Programación y Estadística con R

Resumen

Este curso es una introducción rápida a un «entorno para la computación estadística y los gráficos», que proporciona una amplia variedad de técnicas estadísticas y gráficas: modelización lineal y no lineal, pruebas estadísticas, análisis de series temporales, clasificación, agrupación, etc. Prácticamente todos los análisis estadísticos que se realizan en Bioinformática se pueden llevar a cabo con R. Además, la «minería de datos» está bien cubierta en R: el clustering (a menudo llamado «análisis no supervisado») en muchas de sus variantes (jerárquico, k-means y familia, modelos de mezcla, fuzzy, etc), bi-clustering, clasificación y discriminación (desde el análisis discriminante a los árboles de clasificación, bagging, máquinas de vectores soporte, etc), todos tienen muchos paquetes en R. Así, tareas como la búsqueda de subgrupos homogéneos en conjuntos de genes/sujetos, la identificación de genes que muestran una expresión diferencial (con ajuste para pruebas múltiples), la construcción de algoritmos de predicción de clases para separar a los pacientes de buen y mal pronóstico en función del perfil genético, o la identificación de regiones del genoma con pérdidas/ganancias de ADN (alteraciones del número de copias) pueden llevarse a cabo en R de forma inmediata.

Sandra Mingo Ramírez

UAM - 2024/25

6 de noviembre de 2024 12:51

Universidad Autónoma de Madrid
Bioinformática y Biología Computacional

Código en Github

Índice general

I Introducción en R y estadística **3**
I.1 RStudio y primeras nociones . . . 3
I.2 Ejemplo . . . 4
I.2.1 Introducción al test de la t . . . 4
I.2.2 Problema de las pruebas múltiples . . . 5
I.3 La consola de R para cálculos interactivos . . . 8
I.3.1 Nombrar variables . . . 10
I.3.2 Obtener ayuda . . . 11
I.3.3 Mensajes de error . . . 12
I.3.4 Estilo del código . . . 13
I.4 Leer datos en R y guardarlos desde R . . . 13
I.4.1 Localización de ficheros . . . 14
I.4.2 Missing values - NA . . . 14
I.4.3 Guardar tablas, datos y resultados . . . 15
I.4.4 Guardar una sesión en R: .RData . . . 15
I.5 Scripts . . . 16
I.5.1 Utilizar un script . . . 16
I.6 Estructuras de datos básicas en R . . . 17
I.6.1 Vectores . . . 17
I.6.2 Crear vectores a partir de otros vectores . . . 18
I.6.3 Logical operations . . . 19
I.6.4 Nombres de elementos . . . 21
I.6.5 Acceder y modificar elementos de un vector: indexación y subsetting . . . 22
I.6.6 Interludio: comparación de floats . . . 25
I.6.7 Factores . . . 26
I.6.8 Matrices . . . 29
I.6.9 Listas . . . 33
I.6.10 Dataframes . . . 35
I.7 Números aleatorios y semillas . . . 37
I.8 Plots (gráficos) . . . 37
I.8.1 Lo más básico . . . 37
I.8.2 Personalización de plots: colores, tipos de línea y de puntos . . . 38
I.8.3 Guardar plots . . . 42
I.8.4 Tipos de gráficos . . . 42
I.9 Tablas . . . 45
I.9.1 Más de dos dimensiones y `ftable` . . . 45
I.9.2 Recuperar una tabla de un dataframe . . . 47
I.10 La familia `apply` . . . 47

I.10.1 apply . . . 48
I.10.2 lapply . . . 48
I.10.3 tapply y by . . . 49
I.10.4 aggregate . . . 51
I.10.5 split . . . 53
I.10.6 apply y dejar caer dimensiones en matrices . . . 56
I.10.7 Algunas apreciaciones . . . 56
I.11 Programación en R . . . 57
I.11.1 Flow control . . . 57
I.11.2 Definir funciones . . . 58
I.11.3 Orden de los argumentos, argumentos con y sin nombre . . . 59
I.11.4 Scoping, frames y entornos . . . 60
I.11.5 Los 61
I.11.6 local . . . 63
I.11.7 Evaluación vaga . . . 63
I.12 Debugging y capturar excepciones . . . 63
I.12.1 `traceback` . . . 64
I.12.2 `debug` and `browser` . . . 64
I.12.3 `trace` para ver funciones arbitrarias en sitios arbitrarios . . . 65
I.12.4 Warnings . . . 66
I.12.5 `where` para cuando uno está perdido en dónde está . . . 66
I.12.6 Protección frente a posibles fallos . . . 66
I.12.7 Funciones de debugging que no son exportadas . . . 67
I.13 Programación orientada a objetos: clases S3 y S4 . . . 67
I.13.1 `methods` . . . 67
I.13.2 Creación de clases y métodos . . . 68
I.13.3 Testeo y test-driven development . . . 70
I.13.4 Creación de función de plot . . . 72
I.13.5 Clases S4 . . . 72
I.13.6 Resumen sobre la programación orientada a objetos en R . . . 75

Capítulo I

Introducción en R y estadística

I.1. RStudio y primeras nociones

En RStudio, se puede crear un nuevo fichero en File > New File > R script. Se abre un nuevo fichero en el que se puede programar. En R, la asignación de variables se realiza con <-. En la parte superior derecha, se pueden ver todas las variables que se han asignado en la sesión, los datos y las funciones.

```
x <- 9
y <- matrix(1:20, ncol = 4)
```

En la parte inferior derecha hay una pestaña para poder visualizar los gráficos. Desde ese menú, se puede guardar, pero esto no es recomendable, ya que el gráfico se ajusta al tamaño de la pantalla y luego eso no es reproducible. En otra pestaña aparece un listado de todos los paquetes instalados en el disco duro, aunque luego haya que cargarlos en cada script en el que se desee usar. Al pulsar en el nombre de un paquete, se va a la página de ayuda del mismo. También es posible acceder con:

```
help(rnorm)
```

