

Introducción a R

Alberto Torres Barrán

22 de Mayo de 2018

Índice I

1. Introducción

2. Objetos

3. Estructuras de datos básicas: vector, array, list

4. Estructuras de datos derivadas: data.frame, factor

Índice II

5. Lectura de datos

6. Sentencias de control

7. Funciones

8. Gráficos

9. Paquetes

- ▶ R es un lenguaje de programación y un entorno para manipular datos, cálculo y gráficos.
- ▶ Ventajas:
 - ▶ Proyecto de GNU (*open source*), cualquier puede contribuir al desarrollo.
 - ▶ Gran cantidad de paquetes (12585 a 21 de Mayo de 2018).
 - ▶ La mayoría de nuevas tecnologías y algoritmos relacionados con estadística aparecen primero en R.
 - ▶ Documentación abundante en Internet, muchos grupos de usuarios activos.
- ▶ Desventajas:
 - ▶ Curva de aprendizaje inclinada, como la mayoría de lenguajes de programación.
 - ▶ Menor rendimiento que otros lenguajes de cálculo científico.
 - ▶ Gestión de memoria.

Comparación rendimiento

	Fortran	Python	R	Matlab	Java
fib	0.57	95.45	528.85	4258.12	0.96
parse_int	4.67	20.48	54.30	1525.88	5.43
quicksort	1.10	46.70	248.28	55.87	1.65
mandel	0.87	18.83	58.97	60.09	0.68
pi_sum	0.83	21.07	14.45	1.28	1.00
rand_mat_stat	0.99	22.29	16.88	9.82	4.01
rand_mat_mul	4.05	1.08	1.63	1.12	2.35

Tiempos de benchmark relativos a C (más pequeño es mejor, rendimiento de C = 1.0). Fuente: <http://julialang.org/>

- ▶ R es un lenguaje interpretado, por lo que no es necesario compilar el código fuente.
- ▶ El intérprete de R está disponible para los principales sistemas operativos (Windows, OSX y Linux):
<http://cran.r-project.org>.
- ▶ En este curso vamos a trabajar con el IDE RStudio
<http://www.rstudio.com>.
- ▶ RStudio proporciona un entorno similar al de Matlab.
- ▶ Documentación y manuales: <http://www.r-project.org>.

Comandos de R

- ▶ Distinguen entre mayúsculas y minúsculas.
- ▶ Se clasifican en *asignaciones* (el resultado se guarda) y *expresiones* (el resultado se imprime y se pierde).
- ▶ El operador de asignación es `<-` o `->` (no `=`, como en la mayoría de lenguajes).
- ▶ Los comandos se pueden escribir de forma interactiva en el intérprete o almacenar en un fichero de texto.
- ▶ Para obtener ayuda sobre un determinado comando se utiliza la expresión
 `> help("lm")`
o alternativamente
 `> ?lm`

Scripts de R

- ▶ Si se re-utilizan secuencias de comandos con frecuencia, es conveniente guardarlas en un fichero `script.R`.
- ▶ El fichero se puede ejecutar posteriormente en R con

```
> source("script.R")
```
- ▶ El intérprete de R ejecutará cada una de las líneas del fichero, pero sin imprimir el valor de ninguna variable en la consola.
- ▶ Para ello, son útiles las funciones `cat()` y `print()`, que imprimen por pantalla el valor de las variables pasadas como argumentos.
- ▶ Es importante destacar que el fichero `script.R` tiene que estar en el directorio de trabajo actual, de lo contrario hay que usar la ruta completa.

Objetos

- ▶ Todas las entidades que R manipula se denominan objetos (variables, funciones, etc).
- ▶ Para crear un objeto se utiliza el operador de asignación.
- ▶ Los objetos se almacenan en la memoria RAM del ordenador con un nombre específico.
- ▶ Para cambiar el valor de un objeto hay que asignarlo de nuevo.
- ▶ Listar todos los objetos en memoria
`> ls()`
- ▶ Eliminar objetos `x` y `tmp` de la memoria
`> rm(x, tmp)`
- ▶ Al cerrar la sesión se pueden almacenar todos los objetos en el fichero `.RData`.
- ▶ Al abrir una nueva sesión se cargaran los objetos del fichero `.RData` almacenado en el directorio actual (si existe).

Objetos (cont.)

- ▶ Los tipos de datos básicos son `numeric`, `complex`, `logical` y `character`.
- ▶ El modo de un objeto es el tipo básico de los elementos que contiene, se puede ver con la función `mode()`.
- ▶ Para convertir de un modo a otro se utilizan las funciones `as.character()`, `as.numeric()`, etc.
- ▶ Todos los objetos tienen una clase, que se puede ver con la función `class()`.
- ▶ Algunas funciones producirán un resultado u otro dependiendo de la clase de sus argumentos.
- ▶ Las funciones `is.character()`, `is.numeric()`, etc. devuelven verdadero o falso dependiendo si el objeto es de esa clase o no.

- ▶ Las funciones son un tipo especial de objetos.
- ▶ Dados unos argumentos o parámetros de entrada, realizan una operación sobre los mismos y devuelven un resultado.
- ▶ En ocasiones también tienen parámetros opcionales.
- ▶ En general las funciones no modifican el valor de sus argumentos.
- ▶ Los parámetros de entrada tienen que ser de una clase o clases determinadas por la propia función.
- ▶ En la ayuda se puede ver el número y clase de cada uno de los parámetros que acepta junto con información detallada acerca de la función.

- ▶ Un vector es una secuencia de datos del mismo tipo básico.
- ▶ Creación de vectores:
 1. Función `c()`: combina sus argumentos para formar un vector, intenta convertirlos al mismo tipo (si es posible).
 2. Función `vector()`: tiene dos argumentos, el `tipo` y la `longitud`.
 3. Funciones `numeric()`, `integer()`, etc.: igual que `vector()` pero tienen un único argumento, que es la longitud del vector.
- ▶ Para ver la longitud de un vector se utiliza `length()`.

