FEIR 10: Entorno de trabajo R. RStudio

Apuntes del curso FEIR3, curso 2014/15 actualizados. `Última actualización:` martes 02 abril 2019, 17:02:33

Laura del Río Alonso y Antonio Maurandi López

Índice

1. ¿Qué es R?

R es un potente lenguaje orientado a objetos y destinado al análisis estadístico y la representación de datos. Se trata de software libre que permite su utilización libre y gratuitamente. La comunidad científica internacional lo ha elegido como la *lingua franca* del análisis de datos. Y tiene una gran implantación en universidades y cada vez más en mundo empresarial.

Otra definición: "*R es un paquete estadístico de última generación al mismo tiempo que un lenguaje de programación*".

1.1. ¿Por qué emplear R?

Estadística se puede hacer con miles de paquetes estadísticos, incluso con hojas de cálculo e incluso los más osados con lápiz y papel (aunque la exactitud de las máquinas es una potencia que no podemos desdeñar: cálculo de p-valores, estadísticos exactos...).

¿Qué tiene R que tanto nos gusta?:

- Es libre. Se distribuye bajo licencia GNU, lo cual significa que lo puedes utilizar y ¡mejorar!
- Es multiplataforma, hay versiones para Linux, Windows, Mac, iPhone... ¡web!
- Se puede analizar en R cualquier tipo de datos.
- Es potente. Es muy potente.
- Su capacidad gráfica difícilmente es superada por ningún otro paquete estadístico.
- Es compatible con 'todos' los formatos de datos (`.csv`, `.xls`, `.sav`, `.sas`...)
- Es ampliable, si quieres añadir algo: ¡empaquétalo!
- Hay miles de técnicas estadísticas implementadas, cada día hay más.
- ...

1.2. Instalación de R

La instalación difiere para cada sistema operativo; en Windows es un ejecutable, en GNU/Linux se hace habitualmente ejecutando algunos comandos desde la consola...

La última versión, cuando se elabora este documento, es la *R 3.3.2* que está disponible para su descarga desde la página web de CRAN: http://cran.es.r-project.org/

1.2.1. En Windows

Se hace desde un ejecutable que bajamos desde "Download R 3.3.2 for Windows (62 megabytes, 32/64 bit)" y lo instalamos ejecutándolo; un vídeo de la instalación puede verse aquí.

1.2.2. En GNU/Linux

Dependiendo de la distribución el procedimiento puede variar, a modo de ejemplo indicamos el procedimiento para la instalación en una ditribución Ubuntu:

```
sudo apt-get update
sudo apt-get install r-base
```

Para más detalles sobre el procedimiento ver CRAN o esté sitio web: www.digitalocean.com.

1.2.3. En Mac OX

Tanto la información para la intalación como el fichero necesarios están disponibles aquí: http://cran.es.r-project.org/bin/macosx/

1.2.4. Comprobando la instalación

Lanzar R y salir:

1. En GNU/Linux: `R`
2. en Windows: doble clic en el icono "R"
3. Para salir de la aplicación ejecutamos en ambos S.O.:

   ```
   q()  # para salir
   ```

1.3. Trabajando con R

1.3.1. Ayuda en R

¿Cómo obtenemos ayuda en R? Veamos la siguiente tabla:

Cuadro 1: Funciones de ayuda en R

Función	Acción
`help.start()`	Ayuda general
`help(mean)` o `?mean`	Función de ayuda
`help.search("mean")`	Ayuda online
`RSiteSearch("mean")`	
`apropos("mean", mode ="function")`	Lista todas las funciones que contienen `mean` en el nombre
`data()`	Muestra los conjuntos de datos de ejemplo que hay disponibles

Otra fuente de ayuda son las listas de correo, `R-help-es`: https://stat.ethz.ch/mailman/listinfo/r-help-es. Es una lista de distribución donde uno puede preguntar y responder dudas que surgen al trabajar con R. Esta lista surgió muy cerca de esta universidad y gracias a ella se han realizado ya 6 jornadas de usuarios de R

en español; la primera en Murcia, la segunda en Oviedo y las sucesivas en Madrid, Barcelona, Zaragoza y Santiago de Compostela (visita:www.r-es.org).

1.3.2. Espacio y directorio de trabajo

El *"workspace"* es el espacio de trabajo en que se incluyen todos los objetos definidos por el usuario (ya veremos qué son estos objetos, que incluyen variables, vectores, *dataframes*...), se almacena en memoria intermedia mientras trabajas con R.

Cuando termina una sesión de R el propio R te pregunta si quieres guardar el *"workspace"* para usos futuros. Este espacio, *"workspace"*, se recarga al volver a iniciar la sesión. Directorio de trabajo o *"working directory"* es el directorio donde por defecto "lee" R. También es donde guardará el *workspace* al finalizar la sesión y donde buscará un workspace guardado al inicio. Si quieres que R lea un fichero que no esté en *"working directory"* hay que especificar la ruta completa.

Funciones para manejar el *"workspace"*:

Cuadro 2: Funciones para el manejo del *workspace*

Función	Acción
`getwd()`	Muestra el wd: *working directory*
`setwd("midirectorio")`	Ajusta el wd al especificado
`ls()` o `dir()`	Lista lo que hay en el `wd`
`history()`	Muestra los últimos comandos ejecutados
`savehistory()`	Guarda el historial de comandos, por defecto en `.RHistory`
`loadhistory()`	Carga el historial de comandos
`save.image("mywspace.R")`	Guarda los objetos del *workspace*, por defecto en `.RData`
`load("mywspace.R")`	Carga el *workspace* `mywspace.R`

1.3.3. Cosas varias: asignaciones, variables, comentarios, *inputs*...

Todo lo precedido por almohadillas `#` R lo considera un comentario y no lo *interpreta*. En R empleamos el operador `<-` para hacer asignaciones, y una variable se crea *"al vuelo"*, esto es, en el mismo instante en el que la asignas. Es más, **no puedes declararlas con anterioridad y dejarlas vacías**.

Así:

```
mivariable <- 7  # asigna el número 7 a una variable
```

Si quiero ver qué valor toma mi variable escribo:

```
mivariable
```

```
[1] 7
```

Nota: Cualquier asignación a una variable crea o reutiliza un "objeto" de R.

Si guardo el *"workspace"* , ahora, quedará guardada mi nueva variable en él.

1.3.4. Una sesión "tipo" en R

Veamos expresiones que habitualmente se usan una sesión de R.

```
> getwd()
```

```
[1] "C:/Documents and Settings/Administrador/Mis documentos"
```

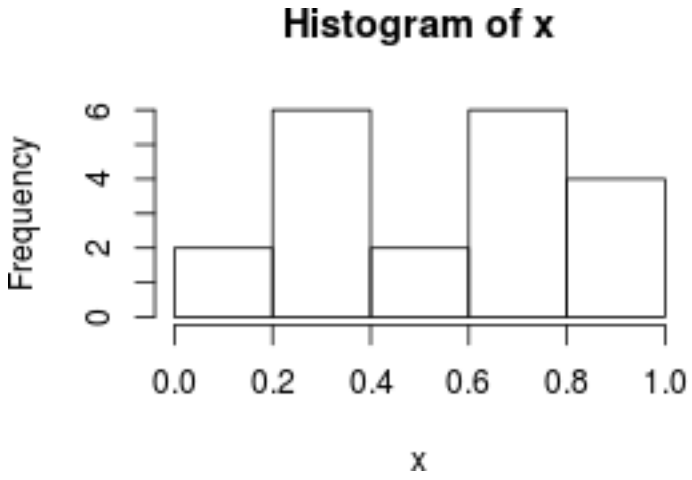


Figura 1: Mi madre que chulo!

```
> setwd( "C:/CursoR_UMU" ) # ¡¡Ojo con la barra!!
```

```
set.seed( 1 )
x <- runif( 20 )
x
```

```
 [1] 0.2655 0.3721 0.5729 0.9082 0.2017 0.8984 0.9447 0.6608 0.6291
[10] 0.0618 0.2060 0.1766 0.6870 0.3841 0.7698 0.4977 0.7176 0.9919
[19] 0.3800 0.7774
```

```
summary( x )
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.062   0.345   0.601   0.555   0.772   0.992
```

```
hist( x )
```

Ahora cerramos la sesión guardando, explícitamente, el historial de expresiones y el *work space* actual (*objetos*).

```
> savehistory()
> save.image()
> q()
```

1.3.5. R como calculadora

Podemos emplear R como una calculadora.

```
45 + 23
```

```
[1] 68
```

```
100 / 4
```

```
[1] 25
```

```
sqrt( 25 )
```

```
[1] 5
```

```
# usando variables
x <- 100 / 4
5
```

```
[1] 5
```

hay definidas algunas constantes...

```
pi
```

```
[1] 3.14
```

```
r    <- 5
area <- 2 * pi + r
area
```

```
[1] 11.3
```

```
c( 25, 100, 2, 3 ) * 5
```

```
[1] 125 500  10  15
```

> **Nota**: Cambia el número de decimales por defecto con `options( digits = 12 )` y para saber en que está establecido usa `getOption( "digits" )`.

```
old <- getOption( "digits" ) # guardamos la opcion actual
getOption( "digits" )
```

```
[1] 3
```

```
options( digits = 12 )  # cambiamos el número de digitos
getOption( "digits" )
```

```
[1] 12
```

```
options( digits = old ) # lo devolvemos al estado inicial
getOption( "digits" )
```

```
[1] 3
```

1.3.6. En R trabajamos con *scripts*

Para cargar un fichero con instrucciones, un *script*, empleamos la función `source()`.

```
source( "fichero_de_comandos.R" )
```

Un fichero que crease un histograma de una curva normal debería de contener, por ejemplo, estas instrucciones:

```
x <- rnorm( 100 )  # crea 100 observaciones aleatorias
                   # de una normal tipificada
hist( x )
```

Así que creamos un fichero de texto plano con el contenido anterior, lo guardamos con el nombre "script.R" (en realidad da lo mismo la extensión, es una convención). Ejecutamos:

```
source( "script.R" )
```

---

* *Ejercicio**: Ejecutar mediante el comando `source()` el fichero `10A-ejSource.R`

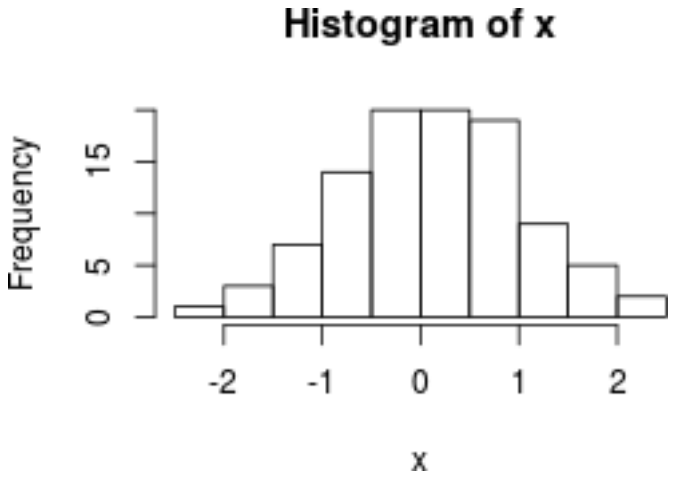


Figura 2: hist script source

1.3.7. Ampliar la funcionalidad de R. *Packages*

Con la instalación simple de R tenemos muchísimas posibilidades, no obstante existen multitud de módulos opcionales que llamamos paquetes, *packages*. Los paquetes son colecciones de funciones y datos.

El directorio de tu PC donde se almacenan los *packages* es denominado `library`.

```
.libPaths()
[1] "C:/Archivos de programa/R/R-2.14.1/library"
```

o

```
.libPaths()
[1] "/home/amaurandi/R/i686-pc-linux-gnu-library/3.2"
[2] "/usr/local/lib/R/site-library"
[3] "/usr/lib/R/site-library"
[4] "/usr/lib/R/library"
```

Para ver qué paquetes tienes instalados empleamos la función `library()`. *No es lo mismo instalar que cargar un paquete*. La instrucción `search()` nos dice que paquetes están instalados y cargados en el sistema listos para usarse.

Instalamos un paquete con la instrucción `install.packages("nombre_paquete")`, sólo instalamos una vez cada paquete, cargamos el paquete con la instrucción `library("nombre_paquete")` y tendremos que ejecutar este comando en cada sesión en que queramos emplear dicho paquete.

```
install.packages("foreign")
```

```
--- Please select a CRAN mirror for use in this session ---
HTTPS CRAN mirror
Selection: 39
probando la URL 'https://ftp.cixug.es/CRAN/src/contrib/foreign_0.8-67.tar.gz'
Content type 'application/x-gzip' length 334175 bytes (326 KB)
==================================================
downloaded 326 KB
....
....
installing to /home/amaurandi/R/x86_64-pc-linux-gnu-library/3.3/foreign/libs
...
* DONE (foreign)
```

CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods

Figura 3: Task Views

```
The downloaded source packages are in
    '/tmp/Rtmp3wi1AY/downloaded_packages'
```

Los paquetes disponibles están en CRAN, específicamente en http://cran.r-project.org/.

Normalmente son necesarios más paquetes que los que vienen por defecto, así que siempre se está instalando paquetes nuevos. A lo largo de ete documento veremos algunos de los más usuales.

Para obtener información sobre un paquete empleamos la función `help( package = "nombre_del_paquete" )`.

> **Nota**: ¡En el momento de escribir este manual existen 10021 paquetes! Seguro que si lo comprobamos ahor amismo habra más!

> **Ejercicio**: Visitar la web de CRAN http://cran.r-project.org/ y consultar los últimos paquetes añdidos/modificados.

> **Nota**. Los *Task View* son colecciones de paquetes para trabajos concretos o para áreas de trabajo concretas. Instalándose una *Task View* automáticamente conjuntos enteros de paquetes se instalan sin tener que ir instalando paquete a paquete.

```
> To automatically install these views, the ctv package needs to be
> installed, e.g., via
> install.packages("ctv")
> library("ctv")
> and then the views can be installed
```

```
> via `install.views` or `update.views` (which first assesses which of
> the packages are already installed and up-to-\>date), e.g.,
> install.views("Econometrics")
> or
> update.views("Econometrics")
```

1.3.8. Otra *"sesión tipo"* de R

Probemos a comunicarnos con R y ver su versatilidad e interactividad.

```
peso <- c( 4.4, 5.4, 6.4, 3.2, 7.5, 3, 6.1, 3.1, 6.1, 7, 3.4 )
# c() concatena y crea un vector
edad <- c( 1, 2, 3, 4, 5, 6, 7, 8, 9, 1 )
edad
```

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

```
peso
```

```
 [1] 4.4 5.4 6.4 3.2 7.5 3.0 6.1 3.1 6.1 7.0 3.4
```

```
mean( peso )  # función media
```

```
[1] 5.05
```

```
mean( edad )
```

```
[1] 4.6
```

```
# cor( peso, edad )  # coeficiente de correlación de Pearson

length( edad )
```

```
[1] 10
```

```
length( peso )  # función longitud/dimensión
```

```
[1] 11
```

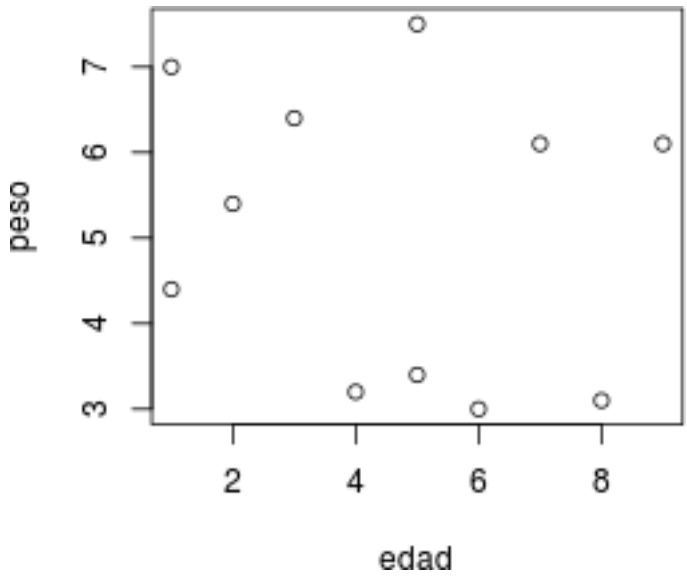
```
edad <- c( edad, 5)  # Nota le añadimos a edad una observación más
length( edad )
```

```
[1] 11
```

```
cor( peso, edad )
```

```
[1] -0.188
```

```
plot( edad, peso )
```




```
q()
```

1.4. RStudio

1.4.1. Una GUI para R multiplataforma

Una de las cosas que menos gustan de R es que *se ve diferente en Windows que en Linux*. Algo que suele molestar a los usuarios mas experimentados en R es utilizar programas en que los menús te limiten (algo parecido le pasa a los usuarios de GNU/Linux). En general, el usuario de R no quiere que los menús de la aplicación le sugieran qué hacer con sus datos, más bien simplemente quiere *"decirle"* al software qué hacer con los datos, es él el que manda. Esto se hace mediante la combinación de expresiones y gracias un poco de formación, experiencia y tesón.