La mayor parte del trabajo «real» con R requerirá la instalación de paquetes. Los paquetes proporcionan funcionalidad adicional. Los paquetes están disponibles en muchas fuentes diferentes, pero posiblemente las principales ahora son CRAN y BioConductor. Si un paquete está disponible en CRAN, puedes hacer lo siguiente:

```
install.packages("nombre-paquete") # 1 paquete
install.packages(c("paquete1", "paquete2")) # varios paquetes
```

En Bioinformática, BioConductor es una fuente bien conocida de muchos paquetes diferentes. Los paquetes de BioConductor pueden instalarse de varias maneras, y existe una herramienta semiautomatizada que permite instalar conjuntos de paquetes BioC. Implican hacer algo como

```
BiocManager::install("nombre-paquete")
```

A veces los paquetes dependen de otros paquetes. Si este es el caso, por defecto, los mecanismos anteriores también instalarán las dependencias. Con algunas interfaces gráficas de usuario (en algunos sistemas operativos) también puede instalar paquetes desde una entrada de menú. Por ejemplo, en Windows, hay una entrada en la barra de menú llamada Paquetes, que permite instalar desde Internet, cambiar los repositorios, instalar desde archivos zip locales, etc. Del mismo modo, desde RStudio hay una entrada para instalar paquetes (en «Herramientas»). Los paquetes también están disponibles desde otros lugares (RForge, github, etc); a menudo encontrarás instrucciones allí.

Siempre puedes simplemente matar RStudio; pero eso no es agradable. En todos los sistemas escribir q() en el símbolo del sistema debería detener R/RStudio. También habrá entradas de menú (por ejemplo, «Salir de RStudio» en «Archivo», etc). A continuación sale la pregunta de si se debe guardar el workspace, y en general querremos decir que no.

I.2. Ejemplo

I.2.1. Introducción al test de la t

En un test de la t, la hipótesis nula (H_0) suele representar lo contrario de lo que se desea demostrar. Por ejemplo, si nuestro objetivo es comprobar si hay diferencias entre dos muestras, la hipótesis nula establece que ambas son iguales. A continuación, se utiliza la fórmula de la t para obtener un valor estadístico, cuya distribución se examina bajo la suposición de que H_0 es cierta. Luego, se calcula la probabilidad de observar un resultado tan extremo o más extremo que el obtenido bajo H_0. Esta probabilidad se denomina p-valor, y su interpretación indica cuánta evidencia hay en contra de H_0: un p-valor bajo sugiere que lo observado es improbable bajo H_0.

$$t = \frac{x_A - x_B}{SD_{x_A, x_B}}$$

Es importante aclarar que el p-valor no representa la probabilidad de que H_0 sea cierta, ni la probabilidad de que H_0 o la hipótesis alternativa (H_1) se cumplan dado los datos. Lo que el p-valor señala es que, o bien H_0 es falsa, o ha ocurrido un evento tan improbable como el valor observado. No se "rechaza" H_0 de manera concluyente, sino que simplemente no se acepta si el p-valor es suficientemente bajo. En este análisis, se compara el resultado observado con todos aquellos más extremos, algo que es distinto de seleccionar el valor que hace los datos lo más probables posible (como se hace en la máxima verosimilitud).

Por ejemplo, una moneda perfectamente equilibrada tiene una probabilidad de 0.5^6 de que al lanzarla seis veces, salga exactamente tres veces cara y tres veces cruz. Aunque este número es pequeño, no implica que la hipótesis alternativa sea necesariamente más probable, ya que otros resultados también podrían ser igualmente o más improbables.

En la mayoría de los casos de comparación de medias, los datos no están restringidos a un único valor.

Cuando H_0 es cierta:

$$Pr(p - valor \leq 0,05) = 0,05$$

$$Pr(p - valor \leq 0,01) = 0,01$$

En muchos casos se comprueba más de una H_0. En un screening, se analizan 20.000 genes y se decide elegir todos aquellos que tengan un p-valor inferior a 0,05. Esa lista, sobre el total de los genes, la probabilidad de rechazar H_0 cuando es cierta, es muy superior al 5 %, aunque se cumpla para cada gen individual. Así, se debe trasladar la lógica al test múltiple, puesto que si no se va a rechazar H_0 en muchas ocasiones cuando no se debería.