- ▶ Los siguientes expresiones se pueden aplicar en vectores:
 - ▶ Operadores aritméticos: `+`, `-`, `*`, `/`, `^`, `%%` y `%/%`.
 - ▶ Funciones: `log`, `exp`, `sin`, `cos`, `tan` y `sqrt`.
 - ▶ Operadores lógicos: `&`, `!` y `|`.
- ▶ Realizan las operaciones elemento a elemento.
- ▶ Los vectores pueden ser de distintas longitudes: los vectores más cortos reciclan sus elementos hasta tener la longitud del más largo.
- ▶ Otras funciones que operan sobre vectores (no elemento a elemento): `sum()`, `prod()`, `max()`, `min()`, `range()`, `mean()`, `var()`, `sd()`, `cumsum()`, `cumprod()`, etc...

- Operador “:”. Genera números enteros en un rango, tiene máxima prioridad.

```
> 1:20
```

- Función `seq()`. Genera secuencias más complejas. Por ejemplo:

```
> seq(-0.5, 0.5, by=.2)
[1] -0.5 -0.3 -0.1  0.1  0.3  0.5
```

- Función `rep()`. Duplica el objeto. Ejemplos:

```
> rep(1, 10)
[1] 1 1 1 1 1 1 1 1 1 1
> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4
```

Selección de subvectores

- ▶ Para seleccionar partes de un vector se utilizan vectores de índices entre corchetes [y].
- ▶ Hay 4 tipos de vectores de índices:
 1. Vector lógico: se seleccionan los valores que son **TRUE**.
 2. Vector de enteros positivos: se seleccionan los elementos con esos índices.
 3. Vector de enteros negativos: se excluyen los elementos con esos índices.
 4. Vector de caracteres: solo si los elementos del vector tienen nombre, selecciona los elementos con dicho nombre.
- ▶ La variable de almacenamiento también puede ser indexada.
- ▶ Poner a 0 los elementos de **x** menores que 5:

```
> x[x < 5] <- 0
```

Ejercicio I

Ejecutar el siguiente código en R, que genera dos vectores de enteros aleatorios elegidos entre el 1 y el 1000 de tamaño 250:

```
> n <- 250  
> x <- sample(1:1000, n, replace=T)  
> y <- sample(1:1000, n, replace=T)
```

A partir de los dos vectores anteriores:

1. Calcular el el máximo y el mínimo de los vectores x e y .
2. Calcular la media de los vectores x e y . Antes de calcularla, ¿que valor esperarías?.
3. Calcula el número de elementos de x divisibles por 2 (el operador módulo es %%).

Ejercicio II

4. Ordenar los vectores, primero usando la función `order()` y luego la función `sort()`.
5. Seleccionar los valores de y menores que 600.
6. Crear las secuencias (funciones `rep` y `seq`):
 - ▶ 1 2 3 4 5 6 7 8 9 10.
 - ▶ 1 2 3 4 1 2 3 4 1 2 3 4.
 - ▶ -1 -0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.8 1.

Arrays

- ▶ Un **array** es una colección de datos del mismo tipo indexada por varios índices.
- ▶ La función **dim()** devuelve la longitud de cada una de las dimensiones del **array**.
- ▶ Los **arrays** se crean con la función **array()** y sus datos se rellenan con el primer índice variando más rápido:

```
> z <- array(x, dim=c(3,5,4))
```

Selección de subarrays I

- ▶ En general, se puede seleccionar una parte del **array** mediante una sucesión de vectores índice.

```
> z[1:2, -(1:4), 2]  
[1] 28 29
```

- ▶ Si un vector índice está vacío se selecciona todo el rango de valores.

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

Matrices

- ▶ Una `matriz` es un `array` de 2 dimensiones.
- ▶ Las matrices se crean con la función `matrix()`, y sus datos se rellenan por columnas (igual que en los arrays).
- ▶ Las funciones `nrow()`, `ncol()` devuelven el número de filas y columnas de forma respectiva.
- ▶ Además de todos los operadores de arrays, existen funciones específicas para matrices:
 - `t()` devuelve la transpuesta de una matriz.
 - `diag()` devuelve la diagonal de una matriz.
 - `crossprod()` realiza el producto cruzado entre matrices (equivalente al operador `%*%`).
 - `eigen()` calcula autovalores y autovectores.
 - `svd()` realiza la descomposición en valores singulares.

Ejercicio matrices I

1. Crear la matriz 4×5 ,

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

Pista: ver el parámetro `byrow` de la función `matrix()`.

2. Extraer los elementos $A[4, 3]$, $A[3, 4]$, $A[2, 5]$ utilizando una matriz de índices.
3. Reemplazar dichos elementos con 0.

Ejercicio matrices II

4. Crear la matriz identidad 5×5 ,

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Pista: mirar documentación de la función `diag()`.

5. Convertir la matriz A anterior en una matriz cuadrada B añadiendo al final una fila de unos (función `rbind()`):

$$B = \begin{pmatrix} A \\ \mathbf{1} \end{pmatrix}$$

6. Calcular la inversa de la matriz B con la función `solve()`.

Ejercicio matrices III

7. Multiplicar B por su inversa B^{-1} .
8. Comprobar si el resultado es exactamente la matriz identidad \mathbf{I} .
9. En caso contrario, calcular el “error” o “precisión” de la operación, definido como:

$$\text{Error} = \frac{1}{N} \sum_{i,j} |(BB^{-1} - \mathbf{I})_{(i,j)}|$$

donde N es el número de elementos de la matriz B .

- ▶ Una lista consiste en una colección ordenada de objetos del mismo o distinto tipo.
- ▶ Los componentes de la lista siempre están numerados.
- ▶ Son accesibles con el operador doble corchete `[[y]]`:

```
> l <- list(nombre="Alberto", numeros=c(3,5,6))  
> l[[1]]  
[1] "Alberto"
```

- ▶ Los componentes también pueden tener nombre. En ese caso también son accesibles de la siguiente forma:

```
> l$nombre  
[1] "Alberto"  
> l[["numeros"]]  
[1] 3 5 6
```


Operaciones sobre listas

- ▶ Se crean con la función `list()`.
- ▶ Se pueden modificar sus elementos con la sintaxis:

```
> l$nombre <- "Juan"
```
- ▶ Si el nombre del elemento no existe, se añade a la lista:

```
> l$edad <- 25
```
- ▶ También se pueden combinar con la función `c()` (similar a como se hacía con vectores).
- ▶ Importante destacar la diferencia entre `l[[1]]` y `l[1]`: el primero devuelve el componente mientras que el segundo devuelve una sublista.