Una programa que soluciona en gran medida estos problemas es **RStudio** http://rstudio.org/.

RStudio es una GUI, *Graphical user interface*, para R programada en `C#`, multiplataforma (Windows, Linux y Mac). Que aúna todos los entornos y asume la filosofía de los expresiones, pero aportando algunas 'ayudas' que hacen más llevadero el día a día.

Otra definición de RStudio: es un entorno libre y de código abierto para el desarrollo integrado (IDE) de R. Se puede ejecutar en el escritorio (Windows, Mac o Linux) o incluso a través de Internet mediante el servidor RStudio.

Entre otras cosas encontramos que RStudio:

- Nos permite abrir varios *scripts* a la vez
- Nos permite ejecutar trozos de código con sólo marcarlo en los *scripts*
- Nos muestra el *workspace*
- Nos muestra el historial
- Nos muestra los objetos del *workspace*
- Integra la ayuda
- Integra la gestión de librerías
- Permite intercalar un lenguaje de marcas, `markdown`, con código de R, con el paquete Rmarkdown y knitr.
- etc.

Actualmente RStudio está en la versión `v1.0.136` y ofrece una gran integración con ficheros en diversos formatos: R scripts (.R), markdown (md), Rmarkdown (.Rmd), LaTeX(.txt), .. que posibilitan desarrollar

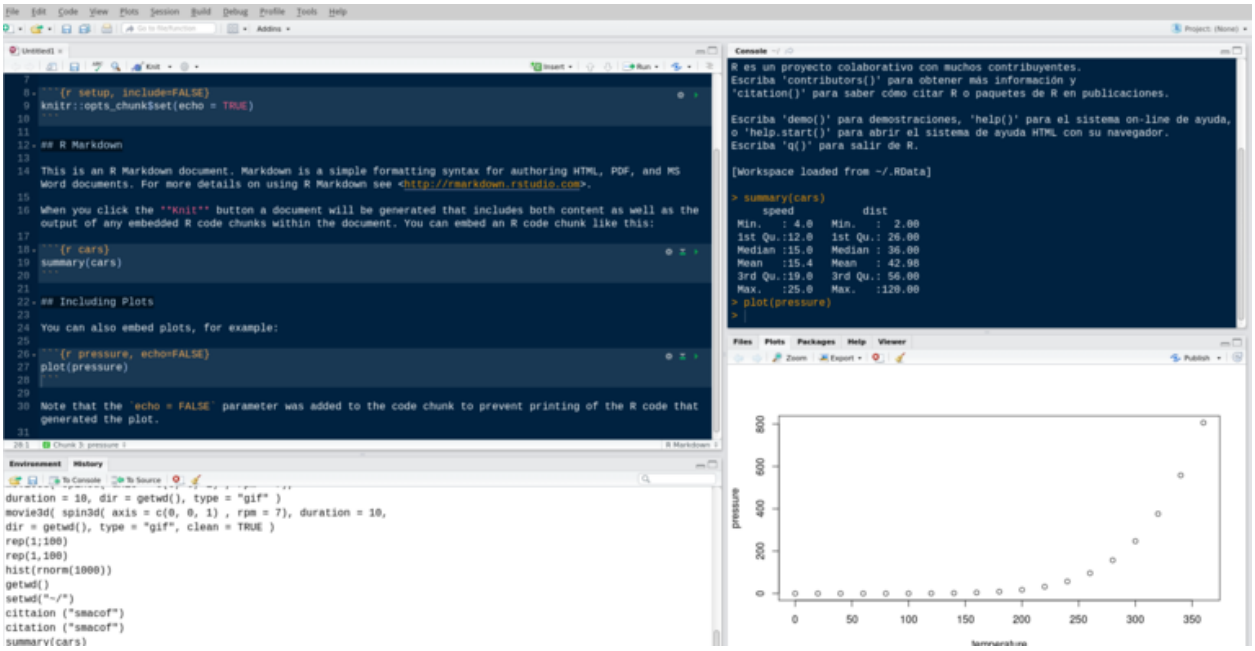


Figura 4: RStudio

cómodamente investigación reproducirle gracias al paquete `knitr` de Yihui Xie (Xie, 2013). Puede ampliar información sobre este tema en el libro de Christopher Gandrud (Gandrud, 2013). La gran gran facilidad para generar documentos dinámicos con RStudio y `knitr` a hecho que RStudio se convierta en la GUI por excelencia para R. Este documento está editado con en RStudio y procesado con los paquetes `knitr` y `rmarkdown`.

2. Estructuras de datos en R

Trabajaremos con ejemplos y fijándonos en los diferentes tipos de datos y de estructuras que R tiene para albergarlos.

Un *data set*, es (normalmente) un "conjunto de datos" que está ordenado por filas y columnas, y donde cada fila representa una observación y cada columna una variable. Dependiendo de las áreas de conocimiento recibe distinto nombre: tabla de datos, matriz de datos,...

Importamos un fichero de datos desde un *csv* (no nos preocupemos, de momento, en los aspectos relacionados con el cómo importar datos).

```
read.table( "files/ejemplo01.csv",
            sep = ';',
            head = T )
```

```
  paciente   admision edad diabetes status
1        1 11/11/2010   23   tipo 1  bueno
2        2 11/01/2009   56   tipo 1  bueno
3        3 15/03/2012   78   tipo 2   malo
4        4 28/04/2008   34   tipo 1  bueno
5        5 09/08/2011   23   tipo 2 mejora
6        6 24/04/2012   12   tipo 1  bueno
7        7 08/08/2008   56   tipo 2  bueno
```

>

Hay diversos tipos de variables, una forma de clasificarlas podría ser esta: *datos nominales*, *ordinales* y *numéricos*.

En el ejemplo anterior:

- Son variables numéricas: `paciente` y `edad`
- Son variables ordinales: `status`
- Son variables nominales: `diabetes`

Nota. Para almacenar los datos R cuenta, como todos los lenguajes de programación, de una gran variedad de estructuras de datos. Estas van de lo más sencillo a las estructuras más complejas.

2.1. Tipos de objetos

2.1.1. Vectores

Son matrices de una dimensión que solamente pueden contener valores homogéneos, ya sean numéricos, alfanuméricos o valores lógicos. Emplearemos la función `c()` para construir vectores (función *"combine"*)

```
x <- c( 1, 2, 3, 4, 5, 6, 7, 8 )
y <- c( "juan", "pepe", "iñaky", "amparito"
        ,"mariano", "juancar", "fulano", "elefante" )
z <- c( TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE, TRUE )
```

Podemos acceder a los elementos un vector usando los corchetes, que indican subíndice, así, x_3, x_6, ...

```
y[ 3 ]
```

```
[1] "iñaky"
```

```
y[ 6 ]
```

```
[1] "juancar"
```

```
y[ 8 ]
```

```
[1] "elefante"
```

```
x[ 1 ]
```

```
[1] 1
```

Es importante ver que un subíndice puede ser una expresión y por lo tanto un conjunto de valores:

```
y[ c( 6, 8 ) ]
```

```
[1] "juancar"  "elefante"
```

Nota: En R los *índices (numeraciones)* comienzan en el 1, no en el 0 como ocurre en muchos lenguajes de programación.

Extra: El operador "`:`" genera secuencias.

```
c( 2:6 )
```

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

```
c( 1:3 )
```

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

Nota: Podemos crear un vector de un cierto tipo y dejarlo vacío para ir *llenándolo* después en un bucle por ejemplo. Se puede hacer con las instrucciones `v<-vector('numeric')` o `v<-vector('character')`.

2.1.2. Matrices

Una matriz es un vector con un atributo adicional (`dim`), que a su vez es un vector numérico de longitud 2 que define el número de filas y columnas. Se crean con la función `matrix()`.

```
matrix( data = NA, nrow = 2, ncol = 2, byrow = F, dimnames = NULL )
```

```
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
```

```
matrix( data = 1:4, nrow = 2, ncol = 2, byrow = F, dimnames = NULL )
```

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

```
matrix( data = 5, nrow = 2, ncol = 2, byrow = F, dimnames = NULL )
```

```
     [,1] [,2]
[1,]    5    5
[2,]    5    5
```

```
matrix( data = 1:6, nrow = 2, ncol = 2, byrow = F, dimnames = NULL )
```

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

```
matrix( data = 1:6, nrow = 2, ncol = 3, byrow = F, dimnames = NULL )
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
x <- matrix( data = 1:6, nrow = 2, ncol = 3, byrow = F, dimnames = NULL )
dim( x )
```

```
[1] 2 3
```

Las matrices son muy versátiles, podemos convertir vectores en matrices.

```
x <- 1:15
x
```

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

```
dim( x )
```

```
NULL
```

```
dim( x ) <- c( 5, 3 )
x
```

```
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
```

```
[5,]    5   10   15
```

```
x <- matrix( 1:10, nrow = 2 ) #¡no tenemos que especificar las columnas!
x
```

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

Los **arrays** son otro tipo similar a las matrices pero pueden tener más de dos dimensiones. Nos nos detenemos en ellos.

2.1.3. Dataframes

Un *dataframe* (a veces se traduce como *'marco de datos'*) es una generalización de las matrices donde cada columna puede contener tipos de datos distintos al resto de columnas, manteniendo la misma longitud. Es lo que más se parece a una tabla de datos de SPSS o SAS, o de cualquier paquete estadístico estándar. Se crean con la función `data.frame()`.

```
nombre <- c( "juan", "pp", "iñaky", "amparo",
             "mariano", "juancar", "fulano", "elefante" )
edad <- c( 23, 24, 45, 67, 32, 56, 78, 45 )
peso <- c( 34, 34, 56, 78, 34, 56, 76, 87 )
length( nombre )
```

```
[1] 8
```

```
length( edad )
```

```
[1] 8
```

```
length( peso )
```

```
[1] 8
```

```
caseid <- c( 1:length( peso ) )
df     <- data.frame( caseid, nombre, edad, peso )
df
```

```
  caseid   nombre edad peso
1      1     juan   23   34
2      2       pp   24   34
3      3    iñaky   45   56
4      4   amparo   67   78
5      5  mariano   32   34
6      6  juancar   56   56
7      7   fulano   78   76
8      8 elefante   45   87
```

Seleccionar columnas concretas de un *dataframe* con corchetes `[ ]`.

```
df[ 1:2 ]
```

```
  caseid   nombre
1      1     juan
2      2       pp
3      3    iñaky
4      4   amparo
5      5  mariano
6      6  juancar
```

```
7      7   fulano
8      8 elefante
```

```
df[ c( 1, 3 ) ]
```

```
  caseid edad
1      1   23
2      2   24
3      3   45
4      4   67
5      5   32
6      6   56
7      7   78
8      8   45
```

Prueba con `df["nombre"]`

Para acceder al vector que forma una de las columnas de un *dataframe* también usamos el operador `$`.

```
df$nombre
```

```
[1] juan     pp       iñaky    amparo   mariano  juancar  fulano
[8] elefante
Levels: amparo elefante fulano iñaky juan juancar mariano pp
```

```
df$peso
```

```
[1] 34 34 56 78 34 56 76 87
```

Ampliar un dataframe:

```
diabetes <- c( "Tipo1", "Tipo1", "Tipo2", "Tipo2",
               "Tipo1", "Tipo1", "Tipo2",  "Tipo1" )
estado   <- c( "bueno", "malo", "bueno", "bueno",
               "bueno", "malo", "bueno", "malo" )
length( diabetes )
```

```
[1] 8
```

```
length( estado )
```

```
[1] 8
```

```
df <- data.frame( df, diabetes, estado )
df
```

```
  caseid   nombre edad peso diabetes estado
1      1     juan   23   34    Tipo1  bueno
2      2       pp   24   34    Tipo1   malo
3      3    iñaky   45   56    Tipo2  bueno
4      4   amparo   67   78    Tipo2  bueno
5      5  mariano   32   34    Tipo1  bueno
6      6  juancar   56   56    Tipo1   malo
7      7   fulano   78   76    Tipo2  bueno
8      8 elefante   45   87    Tipo1   malo
```

Para saber el número de filas de un dataframe podemos usar la función `nrow()`, `ncol()` nos devuelve el número de columnas.

```
nrow( df )
```

```
[1] 8
```

```
ncol( df )
```

```
[1] 6
```

Extra: Crear un tabla de frecuencias. Explorar la función `cbind()` para *ampliar* dataframes.

```
table( df$diabetes, df$estado )
```

```
        bueno malo
  Tipo1     2    3
  Tipo2     3    0
```

Funciones `attach()`, `detach()` dataframes para acceder más *fácilmente*; *si bien no es aconsejable su uso.*

```
df$peso
```

```
[1] 34 34 56 78 34 56 76 87
```

```
attach( df )
```

```
The following objects are masked _by_ .GlobalEnv:

    caseid, diabetes, edad, estado, nombre, peso
```

```
peso
```

```
[1] 34 34 56 78 34 56 76 87
```

```
detach( df )
```

Nota: si ya existe un objeto con ese nombre puede llevarnos a confusión. Una alternativa al attach es el `with()`.

```
with( df, { +peso } )
```

```
[1] 34 34 56 78 34 56 76 87
```

En nuestro ejemplo estamos empleando la variable `caseid` para identificar las observaciones de forma unívoca. Podemos emplear la opción `rownames` en la función `data.frame()` con tal de usar una variable para cuestiones como etiquetar casos, etc.

```
df <- data.frame( caseid,
                  nombre,
                  edad,
                  peso,
                  diabetes,
                  estado,
                  row.names = caseid )
```

Nota: Por motivos de legibilidad y mantenibilidad del código no es recomendable emplear las funciones `attach()`, `detach()` y `with()`.

2.2. Factores

Las variables pueden ser clasificadas como *nominales, ordinales* o *numéricas*. Las variables nominales son variables categóricas sin un orden definido, como *diabetes* en nuestro ejemplo anterior.

```
df$diabetes
```

```
[1] Tipo1 Tipo1 Tipo2 Tipo2 Tipo1 Tipo1 Tipo2 Tipo1
Levels: Tipo1 Tipo2
```

Las variables ordinales, por ejemplo `estatus`, implican orden pero no cantidad.

```
df$estado
```

```
[1] bueno malo  bueno bueno bueno malo  bueno malo
Levels: bueno malo
```

En R prestamos un interés especial a estas variables nominales y ordinales y las llamamos *factores*. Los factores son cruciales porque van a determinar cómo se analizarán los datos y cómo se mostrarán en gráficos. Para convertir un vector en un factor empleamos la función `factor()`.

```
estado   <- c( "bueno", "malo", "bueno", "bueno",
               "bueno", "malo", "bueno", "malo" )
diabetes <- c( "Tipo1", "Tipo1", "Tipo2", "Tipo2",
               "Tipo1", "Tipo1", "Tipo2",  "Tipo1" )
str( diabetes )
```

```
 chr [1:8] "Tipo1" "Tipo1" "Tipo2" "Tipo2" "Tipo1" "Tipo1" ...
```

```
str( estado )
```

```
 chr [1:8] "bueno" "malo" "bueno" "bueno" "bueno" "malo" "bueno" ...
```

```
diabetes <- factor( diabetes )
str( diabetes )
```

```
 Factor w/ 2 levels "Tipo1","Tipo2": 1 1 2 2 1 1 2 1
```

```
estado <- factor( estado )
str( estado )
```

```
 Factor w/ 2 levels "bueno","malo": 1 2 1 1 1 2 1 2
```

> **Nota**: La función `str()` nos muestra la estructura de un objeto. Observa que antes de hacer a diabetes un factor era un vector de `chars`.

Un repaso algo más extenso, sin ser muy largo, a las estructuras de datos y operadores se puede hacer siguiendo el *Curso básico de R* de F. Carmona(Carmona, 2007).

3. Entrada de datos

R es 'compatible' con prácticamente cualquier formato de datos.