I.2.2. Problema de las pruebas múltiples

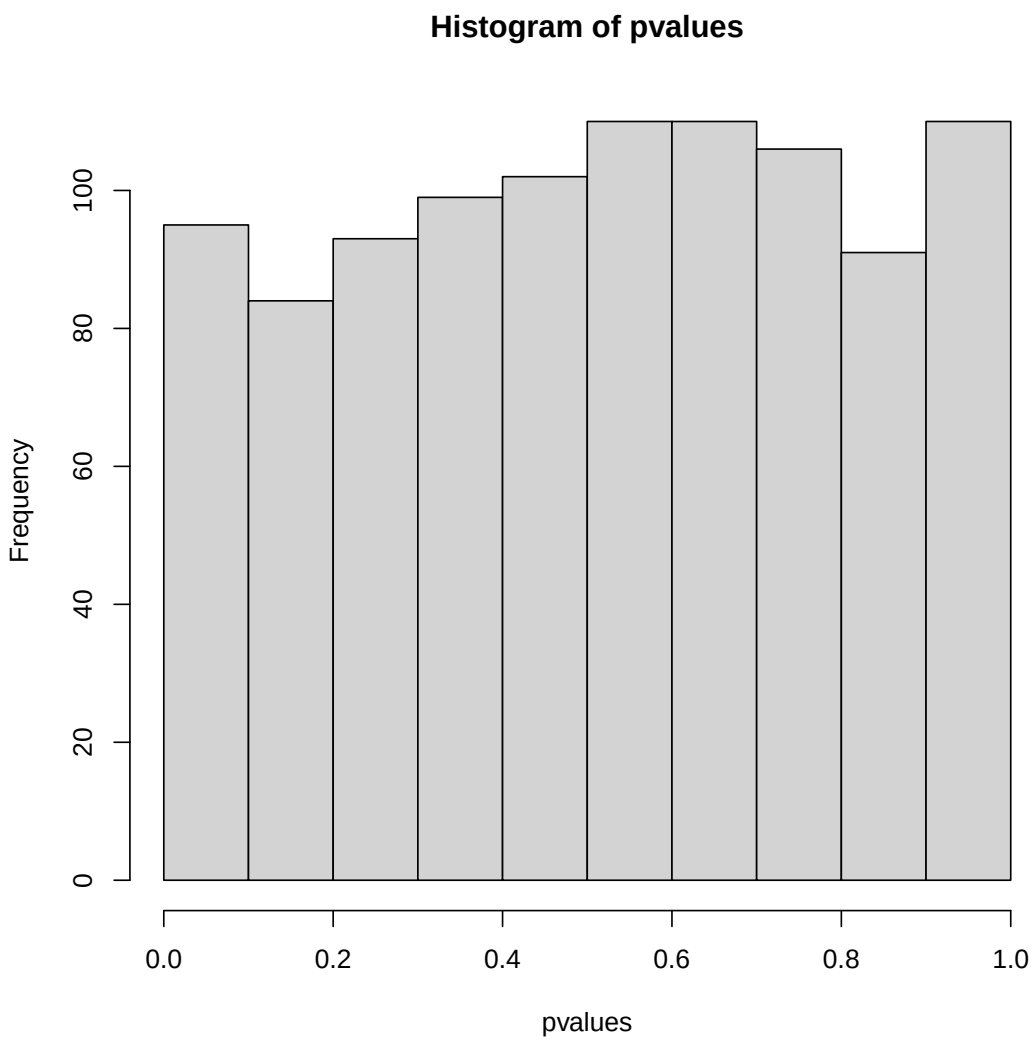
Es posible que hayamos oído hablar del problema de las pruebas múltiples con los microarrays: si observamos los p-valores de un gran número de pruebas, podemos ser inducidos a pensar erróneamente que está ocurriendo algo (es decir, que hay genes expresados de forma diferencial) cuando, en realidad, no hay absolutamente ninguna señal en los datos. A nosotros esto nos convence. Pero tienes un colega testarudo que no lo está. Ha decidido utilizar un ejemplo numérico sencillo para mostrarle el problema. Este es el escenario ficticio: 50 sujetos, de los cuales 30 tienen cáncer y 20 no. Medimos 1000 genes, pero ninguno de los genes tiene diferencias reales entre los dos grupos; para simplificar, todos los genes tienen la misma distribución (una distribución normal). Haremos una prueba t por gen, mostrará un histograma de los valores p e informaremos del número de genes «significativos» (genes con p < 0,05). Este es el código R:

```
randomdata <- matrix(rnorm(50 * 1000), ncol = 50)
class <- factor(c(rep("NC", 20), rep("cancer", 30)))
pvalues <- apply(randomdata, 1,
                 function(x) t.test(x ~ class)$p.value)
```

Para leer el código, se empieza por la función más interna, que en este caso es `rnorm`. Así, primero se generan 50.000 entradas de distribución normal (1000 genes por 50 personas) de los que se quiere realizar 1000 contrastes de hipótesis (uno por gen) y representar el aspecto de la distribución (que será uniforme). Todas las entradas se organizan en una matriz con 50 columnas. Después, se crean los dos grupos que se están analizando mediante repeticiones (función `rep`). El comando de `factor` crea las etiquetas. En R, se puede llamar al test de la t de varias maneras, siendo una estándar con la interfaz de tipo fórmula (x ~ class), dividiendo así x en los distintos niveles que se han creado previamente. La sintaxis siempre es una variable que va cambiando (en este caso, las filas) antes de la virgulilla y una variable constante después de la virgulilla (los distintos niveles). La función `apply` permite aplicar una función a un objeto o conjunto de datos, evitando así tener que realizar un bucle for. El primer

argumento es el objeto, el segundo la dimensión del objeto a lo que se quiere aplicar (si se recorren filas, columnas, etc.), y el tercero la función que se va a aplicar. La función `t.test` devuelve objetos a los que se puede acceder, como el valor `t`, df, p-valor, la media de cada grupo, etc. Se puede acceder al nombre de todos los valores mediante `names(t.test(x ~ class))`. En nuestro caso, `x` es el valor que irá adquiriendo el número de filas a recorrer. En este caso, se define la función en el momento de llamarla, pero también se puede definir antes y utilizarla en el apply. En este caso se define dentro porque es una función corta que solo se utilizará en ese momento, por lo que no es necesario crearla fuera. Si por el contrario fuese una función a la que quisiéramos acceder posteriormente o que fuese compleja con varias líneas, se suele crear fuera. Por último, se accede a los p-valores y se guardan en la variable `pvalues`. Esos p-valores se pueden representar a continuación en un histograma y calcular todos aquellos que sean menores o iguales que 0,05.

```
hist(pvalues)
```



```
sum(pvalues <= 0.05)

## [1] 55
```

Al realizar la suma de una lógica booleana, se coercia para que los valores falsos se conviertan en 0 y los verdaderos en 1. Así, al sumarlos, el resultado es numérico.

En resumen, en este ejemplo hemos visto los siguientes objetos:

- Vectores: colección de uno o más datos del mismo tipo.
- Matrices: conjunto de datos indexados por filas y columnas del mismo tipo.
- Arrays: generalización de una matriz que no tiene límite de dimensiones (pero debe tener una estructura rectangular).
- Data frames: estructura rectangular de dos dimensiones (filas y columnas) en la que cada columna puede ser de un tipo diferente.
- Listas: cajón desastre en el que se pueden meter muchas cosas de muchos tipos distintos. Muchas funciones devuelven listas u objetos que contienen listas.
- Factores: vectores de un tipo especial (variable categórica).
- Funciones: objetos que realizan una operación y devuelven algo.