- ▶ Un factor es un tipo de dato que se utiliza para codificar valores categóricos, por ejemplo: `renta = {alta, baja, media}`.
- ▶ Se crean con la función `factor`:

```
> f <- factor(c("hombre", "mujer", "mujer"))
```
- ▶ Se pueden ver los niveles (valores distintos) con la función `levels()`:

```
> levels(f)
[1] "hombre" "mujer"
```
- ▶ Función `relevel()`: reordena los niveles del factor especificado, poniendo el nivel especificado de primero

Operaciones con factores

- ▶ Función `cut()`: crea un factor dividiendo en rangos un vector numérico de acuerdo a unos puntos de corte
- ▶ Función `tapply()`: aplica una función a cada uno de los elementos de un vector, divididos en los distintos grupos de un determinado factor.
- ▶ Función `by()`: similar a la anterior, pero el objeto sobre el que se aplica la operación agrupada puede ser un `data.frame`.
- ▶ Función `aggregate()`: similar a `tapply()` pero devuelve un `data.frame` y acepta “fórmulas”.
- ▶ Funciones `table()`, `prop.table()`, `margin.table()` y `xtabs()`: crear tablas de contingencia a partir de ciertos factores

Data frames

- ▶ Un data frame es una lista donde cada uno de sus elementos se conocen como variables.
- ▶ Además, tiene las siguientes restricciones:
 - ▶ Los componentes deben de ser vectores, factores, matrices, listas u otros data frames.
 - ▶ Las matrices, listas y data frames contribuyen al nuevo data frame con tantas variables como columnas, elementos o variables tengan.
 - ▶ Los vectores no numéricos se transforman en factores.
 - ▶ Todos los vectores tienen que tener la misma longitud y las matrices el mismo número de filas.
- ▶ Resumiendo, un data frame es como una matriz donde sus columnas pueden ser de tipos diferentes.

Operaciones con data.frames I

- ▶ Por su condición de listas, se puede acceder a cada una de las variables del data frame de las siguientes formas:

```
> mtcars[[1]]  
> mtcars[["mpg"]]  
> mtcars$mpg
```

- ▶ El resultado es un vector del mismo tipo que el elemento del data frame.
- ▶ Para obtener un subconjunto de las columnas se pueden indexar numéricamente o por nombre:

```
> mtcars[1]  
> mtcars["mpg"]
```

- ▶ Al igual que en las listas, el resultado de las operaciones anteriores es otro data frame.

Operaciones con data.frames II

- ▶ Para obtener un subconjunto de las filas se pueden indexar numéricamente, por el nombre o usando un vector lógico:

```
> mtcars[1,]  
> mtcars["Camaro Z28",]  
> mtcars[mtcars$mpg > 30,]
```

- ▶ Por último, se pueden combinar las dos anteriores para acceder a un subconjunto de las filas y de las columnas:

```
> mtcars[4:8, -c(5:11)]  
> mtcars["Camaro Z28", c("mpg", "gear")]  
> mtcars[mtcars$mpg > 30, c(1,4)]
```

- ▶ El resultado de la operación es otro data frame, menos si es un único elemento ó se seleccionan todas las filas:

```
> mtcars[,1]
```

Otras operaciones con data.frames

- ▶ Añadir fila: `rbind()`
- ▶ Añadir columna (variable): `cbind()`
- ▶ Eliminar una variable
 - > `mtcars$mpg <- NULL`
- ▶ Renombrar una variable
 - > `names(mtcars)[1] <- "miles/galon"`
- ▶ Reordenar el data frame (mayor a menor):
 - > `mtcars[order(mtcars$disp, decreasing=TRUE),]`
- ▶ Detectar *missing values*
 - > `mtcars[5, "disp"] <- NA`
 - > `mtcars[complete.cases(mtcars),]`

Combinar data frames

- ▶ La función `merge()` permite combinar (unir) dos data frames
- ▶ Podemos indicar la columna o columnas por las que queremos unir cada uno de los dos data frames con los parámetros opcionales `by.x` y `by.y`
- ▶ Existen 4 tipos (terminología de SQL):
 1. *Inner join* o *natural join*
 2. *Full (outer) join*
 3. *Left (outer) join*
 4. *Right (outer) join*
- ▶ Los parámetros lógicos opcionales `all.x` y `all.y` sirven para indicar que tipo de unión queremos realizar

Ejercicio

Con el data frame `mtcars` (viene cargado en R).

1. Previsualizar el contenido con la función `head()`.
2. Mirar el número de filas y columnas con `nrow()` y `ncol()`.
3. Crear un nuevo data frame con los modelos de coche que consumen menos de 15 millas/galón.
4. Ordenar el data frame anterior por `disp`.
5. Calcular la media de las marchas (`gear`) de los modelos del data frame anterior.
6. Cambiar los nombres de las variables del data frame a `var1`, `var2`, ..., `var11`.

Pista: Mirar la documentación de la función `paste` y usarla para generar el vector (`"var1"`, `"var2"`, ..., `"varN"`) donde N es el número de variables del data frame.

Ejercicio data frames I

Con el data frame `iris` (viene cargado en R).

1. ¿Como está estructurado el data frame? (utilizar las funciones `str()` y `dim()`).
2. ¿De qué tipo es cada una de las variables del data frame?.
3. Utilizar la función `summary()` para obtener un resumen de los estadísticos de las variables.
4. Comprobar con las funciones `mean()`, `range()`, que se obtienen los mismos valores.
5. Cambia los valores de las variables `Sepal.Length` y `Sepal.Width` de las 5 primeras observaciones por NA.
6. ¿Qué pasa si usamos ahora las funciones `mean()`, `range()` con las variables `Sepal.Length` y `Sepal.Width`? ¿Tiene el mismo problema la función `summary()`?