3.1. Desde el teclado

Si tenemos un *dataframe* y queremos editarlo *"a mano"* usamos la función `edit()`.

```
df <- data.frame( edad = numeric(   0 ),
                  sexo = character( 0 ),
```

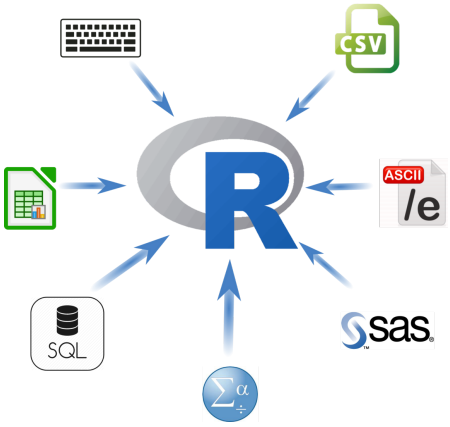


Figura 5: entrada de datos

```
                  peso = numeric(   0 ) )
df
```

```
[1] edad sexo peso
<0 rows> (or 0-length row.names)
```

Nota: lo hemos creado *vacío*. Ahora lo editaremos con `edit()`.

```
df <- edit( df )
```

Eliminar, borrar un objeto se puede hacer con `rm()`

```
rm( df )
```

3.2. Importar datos desde ficheros de texto

La función más importante para importar datos, al menos una de las más usadas es `read.table()`, automáticamente te convierte los datos en un *dataframe*.

```
df <- read.table( file,
                  header = logical_value,
                  sep = "delimiter",
                  row.names = "name" )
```

- `header` será TRUE o FALSE según la primera fila del fichero ASCII represente el nombre de las variables.
- `sep` es el delimitador, es decir el separador de campos. Por defecto es el espacio `" "`pero se puede especificar lo que sea:
  `sep="."`, `sep=","`, `sep=` (tabulación)...
- `row.names` es un parámetro opcional para especificar una o varias variables de identificador de casos.

```
df <- read.table( "files/10A-ejemplo01.csv",
                  sep = ";",
                  head = T )
```

Por defecto las variables de tipo carácter se convierten a factores, si queremos que no lo haga habrá que indicarle `stringsAsFactor = FALSE` como opción.

Hay muchas más opciones, mira la ayuda `help( read.table )`. Algunas muy importantes tratan el tema de los valores faltantes (`na.strings` = "NA") , o de la separación de decimales para números (`dec` = ".").

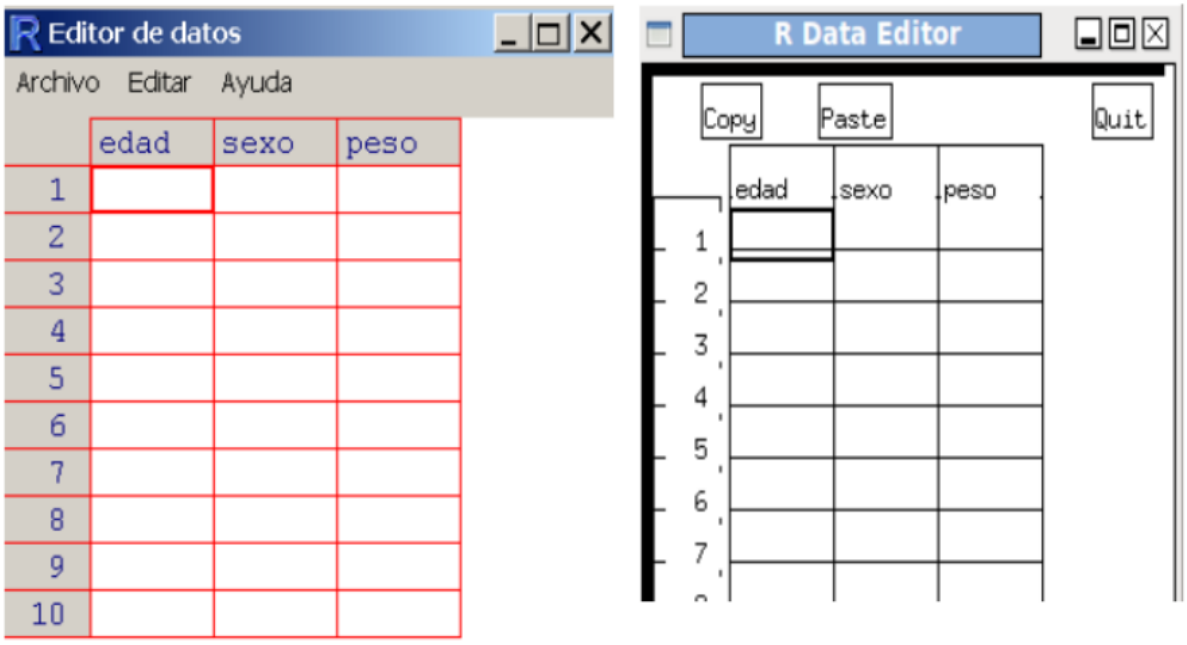


Figura 6: Editores visuales de datos: `edit( df )`

3.3. Desde una hoja de cálculo

Lo mejor para leer ficheros *excel* es guardar éstos en formato *csv*, es decir, texto delimitado por comas. También podemos leer el *xls* directamente con el paquete `RODBC`.

```
> #install.packages("RODBC")
--- Please select a CRAN mirror for use in this session ---
probando la URL 'http://cran.es.r-project.org/bin/windows/contrib/2.14/RODBC_1.3-5.zip'
Content type 'application/zip' length 753058 bytes (735 Kb)
URL abierta
downloaded 735 Kb

package 'RODBC' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\WINDOWS\Temp\RtmpecjIRj\downloaded_packages
> library("RODBC")
Mensajes de aviso perdidos
package 'RODBC' was built under R version 2.14.2
> channel <- odbcConnectExcel ("./files/10A-ejemplo01.xls")
> df3<-sqlFetch(channel, "sheet1")
> df3
  paciente   admision edad diabetes status
1        1 2010-11-11   23   tipo 1  bueno
2        2 2009-01-11   56   tipo 1  bueno
3        3 2012-03-15   78   tipo 2   malo
4        4 2008-04-28   34   tipo 1  bueno
5        5 2011-08-09   23   tipo 2 mejora
6        6 2012-04-24   12   tipo 1  bueno
7        7 2008-08-08   56   tipo 2  bueno
```

```
8        8 2099-01-21   88   tipo 1   malo
> odbcClose(channel)
```

Otras funciones que te pueden interesar: `read.xlsx()`

```
library( xlsx )
ficheroexcel <- "c:/ficheroexcel.xlsx"
mydataframe  <- read.xlsx( ficheroexcel, 1 )
# La instrucción anterior lee la primera hoja
# del fichero ficheroexcel.xlsx
```

3.4. Desde SPSS

Emplearemos la función `spss.get()` del paquete *Hmisc* (se requiere el paquete *foreign*).

```
# install.packages("foreign")
# install.packages("Hmisc")
library( Hmisc )
```

```
Loading required package: lattice
```

```
Loading required package: survival
```

```
Loading required package: Formula
```

```
Loading required package: ggplot2
```

```

Attaching package: 'Hmisc'
```

```
The following objects are masked from 'package:base':

    format.pval, units
```

```
df4 <- spss.get( "files/10A-mydata.sav", use.value.labels = TRUE )
```

```
re-encoding from CP1252
```

```
Warning in read.spss(file, use.value.labels = use.value.labels,
to.data.frame = to.data.frame, : Undeclared level(s) 15750, 15900,
16200, 16350, 16500, 16650, 16800, 16950, 17100, 17250, 17400, 17700,
18150, 18450, 18750, 19200, 19650, 19800, 19950, 20100, 20400, 20550,
20700, 20850, 21000, 21150, 21300, 21450, 21600, 21750, 21900, 22050,
22200, 22350, 22500, 22650, 22800, 22950, 23100, 23250, 23400, 23550,
23700, 23850, 24000, 24150, 24300, 24450, 24600, 24750, 24900, 25050,
25200, 25350, 25500, 25650, 25800, 25950, 26100, 26250, 26400, 26550,
26700, 26850, 27000, 27150, 27300, 27450, 27600, 27750, 27900, 28050,
28200, 28350, 28500, 28650, 28800, 28950, 29100, 29160, 29250, 29340,
29400, 29550, 29700, 29850, 30000, 30150, 30270, 30300, 30450, 30600,
30750, 30900, 31050, 31200, 31350, 31500, 31650, 31950, 32100, 32400,
32550, 32850, 33000, 33150, 33300, 33450, 33540, 33750, 33900, 34410,
34500, 34620, 34800, 34950, 35100, 35250, 35550, 35700, 36000, 36150,
36600, 37050, 37500, 37650, 37800, 38400, 38550, 38700, 38850, 39150,
39300, 39600, 39900, 40050, 40200, 40350, 40800, 41100, 41550, 42000,
42300, 43000, 43410, 43500, 43650, 43950, 44875, 45000, 45150, 45250,
45625, 46000, 46875, 47250, 47550, 48000, 48750, 49000, 50000, 50550,
51000, 51250, 51450, 52125, 52650, 53125, 54000, 54375, 54875, 54900,
55000, 55500, 55750, 56500, 56550, 56750, 57000, 58125, 58750, 59375,
59400, 60000, 60375, 60625, 61250, 61875, 62500, 65000, 66000, 66250,
```

```
66750, 66875, 67500, 68125, 68750, 69250, 70000, 70875, 72500, 73500,
73750, 75000, 78125, 78250, 78500, 80000, 81250, 82500, 83750, 86250,
90625, 91250, 92000, 97000, 100000, 103500, 103750, 110625, 135000
added in variable: salario

Warning in read.spss(file, use.value.labels = use.value.labels,
to.data.frame = to.data.frame, : Undeclared level(s) 9000, 9750,
10050, 10200, 10500, 10650, 10950, 11100, 11225, 11250, 11400, 11550,
12000, 12150, 12300, 12450, 12600, 12750, 12900, 13050, 13200, 13350,
13500, 13800, 13950, 14100, 14250, 14400, 14550, 14700, 15000, 15300,
15600, 15750, 16050, 16500, 16800, 17100, 17250, 17490, 18000, 18750,
19500, 19980, 20250, 20400, 20550, 21000, 21240, 21480, 21750, 21990,
22500, 23250, 23730, 24990, 25000, 25500, 26250, 27000, 27480, 27510,
27750, 28740, 29490, 30000, 30750, 31250, 31500, 31980, 32010, 32490,
33000, 33750, 34980, 35010, 35040, 36240, 36750, 37500, 39990, 42480,
42510, 43500, 44100, 45000, 47490, 52500, 60000, 79980 added in
variable: salini

Warning in read.spss(file, use.value.labels = use.value.labels,
to.data.frame = to.data.frame, : Undeclared level(s) 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98 added in
variable: tiempemp

Warning in read.spss(file, use.value.labels = use.value.labels,
to.data.frame = to.data.frame, : Undeclared level(s) 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 29, 30, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 66, 67, 68, 69, 70, 72, 74, 75, 76, 78, 79, 80, 81, 82, 83,
84, 85, 86, 87, 90, 91, 93, 94, 96, 97, 99, 101, 102, 103, 105, 106,
107, 108, 110, 113, 114, 115, 116, 117, 120, 121, 122, 123, 124, 125,
126, 127, 128, 129, 130, 132, 133, 134, 137, 138, 139, 143, 144, 149,
150, 151, 154, 155, 156, 159, 163, 165, 168, 169, 171, 173, 174, 175,
176, 180, 181, 182, 184, 190, 191, 192, 193, 194, 196, 198, 199, 205,
207, 208, 209, 210, 214, 216, 221, 228, 229, 231, 240, 241, 244, 246,
252, 258, 261, 264, 265, 271, 272, 275, 281, 284, 285, 288, 302, 305,
307, 308, 314, 315, 317, 318, 319, 320, 324, 338, 344, 348, 358, 359,
367, 371, 372, 375, 380, 381, 385, 387, 390, 408, 412, 429, 432, 438,
444, 451, 460, 476 added in variable: expprev
```

```
head( df4 )
```

```
  id   sexo     fechnac educ         catlab salario salini tiempemp
1  1 Hombre 11654150400   15      Directivo   57000  27000       98
2  2 Hombre 11852956800   16 Administrativo   40200  18750       98
3  3  Mujer 10943337600   12 Administrativo   21450  12000       98
4  4  Mujer 11502518400    8 Administrativo   21900  13200       98
5  5 Hombre 11749363200   15 Administrativo   45000  21000       98
6  6 Hombre 11860819200   15 Administrativo   32100  13500       98
  expprev minoría
1     144      No
2      36      No
3     381      No
4     190      No
5     138      No
```

```
6       67       No
```

3.5. Desde SAS

Pues emplear las funciones `read.ssd()` del paquete *foreign* y `sas.get()` del *Hmisc*. Aunque si estos ficheros son posteriores a SAS 9.1 no funcionarán; en ese caso deberás exportar los datos desde *SAS* a *csv* y emplear la función `read.table()` (*que siempre es mejor, más que ir aprendiendo tanta función*).

```
SAS :
proc export data=mydata
outfile="mydata.csv"
dbms=csv;
run;
```

3.6. Desde R

```
mydata <- read.table( "files/mydata.csv", header = TRUE, sep = "," )
```

3.7. Anotaciones

Al inicio de un análisis no necesitamos muchos detalles más que los vistos hasta ahora, pero para interpretar los resultados y que con el tiempo no dejen de tener sentido conviene 'anotarlos'.

- **Etiquetas de variables**
  Para una variable cualquiera, por ejemplo *"edad"*, podríamos querer que tuviera una etiqueta (según la nomenclatura de *spss*), que fuera *"Edad el día de ingreso (años)"*

```
names( df )[ 3 ] <- "Edad el día de ingreso (años)"
names( df )
```

```
[1] "paciente"                      "admision"
[3] "Edad el día de ingreso (años)" "diabetes"
[5] "status"
```

- **Etiquetas de valores**
  Podemos tener un factor codificado, por ejemplo `sexo`, donde 1 es *'Masculino'* y 2 *'Femenino'*. Podemos crear unas etiquetas con el siguiente código.
  Primero le añadimos a nuestro ejemplo una variable más (una columna) de "*"unos"* y *"doses"* representando el sexo codificado.

```
sexo    <- c( 1, 1, 1, 1, 2, 2, 2 )
df      <- data.frame( df, sexo )
df$sexo <- factor( df$sexo )
```

Comprobamos la estructura del *dataframe*:.

```
str( df )
```

```
'data.frame':   7 obs. of  6 variables:
 $ paciente                    : int  1 2 3 4 5 6 7
 $ admision                    : Factor w/ 7 levels "08/08/2008","09/08/2011",..: 4 3 5 7 2 6 1
 $ Edad.el.día.de.ingreso..años.: int  23 56 78 34 23 12 56
 $ diabetes                    : Factor w/ 3 levels "tipo1","tipo 1",..: 2 2 3 1 3 1 3
 $ status                      : Factor w/ 3 levels "bueno","malo",..: 1 1 2 1 3 1 1
 $ sexo                        : Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2
```

Creamos las etiquetas

```
df$sexo <- factor( df$sexo,
                   levels = c( 1, 2 ),
                   labels = c( "masculino", "femenino" )
                  )
```

Ejercicio. Mira ahora la estructura del *dataframe* con `str(df)`.

3.8. Algunas funciones útiles

Cuadro 6: Algunas funciones útiles.

Función	Acción
`length(obj)`	Número de componentes, elementos
`dim(obj)`	Dimensión de un objeto
`str(obj)`	Estructura de un objeto
`class(obj)`	Clase (`class`) o tipo de objeto
`names(obj)`	Nombres de los componentes de un objeto
`c(obj,obj,...)`	Combina objetos en un vector
`head(obj)`	Lista la primera parte de un objeto
`tail(obj)`	Lista la última parte (cola) de un objeto
`head(obj)`	Lista la primera parte de un objeto
`tail(obj)`	Lista la última parte (cola) de un objeto
`ls()`	Lista los objetos actuales
`rm(obj)`	Borra un objeto
`newobj <- edit(obj)`	Edita un objeto y lo guarda
`fix(obj)`	Edita sobre un objeto ya creado

4. Manipulación básica de objetos en R

Trabajaremos desde ahora con la *GUI RStudio*, veremos que todo es más sencillo y procuraremos coger un poco de práctica en trabajo sobre *scripts*, que es como la mayor parte de los especialistas trabajan. Pronto veremos que es lo más cómodo.

Vamos a crear un *data set* y trabajaremos con él a lo largo del capítulo; Trabajaremos con él en esta sección. El fichero con el *script* de creación del `dataframe` inicial lo puedes encontrar en `manipulacion-basica.R`

```
# getwd()
manager <- c( 1:5 )
date    <- c( "10/11/08", "10/12/08", "10/13/08", "10/14/08", "10/15/08" )
country <- c( "US", "US", "UK", "UK", "UK" )
gender  <- c( "M", "F", "F", NA, "F" )
age     <- c ( NA, 45, 25, 39, 99 )
q1      <- c( 5, 3, 3, 3, 2 )
q2      <- c( 5, 5, 5, NA, 2 )
q3      <- c( 5, 5, 2, NA, 1 )
df     <- data.frame( manager, date, country, gender, age, q1, q2, q3,
                      stringsAsFactors = FALSE )
df
```

```
  manager     date country gender age q1 q2 q3
1       1 10/11/08      US      M  NA  5  5  5
```

	manager	date	country	gender	age	q1	q2	q3
1	1	10/11/08	US	M	NA	5	5	5
2	2	10/12/08	US	F	45	3	5	5
3	3	10/13/08	UK	F	25	3	5	2
4	4	10/14/08	UK	NA	39	3	NA	NA
5	5	10/15/08	UK	F	99	2	2	1

Figura 7: Vista del navegador de objetos

```
2       2 10/12/08      US      F  45  3  5  5
3       3 10/13/08      UK      F  25  3  5  2
4       4 10/14/08      UK   <NA>  39  3 NA NA
5       5 10/15/08      UK      F  99  2  2  1
```

4.1. Operadores aritméticos

Si queremos crear una nueva variable usando operadores aritméticos es tan sencillo como saberse los operadores.

