # R data set
?VADeaths
barplot(VADeaths)
barplot(VADeaths, col=1:5,legend = rownames(VADeaths),
        ylab="Mortality per 1000")
title(main = "Death Rates in Virginia", font.main = 4)
barplot(VADeaths, col=1:5,legend = rownames(VADeaths),
        ylab="Mortality per 1000",beside=T)
title(main = "Death Rates in Virginia", font.main = 4)
```

   Además vale la pena explorar las opciones de la función `barchart` de la librería `lattice` [6].

Existen muchas librerías de R para gráficos de muy alta calidad. Para tener una idea de las posibilidades que existen con R, es recomendable explorar los paquetes presentados en la página web *R Graph Gallery*: `http://addictedtor.free.fr/graphiques/thumbs.php`

Actividad 4

Definición de funciones y programación básica en R

Contenido

Una de las ventajas de R es la posibilidad de poder definir 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 brevemente algunas de las instrucciones existentes para el control de flujos de programas.

4.1. Definición de funciones

1. R permite al usuario el personalizar algunas de las funciones incorporadas en el lenguaje así como el definir 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):

```
matrix
sqrt
```

2. Para ver cómo definir una función en R, veamos un ejemplo: como la función `var` no ofrece ninguna opción para calcular el estimador de máxima verosimilitud de la varianza, o sea $\mathrm{Var}_{ML}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$, podemos definir una función, `varML`, que lo realice:

```
fix(varML)              # Ejecutamos el editor y en él escribimos:
function(x)             # vector que se pasa a la función varML
{
  n<-length(x)
  val<-var(x)*(n-1)/n   # la variable val es temporal
  val                   # la función devuelve el contenido de val
}
```

Comprobemos que la función `varML` existe pero ningún objeto identificado como `n` o `val` y que calcula la varianza sesgada:

```
varML
val
x<-1:10
mean(x)
var(x)
varML(x)
```

3. Para definir una función de un código más largo, es recomendable programarla en una ventana *script*. En ella, en vez de ejecutar `fix(varML)`, se ha de usar el comando `function`:

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

4. Para conseguir que una función devuelva los resultados de más de un cálculo, por ejemplo ambas formas de calcular la varianza, hay que guardar los valores en una lista o utilizar las funciones `print` y/o `cat`. Para ilustrarlo podemos definir tres funciones y veremos qué resultados devuelven aplicadas al mismo vector:

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

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

vars3<- function(x){
  val1<-var(x)
  val2<-varML(x)
  list("Varianza"=val1,"Varianza sesgada"=val2)
}
```

5. Otro ejemplo: puede ser útil definir una función, llamémosla `edafigs`, que genere los siguientes cuatro gráficos:

   a) histograma
   b) boxplot
   c) gráfico de densidad
   d) Q-Q plot

   Un posible código para esta función es el siguiente:

```
edafigs<-function(x)
{
   win.graph()
   par(mfrow=c(2,2))
   hist(x)
   boxplot(x)
   iqd<-summary(x)[5]-summary(x)[2]
   plot(density(x,width=2*iqd),xlab="x", ylab="",type="l")
   qqnorm(x)
   qqline(x)
}
```

   A continuación podemos ejecutar `edafigs` con diferentes vectores:

```
x<-rnorm(100,12,3)
y<-rchisq(100,5)
edafigs
edafigs(x)
edafigs(y)
```

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:

```
log
log(100)
log(100,exp(1))
log(100,10)
```

7. Si definimos 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 función sencilla que produce un *scatterplot* de dos vectores numéricos `x` e `y`:

```
plotxy<-function(x,y,...)
{
   win.graph()
   par(...)
   plot(x,y)
}

x<-seq(1,15,0.2)
y<-2*x+rnorm(length(x),sd=1)
plotxy(x,y)
plotxy(x,y,pch=19,font.lab=2,col=4)
```

8. Las funciones pueden ser guardadas como elementos de un espacio de trabajo. No obstante, se recomienda guardar el *script* y cargarlo en cada sesión que sea necesario mediante la función `source()` o desde la barra de herramientas:

   Archivo ▶ Interpretar código fuente R...

4.2. Programación básica en R

1. Como en cualquier otro lenguaje de programación, existen instrucciones en R para controlar el flujo de un programa, por ejemplo el *script* de una simulación: `for`, `if else`, `while`, `repeat`, etc. Veremos a continuación algunos ejemplos:

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

x<-rbinom(10,1,0.5)
for(i in 1:10){
 if(x[i]==1) cat(paste("x[",i,"] es igual a 1",sep=""),"\n")
 else cat(paste("x[",i,"] es igual a 0",sep=""),"\n")
}

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

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

```
help(Control)
```

2. La función `ifelse(test,A,B)` comprueba si se cumple la condición `test` y en caso afirmativo devuelve `A`, en caso contrario `B`:

```
ifelse(log(100)<4,"exp(4)>100","exp(4)<100")

x<-rep(1:2,3)
s<-ifelse(x==1,'Hombre','Mujer')
s
```