Ejercicio data frames II

7. Ver la documentación de `mean()`, `range()`, etc. ¿Qué parámetro habría que cambiar para arreglar el problema anterior?.
8. Visto lo anterior, ¿por qué es importante codificar los *missing values* como `NA` y no como 0, por ejemplo?
9. Eliminar los valores `NA` usando `na.omit()`.
10. Calcular la media de la variable `Petal.Length` para cada uno de las distintas especies (`Species`). **Pista:** usar la función `tapply()`.

Cargar datos desde fichero

- ▶ Los datos suelen leerse desde archivos de texto externos.
- ▶ La función principal es `read.table(fichero)`, que lee el contenido de *fichero* y devuelve una variable de tipo `data.frame`.
- ▶ Existen otras como `read.csv()` o `read.delim()` pero simplemente llaman a `read.table()` con distintos parámetros por defecto.
- ▶ Igual que en los scripts, el *fichero* tiene que estar en el directorio de trabajo o en su defecto escribir la ruta completa.
- ▶ Para escribir un `data.frame` en un fichero de texto se puede usar la función `write.table()`.

Función read.table() I

Tiene los siguientes parámetros opcionales:

header Si TRUE, la primera fila es el nombre de las variables.

sep Carácter que separa las columnas. Por defecto se separan por uno o más espacios en blanco o tabuladores. Para archivos CSV, poner `sep=","` o `sep=";"`.

na.strings Vector con cadenas que se interpretan como *missing values*. Por defecto "NA". Por ejemplo: `na.strings=c("-", "-9999.0")`.

colClasses Vector de clases para las columnas. Por defecto todas las columnas se leen como texto y a continuación se convierten a valores lógicos, enteros, flotantes, números complejos o factores.

Función read.table() II

- `nrows` Número máximo de líneas a leer del fichero.
- `comment.char` Un vector con un único carácter que indica líneas a ignorar en el fichero (comentarios).
- `skip` Número de líneas a ignorar al principio del fichero.
- `dec` Carácter para separar los decimales, “.” por defecto. Útil cuando los decimales se separan con “,”.
- `fill` Si TRUE, en el caso de que las líneas tengan longitudes diferentes se añaden variables en blanco. Por defecto, el número de columnas se determina a partir de las 5 primeras líneas, y si no coincide en el resto se produce un error.

Tiempo de ejecución

- ▶ La función `system.time()` se puede usar para medir el tiempo de ejecución de una expresión de R.
- ▶ Devuelve un vector con tres valores:
 - `user` Tiempo ejecutando código de usuario.
 - `system` Tiempo realizando llamadas al sistema (por ejemplo, operaciones entrada/salida).
 - `elapsed` Tiempo real que tardó la expresión en ejecutarse.
- ▶ Generalmente, nos interesa el tercer valor.
- ▶ Para medir el tiempo de un segmento de código arbitrario, se puede utilizar `proc.time()`.
- ▶ Esta función se usa realizando dos llamadas, una al principio del segmento y otra al final, y después se restan los valores.

Ejecución condicional

- ▶ La sintaxis de la construcción condicional es:
 > `if (cond) expr_true else expr_false`
- ▶ *cond* es cualquier expresión de R cuyo resultado es un valor lógico.
- ▶ A menudo se usan los operadores `&&` (and lógico) y `||` (or lógico) para combinar varias condiciones.
- ▶ Si se utilizan con vectores de tamaño mayor que 1, solo miran el primer elemento.
- ▶ A diferencia de los anteriores, `&` y `|` operan elemento a elemento de un vector.
- ▶ La función `ifelse()` es una versión vectorizada de la construcción condicional.

- Hay tres tipos de bucles:

for Se ejecuta un número fijo de veces:

> **for** (*var in seq*) *expr*

while Se ejecuta mientras se cumple una condición:

> **while**(*cond*) *expr*

repeat Se ejecuta indefinidamente:

> **repeat** *expr*

- La instrucción **break** termina cualquiera de los tres bucles anteriores.
- La instrucción **next** pasa a la siguiente iteración.
- Una expresión puede contener a su vez un número arbitrario de expresiones, en cuyo caso se agrupan entre llaves {*expr_1*; ...; *expr_n*}

Funciones

- ▶ R permite crear objetos de tipo **function** que constituyen nuevas funciones del lenguaje.
- ▶ Para definir una función hay que hacer una asignación de la forma:

```
> nombre_fun <- function(arg1, arg2, ...) expr
```

- ▶ Típicamente, las funciones devuelven un valor con la expresión **return**(*value*).
- ▶ Si se llega al final de la función y no hubo ningún **return**, se devuelve el valor de la última expresión.
- ▶ Ejemplo:

```
> media <- function(vector) {  
  ret <- sum(vector)/length(vector)  
  return(ret)  
}
```

```
> m <- media(1:10000)
```

Argumentos con nombre. Valores predeterminados

- ▶ Se pueden pasar los argumentos a una función dado su nombre, en cuyo caso el orden es irrelevante:

```
> myfun <- function(datos, resp, impr) {}
```

- ▶ Las siguientes llamadas a `myfun` son equivalentes:

```
> myfun(X, y, TRUE)
```

```
> myfun(datos=X, resp=y, impr=TRUE)
```

```
> myfun(impr=TRUE, resp=y, datos=X)
```

- ▶ También se pueden definir valores por defecto, en cuyo caso se pueden omitir cuando se llama a la función:

```
> myfun <- function(datos, resp, impr=TRUE) {}
```

- ▶ Las siguientes llamadas también son equivalentes:

```
> myfun(X, y)
```

```
> myfun(resp=y, datos=X)
```

Funcionales: familia `apply`

- ▶ Los funcionales son funciones que toman otras funciones como argumentos.
- ▶ Un ejemplo es la función `apply` y derivadas, que se pueden usar para reemplazar algunos bucles y hacer el código más legible (y algo más eficiente):
 - ▶ `apply`: aplica una función a las filas o columnas de una matriz (o array multidimensional).
 - ▶ `lapply`: igual que `apply` pero para listas y devuelve otra lista.
 - ▶ `sapply`: como `lapply` pero devuelve un vector en lugar de una lista.
 - ▶ `tapply`: aplica una función a cada uno de los subconjuntos de un vector, y estos subconjuntos vienen definidos por un factor.