Cuadro 7: Operadores aritméticos

Operador	Descripción
`+`	Suma
`-`	Resta
`*`	Multiplicación
`*`	Multiplicación
`/`	División
`^` ó `**`	Potencias
`%%`	Módulo (x mod y) `5%%2` es 1
`%/%`	División entera `5%/%2` es 2

Así si queremos crear una variable `q1mas2` que sea `q1 + q2`

```
q1masq2 <- q1 + q2
```

Un método interesante para crear nuevas variables y que automáticamente se incluyan en el *dataframe* es el empleo de la función `transform()`.

```
# creamos variables con la función transform
df <- transform( df,
                 sumx  = q1 + q2,
                 meanx = ( q1 + q2 ) / 2 )
df
```

```
  manager     date country gender age q1 q2 q3 sumx meanx
1       1 10/11/08      US      M  NA  5  5  5   10     5
2       2 10/12/08      US      F  45  3  5  5    8     4
3       3 10/13/08      UK      F  25  3  5  2    8     4
4       4 10/14/08      UK   <NA>  39  3 NA NA   NA    NA
5       5 10/15/08      UK      F  99  2  2  1    4     2
```

Otra forma de incluir variables automaticamente en un dataframe es definiendolas *al vuelo* como parte del mismo:

```
df$nueva <- df$q1 * 2
df
```

4.2. Recodificación de variables

Para recodificar variables es interesante hablar, en primer lugar, de los operadores lógicos, pues a menudo recodificamos según una regla lógica.

Cuadro 8: Operadores lógicos

Operador	Descripción
`<`	Menor que
`<=`	Menor o igual que
`>`	Mayor que
`>=`	Mayor o igual que
`==`	Exactamente igual a
`==`	Exactamente igual a
`!=`	No igual a/que
`!x`	Diferente de `x`
`x | y`	`x` o `y`
`x & y`	`x` e `y`
`isTRUE(x)`	Evalua si `x` es una expesión verdadera

```
# Creamos la variable agecat (categoría de edad),
# le asignamos el código 'anciano' si age>75,
# si está entre 44 y 77 le asignamos 'maduro' y
# si está º por debajo de 44 'joven'.
df$agecat[ df$age > 75 ]                 <- "anciano"
df$agecat[ df$age <= 75 & df$age > 44 ] <- "maduro"
df$agecat[ df$age <= 44 ]                <- "joven"
df
```

```
  manager     date country gender age q1 q2 q3 sumx meanx nueva
1       1 10/11/08      US      M  NA  5  5  5   10     5    10
2       2 10/12/08      US      F  45  3  5  5    8     4     6
3       3 10/13/08      UK      F  25  3  5  2    8     4     6
4       4 10/14/08      UK   <NA>  39  3 NA NA   NA    NA     6
5       5 10/15/08      UK      F  99  2  2  1    4     2     4
   agecat
1    <NA>
2  maduro
3   joven
4   joven
5 anciano
```

La expresión `variable[ condición ] <- expresión` hará sólo la asignación cuando la condición sea TRUE.

4.3. Renombrar variables

Para renombrar variables lo más burdo sería recurrir a la función `fix( df )` (edición de la tabla) sustituyendo *"a mano"* los nombres. En contra se suele emplear el comando `rename`; en RStudio podemos editar los *data frames* clicando en la descripción del objeto, en la pestaña `Environment`.

```
# install.packages( 'reshape' ) # renombrar variables
library( reshape )

df <- rename( df,
```

```
            c(manager = "manID", date = "testdate") )

# Otra opción es con la función names(),
# sin necesidad de más paquetes que los básicos.
names( df )
```

```
 [1] "manID"    "testdate" "country"  "gender"   "age"      "q1"
 [7] "q2"       "q3"       "sumx"     "meanx"    "nueva"    "agecat"
```

```
names( df )[ 3 ] <- "pais"

# del mismo modo
names( df )[ 6:8 ] <- c( "it1", "it2", "it3" )
df
```

```
  manID testdate pais gender age it1 it2 it3 sumx meanx nueva  agecat
1     1 10/11/08   US      M  NA   5   5   5   10     5    10    <NA>
2     2 10/12/08   US      F  45   3   5   5    8     4     6  maduro
3     3 10/13/08   UK      F  25   3   5   2    8     4     6   joven
4     4 10/14/08   UK   <NA>  39   3  NA  NA   NA    NA     6   joven
5     5 10/15/08   UK      F  99   2   2   1    4     2     4 anciano
```

> **Nota**: Fijate en que hemos comentado la instalación del paquete, `install.packages()`, pues sólo es necesaria si no has instalado previamente la librería, y que la carga de la librería, `library()` sólo es necesaria cuando en la sesión de trabajo la necesitamos por primera vez.

4.4. Valores perdidos (*missing values*)

El tratamiento de datos perdidos, más conocidos como *missing* es fundamental para la correcta interpretación de un análisis.

- En R valores imposibles, *por ejemplo el resultando de* $\sqrt{-9}$, se representan por el código: `NaN` (*Not a Number*).
- Los valores perdidos (*simplemente, hemos perdido algún dato y no sabemos qué valor toma*) por el código: `NA` (*Not available*).

Hay muchas funciones para identificar estos valores, la más usada es `is.na()`.

```
# missings
y <- c( 1, 2, 3, NA )
is.na( y )  # ¿cuáles son NA?,
```

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

```
# is.na() devuelve un objeto del mismo tamaño que el que recibe.
is.na( df[ , 1:10 ] )
```

```
     manID testdate  pais gender   age   it1   it2   it3  sumx meanx
[1,] FALSE    FALSE FALSE  FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[3,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[4,] FALSE    FALSE FALSE   TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[5,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

4.4.1. Recodificar valores a *missing*

Para recodificar valores faltantes podemos empelar las mismas técnicas de recodificación que ya hemos visto.

```
# recodificar
df$age[ is.na( df$age ) ] <- 99
# observa qué ocurre con 'is.na(df)'
is.na( df[ , 1:10 ] )
```

```
     manID testdate  pais gender   age   it1   it2   it3  sumx meanx
[1,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[3,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[4,] FALSE    FALSE FALSE   TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[5,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# ahora devolvemos el df a su estado original.
df$age[ df$age == 99 ] <- NA
is.na( df[ , 1:10 ] )
```

```
     manID testdate  pais gender   age   it1   it2   it3  sumx meanx
[1,] FALSE    FALSE FALSE  FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[3,] FALSE    FALSE FALSE  FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[4,] FALSE    FALSE FALSE   TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[5,] FALSE    FALSE FALSE  FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

4.4.2. Excluir valores *missing* del análisis

En muchas funciones se incorpora un argumento para excluir estos valores. Emplearemos la opción `na.rm = TRUE` para que **no** considere los valores faltantes.

```
# excluir los missings del análisis
x <- c( 1, 2, NA, 3 )
y <- x[ 1 ] + x[ 2 ] + x[ 3 ] + x[ 4 ]
z <- sum( x )
y
```

```
[1] NA
```

```
z
```

```
[1] NA
```

```
# ¡¡¡¡Ambos son NA!!!!
sum( x, na.rm = T )  # ¡Ahora no!
```

```
[1] 6
```

De una forma más general podemos emplear la función `na.omit()` que elimina cualquier fila de un *data frame* que tenga valores perdidos.

```
df <- na.omit( df )
df
```

```
  manID testdate pais gender age it1 it2 it3 sumx meanx nueva agecat
2     2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
3     3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
```

> **Nota**: Si tenemos una tabla de datos con valores perdidos, pero trabajamos con un subconjunto de variables que no los presentan valores, no debemos eliminar *a priori* las observaciones incompletas, pues esto nos afectaría eliminando observaciones válidas.

4.5. Conversión de tipos

Muchas veces necesitamos que R entienda un número como un carácter, o viceversa. En R cuando añadimos a un vector numérico un elemento no numérico convierte todo el vector en no numérico automáticamente.

Cuadro 9: Conversión de tipos

lógico	convierte
`is.numeric()`	`as.numeric()`
`is.character()`	`as.character()`
`is.vector()`	`as.vector()`
`is.matrix()`	`as.matrix()`
`is.data.frame()`	`as.data.frame()`

```
# conversión de tipos.
a <- c( 1, 2, 3 )
a
```

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

```
is.numeric( a )
```

```
[1] TRUE
```

```
is.vector( a )
```

```
[1] TRUE
```

```
a <- as.character( a )
a
```

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

```
is.numeric( a )
```

```
[1] FALSE
```

```
is.vector( a )
```

```
[1] TRUE
```

4.6. Ordenar datos

Para ordenar un vector es suficiente con la función `sort`; Pero para ordenar las filas de un *data frame* empelamos la función `order()`, que nos genera un índice de orden, por defecto ordena ascendentemente. Empleamos el signo *'menos'* para cambiar el sentido.

```
df
```

```
  manID testdate pais gender age it1 it2 it3 sumx meanx nueva agecat
2     2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
3     3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
```

```
df_ordenado <- df[ order( df$age ),  ]
df_ordenado
```

```
  manID testdate pais gender age it1 it2 it3 sumx meanx nueva agecat
3     3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
2     2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
```

```
df_ordenado2 <- df[ order( df$gender , -df$age ),  ]
df_ordenado2
```

```
  manID testdate pais gender age it1 it2 it3 sumx meanx nueva agecat
2     2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
3     3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
```

4.7. Ampliar/unir conjuntos de datos

Podemos ampliar un *data frame* añadiendo variables (columnas) o casos (filas). Y añadir columnas con las funciones `merge()` y `cbind()`.

```
dfA <- df  # nos creamos un par de dataframes aunque sean iguales
dfB <- df
df.total <- merge( dfA, dfB,
                   by = "manID" )

df.total
```

```
  manID testdate.x pais.x gender.x age.x it1.x it2.x it3.x sumx.x
1     2   10/12/08     US        F    45     3     5     5      8
2     3   10/13/08     UK        F    25     3     5     2      8
  meanx.x nueva.x agecat.x testdate.y pais.y gender.y age.y it1.y
1       4       6   maduro   10/12/08     US        F    45     3
2       4       6    joven   10/13/08     UK        F    25     3
  it2.y it3.y sumx.y meanx.y nueva.y agecat.y
1     5     5      8       4       6   maduro
2     5     2      8       4       6    joven
```

```
# unir matrices
A        <- matrix( 1:9, 3, 3 )
B        <- matrix( 1:9, 3, 3 )
AB.total <- cbind( A, B )
AB.total
```

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

```
# unir matrices
lA        <- matrix( c( "a", "b", "c", "d" ), 2, 2 )
lB        <- matrix( c( "A", "B", "C", "D" ), 2, 2 )
lAB.total <- cbind( A, B )
lAB.total
```

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

Para unir filas (observaciones) empleamos la función `rbind(dfA, dfB)`.

Nota: Ambos *data frames* han de tener las mismas columnas.

```
df.total2 <- rbind( dfA, dfB )
df.total2
```

```
   manID testdate pais gender age it1 it2 it3 sumx meanx nueva agecat
2      2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
3      3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
21     2 10/12/08   US      F  45   3   5   5    8     4     6 maduro
31     3 10/13/08   UK      F  25   3   5   2    8     4     6  joven
```

4.8. Seleccionar subconjuntos de un *data frame*

Muy a menudo necesitamos efectuar operaciones sobre una parte de un conjunto de datos, ya sea de un subconjunto de variables o de un subconjunto de observaciones. R es muy ágil en estas operaciones y hay muchas formas de acceder a observaciones o variables concretas.

Usaremos los elementos de un *data frame*, los corchetes para acceder a partes de él.

`data.frame[índice de fila, índice de columna]`

> **Nota**: la regla es *"filas por columnas"*.

4.8.1. Seleccionar variables (columnas)

`nuevo_df<- df[ , índices]`

Observar que dejamos en blanco la primera posición entre los corchetes para referirnos a la columnas

```
# Seleccionar columnas de un dataframe
df <- data.frame( manager, date, country, gender, age, q1, q2, q3,
                  stringsAsFactors = FALSE )
df
```

```
  manager     date country gender age q1 q2 q3
1       1 10/11/08      US      M  NA  5  5  5
2       2 10/12/08      US      F  45  3  5  5
3       3 10/13/08      UK      F  25  3  5  2
4       4 10/14/08      UK   <NA>  39  3 NA NA
5       5 10/15/08      UK      F  99  2  2  1
```

```
new_df <- df[ , c( 6:8 ) ]
new_df
```

```
  q1 q2 q3
1  5  5  5
2  3  5  5
3  3  5  2
4  3 NA NA
5  2  2  1
```

```
new_df2 <- df[ , "country" ]
new_df2    # por el nombre de la 'columma'
```

```
[1] "US" "US" "UK" "UK" "UK"
```

```
new_df3 <- df[ , 3 ]
new_df3    # por el índice
```

```
[1] "US" "US" "UK" "UK" "UK"
```

```
# aunque más generalmente si especificamos el nombre no hace falta tratar
# con los índices
```

```
new_df4 <- df[ "country" ]
new_df4    # por el nombre de la 'columna'
```

```
  country
1      US
2      US
3      UK
4      UK
5      UK
```

```
str( new_df4 )
```

```
'data.frame':   5 obs. of  1 variable:
 $ country: chr  "US" "US" "UK" "UK" ...
```

```
str( new_df3 )
```

```
 chr [1:5] "US" "US" "UK" "UK" "UK"
```

Igualmente podríamos haber construido un vector de **chars**.

```
variables <- c( "country", "date")
new_df    <- df[ , variables ]`
```

Seleccionar para eliminar: Atención al signo '-' delante del índice.

```
# eliminamos las variables q3 y q4, que tienen índices 8 y 9
new_df5 <- df[ c( -8, -9 ) ]
new_df5
```

```
  manager     date country gender age q1 q2
1       1 10/11/08      US      M  NA  5  5
2       2 10/12/08      US      F  45  3  5
3       3 10/13/08      UK      F  25  3  5
4       4 10/14/08      UK   <NA>  39  3 NA
5       5 10/15/08      UK      F  99  2  2
```

4.8.2. Seleccionar observaciones (filas)

Haremos uso de las mismas reglas que para las columnas pero dejando vacía la segunda posición del corchete y jugando con los operadores lógicos.

```
nuevo_df<- df[índices, ]
```

```
# Seleccionar observaciones
df
```

```
  manager     date country gender age q1 q2 q3
1       1 10/11/08      US      M  NA  5  5  5
2       2 10/12/08      US      F  45  3  5  5
3       3 10/13/08      UK      F  25  3  5  2
4       4 10/14/08      UK   <NA>  39  3 NA NA
5       5 10/15/08      UK      F  99  2  2  1
```

```
# seleccionar las 3 primeras filas
df6 <- df[ 1:3, ]
df6
```

```
  manager     date country gender age q1 q2 q3
1       1 10/11/08      US      M  NA  5  5  5
2       2 10/12/08      US      F  45  3  5  5
```

```
3       3 10/13/08      UK      F  25  3  5  2
```

```
df7 <- df[ which( df$gender == "F" & df$country == "UK" ),  ]
df7
```

```
  manager     date country gender age q1 q2 q3
3       3 10/13/08      UK      F  25  3  5  2
5       5 10/15/08      UK      F  99  2  2  1
```

Una magnifico texto de referencia se puede encontrar en el texto de Joseph Adler, *R in a nutshell*(Adler, 2012)

5. Funciones

No vamos a detenernos muchos en los detalles de las funciones, durante el curso trabajaremos con muchas de ellas y sus paticularidades, sólo decir que hay miles y miles, agrupadas en paquetes, y cada día hay más paquetes que aporta la comunidad y que se encuentran en CRAN como ya mencionamos al principio del texto. La página *Rdocumentation*, en www.rdocumentation.org a la hora de redactar este documento recogía entre los distintos repositorios de R un total de 1,757,257 funciones.