En el siguiente código se muestran las distintas maneras de acceder a una matriz. La indexación funciona [filas, columnas], y si un campo está sin rellenar implica todos sus datos.

```
randomdata[, 1]
randomdata[2, ]
randomdata[, 2]
randomdata[2, 3]
```

Al ejecutar la variable `class` creada anteriormente, no solo devuelve la lista de los elementos con las distintas etiquetas, si no que también muestra al final los distintos niveles. Como `factor` por detrás les asignó un valor entero que corresponda a la etiqueta dada, cuando se pide convertir en numérico, se devuelve el entero. La asignación de los valores se realiza por orden alfanumérico.

```
class
as.numeric(class)
```

```
pvalues[1]

t.test(randomdata[1, ] ~ class)
```

```r
t.test(randomdata[1, ] ~ class)$p.value

pvalues[1:10] < 0.05

sum(c(TRUE, TRUE, FALSE))

hist(c(1, 2, 7, 7, 7, 8, 8))
```

```r
## For ease
rd2 <- randomdata[1:10, ]

## Where we will store results
pv2 <- vector(length = 10)

for(i in 1:10) {
    pv2[i] <- t.test(rd2[i, ] ~ class)$p.value
}

pv2

## Compare with
pvalues[1:10]
```

Ahora usamos apply. No lo hemos dicho explícitamente, pero cuando usamos apply estamos pasando una función (nuestra función anónima) a otra función. Esto es algo muy común y fácil en R: pasar funciones a otras funciones.

```r
apply(rd2, 1, function(z) t.test(z ~ class)$p.value)
```

Esta es otra forma de hacerlo, pero es más verbosa (quizás incluso innecesariamente verbosa):

```r
myfunction <- function(y, classfactor = class) {
    t.test(y ~ classfactor)$p.value
}

apply(rd2, 1, myfunction)
```

I.3. La consola de R para cálculos interactivos

Independientemente de cómo interactuemos con R, una vez que iniciemos una sesión interactiva de R, siempre habrá una consola, que es donde podemos introducir comandos para que sean ejecutados por R. En RStudio, por ejemplo, la consola suele estar situada en la parte inferior izquierda.Todos los prompts en la consola empiezan con >.

```
1 + 2
```

```
## [1] 3
```

Mira la salida. En este documento, los trozos de código, si muestran salida, mostrarán la salida precedida por ##. En R (como en Python), # es el carácter de comentario. En la consola, NO veremos el ## precediendo a la salida. Esto es sólo la forma en que está formateado en este documento (al igual que no se ve el > antes del comando). Fíjate también en que ves un [1], antes del 3. Esto se debe a que la salida de esa operación es, en realidad, un vector de longitud 1, y R está mostrando su índice. Aquí no ayuda mucho, pero lo haría si imprimiéramos 40 números:

```
1:40
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
## [21] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Se puede asignar 1 + 2 a una variable mediante <-. También se puede utilizar =, pero no se aconseja. Esto se debe a que se suele utilizar = cuando se pasan argumentos a una función, y utilizar la flecha permite diferenciar a simplevista las asignaciones. Para ver el valor de una variable, se puede escribir simplemente el nombre de la variable, utilizar print o hacer la asignación entre paréntesis (eso realiza la asignación y muestra el resultado por pantalla).

```
(v1 <- 1 + 2)
```

```
## [1] 3
```

```
print(v1)
```

```
## [1] 3
```

```
v1
```

```
## [1] 3
```

Se pueden separar dos comandos con un punto y coma (;), pero utilizarlo no suele ser una buena idea, solo en casos muy concretos.

```
v1 <- 1 + 2; v1
```

```
## [1] 3
```

Es posible dividir comandos en varias líneas si R puede entender que la expresión no se ha terminado:

```
v2 <-  4 - ( 3 * [Enter]
2)
```

Cuando se hace esto, se ve un + que indica que la línea se continúa y que R sigue esperando más input. No obstante, hay ocasiones en las que esto puede ser confuso, y se puede cancelar mediante Ctrl + c en Linux o pulsando Escape para abortar la operación.

Los paréntesis se ponen cuando el usuario opine que es apropiado y que facilite el entendimiento de una expresión. R utiliza las normas de precedencia usuales, pero en caso de duda, se pueden utilizar paréntesis.

```
v11 <- 3 * ( 5 + sqrt(13) - 3^(1/(4 + 1)))
```

I.3.1. Nombrar variables

Anteriormente hemos creado las variables v1 y v2. Los nombres de las variables deben comenzar con una letra. También pueden empezar por un punto, pero entonces estarán ocultas. A continuación se pueden mezclar letras, números, puntos y barras bajas. Los nombres de las variables son case-sensitive, es decir, se diferencia entre las mayúsculas y minúsculas (v1 es diferente a V1). Una vez que se ha creado una variable, se puede utilizar la variable en lugar del contenido:

```
v3 <- 5
(v4 <- v1 + v3)

## [1] 8

(v5 <- v1 * v3)

## [1] 15

(v6 <- v1 / v3)

## [1] 0.6
```

Las asignaciones posteriores sobreescriben las asignaciones previas.

```
(z2 <- 33)

## [1] 33

z2 <- 999
z2
```

```
## [1] 999

z2 <- "Now z2 is a sentence"
z2

## [1] "Now z2 is a sentence"
```

Se puede borrar una variable de la siguiente forma:

```
rm(z2)
```

I.3.2. Obtener ayuda

Se puede acceder a la página de ayuda mediante:

```
help(mean)
```

También se puede utilizar la siguiente sintaxis:

```
?mean
```

Hay otras formas de buscar ayuda sobre cómo hacer algo con R. Se puede buscar en Google, utilizar StackOverflow, etc. También hay un paquete `sos` que ayuda a buscar funciones y demás en paquetes que no están instalados, hacer un ranking de resultados de búsqueda, etc. A su vez, RStudio incluye un navegador de ayuda integrado. Todas las ayudas cuentan con una descripción de la función, los argumentos que admiten (y su orden en caso de pasarlos sin nombre; en general es mejor añadir el nombre de cada parámetro a la hora de pasarlo) y el valor, es decir, lo que devuelve. En algunos casos se especifican las fuentes y referencias. También hay una sección de ejemplos de uso de la función.

Lo visto anteriormente proporciona información de funciones concretas. No obstante, hay veces que no sabemos exactamente cómo se llama la función que buscamos. Para ello, se puede utilizar las siguientes formas:

```
apropos("normal")

## [1] "normal_print"  "normalizePath"

# help.search("normal")
```

El comando `apropos` busca todos los paquetes que contengan en el nombre lo que se esté buscando. Por el contrario, `help.search` busca todos aquellos paquetes que, en la página de ayuda, tengan lo que se esté buscando.

La función `args` devuelve los argumentos que se le puede pasar a una función.