Estadística y muestreos aleatorios

- ▶ Las funciones para calcular estadísticos de una muestra son: `mean` (media), `var` (varianza), `sd` (desviación típica), `quantile` (cuantiles), `max` (máximo), `min` (mínimo), `range` (rango), `cov` (covarianza) y `cor` (correlación).
- ▶ También existe la función genérica `summary()`, que devuelve un resumen del objeto que se le pasa como primer argumento.
- ▶ Para generar muestreos aleatorios y permutaciones se utiliza la función `sample()`. Esta función también sirve para generar números enteros aleatorios.
- ▶ Otras funciones útiles son `unique()` y `table()`, que calculan el número de elementos únicos y cuantos hay de cada uno de ellos, respectivamente.

Distribuciones de probabilidad

- ▶ R contiene un amplio conjunto de tablas estadísticas.
- ▶ Para cada una de ellas, está disponible:
 - ▶ La función de densidad (pdf), añadiendo `d`.
 - ▶ La función de distribución (cdf), añadiendo `p`.
 - ▶ La función de distribución inversa, añadiendo `q`.
 - ▶ Generación de números pseudo-aleatorios, añadiendo `r`.
- ▶ Las más comunes son: `beta`, `binom`, `cauchy`, `gamma`, `lnorm`, `norm`, `pois`, `t`, y `unif`.
- ▶ Ejemplo: para la distribución normal tenemos las funciones `dnorm`, `pnorm`, `qnorm` y `rnorm`.
- ▶ Otras funciones útiles para estudiar la distribución de unos datos son `density()`, para estimar la función de densidad y `ecdf()`, para calcular la función de distribución empírica.

Ejercicio

- ▶ Escribir una función `mediaColumnas` que calcule la media de las columnas de un `data.frame` que se pasa como argumento usando un bucle y las devuelva en un vector. La función debe comprobar que el argumento es un `data.frame`.
- ▶ Generar un `data.frame` de 1,000 filas y 100,000 columnas con números aleatorios con distribución normal con media 10 y desviación 2.
- ▶ Utilizar la función anterior para calcular la media de las columnas de los datos del `data.frame`.
- ▶ Hacer la misma operación anterior pero ahora con la función `sapply`.
- ▶ Comparar el resultado y el tiempo de ejecución de los dos procedimientos anteriores.

- ▶ Los comandos gráficos en R se dividen en 2 tipos:
 - ▶ Alto nivel: funciones que crean un nuevo gráfico, con ejes, etiquetas, títulos, etc.
 - ▶ Bajo nivel: funciones que añaden elementos a un gráfico ya existente, como puntos, líneas, etc.
- ▶ Para exportar un gráfico a un fichero, por ejemplo en PDF, hay que inicializar el dispositivo gráfico con

```
> pdf("fig.pdf")
```

a continuación realizar el gráfico y por último cerrarlo con

```
> dev.off().
```
- ▶ Alternativamente, desde RStudio se puede exportar con la opción “Export” de la interfaz gráfica.

Función `plot()`

- ▶ Es genérica, es decir, el tipo de gráfico que produce depende de la clase del primer argumento.
- ▶ La forma con un argumento, `plot(x)`, produce:
 - ▶ Factor: diagrama de barras.
 - ▶ Vector numérico: gráfico de sus elementos sobre el índice.
 - ▶ Matrix: diagrama de dispersión de la segunda columna sobre la primera.
 - ▶ Data frame: diagramas de dispersión de todos los pares de variables.
- ▶ La forma con dos argumentos, `plot(x,y)`, produce:
 - ▶ Vectores numéricos: gráfico de dispersión de `y` sobre `x`.
 - ▶ `x` factor, y vector numérico: diagrama de cajas de `y` para cada factor de `x`.
- ▶ Para ver los métodos para otros tipos de objetos se usa `methods(plot)`.

Funciones de alto nivel

- ▶ Gráficos cuantil-cuantil, que sirven para comparar dos distribuciones:
 - ▶ `qqnorm(x)`, compara una muestra con la distribución normal.
 - ▶ `qqline(x)`, añade una línea de referencia.
 - ▶ `qqplot(x,y)`, compara dos muestras `x` e `y`.
- ▶ Histogramas: representa el vector numérico `x` en forma de barras, `hist(x)`.
- ▶ Gráficos 3D:
 - ▶ `image(x,y,z)`, rejilla de rectángulos con colores que se corresponden a los valores en `z` (*heatmap*).
 - ▶ `contour(x,y,z)`, curvas de nivel de `z`.
 - ▶ `persp(x,y,z)`, superficie 3D de `z` sobre el plano `x-y`.
- ▶ Diagramas de cajas y “bigotes”, `boxplot(x)`.
- ▶ Curvas de funciones, `curve(expr)`.

Argumentos de las funciones de alto nivel

<code>add</code>	Si <code>TRUE</code> , hace que la función se comporte como una de bajo nivel y se añada el gráfico actual.
<code>axes</code>	Si <code>FALSE</code> no se dibujan los ejes. Útil para dibujar tus propios ejes.
<code>log</code>	Represente ejes en escala logarítmica, valores posibles: “x”, “y” o “xy”.
<code>type</code>	Tipo del gráfico, valores posibles: “p” (puntos), “l” (líneas), “o” (puntos y líneas), etc.
<code>xlab</code>	Etiqueta del eje x.
<code>ylab</code>	Etiqueta del eje y.
<code>xlim</code>	Límite del eje x.
<code>ylim</code>	Límite del eje y.
<code>main</code>	Título del gráfico.
<code>sub</code>	Subtítulo del gráfico.