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

help(Logic)

4.3. Generación aleatoria de datos

1. Ya vimos anteriormente que la generación aleatoria de datos se consigue con la función `r` seguida del nombre de la distribución de la cuál se desee generar. Por ejemplo, para generar 50 valores de una distribución normal de media 5 y desviación estándar 2 y otro de una distribución de Poisson con media 17 es suficiente escribir las siguientes instrucciones:

   ```
   rnorm(50,5,2)
   rpoisson(17)
   ```

2. 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 requieren la generación de datos aleatorias y queremos asegurarnos de poder reproducir los resultados obtenidos:

   ```
   rnorm(15)
   set.seed(123)
   rnorm(15)
   set.seed(123)
   rnorm(15)
   ```

3. En la Tabla 4.1[1] podemos ver los nombres de varias distribuciones en R. En la mayoría de ellas existen valores por defecto de los parámetros que se pueden conocer mediante

   `help(r`*`distribution`*`)`.

   Notad que las funciones `d`*`distribution`*, `p`*`distribution`* y `q`*`distribution`* calculan los valores de las funciones de densidad y de distribución y los cuantiles, respectivamente, según la ley *distribution*.

[1] de Venables *et al.* [7]

Tabla 4.1: 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`

Actividad 5

Pruebas estadísticas para dos poblaciones y modelos de regresión

Contenido

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

5.1. Pruebas estadísticas para dos poblaciones

5.1.1. Pruebas de independencia para dos variables categóricas

1. Dadas muestras de dos variables categóricas suele interesar si existe independencia entre éstas. 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. Veremos su aplicación a los datos de `state.x77` de la Sección 3.1 examinando si existe una relación entre la región de un estado y el salario medio.

```
state.x77
state.region
states77<-data.frame(state.x77,'Region'=state.region)
names(states77)
states77$Income.c<-cut(state.x77[,"Income"],c(3,4,5,7)*1000,
                       labels=c("<4000","4000-5000",">5000"))

attach(states77)
table(Region,Income.c)
chisq.test(Region,Income.c)
```

```
fisher.test(Region,Income.c)
fisher.test(table(Region,Income.c))
detach(states77)
```

2. Podemos guardar el resultado de cada una de estas funciones como un objeto y referirnos a cada uno de sus elementos.

```
attach(states77)
ct<-chisq.test(Region,Income.c)
ft<-fisher.test(Region,Income.c)
detach(states77)
names(ct)
is.list(ct)
names(ft)
ct[1]
ft[1]
ct$observed
ct$expected
```

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

1. Si queremos comparar la media de dos poblaciones de las disponemos de dos muestras, podemos aplicar la prueba t:

```
x<-rnorm(50,3,2)
y<-rnorm(75,4,1.5)
t.test(x,y)
t.test(x,y,alternative="less")
```

2. Generalmente, 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 la librería `datasets`.

```
sleep
?sleep
attach(sleep)
by(extra,group,summary)
dotchart(extra, groups=group,pch=19,cex=1.15,color="steelblue")
detach(sleep)
tt<-t.test(extra~group,data=sleep)
tt
names(tt)
tt$estimate
```

3. Aplicada a un vector numérico de una sola población, la función `t.test` puede contrastar la hipótesis de que la media es igual a μ (por defecto, $\mu = 0$). Además puede servir para calcular un intervalo de confianza para la media:

```
x<-rnorm(50,3,2)
t.test(x,mu=2.5)
round(t.test(x)$conf.int,3)
```

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

```
x<-rnorm(50,3,2)
y<-rnorm(50,4,1.5)
t.test(x,y,paired=T)
```

5. Por defecto, la función `t.test` supone varianzas desiguales en ambas poblaciones. Para comprobar si no podemos suponer lo contrario, podemos aplicar las funciones `var.test` y `levene.test`. Ésta última, que no supone distribución normal de los datos, está disponible en la librería `car` [8].

```
var.test(extra~group,data=sleep)
library(car)
levene.test(sleep$extra,sleep$group)
t.test(extra~group,data=sleep,var.equal=T)
```

6. Una alternativa a las pruebas paramétricas son las pruebas no-paramétricas. Por ejemplo, en caso de dos poblaciones podemos contrastar la igualdad de las medianas de ambas mediante la prueba de Wilcoxon. En R la ejecutamos mediante la función `wilcox.test`

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

   Si existen empates entre los datos, las funciones `wilcox.exact` y `wilcox_test` de las librerías `exactRankTests` [9] y `coin` [10], respectivamente, pueden calcular los valores de p de manera exacta. Por ejemplo:

```
library(exactRankTests)
wilcox.exact(extra~group,data=sleep)
wilcox.exact(extra~group,data=sleep,conf.int=T)
library(coin)
wilcox_test(extra~group,data=sleep,distribution="exact")
```