5.1. Funciones matemáticas

Para empezar sí debemos de saber que existen un buen número de funciones matemáticas estándar, su sintaxis es muy parecida a otros lenguajes de programación.
Algunas son:

Cuadro 10: Funciones matemáticas

Función	Devuelve
`abs(x)`	Valor absoluto de `x`, `abs(-4)` devuelve 4
`sqrt(x)`	Raíz cuadrada de `x`, `sqrt(25)` devuelve 5
`ceiling(x)`	Entero más pequeño mayor que `x`, `ceiling(4.6)` devuelve 5
`floor(x)`	Entero más grande no mayor que `x`, `floor(4.6)` devuelve 4
`trunc(x)`	Truncamiento de `x`
`round(x, digits=n)`	Redondea `x` a un número específico de decimales, por defecto `n=0`
`log(x,base=n)`	Logaritmos (`log` devuelve el logaritmo natural, log(exp(1)) devuelve 1

5.2. Funciones estadísticas

Las funciones estadísticas más habituales se recogen en la tabla siguiente.

Cuadro 11: Funciones estadísticas

Función	Devuelve
`mean(x)`	Media
`median(x)`	Mediana
`sd(x)`	Desviación estándar
`var(x)`	Varianza
`sum(x)`	Suma
`cumsum(x)`	Suma acumulada
`range(x)`	Rango

Función	Devuelve
`min(x)`	Mínimo
`max(x)`	Máximo
`table(x)`	Tabla de frecuencias de repetición de los valores de `x`
`scale(x,center=TRUE, scale=TRUE)`	Estandarizar, devuelve una varialbe con $\bar{x} = 0$ y $s^2 = 1$

Ejercicio: Probar estas funciones con `x<-c( 1, 2, 3, 4, 5, 6, 7, 8, 9 )`

Existen también funciones de densidad de probabilidad, test estadísticos... que veremos más adelante.

Las funciones de caracteres (`strings`, `chars`), que buscan, sustituyen, cuentan caracteres en cadenas, etc. nos las vamos a saltar también porque no son *"importantes"*"para nuestros objetivos.

5.3. Miscelánea de funciones interesantes

`length()`

```
# Miscelánea de funciones interesantes medir objetos
x <- c( 1, 2, 3, 4, 5, 6, 7, 8, 9 )
length( x )
```

```
[1] 9
```

`seq()`

```
# generar secuencias con saltos
seq( 1, 10, 2 )     # seq(from, to, by)
```

```
[1] 1 3 5 7 9
```

```
seq( 1, 20, 3 )
```

```
[1]  1  4  7 10 13 16 19
```

`rep()`, `sort()`

```
# repetir
rep( 1, 5 )    # rep(ob, número de repet)

rep( c( "A", "B" ), 5 )

rep( 1:3, 2 )

# series repetidas y ordenadas
sort( rep( 1:3, 2 ) )
```

`cat()`

```
# concatenar
cat( "hola", "amigo" )
```

```
hola amigo
```

```
cat( "hola",
    "amigo",
    "\n",
```

```
    "¿cómo estas?",
    file = "fichero-cocatenado.txt" ) # escribimos en un fichero
```

5.4. Aplicando funciones a objetos

Una funcionalidad que resulta muy útil al trabajar con R es que **podemos aplicar funciones a una gran cantidad de objetos**: vectores, *arrays*, matrices, *data frames*...

```
# aplicar funciones a objetos 'complejos'
y <- c( 1.23, 4.56, 7.89 )
round( y )
```

```
[1] 1 5 8
```

```
z <- matrix( runif( 12 ), 3 )
z
```

```
      [,1]  [,2]   [,3]  [,4]
[1,] 0.262 0.510 0.2576 0.854
[2,] 0.165 0.924 0.0465 0.347
[3,] 0.322 0.511 0.4179 0.131
```

```
log( z )
```

```
      [,1]    [,2]   [,3]   [,4]
[1,] -1.34 -0.6731 -1.356 -0.158
[2,] -1.80 -0.0791 -3.069 -1.058
[3,] -1.13 -0.6715 -0.873 -2.029
```

```
mean( z )  # devuelve un escalar
```

```
[1] 0.396
```

5.5. Aplicando funciones a los elementos de los objetos (`apply`)

En el último ejemplo `mean(z)` devuelve un escalar cuando le habíamos proporcionado una `matriz`, ¿qué hacemos si queremos las medias por filas?

Para eso en R podemos emplear la función `apply()`. Si quieres saber más prueba `?apply`.

```
apply( z, 1, mean )  # medias de las filas
```

```
[1] 0.471 0.371 0.346
```

```
apply( z, 2, mean )  # medias de las columnas
```

```
[1] 0.250 0.648 0.241 0.444
```

`apply(x, MARGIN, FUN)`

- `FUN`: cualquier función
- `MARGIN`: índice de la dimensión(1=fila, 2=columna) a la que se la quieres aplicar

Nota: `apply()` es una potentísima herramienta; existe una familia de funciones sejemantes `sapply`, `lapply`, `tapply`...

5.6. Control de flujo

Sobre control de flujo no vamos a tratar nada en este curso, sólo mencionaremos que como en cualquier lenguaje de programación existe y es fundamental para un trabajo avanzado. Lo mencionamos por si te animas a explorarlo por que crees que puedes sacarle partido.

Aunque a continuación seguiremos hablando de funciones, dada su *naturaleza habitual dentro de los lenguajes de programación* nos referiremos a ellas también con el nombre de sentencias.

5.6.1. Sentencia `for`

Genera lo que se denomina un bucle de evaluación. Evaluará una o varias expresiones hasta que la variable no esté contenida en una secuencia dada:

```
for ( variable in secuencia ) { expresión/ones }
```

```
# FOR
x <- 0
x
```

```
[1] 0
```

```
# Pueden obviarse las llaves al contener una sola expresión
for ( i in 1:3 ) { x <- x + 1 }
x
```

```
[1] 3
```

```
for ( i in 1:2 ) { print( "¡¡Viva el software libre!!" ) }
```

```
[1] "¡¡Viva el software libre!!"
[1] "¡¡Viva el software libre!!"
```

5.6.2. Sentencias `while`

Un bucle `while` evalua expresiones mientras se cumpla una condición.

```
while (condición) sentencia
```

```
i <- 5
while ( i > 0 ) {
    print( "¡Perfecto mundo!" )
    i <- i - 1
}
```

```
[1] "¡Perfecto mundo!"
[1] "¡Perfecto mundo!"
[1] "¡Perfecto mundo!"
[1] "¡Perfecto mundo!"
[1] "¡Perfecto mundo!"
```

NOTA: En R, en la practica, `while` apenas se emplea siendo sustituidas por una combianción `for` e `if`.

5.6.3. Sentencias `if-else`

Evalúa expresiones si se da una condición (`if`), si no se cumple evalúan expresión alternativa (`else`). La utilización de `else` es opcional.

```
if (condición) sentencia
ó
if (condición) sentencia1 else sentencia2
```

```
# IF-ELSE
if ( TRUE ) {
    print( "esta siempre se ejecuta" )
}

if ( FALSE ) {
    print( "esta nunca se ejecuta" )
    print( "pero nunca, nunca" )
} else {
    print( "¿lo ves?" )
    print( ";-)" )
}
```

Nota: Las sentencias `for`, `while` e `if-them` puede anidarse.

5.7. Funciones escritas por el usuario

Una de las mayores potencialidades de R es la posibilidad de escribir funciones *ad hoc*. Si además consideramos que estas se pueden empaquetar en grupos y tener siempre disponibles para todas las sesiones nos podemos hacer una idea de lo potente y personalizable que puede llegar a ser R.

Un ejemplo muy sencillo:

```
# Definimos
euros <- function( pesetas ) pesetas/166.386

#Usamos
euros( 1000 )
```

La sintaxis, com vemos, es sencilla; partimos de un nombre, un procedimiento y un conjunto de argumentos:

```
mi_funcion <- function( arg1, arg2, ... ) {
    expresiones, para realizar el procedimiento
    return( objeto )
}
```

```
f.potencia <- function( num, exp = 2 ) {
    return( num * exp )
}
f.potencia( 5, 2 )
```

```
[1] 10
```

```
f.potencia( 5, 5 )
```

```
[1] 25
```

```
f.potencia( 5 )
```

```
[1] 10
```

Ejercicio: Crear una función que acepte dos argumentos, uno que sea un `char` con dos posibilidades `param` o `noparam`, que por defecto sea el valor "param". El segundo argumento que sea un vector numérico. Y que nos devuelva para "param", la media, la desviación típica y la

varianza del vector. Y para "noparam", la mediana, los cuartiles 0.25 y 0.75, máximo y mínimo del vector.

Ejercicio: Crear una función que resuelva la ecuación de segundo grado. Y probarla para la ecuación $x^2 + 5x - 16 = 0$

```
# posible solución
e2grado <- function( a, b, c ) {
    discriminante <- ( b ^ 2 ) - ( 4 * a * c )
    solucion1 <- ( -b + sqrt( discriminante ) ) / ( 2 * a )
    solucion2 <- ( -b - sqrt( discriminante ) ) / ( 2 * a )
    c( solucion1, solucion2 )
}
e2grado( 1, 5, -16 )
```

```
[1]  2.22 -7.22
```

Podemos ahondar en muchos textos sobre la creación de funciones por el usuario, un texto sencillo es el de Emmanuel Paradis, *R para principiantes*(Paradis, 2003), también el de R. Kabacoff, *R in Action*(Kabacoff, 2011).

5.7.1. Cargar una función *ad hoc* al inicio de la sesión de R

Basta con crear un fichero que contenga la función, o funciones o lo que deseemos calcular, a este fichero lo llamamos `script`. Lo cargamos al inicio de la sesión.

Nota: En Linux una forma sencilla de hacerlo es modificar el fichero `Rprofile.site` que por defecto se encuentra en `/usr/lib/R/etc/`. Modificaremos la función `.First` en `Rprofile.site`, cambiando única línea de código del fichero `.Firts`.

```
# .First <- function() cat("\n   Welcome to R!\n\n")
```

Por ejemplo para hacer que el *script* `fdescriptivos.r` se ejecute al inicio escribiremos:

```
.First <- function(){
                     cat( "\n   Welcome to R!\n\n" )
                     source( "/usr/lib/R/etc/fdescriptivos.R" )
                    }
```

6. Estadística descriptiva con R

6.1. Algunas definiciones

Vamos pasar, aunque muy superficialmente, por conceptos estadísticos importantes; lo haremos de una manera intuitiva ya que nos será útil para encarar más adelante el concepto de contraste estadístico y *p-valor* más eficientemente.

6.2. Población y muestra. Variables

- **Población**: Llamamos *población estadística*, *universo* o *colectivo* al conjunto de referencia sobre el cual van a recaer las observaciones.

- **Individuos**: Se llama *unidad estadística* o *individuo* a cada uno de los elementos que componen la población estadística. El individuo es un ente observable que no tiene por qué ser una persona, puede ser un objeto, un ser vivo, o incluso algo abstracto.
- **Muestra**: Es un subconjunto de elementos de la población. Se suelen tomar muestras cuando es difícil o costosa la observación de todos los elementos de la población estadística.
- **Censo**: Decimos que realizamos un censo cuando se observan todos los elementos de la población estadística.
- **Caracteres**: La observación del individuo la describimos mediante uno o más caracteres. El carácter es, por tanto, una cualidad o propiedad inherente en el individuo.

Tipos de caracteres:

- Cualitativos : aquellos que son categóricos, pero no son numéricos.
  - por ej. "color de los ojos", "profesión", "marca de coche"...
- Ordinales : aquellos que pueden ordenarse, pero no son numéricos.
  - por ej. "preguntas de encuesta sobre el grado de satisfacción de algo"
  - Mucho, poco, nada. Bueno, regular, malo...
- Cuantitativos : son numéricos.
  - por ej. "peso", "talla", "número de hijos", "número de libros leídos al mes"...

Modalidad o valor: Un carácter puede mostrar distintas modalidades o valores, es decir, son distintas manifestaciones o situaciones posibles que puede presentar un carácter estadístico. Las modalidades o valores son incompatibles y exhaustivos.

- Generalmente se utiliza el término *modalidad* cuando hablamos de caracteres cualitativos y el término *valor* cuando estudiamos caracteres cuantitativos.
- Por ejemplo: el carácter cualitativo "Estado Civil" puede adoptar las modalidades : casado, soltero, viudo. El carácter cuantitativo "Edad" puede tomar los valores : diez, once, doce, quince años...

Variable estadística: Al conjunto de los distintos valores numéricos que adopta un carácter cuantitativo se llama variable estadística.

Tipos de variables estadísticas :

- **Discretas**: Aquellas que toman valores aislados (números naturales), y que no pueden tomar ningún valor intermedio entre dos consecutivos fijados.
  - por ej. "núm. de goles marcados", "núm. de hijos", "núm. de discos comprados", "núm. de pulsaciones"...
- **Continuas**: Aquellas que toman infinitos valores (números reales) en un intervalo dado, de forma que pueden tomar cualquier valor intermedio, al menos teóricamente, en su rango de variación.
  - por ej. "talla", "peso", "presión sanguínea", "temperatura"...

Observación: Una observación es el conjunto de modalidades o valores de cada variable estadística medidos en un mismo individuo.

```
- p.ej. en una población de 100 individuos podemos estudiar,
  de forma individual, tres caracteres:

   - "edad : 18, 19, ...",
   - "sexo : Hombre, Mujer" y
   - "ha votado en las elecciones : Sí, No".

  Realizamos 100 observaciones con tres datos cada una, es decir,
```

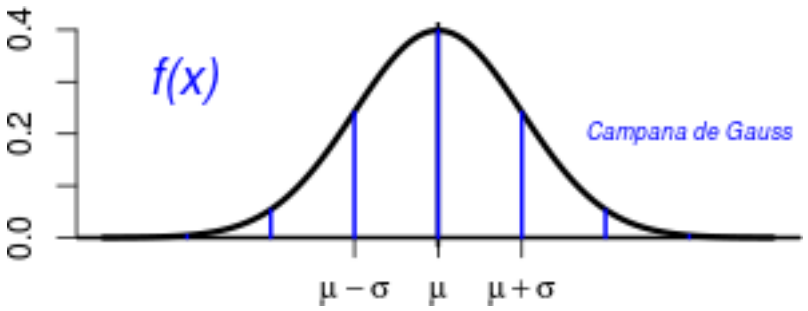


Figura 8: Función de densidad de la distribución normal: campana de Gauss

```
una de las observaciones puede ser (43, H, S).
```

6.3. Distribución de probabilidad y función de densidad de una variable aleatoria

Intuitivamente, una *variable aleatoria* puede tomarse como una cantidad cuyo valor no es fijo pero puede tomar diferentes valores; una *distribución de probabilidad* se usa para describir la probabilidad de que se den los diferentes valores (se denota usualmente por $F(x)$).

$$F_x(x) = P(x \leq x)$$

La distribución de probabilidad de una v.a. describe teóricamente la forma en que varían los resultados de un experimento aleatorio. Intuitivamente se trataría de una lista de los resultados posibles de un experimento con las probabilidades que se esperarían ver asociadas con cada resultado.

La *función de densidad de probabilidad*, ***función de densidad***, o, simplemente, ***densidad de una variable aleatoria continua*** es una función, usualmente denominada $f(x)$ que describe la densidad de la probabilidad en cada punto del espacio de tal manera que la probabilidad de que la variable aleatoria tome un valor dentro de un determinado conjunto sea la integral de la función de densidad sobre dicho conjunto("Función de densidad de probabilidad - wikipedia, la enciclopedia libre," n.d.).

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

6.4. Parámetros y estadísticos

- **Parámetro**: Es una cantidad numérica calculada sobre una población.
  - La altura media de los individuos de un país
  - La idea es resumir toda la información que hay en la población en unos pocos números (parámetros).
- **Estadístico**: Ídem (cambiar población por muestra; es decir "Es una cantidad numérica calculada sobre una muestra".
  - La altura media de los que estamos en este aula.
  - ¿Somos una muestra (¿representativa?) de la población?

Si un estadístico se usa para aproximar un parámetro también se le suele llamar **estimador**.