Funciones de bajo nivel

<code>points(x,y)</code>	Puntos con coordenadas (x,y).
<code>lines(x,y)</code>	Línea que pasa por todos los puntos (x,y).
<code>abline(a,b)</code>	Recta con pendiente <i>a</i> y ordenada <i>b</i> .
<code>abline(h=y)</code>	Recta horizontal en el punto <i>y</i> .
<code>abline(v=x)</code>	Recta vertical en el punto <i>x</i> .
<code>polygon(x,y)</code>	Polígono cuyos vértices son los elementos de (x,y).
<code>text(x,y,lab)</code>	Texto en las coordenadas (x,y).
<code>legend(pos,leg,...)</code>	Leyenda en la posición especificada.
<code>title(main,sub)</code>	Título principal y subtítulo.
<code>axis(lado,...)</code>	Eje en el lado indicado por argumento, de 1 a 4.

Parámetros gráficos

- ▶ Muchos aspectos de los gráficos se pueden personalizar a través de opciones.
- ▶ Estas opciones se pueden especificar de forma temporal a través de las funciones de alto nivel o hasta que se termine la sesión con la función `par()`.
- ▶ Algunas de las opciones más usadas son:
 - `pch` Carácter que se utiliza para dibujar un punto.
 - `lty` Tipo de línea.
 - `lwd` Anchura de la línea, en múltiplos de la anchura base.
 - `col` Color de los textos, líneas, texto, imágenes.
 - `font` Fuente que se utiliza para el texto.
 - `cex` Tamaño del texto y los elementos gráficos.
- ▶ Para más información sobre los valores que pueden tomar las opciones anteriores `help(par)` o [esta web](#).

Hay varias formas de crear gráficos múltiples:

1. Función `par()`, modificando las opciones:

- ▶ `mfrow=c(n, m)`, crea una matriz de $n \times m$ figuras que se va rellenando por filas.
- ▶ `mfcol=c(n, m)`, crea una matriz de $n \times m$ figuras que se va rellenando por columnas.
- ▶ `fig=c(x1, x2, y1, y2)`, coloca la figura dentro del rectángulo definido por las coordenadas (x_1, y_1) y (x_2, y_2) . Estas coordenadas tienen un valor entre 0 y 1, donde (0, 0) es la esquina superior izquierda. Esta opción crea un nuevo gráfico, por lo que es necesario combinarla con `new=TRUE`.

2. Función `layout(m)`, donde m es una matriz que contiene la localización de cada figura. Las filas y columnas de la rejilla de figuras pueden tener distinto tamaño (opciones `widths` y `heights`).

Ejercicio ficheros, muestreos y gráficos I

- ▶ Abrir el fichero `diamonds.csv` en un editor de texto (Excel, WordPad, etc.) e identificar cual es el separador de las columnas y si tiene o no cabecera.
- ▶ Cargar el fichero `diamonds.csv` en R con `read.table`, poniendo especial atención en los valores de los parámetros opcionales que definen la separación entre columnas y la cabecera.
- ▶ Ver cuantas filas (diamantes) y columnas (variables) tiene el conjunto de datos.
- ▶ Hacer un gráfico de barras con la cantidad de diamantes que hay para cada corte (variable `cut`).
- ▶ Escoger aleatoriamente 10000 diamantes y guardarlos en un nuevo `data.frame` (función `sample`).

Ejercicio ficheros, muestreos y gráficos II

La correlación mide la fuerza de una relación lineal entre dos variables. Toma valores entre 0 y ± 1 , donde 0 es poca dependencia y 1 máxima dependencia (el signo indica la dirección). En R se calcula con la función `cor`. Sabiendo lo anterior y sobre la muestra reducida de 10000 diamantes:

- ▶ Extraer como vectores los valores de las variables precio y quilate (`price` y `carat`).
- ▶ Calcular la media y mediana del precio.
- ▶ Hacer un histograma para visualizar la distribución de los precios.
- ▶ Calcular la correlación de las variables anteriores.
- ▶ Visualizar dicha correlación haciendo un gráfico de dispersión del precio sobre los quilates.

Instalar paquetes de CRAN

- ▶ Se pueden instalar nuevos paquetes con el comando
`> install.packages("nombre_paquete")`
- ▶ Para cargarlo en el entorno de R se usa el comando
`> library(nombre_paquete)`
- ▶ Alternativamente, en RStudio se puede hacer el procedimiento anterior de forma gráfica.
- ▶ Para instalar un paquete, hay ir a la pestaña “Packages” y hacer click en “Install”.
- ▶ Los paquetes instalados aparecen en la lista inferior, para cargarlos basta con seleccionarlos en el recuadro de la izquierda.

Colecciones de paquetes útiles

- ▶ En [este](#) enlace se puede ver una lista muy reciente de paquetes útiles.
- ▶ [Machine Learning in R](#) es una colección de paquetes de aprendizaje automático.
- ▶ [High-Performance computing in R](#) es una colección de paquetes de útiles para la computación de alto rendimiento.
- ▶ Recientemente aplicaciones web que permiten la visualización de resultados también gozan de gran popularidad, por ejemplo los notebooks de [Jupyter](#).
- ▶ En R destaca [Shiny](#), que permite convertir código R en aplicaciones web interactivas.

- ▶ Conjunto de paquetes creados por Hadley Wickham que comparten una misma API y contienen funciones para el análisis de datos:
 - ▶ `ggplot2`, para hacer gráficos avanzados.
 - ▶ `dplyr`, para manipular datos.
 - ▶ `tidyr`, para limpiar datos.
 - ▶ `readr`, para importar datos.
 - ▶ `purrr`, para programación funcional.
 - ▶ `tibble`, implementa *tibbles*, una versión moderna de los `data.frames`.
- ▶ El paquete *tidyverse* instala y carga los paquetes anteriores.
- ▶ También instala otros paquetes que pueden ser útiles aunque no los carga por defecto.
- ▶ Para más información y la lista completa de paquetes:
<https://github.com/tidyverse/tidyverse>.