```r
args(rnorm)

## function (n, mean = 0, sd = 1)
## NULL
```

I.3.3. Mensajes de error

Los mensajes de error pueden ser un poco crípticos, pero en muchos casos leerlos ayuda a entender qué está pasando y cómo solucionar el problema. La mejor forma de parsear el mensaje de error es ir a la última línea que se ha ejecutado e ir ascendiendo para ver dónde puede estar el problema. A continuación se muestran algunos ejemplos de mensajes de errores:

```r
apply(something, 1, mean)

## Error: objeto 'something' no encontrado

apply(v3, 1, mean) # en la ayuda se especifica qué es X

## Error in apply(v3, 1, mean): dim(X) debe tener una longitud positiva

apply(F, 1, mean)

## Error in apply(F, 1, mean): dim(X) debe tener una longitud positiva

log("23")

## Error in log("23"): Argumento no numérico para una función matemática

rnorm("a")

## Warning in rnorm("a"): NAs introducidos por coerción
## Error in rnorm("a"): invalid arguments

lug(23) # debería ser log

## Error in lug(23): no se pudo encontrar la función "lug"

rnorm(23, 1, 1, 1, 34)

## Error in rnorm(23, 1, 1, 1, 34): los argumentos no fueron usados
(1, 34)

x <- 1:10
y <- 11:21
plot(x, y)
```

```
## Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and 'y' lengths
differ

lm(y ~ x)

## Error in model.frame.default(formula = y ~ x, drop.unused.levels
= TRUE): las longitudes variables difieren (encontradas para 'x')

z <- 1:10
t.test(x ~ z)

## Error in t.test.formula(x ~ z): grouping factor must have exactly
2 levels
```

En la consola, poniendo el nombre de la función, se puede acceder al código que realiza la función por detrás. Esto puede ser útil cuando la página de ayuda no sea suficiente para intentar localizar lo que intenta hacer la función y por qué falla.

I.3.4. Estilo del código

Aunque el código se escriba para la máquina, también debe ser legible por humanos, tanto uno mismo del futuro como otras personas. Por tanto, se recomienda no extenderse más allá de la columna 80 y utilizar espacios. Hay muchas guías de estilo de código, pero esas dos normas son las más básicas: si una línea de código es excesivamente larga, cuesta leerla entera al no poder verla completa a simple vista y tener que scrollear.

Existe un paquete llamado `lintr` que permite corregir el estilo del código.

Los comentarios también forman parte del estilo de código. Se suele separar la documentación para el usuario de la función (documentación de cabecera) de la documentación dentro del código que explica por qué se hacen algunas cosas.

I.4. Leer datos en R y guardarlos desde R

Hay muchas formas de cargar datos en R. Un ejemplo es `read.table` que sirve para todo tipo de datos, pero también hay algunos comandos más concretos como `read_csv`.

```
X <- read.table("data/hit-table-500-text.txt")
head(X)
## We could save what we care about in variables with better names
align.length <- X[, 5]
score <- X[, 13]
summary(X)
```

El objeto no es una matriz, si no un data frame. Otro ejemplo sería el siguiente:

```r
another.data.set <- read.table("data/AnotherDataSet.txt", header = TRUE)
summary(another.data.set)
```

```
##       ID                  Age           Sex
##  Length:5           Min.   :12.0   Length:5
##  Class :character   1st Qu.:13.0   Class :character
##  Mode  :character   Median :14.0   Mode  :character
##                     Mean   :14.8
##                     3rd Qu.:16.0
##                     Max.   :19.0
##        Y
##  Min.   :22.00
##  1st Qu.:23.40
##  Median :24.30
##  Mean   :24.14
##  3rd Qu.:25.00
##  Max.   :26.00
```

Si se pone que no hay cabecera, parece que se lee lo mismo, pero en realidad hay algunas diferencias. Cuando se especifica que hay una cabecera, la primera línea con la descripción de las columnas no está numerada, mientras que cuando no se especifica, sí se numera y se considera como la primera fila, y esto es un error. R, por defecto, pone que cabecera es falso. Cuando no se sabe si un documento tiene o no cabecera, primero se carga el documento y luego se comprueba si el contenido se ha cargado bien. Por defecto, las columnas están separadas por espacios o tabuladores.

I.4.1. Localización de ficheros

Para que R pueda leer los ficheros, debe saber dónde buscarlos. Si los ficheros se encuentran en el directorio de trabajo, no hay ningún problema, ya que R los encuentra directamente. Para conocer el directorio de trabajo, se utiliza el comando `getwd()`. Si el fichero no se encuentra en el directorio de trabajo, hay varias opciones: proporcionar el path completo o mover el directorio de trabajo al lugar donde se encuentran los ficheros mediante `setwd()`. Para esto, es recomendable evitar en el nombre de directorios espacios, acentos y otros caracteres no ASCII.

I.4.2. Missing values - NA

Los missing values son algo muy común en estadística. Lo más sencillo es llamarlos como NA de not available. Otra forma es NaN, not a number.

Puedes especificar el carácter que R debe interpretar como valor omitido, pero los dos procedimientos estándares son sustituir el valor como NA o sustituirlo por nada. Cuando haces cualquiera de los dos, en los datos que se leen deberías ver un NA. Lo mejor es, como de costumbre, ser explícito: utilizar un NA en sus datos originales, o utilizar alguna otra cadena de caracteres especiales para identificarlos. Lo más probable

es que desees utilizar NA (o utilizar alguna otra combinación de caracteres y ser explícito), especialmente para las variables de carácter.

Por defecto, R considera cualquier secuencia de blancos y tabuladores como separadores. Por tanto, si un missing value se representa con un espacio, sería necesario especificar el separador (por ejemplo, `sep = ""`) para que no dé error (al considerar R el espacio como parte del separador).

Al utilizar `summary`, en las columnas que sean de tipo int aparece un contador con las filas que contienen un NA. Sin embargo, esto no es así en las columnas cuyo contenido sea texto. Por tanto, no nos podemos fiar si summary no nos dice que no hay, hay que comprobar que efectivamente no haya.

I.4.3. Guardar tablas, datos y resultados

Es posible guardar los datos en uan matriz o de forma tabular con `write.table`:

```
write.table(X, file = "datos_guardados.txt")
```

El problema que tiene esto es que en el documento de salida tiene una columna adicional que indica el número de línea, y se emplean los espacios como separadores. Todo esto se puede especificar mediante argumentos concretos en la función:

```
write.table(X, file = "datos_guardados.txt", sep = "\t",
            quote = FALSE, row.names = FALSE)
```

En algunos casos, puede que los nombres de las filas sean importantes (por ejemplo, que sean el identificador). En ese caso, sería interesante guardar los nombres de las filas como columna en el dataframe:

```
X$columna_nueva <- rownames(X)
```

I.4.4. Guardar una sesión en R: .RData

R permite guardar una imagen de la sesión actual en un fichero de extensión .RData. Esto se realiza mediante la función `save.image`:

```
save.image(file = "this.RData")
getwd() #donde se guarda
```

Esta función guarda el entorno global, es decir, lo que se haya añadido por el usuario: variables, ficheros (incluso los ocultos), funciones, pero no los paquetes. También se guarda el estado del generador de números aleatorios si se ha utilizado. También existe la posibilidad de guardar un objeto concreto. Esto se logra mediante `save(datos-a-guardar, file = "datos-guardados.RData")`.

En una nueva terminal de R, se pueden cargar las imágenes (ya sea la total o de unos objetos concretos) con `load("datos-guardados.RData")`.

Por último, es posible utilizar `saveRDS` para guardar objetos individuales serializados (en binario) y `readRDS` para leerlos posteriormente. Sirve para un único objeto, pero permite poder asignarlo a un nombre que se decide al cargarlo.

I.5. Scripts

Mantener todo el código en uno o varios scripts y ejecutarlo directamente desde el script y no desde la consola tiene varias ventajas:

- Permite mantener un registro de todo lo que se ha hecho y tenerlo organizado, con comentarios, etc.
- Permite realizar cálculos no interactivos. Por ejemplo, ejecutar un análisis muy largo o volver a ejecutar todo el análisis y los gráficos sin querer.

I.5.1. Utilizar un script

Hay dos maneras básicas de utilizar un script:

- De forma interactiva; lo que se ha hecho hasta entonces. Por ejemplo, RStudio permite seleccionar una parte del código y lanzarlo al intérprete de R, ejecutándolo desde la consola.
- De forma no interactiva:
  - Utilizando `source("script.R")`. En la sesión de R en la que se haya puesto esto, se importan las variables, funciones (y todo) del script. La diferencia es que, como es no interactivo, si se llaman a funciones (como por ejemplo, `mean(x)`), no se muestra el resultado por pantalla; para ello sería necesario utilizar print.
  - Desde la shell. Esto tiene la ventaja de no tener que mantener una ventana abierta con R hasta que el código finalice, por lo que es muy cómodo para los trabajos muy largos. La forma preferida es:

    ```
    R --vanilla < script1.R > script1.Rout
    ```

    La opción de vanilla permite que la sesión sea lo más reproducible posible, es decir, sin cargar librerías adicionales, sesiones de R anteriores, etc. Otra manera muy similar es `R --vanilla -f script1.R > script1.Rout` Con esto lo que conseguimos es que el resultado del script1 se guarde directamente en otro fichero.

I.6. Estructuras de datos básicas en R

I.6.1. Vectores

Los vectores son la estructura de datos más simple de R. Guardan una serie de objetos del mismo tipo, uno detrás de otro, en una sola dimensión.

```
v1 <- c(1, 2, 3) #vector de números enteros
#                 (se guardan como floats si no se fuerza)
v2 <- c("a", "b", "cucu") #vector de strings
v3 <- c(1.9, 2.5, 0.6) #vector de números float
v4 <- c(4, "a") #convierte el 4 en "4"
```

La c viene de concatenar, ya que hace precisamente eso: concatena lo que se le ponga a continuación.

Muchas funciones operan directamente en vectores enteros sin necesidad de realizar un loop sobre cada uno de los objetos en él:

```
log(v1)

## [1] 0.0000000 0.6931472 1.0986123

exp(v3)

## [1]  6.685894 12.182494  1.822119

2 * v1

## [1] 2 4 6

v3/0.7

## [1] 2.7142857 3.5714286 0.8571429
```

I.6.1.1. Funciones para crear vectores

Se pueden crear vectores concatenando elementos, pero hay otras dos funciones para crearlos que tienen algo de estructura: `seq` (de secuencia) y `rep` (de repetición). La función `seq` tiene cuatro formas de invocación:

```
seq(from = 1, to = 10)

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

```r
seq(from = 1, to = 10, by = 2)
```

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

```r
seq(from = 1, to = 10, length.out = 3)
```

```
## [1]  1.0  5.5 10.0
```

```r
1:5
```

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

`rep` también tiene varias invocaciones comunes:

```r
rep(2, 5)
```

```
## [1] 2 2 2 2 2
```

```r
rep(1:3, 2)
```

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

```r
rep(1:3, 2:4)
```

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

En este caso, es importante que el segundo argumento de `rep` sea un único valor (y repita todos los elementos del primer argumento las veces indicadas) o un conjunto de valores de las mismas dimensiones que el primer argumento (y se asigne a cada valor su respectivo número de repetición).

I.6.2. Crear vectores a partir de otros vectores

Se pueden concatenar dos vectores:

```r
v1 <- 1:4
v2 <- 7:12
(v3 <- c(v1, v2))
```

```
##  [1]  1  2  3  4  7  8  9 10 11 12
```

Si se emplean operaciones aritméticas en vectores que no son de la misma longitud, se utiliza la **regla de reciclaje**, es decir, se reutiliza el vector más pequeño cuando

llega a su fin las veces necesarias hasta haber terminado las operaciones con el vector grande:

```
v1 <- 1:3
v2 <- 11:12
v1 + v2

## Warning in v1 + v2: longitud de objeto mayor no es múltiplo de la
longitud de uno menor

## [1] 12 14 14
```

En ocasiones se produce un warning que avisa sobre la reutilización de uno de los vectores. Sin embargo, esto no ocurre siempre, ua que el warning se suprime cuando el vector a reutilizar se repite una ronda concreta (y no se quede a medias durante el reciclaje).