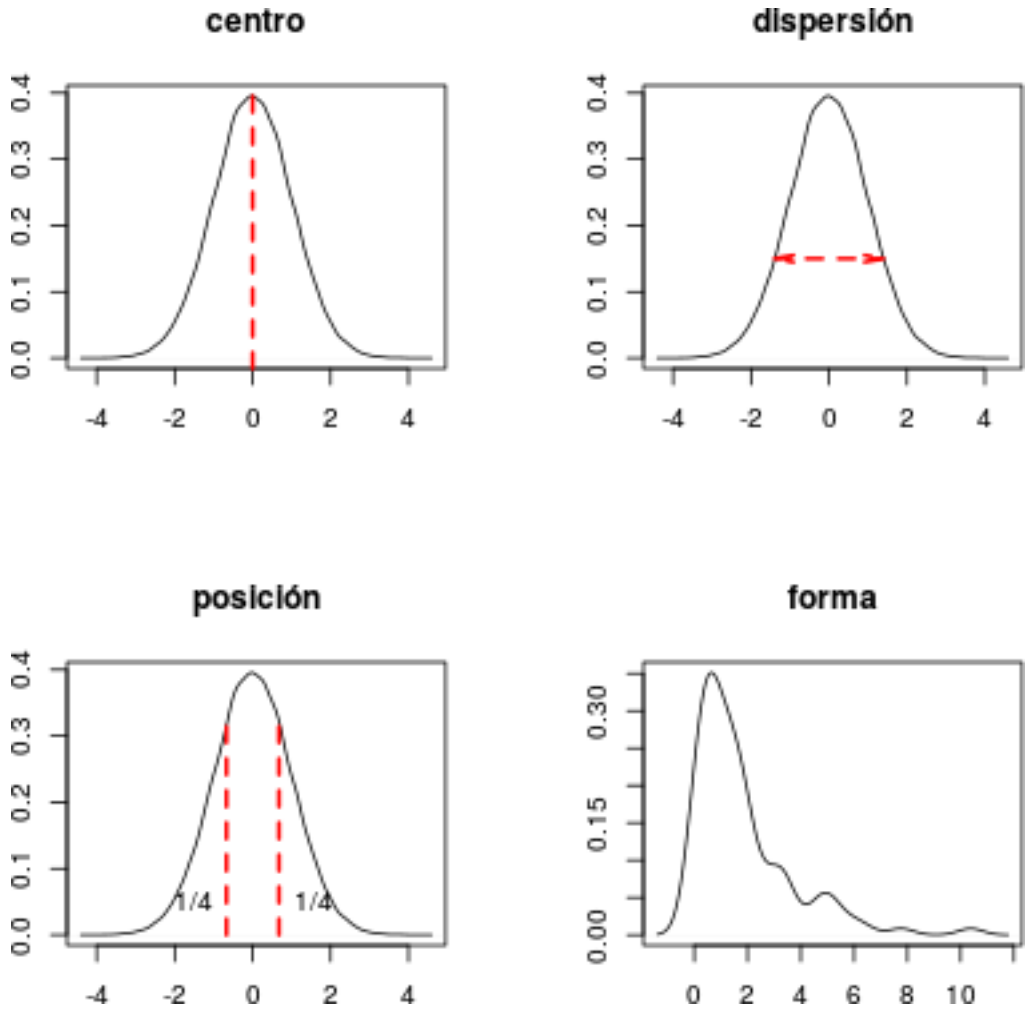


Figura 9: Tipos de esadísticos.

6.5. Tipos de estadísticos

Los estadísticos se calculan, y estos estiman parámetros.

Hay diferentes tipos según las *cosas* que queramos saber de la distribución de una variable.

6.6. Medidas de posición

Dividen un conjunto ordenado de datos en grupos con la misma cantidad de individuos.

Las más populares: cuantiles, percentiles, cuartiles, deciles...

- Se define el cuantil de orden α como un valor de la variable por debajo del cual se encuentra una frecuencia acumuladada α.
- El percentil de orden k es el cuantil de orden $\frac{k}{100}$.

  Ejemplo: El 5 % de los recién nacidos tiene un peso demasiado bajo. ¿Qué peso se considera "demasiado bajo"? Percentil 5 o cuantil 0.05.

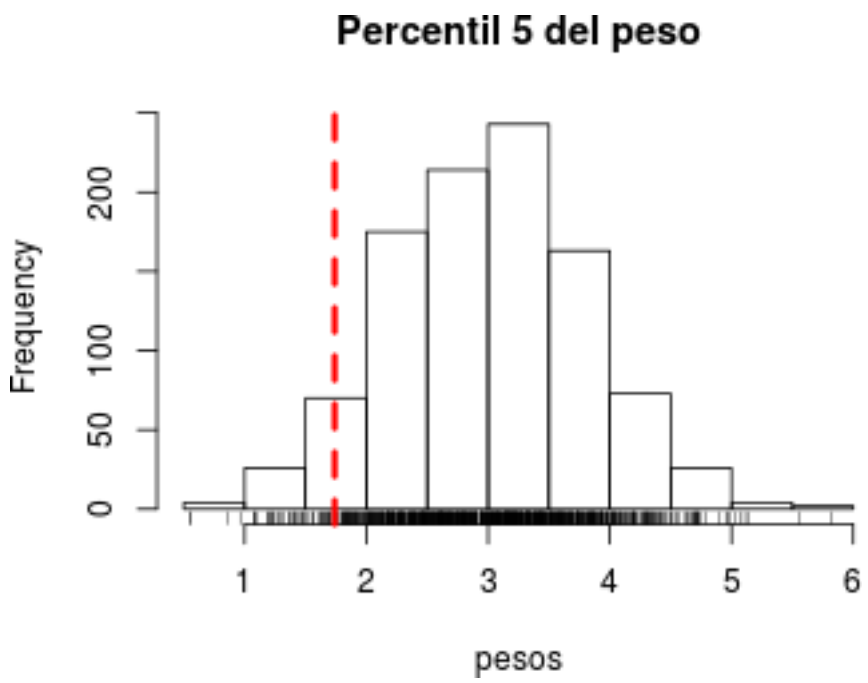


Figura 10: Ejemplo de percentil 5.

```
pesos <- rnorm( 1000, 3, 0.8 )
hist( pesos )
```

```
quantile( pesos, 0.05 )
```

```
  5%
1.71
```

Ejemplo: ¿Qué peso es superado sólo por el 25 % de los individuos? Percentil 75, tercer cuartil o cuantil 0.75.

```
quantile( pesos, 0.75 )
```

```
75%
3.5
```

6.7. Medidas de centralización o tendencia central

Indican valores con respecto a los que los datos 'parecen' agruparse.

- Media, mediana, moda...

Media

Media (*mean*). Es la media aritmética (promedio) de los valores de una variable. Suma de los valores dividido por el tamaño muestral.

```
mean( pesos )
```

```
[1] 3
```

```
x <- c( 1, 2, 3, 4, 5 )
mean(x)
```

```
[1] 3
```

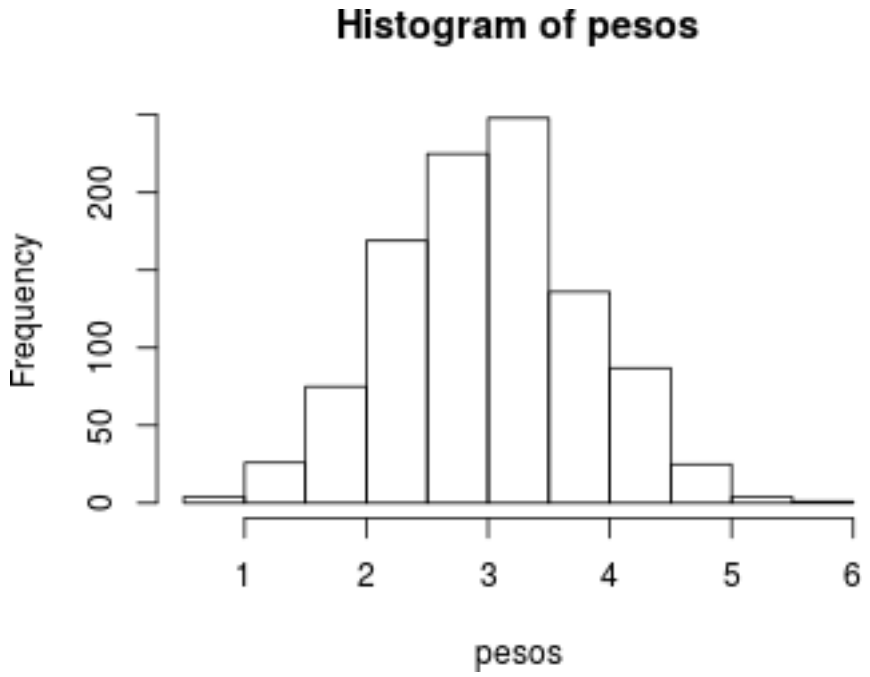


Figura 11: Histograma para los datos de peso.

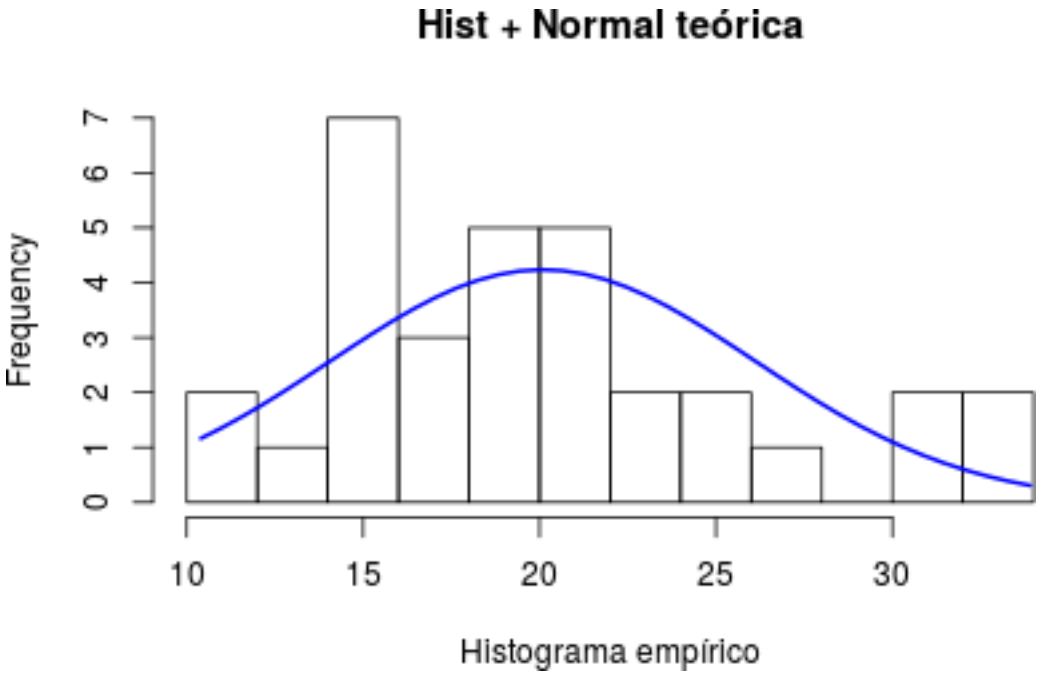


Figura 12: Histograma normal y curva normal teórica asociada.

```
media <- ( 1 + 2 + 3 + 4 + 5 ) / 5
media
```

```
[1] 3
```

- La media es conveniente cuando los datos se concentran simétricamente con respecto a ese valor. Es muy sensible a valores extremos (en estos casos hay otras 'medias', menos intuitivas pero que pueden ser útiles: media geométrica, ponderada...)

  La media representa el centro de gravedad de los datos.

Mediana

Mediana ('median'). Es un valor que divide a las observaciones en dos grupos con el mismo número de individuos (percentil 50). Si el número de datos es par, se elige la media de los dos datos centrales.

- Mediana de 1, 2, 4, **5**, 6, 6, 8 es 5
- Mediana de 1, 2, 4, **5, 6**, 6, 8, 9 es $\frac{5+6}{2} = 5{,}5$
- Es conveniente cuando los datos son asimétricos. No es sensible a valores extremos.
- Mediana de 1, 2, 4, 5, 6, 6, 800 es 5. ¡La media es 117, 7!

```
median( pesos )
```

```
[1] 3
```

```
x <- c( 1, 2, 4, 5, 6, 6, 8 )
median( x )
```

```
[1] 5
```

```
y <- c( 1, 2, 4, 5, 6, 6, 8, 9 )
z <- c( 1, 2, 4, 5, 6, 6, 800 )
median( z )
```

```
[1] 5
```

Moda

Es el valor donde la distribución de frecuencia alcanza un máximo. Puede haber más de uno (2 modas: *bimodal*, 3: *trimodal* ...)

```
# paquete "PrettyR" o crear una función
# install.packages( 'prettyR', dependencies = T )
library( "prettyR" )
help( package = "prettyR" )
```

```
# Otra forma, crearnos una función x discreta
modad <- function( x ) as.numeric( names( which.max( table( x ) ) ) )
# no funciona si hay más de una moda o si es constante!

modad( c( 1, 1, 3, 4, 5, 6, 6, 6) )
```

```
[1] 6
```

```
modac <- function( x ) {
    dd <- density( x )
    dd$x[ which.max( dd$y )]
}
modad( z )
```

```
[1] 6
```

Nota: Un inciso sobre el cálculo de medias

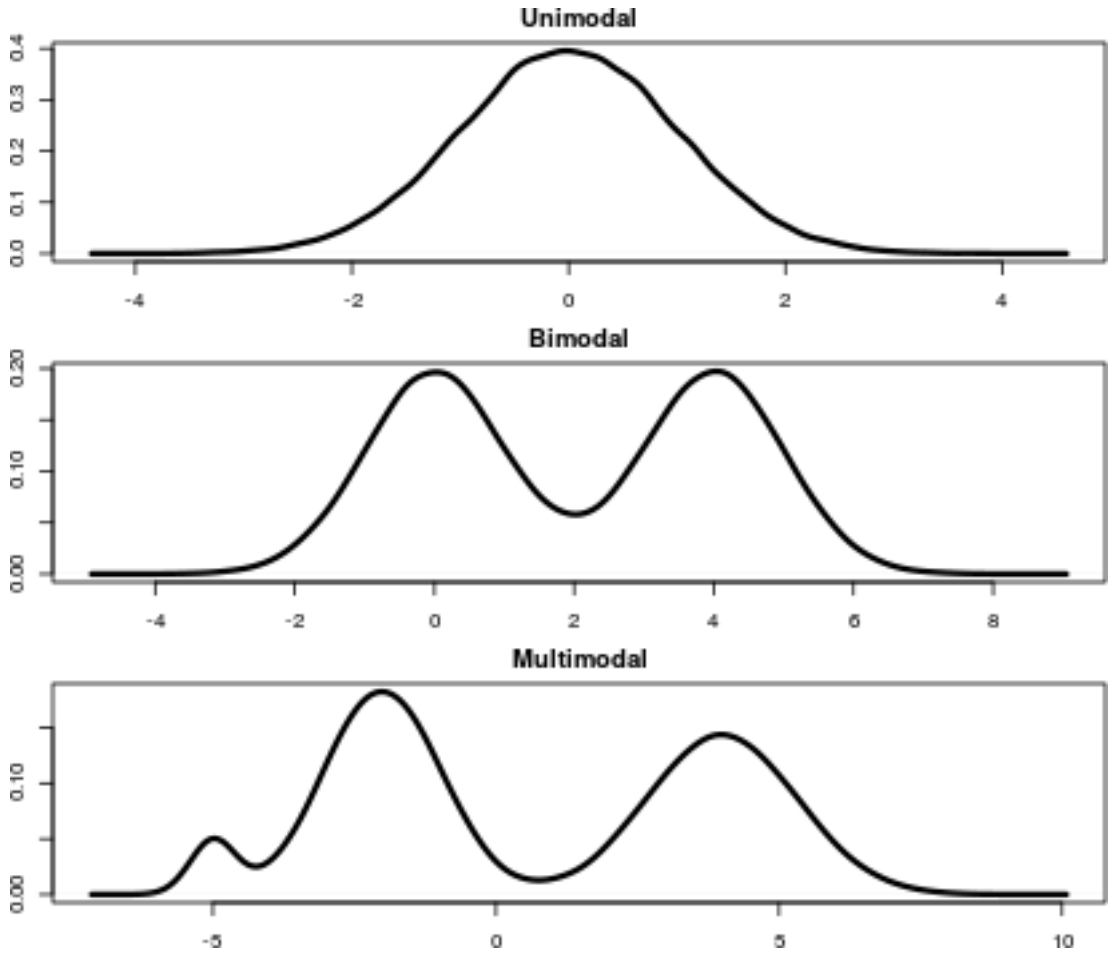


Figura 13: Ejempo de distrbución unimodal, la moda está en el intervalo 4–4.5.

- Datos sin agrupar: $x_1, x_2, \ldots, x_7 : \bar{x} = \dfrac{\sum_i x_i}{n}$
- Datos organizados en tabla: Si está en intervalos usar como x_i las marcas de clase. Si no ignorar la columna de intervalos: $\bar{x} = \dfrac{\sum_i x_i n_i}{n}$

Cuadro 15: Datos organizados en tabla

Intervalo	marca de clase	frecuencia
$L_0 - L_1$	x_1	n_1
$L_1 - L_2$	x_2	n_2
$L_2 - L_3$	x_3	n_3
$\ldots$	$\ldots$	$\ldots$
$L_{k-1} - L_k$	x_k	n_k

6.8. Medidas de dispersión

Indican la mayor o menor concentración de los datos con respecto a las medidas de centralización. Los más usuales son: la desviación típica, el coeficiente de variación, el rango, la varianza...

6.8.1. Fuentes de variabilidad

Imaginemos que los estudiantes de Bioestadística reciben diferentes calificaciones en la asignatura (variabilidad). ¿A qué puede deberse?