5.2. 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. A continuación volvemos a utilizar el *data frame* states77 creado anteriormente (en la Sección 5.1.1) queriendo ajustar un modelo lineal para la variable esperanza de vida.

```
names(states77)
round(cor(state.x77),2)
lm(Life.Exp~Illiteracy+Murder+Income.c,data=states77)
lmod<-lm(Life.Exp~Illiteracy+Murder+Income.c,data=states77)
lmod
```

2. Si aplicamos la función `summary` a un objeto `lm`, obtenemos un *output* más extenso e informativo:

```
class(lmod)
?summary.lm
summary(lmod)
sumod<-summary(lmod)
names(lmod)
names(sumod)
lmod$coef
round(sumod$coef,3)
```

3. En combinación con la función `abline`, la función `lm` permite sobreponer una recta de regresión a un gráfico de dispersión:

```
attach(states77)
win.graph()
par(pch=19,cex=1.25,font.lab=2,font.axis=2,las=1)
plot(Murder,Life.Exp)
abline(lm(Life.Exp~Murder),lwd=3,col=2)
detach(states77)
```

4. Con la función `by` (Sección 3.1) podemos ajustar un modelo de regresión para cada uno de los niveles de un factor. Por ejemplo:

```
by(states77,Region,function(x) lm(Life.Exp~Murder,data=x))
bystat<-by(states77,Region,function(x) summary(lm(Life.Exp~Murder,data=x)))
is.list(bystat)
bystat$West
names(bystat$West)
bystat$West$r.squared
```

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

```
attach(states77)
Income.c
Income.c<-factor(Income.c,levels=c(">5000","4000-5000","<4000"))
Income.c
summary(lm(Life.Exp~Illiteracy+Murder+Income.c))
detach(states77)
```

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

```
?plot.lm
plot(lmod)
win.graph()
par(mfrow=c(2,2),font.lab=2,las=1,font.axis=2)
plot(lmod)

residuals(lmod)
qqnorm(residuals(lmod))
qqline(residuals(lmod))
```

7. Existe la posibilidad de escoger las variables de un modelo de regresión según el procedimiento de la *stepwise forward/backward selection*. La función a utilizar es `step`:

```
step(lmod)
slm<-step(lm(Life.Exp~Illiteracy+Murder+HS.Grad+Frost+Area+Region+Income.c,
             data=states77))
summary(slm)
```

Bibliografía

[1] Ligges, U. (2003). R-WinEdt. In: Hornik, K., Leisch, F. and Zeileis, A. (eds.), Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, TU Wien, Vienna, Austria. ISSN 1609-395X, http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

[2] Fox, J., with contributions from M. Ash, T. Boye, S. Calza, A. Chang, P. Grosjean, R. Heiberger, G. Jay Kerns, R. Lancelot, M. Lesnoff, S. Messad, M. Maechler, E. Neuwirth, D. Putler, M. Ristic and Peter Wolf (2007). Rcmdr: R Commander. R package version 1.3-5. http://www.r-project.org, http://socserv.socsci.mcmaster.ca/jfox/Misc/Rcmdr/.

[3] Frank E Harrell Jr and with contributions from many other users (2007). Hmisc: Harrell Miscellaneous. R package version 3.4-2. http://biostat.mc.vanderbilt.edu/s/Hmisc, http://biostat.mc.vanderbilt.edu/twiki/pub/Main/RS/sintro.pdf, http://biostat.mc.vanderbilt.edu/twiki/pub/Main/StatReport/summary.pdf, http://biostat.mc.vanderbilt.edu/trac/Hmisc

[4] R Development Core Team (2007). R Data Import/Export. Version 2.5.1 (2007-06-27).

[5] John Hendrickx. catspec: Special models for categorical variables. R package version 0.92. http://www.xs4all.nl/ jhckx

[6] Deepayan Sarkar (2007). lattice: Lattice Graphics. R package version 0.16-5.

[7] W. N. Venables, D. M. Smith and the R Development Core Team (2007). An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics. Version 2.5.1 (2007-06-27).

[8] John Fox. I am grateful to Douglas Bates, David Firth, Michael Friendly, Gregor Gorjanc, Georges Monette, Henric Nilsson, Brian Ripley, Sanford Weisberg, and Achim Zeleis for various suggestions and contributions. (2005). car: Companion to Applied Regression. R package version 1.0-17. http://www.r-project.org, http://socserv.socsci.mcmaster.ca/jfox/

[9] Torsten Hothorn and Kurt Hornik (2006). exactRankTests: Exact Distributions for Rank and Permutation Tests. R package version 0.8-15.

[10] Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel and Achim Zeileis (2006). A Lego System for Conditional Inference. The American Statistician 60(3), 257-263.

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
```