- ▶ Paquete de alto nivel para crear gráficos estadísticos en R.
- ▶ Implementa una “gramática de gráficos”, dividiéndolos en múltiples componentes.
- ▶ Algunas de las ventajas sobre los gráficos base de R son:
 - ▶ Leyenda automática.
 - ▶ Se pueden hacer gráficos condicionados a distintos grupos en los datos de manera sencilla.
 - ▶ Es más fácil superponer diferentes elementos en un único gráfico.
- ▶ Tiene dos funciones principales, `qplot`, que se puede usar prácticamente como alternativa a `plot` y `ggplot`, para un mayor control.

Componentes de una capa

- mapping** Se definen con `aes()` (*aesthetics*) y describen como las variables se asignan a propiedades visuales.
- data** Data frame con los valores que queremos usar en el gráfico, pueden ser distintos en cada capa.
- geom** Objetos geométricos, son los que “pintan” la capa. Por ejemplo `geom_points` crea un gráfico de dispersión mientras que `geom_lines` crea un gráfico de líneas.
- stat** Transforman los datos. Suelen estar asociados a un geom en particular, por ejemplo `geom_histogram` utiliza `stat_bin` para agrupar los datos en intervalos.
- position** Pequeños ajustes en la posición de los elementos de la capa.

Construir un gráfico capa a capa

- ▶ Primero definimos el *mapping* de los datos:

```
p <- ggplot(mtcars, aes(x=mpg, y=wt, color=gear))
```
- ▶ Esto no muestra ningún gráfico, únicamente indica que queremos unos ejes cartesianos donde el eje x representa los valores de `mpg`, el eje y `wt` y vamos a usar diferentes colores según el valor de `gear`.
- ▶ Dicha representación podría ser con puntos, barras, líneas, cajas, etc.
- ▶ Para mostrar el gráfico, tenemos que añadir una capa especificando que tipo de representación o gráfico queremos, por ejemplo, `p + geom_point()`.
- ▶ Por último podemos añadir más capas, por ejemplo, `geom_smooth()`.

Facetas (*facets*)

- ▶ A veces es útil dividir los datos por una o más variables y hacer gráficas de los subconjuntos en una misma figura.
- ▶ Esto se puede hacer de formas sencilla en **ggplot2** con las capas:
 - ▶ **facet_grid()**. Divide los datos por una o dos variables, en dirección horizontal y/o vertical. Toma como parámetro una fórmula de la forma **vertical ~ horizontal**.
 - ▶ **facet_wrap()**. Divide los datos por una variable, agrupando los distintos sub-gráficos en un cierto número de columnas o filas. Toma como parámetro una fórmula de la forma **~ variable**.
- ▶ Ejemplo: ver la distribución del precio de los diamantes agrupados por color y corte:

```
> ggplot(diamonds, aes(price)) +  
+ geom_histogram() + facet_grid(color ~ cut)
```

Paquete dplyr

- ▶ Paquete muy popular para manipular grandes conjuntos de datos.
- ▶ Implementado en su mayoría en C++, por lo que ofrece mejor rendimiento que las operaciones equivalentes en R estándar.
- ▶ Utiliza la misma sintaxis sin importar donde están almacenados los datos (`data.frame`, BD, ...).
- ▶ Tiene 5 operaciones principales:
 - ▶ `filter` y `slice`: seleccionar filas por condición o posición.
 - ▶ `select`: seleccionar columnas.
 - ▶ `mutate`: crear nuevas columnas en función de las ya existentes.
 - ▶ `arrange`: ordenar por columnas.
 - ▶ `summarise`: colapsar el `data.frame` a una fila.

Operaciones agrupadas

- ▶ Similar a la función `aggregate()`, en `dplyr` tenemos la función `group_by()` para agrupar por una o más variables.
- ▶ Por si misma la función `group_by()` no realiza ninguna operación, pero es muy potente combinada con los 5 verbos anteriores.
- ▶ Después de llamar a la función `group_by()` las operaciones cambian su funcionamiento de la siguiente manera:
 - ▶ `slice`: seleccionar filas dentro de cada grupo.
 - ▶ `select`: igual que sin agrupar, pero mantiene siempre las variables por las que se agrupa.
 - ▶ `arrange`: ordena primero por las variables agrupadas.
 - ▶ `summarise`: colapsa el `data.frame` por las variables agrupadas y de acuerdo a una función, por ejemplo: `mean()`, `sum()`, `n()`, etc.

Ejercicio dplyr

Con el dataset `diamonds`:

- ▶ Filtrar los diamantes con corte “Ideal”.
- ▶ Seleccionar las columnas `carat`, `cut`, `color`, `price` y `clarity`.
- ▶ Crear una nueva columna precio/quilate.
- ▶ Agrupar los diamantes por color.
- ▶ Calcular la media del precio/quilate para cada uno de los grupos anteriores.
- ▶ Ordenar por precio/quilate de forma descendente.

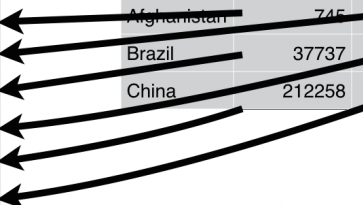
- ▶ El 80% del tiempo del análisis de datos se pasa limpiando y preparando datos (Dasu and Johnson 2003).
- ▶ Una vez cargados los datos, es conveniente estructurarlos de forma que el procesamiento posterior sea lo más sencillo posible.
- ▶ Una estructura muy común son los datos ordenados o *tidy data*.
- ▶ Hadley Wickham (2014) los define como aquellos donde:
 1. Cada variable forma una columna.
 2. Cada observación o muestra forma una fila.
 3. Cada tipo de unidad de observación forma una tabla.
- ▶ Ejemplos de datos tabulares no “ordenados”:
 - ▶ Las cabeceras contienen valores y no nombres de variables.
 - ▶ Múltiples variables están codificadas en la misma columna.