I.6.3. Logical operations

Se pueden comparar los elementos de un vector con algo para obtener un vector de elementos lógicos TRUE y FALSE. Esto es común en varios lenguajes de programación, pero hay que tener en cuenta la diferencia entre | y || y entre && y &. También se puede usar `xor` para obtener TRUE cuando solo uno de las condiciones es verdadera (no ambas).

```
v1 <- 1:5
v1 < 3

## [1]  TRUE  TRUE FALSE FALSE FALSE

(v2 <- (v1 < 3))

## [1]  TRUE  TRUE FALSE FALSE FALSE

v11 <- c(1, 1, 3, 5, 4)
v1 == v11

## [1]  TRUE FALSE  TRUE FALSE FALSE

v1 != v11

## [1] FALSE  TRUE FALSE  TRUE  TRUE

!(v1 == v11)
```

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

```
identical(v1, v11)
```

```
## [1] FALSE
```

```
v3 <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
!v3
```

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

```
v2 & v3
```

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

```
v2 | v3
```

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

```
(v1 > 3) & (v11 >= 2)
```

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

```
(v1 > 3) | (v11 >= 2)
```

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

```
xor(v2, v3)
```

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

I.6.3.1. Valores lógicos 0 y 1

En R, al igual que en otros lenguajes de programación, se pueden utilizar valores lógicos como si fuesen numéricos: se puede tratar `TRUE` como 1 y `FALSE` como 0. Además, `TRUE` puede ser cualquier otro número diferente a 0.

El operador `which` opera en un vector lógico, no en el vector directamente, y devuelve las posiciones que son verdaderas. `length` cuenta la longitud de la salida:

```
vv <- c(1, 3, 10, 2, 9, 5, 4, 6:8)
length(which(vv < 5))
```

```
## [1] 4
```

Es importante remarcar no utilizar T para TRUE y F para FALSE, aunque se pueda hacer. Esto se debe a que se puede redefinir el valor de T y F a que no correspondan a TRUE y FALSE (lo cual es muy difícil de debuggear), mientras que TRUE y FALSE siempre significarán lo mismo al no poder redefinirse.

I.6.3.2. Cortocircuito de operaciones lógicas

Los operadores && y || son cortocircuitos. Los dobles caracteres evalúan el segundo elemento sólo si la evaluación del primero no permite saber el resultado de la operación. Cuando se va a hacer un and y la primera condición es FALSE, no hace falta evaluar la segunda condición, ya que se conoce el resultado (de igual forma si en un or la primera condición es TRUE). Así, esto se puede utilizar para condicionar la ejecución de la segunda condición:

```
a <- "hola"
if (is.numeric(a) && log(a)) cat("\n we entered in the if")
```

En el ejemplo anterior, sólo se quiere evaluar el logaritmo de un número. Por ello, con &&, primero se evalúa si la variable es un número y, en caso afirmativo, se ejecuta el logaritmo. En caso de que la variable no sea numérica (como es el caso del ejemplo), utilizar un solo & resultaría en un error, y no sería lo que nos interesa.

```
a1 <- c(TRUE, FALSE)
b1 <- c(TRUE, TRUE)

a1 && b1

## Error in a1 && b1: 'length = 2' in coercion to 'logical(1)'

a1 || b1

## Error in a1 || b1: 'length = 2' in coercion to 'logical(1)'
```

Hay que tener en cuenta que no se deben utilizar vectores con más de un elemento con cortocircuitos, ya que sólo se evalúa el primer valor.

I.6.4. Nombres de elementos

Los elementos de un vector pueden tener nombres (que deben ser únicos). Esto permite acceder a los vectores utilizando nombres en lugar de posiciones, lo que puede ser más intuitivo.

```r
ages <- c(Juan = 23, Maria = 35, Irene = 12, Ana = 93)
names(ages)

## [1] "Juan"  "Maria" "Irene" "Ana"

ages

##  Juan Maria Irene   Ana
##    23    35    12    93

ages["Juan"]

## Juan
##   23
```

I.6.5. Acceder y modificar elementos de un vector: indexación y subsetting

I.6.5.1. Indexación de vectores

Hay cuatro formas para acceder a elementos específicos de un vector:

- Especificando las posiciones: mediante índices
- Dando los nombres de los elementos
- Utilizando un vector lógico
- Utilizando cualquier expresión que genere cualquiera de las anteriores.

Las posiciones y nombres se dan entre corchetes ([]).

Especificando las posiciones deseadas:

```r
(w <- 9:18)

##  [1]  9 10 11 12 13 14 15 16 17 18

w[1]

## [1] 9

w[2]

## [1] 10

w[c(4, 3, 2)]

## [1] 12 11 10
```

```
w[c(1, 3)] ## not the same as

## [1]  9 11

w[c(3, 1)]

## [1] 11  9
```

```
w[1:2]

## [1]  9 10

w[3:6]

## [1] 11 12 13 14

w[seq(1, 8, by = 3)]

## [1]  9 12 15

vv <- seq(1, 8, by = 3)
w[vv]

## [1]  9 12 15
```

Especificando las posiciones que no se desean (el vector original no se modifica):

```
w[-1]

## [1] 10 11 12 13 14 15 16 17 18

w[-c(1, 3)] ## of course, the same as following

## [1] 10 12 13 14 15 16 17 18

w[-c(3, 1)]

## [1] 10 12 13 14 15 16 17 18
```

Utilizando nombres

```
ages <- c(Juan = 23, Maria = 35, Irene = 12, Ana = 93)
ages["Irene"]

## Irene
##    12

ages[c("Irene", "Juan", "Irene")]

## Irene  Juan Irene
##    12    23    12
```

Utilizando un vector lógico . . .

```
ages[c(FALSE, TRUE, TRUE, FALSE)]

## Maria Irene
##    35    12

## what are thes doing? Avoid these things
ages[c(FALSE, TRUE)]

## Maria   Ana
##    35    93

ages[c(TRUE, TRUE, FALSE)]

##  Juan Maria   Ana
##    23    35    93
```