Lo que vemos es que hay diferencias individuales en el conocimiento de la materia. ¿Podría haber otras razones (fuentes de variabilidad)?
Supongamos, por ejemplo, que todos los alumnos poseen el mismo nivel de conocimiento. ¿Las notas serían las mismas en todos? Seguramente no.

Causas:

- Dormir poco el día del examen.
- Diferencias individuales en la habilidad para hacer un examen.
- El examen no es una medida perfecta del conocimiento.
- Variabilidad por error de medida.
- En alguna pregunta difícil, se duda entre varias opciones, y al azar se elige la mala.
- Variabilidad por azar, aleatoriedad.

6.8.2. Estadísticos de dispersión

- **Amplitud o Rango (*range*)** Diferencia entre observaciones extremas.
- **Rango intercuartílico** (*interquartile range*, IQR): Es la distancia entre primer y tercer cuartil. Rango intercuartílico: $P_{75} - P_{25} = Q_3 - Q_1$
- **Varianza** s^2
  Mide el promedio de las desviaciones (al cuadrado) de las observaciones con respecto a la media (*variance*):
  - Es sensible a valores extremos (alejados de la media).
  - Sus unidades son el cuadrado de las de la variable.

$$s^2 = \frac{1}{n}(x_i - \bar{x})^2$$

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}$$

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n} - 1$$

- **Desviación típica** (*standard deviation*, SD, sd...)
  Es la raíz cuadrada de la varianza.
  - Tiene las misma dimensionalidad (unidades) que la variable. Versión 'estética' de la varianza.

$$s = \sqrt{s^2}$$

- **Coeficiente de variación**
  Es la razón entre la desviación típica y la media.
  - Mide la desviación típica en forma de "qué tamaño tiene con respecto a la media"
  - También se la denomina variabilidad relativa.
  - Es una cantidad adimensional. Interesante para comparar la variabilidad de diferentes variables.
  - Se expresa como porcentaje.
  - Si el peso tiene CV=30 % y la altura tiene CV=10 %, los individuos presentan más dispersión en peso que en altura.

$$CV = \frac{s}{\bar{x}} \cdot 100$$

```
# Dispresión
pesos <- rnorm( 1000, 3, 0.8 )
range( pesos )
```

```
[1] 0.14 5.92
```

```
IQR <- ( quantile( pesos, 0.75, names = F ) -
         quantile( pesos, 0.25, names = F ) )
IQR
```

```
[1] 1.08
```

```
var( pesos )
```

```
[1] 0.667
```

```
sd( pesos )
```

```
[1] 0.816
```

```
CV <- sd( pesos ) / mean( pesos )
CV
```

```
[1] 0.272
```

6.9. Medidas de forma

Las medidas de forma son la *asimetría* y la *curtosis*.

- **Asimetría**
  - Las discrepancias entre las medidas de centralización son indicación de asimetría.
  - Coeficiente de asimetría (positiva o negativa).

  - Distribución simétrica (asimetría nula).
- **Curtosis o apuntamiento** La curtosis nos indica el grado de apuntamiento (aplastamiento) de una distribución con respecto a la distribución normal o gaussiana. Es una medida adimensional.
  - Platicúrtica (aplanada): curtosis < 0
  - Mesocúrtica (como la normal): curtosis$= 0$
  - Leptocúrtica (apuntada): curtosis > 0
- **Regla aproximativa** (para ambos estadísticos)
  - Curtosis o coeficiente de asimetría entre -1 y 1, es generalmente considerada una muy ligera desviación de la normalidad.
  - Entre -2 y 2 tampoco es malo del todo, según el caso.

6.10. Error típico de la media (SEM)

Error típico de la media: Cuando estimamos la media a partir de una muestra de un determinado tamaño (n) los valores que toma la media en las diferentes muestras varía. A la desviación típica de los valores que toma el estadístico se le denomina error típico de la media(Swinscow & Campbell, 2002).

Da una idea de la variabilidad del estadístico.

> **Nota**: No de la distribución de la variable.
>
> $$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$$
> $$s_{\bar{x}} = \frac{\widehat{s}}{\sqrt{n}}$$
>
> **Nota:** Un 'error' muy frecuente de los investigadores es mostrar en los gráficos medias y errores típicos de la media en lugar de la desviación típica y, además, utilizar barras para representar los valores medios.

- Pero ¿es un error o es intencionado?
- ¿Por qué solemos mostrar gráficos de barras y errores típicos de la media (SEM)?

Martin Krzywinski y Naomi Altman hacen una interesante discusión sobre que barras de error emplear en Nature Methods *"Points of significance"* (Krzywinski & Altman, 2013).

6.11. Algunos gráficos útiles: un repaso rápido

6.11.1. Diagramas de cajas (*boxplot*)

Un diagrama de caja, según John Tukey (1977), es un gráfico, basado en cuartiles, mediante el cual se visualiza un conjunto de datos. Está compuesto por un rectángulo (*caja*) y dos brazos (*bigotes*); también reciben el nombre de *diagramas de cajas y bigotes*.

Es un gráfico que suministra información sobre los valores mínimo y máximo, los cuartiles Q1, Q2 o mediana y Q3, y sobre la existencia de valores atípicos y simetría de la distribución.

$$\begin{array}{lcl} L_i & = & Q_1 - 1{,}5 \cdot IRQ \\ L_s & = & Q_3 - 1{,}5 \cdot IRQ \\ IQR & = & Q_3 - Q_1 \end{array}$$

Los valores atípicos son los inferiores a L_i y los superiores a L_s

```
boxplot( pesos )
```

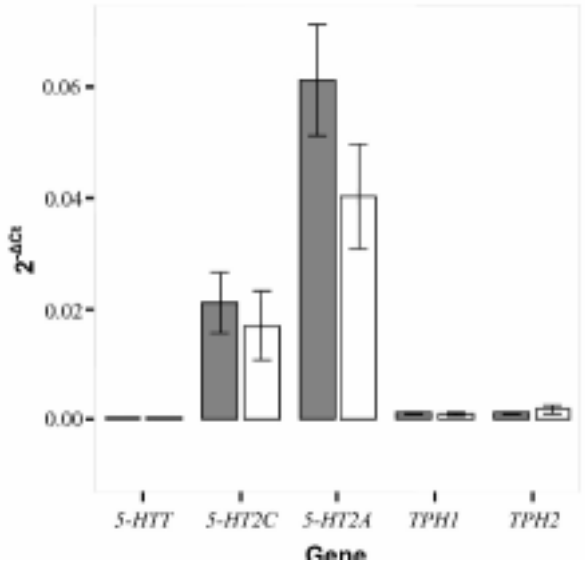


Figura 14: *Data represents the mean+SEM for the individual cells studied (N=7 for both groups). Current trace calibration bar represents 1000 pA...*

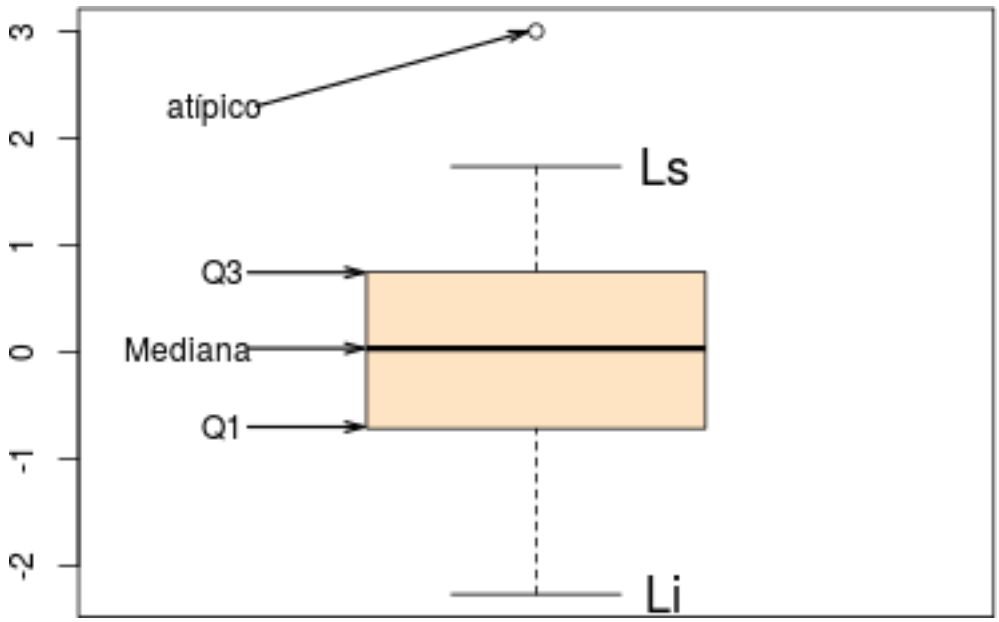


Figura 15: Elementos de un `barplot`.

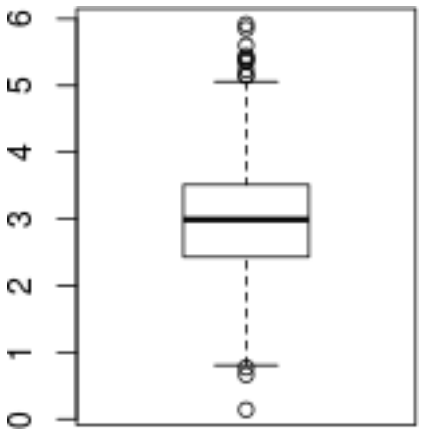


Figura 16: Descripción de la datos de `peso` mediante un `barplot`.

```
p1 <- rnorm( 1000, 3, 0.8 )
p2 <- rnorm( 1000, 2, 0.5 )
p <- c(p1, p2)
grupo <- c( rep( "M", 1000 ), rep( "H", 1000 ) )
df <- data.frame( p, grupo )
str( df )
```

```
'data.frame':   2000 obs. of  2 variables:
 $ p    : num  3.49 2.68 3.84 3.48 3.81 ...
 $ grupo: Factor w/ 2 levels "H","M": 2 2 2 2 2 2 2 2 2 2 ...
```

```
head( df )
```

```
     p grupo
1 3.49     M
2 2.68     M
3 3.84     M
4 3.48     M
5 3.81     M
6 3.49     M
```

```
par( mfrow = c( 1, 2 ), mar=c(5.1,4.1,4.1,2.1), oma=c(0,0,0,0) )
boxplot( pesos )
title( "Una variable" )
boxplot( df$p ~ df$grupo )
title( "Dos grupos de una variable" )
```

- Proporcionan una visión general de la simetría de la distribución de los datos, si la media no está en el centro del rectángulo, la distribución no es simétrica.
- Son útiles para ver la presencia de valores atípicos (*outliers*).
- Muy útiles para comparar distribuciones.

6.12. Histograma

Un histograma es una representación gráfica de una variable en forma de barras, donde la superficie de cada barra es proporcional a la frecuencia de los valores representados. Representar histogramas en R es tan sencillo como crear un objeto histograma, con la función `hist`.

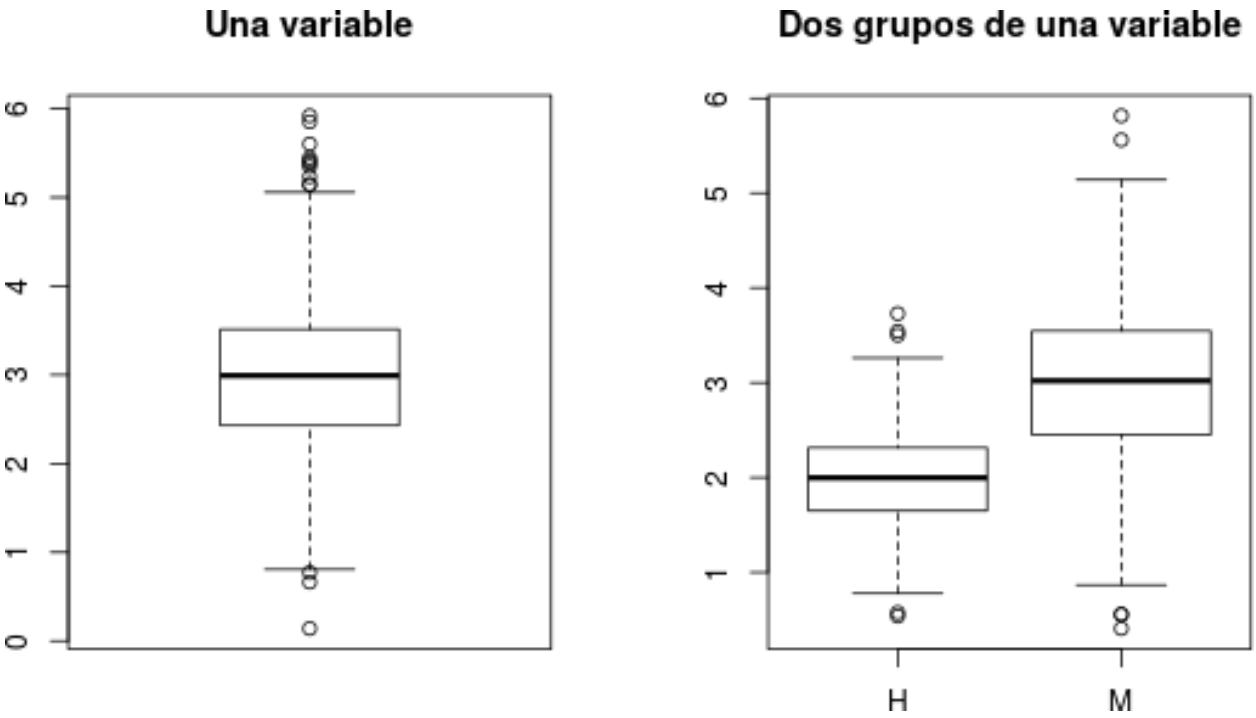


Figura 17: Uso de `boxplot` para una variable sin y con la especifiación de grupos definidos por otra variable adicional.

Prueba el siguiente código y observa qué ocurre.

```
# histogramas

head( df )
```

```
     p grupo
1 3.49     M
2 2.68     M
3 3.84     M
4 3.48     M
5 3.81     M
6 3.49     M
```

```
str( df )
```

```
'data.frame':   2000 obs. of  2 variables:
 $ p    : num  3.49 2.68 3.84 3.48 3.81 ...
 $ grupo: Factor w/ 2 levels "H","M": 2 2 2 2 2 2 2 2 2 2 ...
```

```
par( mfrow = c( 3, 2 ), mar=c(5.1,4.1,4.1,2.1), oma=c(0,0,0,0) )
hist( df$p, main = "Por defecto" )

hist( df$p, breaks = 20, main = "Fijando el número de clases (20)" )

hist( df$p, breaks = 20, probability = TRUE,
      main = "Probabilidad en lugar de frecuencia" )
rug( jitter( p ) )  # aspecto: añadimos marcas en el
                    # eje OX de obs, con un poco de ruido
```

```
 # Personalizando el histograma
hist( df$p, breaks = 20, probability = TRUE,
      main = "Histograma de los pesos " )

hist( df$p, breaks = 20, probability = TRUE,
      main = "Histograma de los pesos (coloreado)",
      col = c("red", "yellow", "purple") )


colores <- c( "red", "yellow", "purple" )
hist( df$p,
      breaks = 20,
      probability = TRUE,
      xlab = "peso en kg",
      ylab = "frecuencia",
      col = colores,
      main = "Un histograma muy personal")

# le añadimos su curva de densidad
lines( density( df$p ), lwd = 2 )

# añadimos más cosas a la curva
lines( density(df$p),
       col = "blue",
       lty = 2,
       ps = 20 )
```

Nota: Los histogramas fueron inventados por Florence Nightingale (1820-1910), Orden del Mérito del Reino Unido, británica, una de las pioneras en la práctica de la enfermería. Se la considera la madre de la enfermería moderna y creadora del primer modelo conceptual de enfermería. Destacó desde muy joven en matemática, aplicando después sus conocimientos de estadística a la epidemiología y a la estadística sanitaria. Inventó los gráficos de sectores o histogramas para exponer los resultados de sus reformas. En 1858, fue la primera mujer miembro de la *Statistical Society*. También fue miembro honorario de la *American Statistical Association*.
Durante la guerra de Secesión en 1861 fue llamada por el gobierno de la Unión para que organizara sus hospitales de campaña durante la guerra, la cual redujo del 44 % al 2.2 % los heridos que fallecian en los hospitales ("Florence nightingale - wikipedia, la enciclopedia libre," n.d.).