Paquete `tidyr`

- ▶ Paquete que implementa operaciones para limpieza de datos.
- ▶ En algunas funciones reemplaza al popular paquete `reshape2`, aunque está más limitado.
- ▶ Las operaciones que implementa son:
 - ▶ `gather`: convierte tablas en formato “ancho” a formato “largo”.
 - ▶ `spread`: operación contraria a `gather`.
 - ▶ `unite`: junta varias columnas en una.
 - ▶ `separate`: separa columnas.
- ▶ Ejemplos:
 - ▶ Cuando las cabeceras contienen valores y no nombres de variables (formato “ancho”), usaríamos la función `gather`.
 - ▶ Cuando varias variables están codificadas en la misma columna, usaríamos `separate`.

Ejemplo gather

country	year	cases		country	1999	2000
Afghanistan	1999	745	←	Afghanistan	745	2666
Afghanistan	2000	2666	←	Brazil	37737	80488
Brazil	1999	37737	←	China	212258	213766
Brazil	2000	80488	←			
China	1999	212258	←			
China	2000	213766	←			



Fuente

Ejemplo spread

country	year	key	value		country	year	cases	population
Afghanistan	1999	cases	745	→	Afghanistan	1999	745	19987071
Afghanistan	1999	population	19987071	→	Afghanistan	2000	2666	20595360
Afghanistan	2000	cases	2666	→	Brazil	1999	37737	172006362
Afghanistan	2000	population	20595360	→	Brazil	2000	80488	174504898
Brazil	1999	cases	37737	→	China	1999	212258	1272915272
Brazil	1999	population	172006362	→	China	2000	213766	1280428583
Brazil	2000	cases	80488	→				
Brazil	2000	population	174504898	→				
China	1999	cases	212258	→				
China	1999	population	1272915272	→				
China	2000	cases	213766	→				
China	2000	population	1280428583	→				

Fuente

- ▶ Los datos suelen leerse desde archivos de texto externos.
- ▶ Las funciones principales del paquete **readr** son:
 - ▶ `read_delim()`, para ficheros separados por un delimitador
 - ▶ `read_csv()`, para ficheros separados por comas
- ▶ Ambos tienen un parámetro que es el nombre del fichero y devuelven un tibble.
- ▶ Igual que en los scripts, el *fichero* tiene que estar en el directorio de trabajo o en su defecto escribir la ruta completa.
- ▶ Para escribir un data.frame o tibble en un fichero de texto se puede usar la función `write_csv()`.

Parámetros opcionales I

Tiene los siguientes parámetros opcionales:

- `col_names` Si TRUE, la primera fila es el nombre de las variables. También se le puede pasar un vector de cadenas de caracteres con los nombres.
- `delim` Carácter que separa las columnas (solo en `read_delim()`).
- `na` Vector con cadenas que se interpretan como *missing values*. Por defecto "NA" y la cadena vacía.
- `col_types` Vector de clases para las columnas (ver documentación de `col()`). Por defecto se intenta adivinar el tipo de cada columna a partir de las 1000 primeras líneas.

Parámetros opcionales II

`n_max` Número máximo de líneas a leer del fichero.

`skip` Número de líneas a ignorar al principio del fichero.

`locale` Objeto que nos permite cambiar el encoding, separador decimal y formato de fechas. Ver documentación de `locale()`.

`comment` Una cadena de caracteres que identifica líneas de texto a ignorar (comentarios).

`trim_ws` Si vale `TRUE`, se eliminan los espacios en blanco al principio y al final de cada campo.

- ▶ Las funciones anteriores solo son capaces de leer ficheros de texto en los formatos más comunes.
- ▶ Para otros formatos, existen paquetes específicos:
 - ▶ `haven` para ficheros SPSS, Stata, y SAS.
 - ▶ `readxl` para ficheros de Excel (tanto `.xls` como `.xlsx`).
 - ▶ `DBI`, junto con otro paquete específico dependiendo del tipo de BD (por ejemplo `RMySQL`, `RSQLite`, `RPostgreSQL`, etc.) permite ejecutar *queries* contra una base de datos, devolviendo un `data.frame`.
 - ▶ `jsonlite`, para ficheros JSON.
 - ▶ `xml2`, para ficheros XML.

Ejercicio ventas

- ▶ Cargar el conjunto de datos `ventas.csv` en R.
- ▶ Ver que columnas tiene y su tipo.
- ▶ Calcular la diferencia media en valor absoluto entre las ventas y su previsión.
- ▶ Eliminar la variable `Prevision`.
- ▶ Calcular la matriz de correlación de las `Ventas` para todos los distintos productos (identificados con su código).
- ▶ Transformar la matriz de correlación anterior en un `data.frame` que esté en formato largo. Pista: identificar que variables deberían ir en las columnas.
- ▶ Hacer un *heatmap* que represente la matriz de correlación anterior. Pista: `geom_tile`.

Libros y manuales

En general, se pueden encontrar muchos manuales en las secciones *Manuals* y *Contributed* de [CRAN](#), así como ejemplos en la web [R Pubs](#). Algunos recursos más específicos:

- Libros**
- ▶ R for Data Science [\[url\]](#).
 - ▶ An Introduction to Statistical Learning with Applications in R [\[url\]](#).

- E-Books**
- ▶ YaRrr! The Pirate's Guide to R [\[url\]](#).
 - ▶ The R Inferno [\[url\]](#).
 - ▶ R Programming [\[url\]](#).

- Blogs**
- ▶ RTutorial [\[url\]](#).
 - ▶ Quick-R [\[url\]](#).
 - ▶ RStudio [\[url\]](#).
 - ▶ RBloggers [\[url\]](#).

- ▶ [StackOverflow](#): las preguntas con el tag R contienen mucha información y problemas resueltos. Además, las nuevas preguntas se responden en cuestión de horas.
- ▶ [CrossValidated](#): no es una comunidad específica de R (más bien de estadística), pero hay mucha información acerca de cómo realizar procedimientos concretos de análisis de datos y aprendizaje automático en R.
- ▶ [@RLangTip](#): Twitter que publica consejos y trucos diarios.
- ▶ [R Programming for Data Analysis](#): Comunidad de Google+.
- ▶ [Statistics and R](#): Otra comunidad de Google+.
- ▶ [The R Project for Statistical Computing](#): Grupo de LinkedIn.