Un paquete muy interesante que nos permite combinar histogramas y *boxplots* es `sfsmisc` con la función `histBxp()`.

```
library( sfsmisc )
v <- rnorm( 250 )
histBxp( v,
         main = "",
         xlab = "variable v",
         probability = F,
         boxcol = 0,
         medcol = 1 )
```

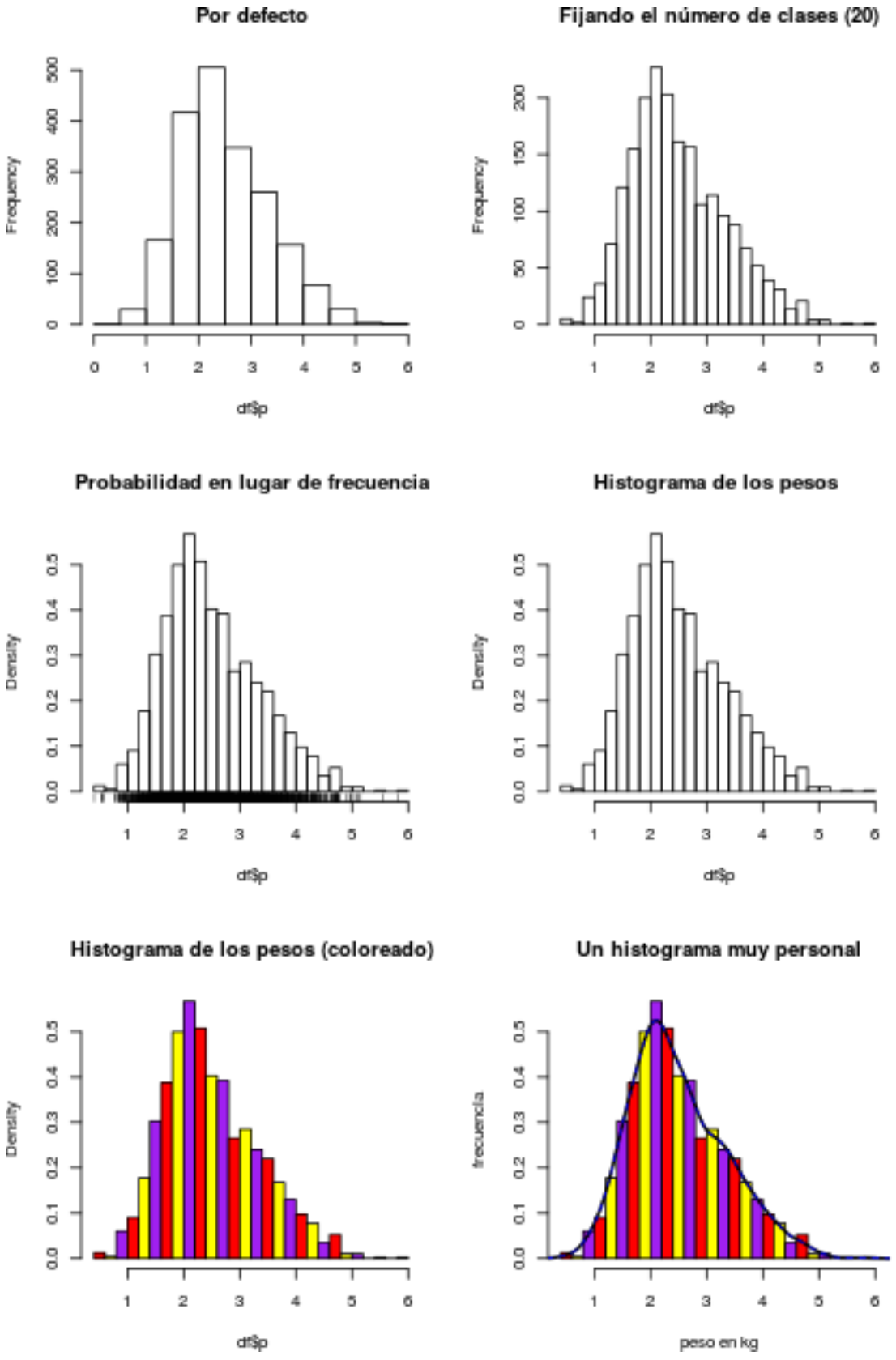
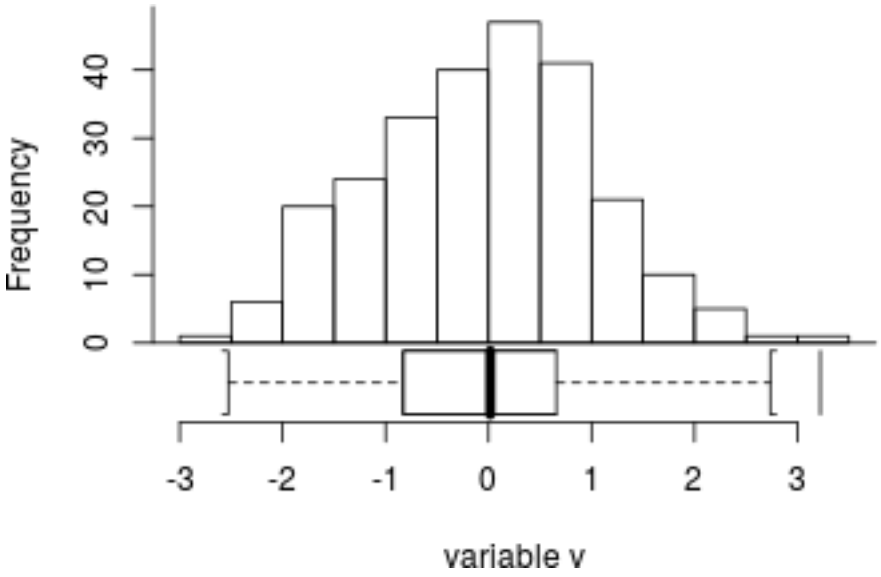


Figura 18: Distintas formas de preparar un histograma.



6.13. Algunas funciones útiles para obtener estadísticos descriptivos

6.13.1. `summary()`

Hemos presentado a lo largo del texto muchas funciones para obtener descriptivos, la más recurrida es `summary()`, pero hay muchas más y algunas no están en los paquetes por defecto. Veamos algunos ejemplos.

```
p1 <- rnorm( 1000, 3, 0.8 )
p2 <- rnorm( 1000, 2, 0.5 )
p <- c( p1, p2 )
altura <- c( rnorm( 1000, 87, 7 ), rnorm( 1000, 97, 6 ) )
# length(p); hist(p)
grupo <- c( rep( "M", 1000 ), rep( "H", 1000 ) )
df <- data.frame( p, altura, grupo )
head( df )
```

```
     p altura grupo
1 2.24   87.5     M
2 2.31   90.1     M
3 3.07  102.0     M
4 2.67   86.0     M
5 1.82   86.7     M
6 4.19   79.8     M
```

```
str( df )
```

```
'data.frame':   2000 obs. of  3 variables:
 $ p     : num  2.24 2.31 3.07 2.67 1.82 ...
 $ altura: num  87.5 90.1 102 86 86.7 ...
 $ grupo : Factor w/ 2 levels "H","M": 2 2 2 2 2 2 2 2 2 2 ...
```

```
summary( df )
```

```
       p            altura      grupo
 Min.   :0.50   Min.   : 67.0   H:1000
 1st Qu.:1.88   1st Qu.: 86.1   M:1000
 Median :2.36   Median : 92.3
```

```
 Mean   :2.49   Mean   : 92.0
 3rd Qu.:3.05   3rd Qu.: 97.9
 Max.   :5.30   Max.   :120.3
```

```
summary( df[ which( df$grupo == "M" ), ] )
```

```
       p            altura      grupo
 Min.   :0.55   Min.   : 67.0   H:   0
 1st Qu.:2.45   1st Qu.: 82.2   M:1000
 Median :3.00   Median : 86.9
 Mean   :2.99   Mean   : 86.9
 3rd Qu.:3.58   3rd Qu.: 91.5
 Max.   :5.30   Max.   :110.9
```

```
# también podemos asignar dataframes a cada grupo para simplificar la
# sintaxis
df.M <- df[ which( df$grupo == "M" ),  ]
summary( df.M )
```

```
       p            altura      grupo
 Min.   :0.55   Min.   : 67.0   H:   0
 1st Qu.:2.45   1st Qu.: 82.2   M:1000
 Median :3.00   Median : 86.9
 Mean   :2.99   Mean   : 86.9
 3rd Qu.:3.58   3rd Qu.: 91.5
 Max.   :5.30   Max.   :110.9
```

```
df.H <- df[ which( df$grupo == "H" ), ]
summary( df.H )
```

```
       p            altura      grupo
 Min.   :0.50   Min.   : 75.2   H:1000
 1st Qu.:1.65   1st Qu.: 93.3   M:   0
 Median :1.99   Median : 97.1
 Mean   :1.98   Mean   : 97.2
 3rd Qu.:2.31   3rd Qu.:101.3
 Max.   :3.45   Max.   :120.3
```

6.13.2. `stat.desc()` del paquete `pastecs`

La función `stat.desc()` del paquete `pastecs`, tiene varias opciones muy interesantes:

stat.desc(x, basic = TRUE, desc = TRUE, norm = FALSE, p=0.95)

- `basic`: nº de valores, nº de nulls, nº de missing, min, max, rango, suma...
- `desc`: mediana, media, SEM, IC(95 %), var, sd, CV,
- `norm`: (OJO: no fijado en TRUE por defecto), curtosis, asimetría, Shapiro-Wilk

```
# del paquete pastecs, install.packages('pastecs')
# la función stat.desc()
library( "pastecs" )
```

```
Attaching package: 'pastecs'
```

```
The following object is masked from 'package:sfsmisc':

    last
```

```
stat.desc( df )
```

```
                     p      altura grupo
nbr.val      2000.0000   2000.0000    NA
nbr.null        0.0000      0.0000    NA
nbr.na          0.0000      0.0000    NA
min             0.5013     67.0228    NA
max             5.3044    120.3445    NA
range           4.8031     53.3217    NA
sum          4975.0423 184051.0781    NA
median          2.3650     92.3359    NA
mean            2.4875     92.0255    NA
SE.mean         0.0188      0.1876    NA
CI.mean.0.95    0.0368      0.3678    NA
var             0.7048     70.3520    NA
std.dev         0.8395      8.3876    NA
coef.var        0.3375      0.0911    NA
```

Nota: Sin incluir la variable *'grupo'* y con la opción `norm = T`.

```
stat.desc( df[ -3 ],
           norm = TRUE )
```

```
                     p       altura
nbr.val       2.00e+03   2000.00000
nbr.null      0.00e+00      0.00000
nbr.na        0.00e+00      0.00000
min           5.01e-01     67.02279
max           5.30e+00    120.34447
range         4.80e+00     53.32169
sum           4.98e+03 184051.07808
median        2.36e+00     92.33595
mean          2.49e+00     92.02554
SE.mean       1.88e-02      0.18755
CI.mean.0.95  3.68e-02      0.36782
var           7.05e-01     70.35205
std.dev       8.40e-01      8.38761
coef.var      3.37e-01      0.09114
skewness      4.86e-01     -0.11405
skew.2SE      4.44e+00     -1.04191
kurtosis     -1.83e-01     -0.27808
kurt.2SE     -8.37e-01     -1.27082
normtest.W    9.80e-01      0.99744
normtest.p    3.04e-16      0.00239
```

```
stat.desc( df.M[ -3 ],
           basic = FALSE,
           norm = TRUE )
```

```
                   p  altura
median        3.0002 86.8914
mean          2.9923 86.8600
SE.mean       0.0255  0.2228
CI.mean.0.95  0.0500  0.4372
var           0.6503 49.6467
std.dev       0.8064  7.0460
coef.var      0.2695  0.0811
```

```
skewness     -0.0480  0.1121
skew.2SE     -0.3103  0.7245
kurtosis     -0.2857  0.1051
kurt.2SE     -0.9244  0.3399
normtest.W    0.9979  0.9980
normtest.p    0.2320  0.2762
```

6.13.3. `descrive()` de `Hmisc`

Hay más funciones que ofrecen descriptivos, por ejemplo `Hmisc` (y muchas más).

```
describe( df$p )
```

```
df$p
       n  missing distinct     Info     Mean      Gmd      .05
    2000        0     2000        1    2.488   0.9457    1.315
     .10      .25      .50      .75      .90      .95
   1.491    1.883    2.365    3.046    3.693    4.002

lowest : 0.501 0.504 0.548 0.626 0.628, highest: 5.020 5.063 5.149 5.205 5.304
```

```
head( df$p )
```

```
[1] 2.24 2.31 3.07 2.67 1.82 4.19
```

```
describe( df$altura )
```

```
df$altura
       n  missing distinct     Info     Mean      Gmd      .05
    2000        0     2000        1    92.03    9.522    77.76
     .10      .25      .50      .75      .90      .95
   80.98    86.11    92.34    97.89   102.95   105.20

lowest :  67.0  67.6  67.8  68.6  69.3, highest: 114.4 114.6 115.3 115.8 120.3
```

```
head( df$altura, 25 )
```

```
 [1]  87.5  90.1 102.0  86.0  86.7  79.8  98.5  78.9  85.5  85.4  85.1
[12]  96.5  91.0  95.5  88.1  87.0  81.0  96.1  98.7  90.4  70.6  90.3
[23]  91.1  85.1  86.6
```

6.14. Función tapply()

Con la función `tapply()` nos podemos construir fácilmente nuestras tablas de descriptivos de una forma muy elegante.

```
# tapply
tapply( df$p, df$g, mean )
```

```
   H    M
1.98 2.99
```

```
m  <- tapply( df$p, df$g, mean )
s  <- tapply( df$p, df$g, sd )
m2 <- tapply( df$p, df$g, median )
n  <- tapply( df$p, df$g, length )
cbind( media = m,
       sd = s,
```

```
      mediana = m2,
      n )
```

```
  media    sd mediana    n
H  1.98 0.500    1.99 1000
M  2.99 0.806    3.00 1000
```

No es difícil construir una función para estadísticos descriptivos *ad hoc* haciendo uso de la función `tapply()`.

```
f.descriptivos <- function ( variable, factor ) {
    df <- na.omit( data.frame(variable, factor ) )
    m  <- tapply( df$variable , df$factor, mean )
    s  <- tapply( df$variable , df$factor, sd )
    me <- tapply( df$variable, df$factor, median )
    n  <- tapply( df$variable , df$factor, length )
    cbind( media = m, sd = s, mediana = me, n )
}
```

6.15. Tablas de frecuencias y probabilidades

Esto lo veremos más extensamente cuando hablemos de contrastes no paramétricos

```
pais <- c( "ES", "ES", "ES", "US", "US" )
sexo <- c( "F", "F", "M", "F", "M" )

t <- table( pais, sexo )  # tabla de frecuencias absolutas
t
```

```
    sexo
pais F M
  ES 2 1
  US 1 1
```

```
# frec relativas
prop.table( t )  # porcentajes totales
```

```
    sexo
pais   F   M
  ES 0.4 0.2
  US 0.2 0.2
```

```
prop.table( t ) * 100
```

```
    sexo
pais  F  M
  ES 40 20
  US 20 20
```

```
# porcentajes por filas
prop.table( t, 1 )
```

```
    sexo
pais     F     M
  ES 0.667 0.333
  US 0.500 0.500
```

```
# porcentajes por columnas
prop.table( t, 2 )
```

```
    sexo
pais     F     M
  ES 0.667 0.500
  US 0.333 0.500
```

Volver al índice del curso

Referencias y bibliografía

Adler, J. (2012). *R in a nutshell* (2nd ed.). O'Reilly Media.

Carmona, F. (2007). Curso básico de r. Retrieved March 5, 2013, from http://www.ub.edu/stat/docencia/EADB/Curso%20basico%20de%20R.pdf

Florence nightingale - wikipedia, la enciclopedia libre. (n.d.). Retrieved October 22, 2014, from http://es.wikipedia.org/wiki/Florence_Nightingale

Función de densidad de probabilidad - wikipedia, la enciclopedia libre. (n.d.). Retrieved October 22, 2014, from http://es.wikipedia.org/wiki/Funci%C3%B3n_de_densidad_de_probabilidad

Gandrud, C. (2013). *Reproducible research with r and rstudio*. Chapman & Hall/CRC Press. Retrieved from http://www.crcpress.com/product/isbn/9781466572843

Kabacoff, R. (2011). *R in action* (1st ed.). Shelter Island, NY: Manning Publications.

Krzywinski, M., & Altman, N. (2013). Points of significance: Error bars, *10*(10), 921–922. Retrieved from http://dx.doi.org/10.1038/nmeth.2659

Paradis, E. (2003). *R para principiantes*. 2002, Emmanuel Paradis. Retrieved from http://cran.r-project.org/doc/contrib/rdebuts_es.pdf

Swinscow, T. D. V., & Campbell, M. (2002). *Statistics at square one the BMJ* (10th ed.). BMJ Publishing Group. Retrieved from http://www.bmj.com/about-bmj/resources-readers/publications/statistics-square-one

Xie, Y. (2013). *Dynamic documents with R and knitr*. Chapman & Hall/CRC. Retrieved from https://github.com/yihui/knitr-book/