. . . o algo que es un vector lógico implícito

```
## All less than 12
w[w < 12]

## [1]  9 10 11

## same, but more confusing (here, not always)
w[!(w >= 12)]

## [1]  9 10 11

## All less than the median
w[w < median(w)]

## [1]  9 10 11 12 13
```

Si se puede acceder, también se puede modificar:

```
ages["Irene"] <- 19
ages

##  Juan Maria Irene   Ana
##    23    35    19    93

w[1] <- 9999
w

##  [1] 9999   10   11   12   13   14   15   16   17   18

w[vv] <- 103
w

##  [1] 103  10  11 103  13  14 103  16  17  18
```

Pero compara esto:

```
w[] <- 77
w[] <- 17:55

## Warning in w[] <- 17:55: número de elementos para sustituir no es
un múltiplo de la longitud del reemplazo

w <- 17:55
```

I.6.6. Interludio: comparación de floats

Comparar valores numéricos muy similares puede ser complicado y muy delicado debido al redondeo y algunos números que no se pueden representar exactamente en notación binaria. De forma predeterminada, R muestra 7 dígitos significativos.

```
x  <-  1.999999
x

## [1] 1.999999

x - 2

## [1] -1e-06

x <- 1.9999999999999
x
```

```
## [1] 2

x-2

## [1] -9.992007e-14
```

Todos los dígitos están presentes, pero en el segundo caso, no se muestran. Además, `x-2` no es exactamente -1×10^{-13}. En R se suelen redondear los números con una precisión de 53 dígitos binarios, por lo que dos números decimales no serán iguales de forma diable a menos que hayan sido calculados por el mismo algoritmo, y ni siquiera entonces:

```
a <- sqrt(2)
a * a == 2
[1] FALSE
a * a - 2
[1] 4.440892e-16
```

Otro ejemplo:

```
0.1 + 0.2 == 0.3

## [1] FALSE

(0.1 + 0.2) - 0.3

## [1] 5.551115e-17
```

En resumen: desconfía extremadamente siempre que veas una comparación de igualdad de dos números en coma flotante; es poco probable que haga lo que quieres. Si sabes lo que estás haciendo, echa un vistazo a `all.equal` para comparaciones de igualdad de objetos casi iguales.

I.6.7. Factores

Los factores son unos tipos especiales de vectores. Los necesitamos para diferenciar entre un vector de caracteres y un vector que representa variables categóricas. El vector `char.vec <- c("abc", "de", "fghi")` contiene varias cadenas de caracteres. Supongamos ahora que tenemos un estudio en el que registramos el sexo de los participantes. Cuando analizamos los datos queremos que R sepa que se trata de una variable categórica, donde cada etiqueta representa un posible valor de la categoría:

```
Sex.version1 <- factor(c("Female", "Female", "Female",
                         "Male", "Male"))
Sex.version2 <- factor(c("XX", "XX", "XX", "XY", "XY"))
```

```
Sex.version3 <- factor(c("Feminine", "Feminine", "Feminine",
                         "Masculine", "Masculine"))
Sex.version4 <- factor(c("fe", "fe", "fe", "ma", "ma"))
```

Queremos que todas esas codificaciones del sexo de cinco sujetos arrojen los mismos resultados de análisis, independientemente de lo que digan exactamente las etiquetas. Cada conjunto de etiquetas puede tener sus pros y sus contras (por ejemplo, la tercera probablemente está codificando el género, no el sexo; la última es demasiado críptica; la segunda sólo funciona para algunas especies; etc.). Independientemente de las etiquetas, lo que hay que tener en cuenta es que los tres primeros sujetos son del mismo tipo y los dos últimos son de un tipo diferente.

Reconocer los factores es esencial cuando se trata de variables que parecen números legítimos:

```
postal.code <- c(28001, 28001, 28016, 28430, 28460)
somey <- c(10, 20, 30, 40, 50)
summary(aov(somey ~ postal.code))

##             Df Sum Sq Mean Sq F value Pr(>F)
## postal.code  1  782.5   782.5   10.79 0.0462 *
## Residuals    3  217.5    72.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Lo anterior está haciendo algo tonto: está ajustando una regresión lineal, porque está tomando postal.code como un valor numérico legítimo. Pero sabemos que no tiene sentido que 28009 y 28016 (dos distritos de Madrid) estén separados por 7 unidades mientras que 28430 y 28410 estén separados por 20 unidades (dos pueblos cercanos al norte de Madrid), ni esperamos encontrar relaciones lineales con (el número del) código postal en sí.

A veces, al leer datos, una variable se convierte en factor, pero en realidad es una variable numérica. ¿Cómo convertirla en el conjunto original de números? Esto no funciona:

```
x <- c(34, 89, 1000)
y <- factor(x)
y

## [1] 34   89   1000
## Levels: 34 89 1000

as.numeric(y)

## [1] 1 2 3

y
```

```
## [1] 34   89   1000
## Levels: 34 89 1000
```

Los valores se han recodificado. Una forma sencilla de hacerlo es la siguiente:

```
as.numeric(as.character(y))

## [1]   34   89 1000
```

I.6.7.1. Factores y símbolos, colores, etc en gráficos

Muchas veces se puede ver código como el siguiente:

```
plot(y ~ x, col = c("red", "blue")[group])
```

donde group es un factor de la longitud de x o y con dos niveles (si tuviese más, habría que proporcionar más colores).

Otro ejemplo:

```
legend(1, 2, legend = c("A", "B"), pch = c(1, 2),
       col = c("red", "blue")[factor(levels(group))])
```

En este caso, los colores se van a sacar en el mismo orden que los puntos. Aunque sea enreversado, lo que se pide son los niveles del grupo y convertirlo en un factor. Así, los niveles se ponen en el orden que se tienen, y los colores se adjudican en ese mismo orden.

Un último ejemplo:

```
gr <- c("B", "A", "A", "B", "A")
group <- factor(gr)
c("red", "blue")[gr]

## [1] NA NA NA NA NA

c("red", "blue")[group]

## [1] "blue" "red"  "red"  "blue" "red"

c("red", "blue")[levels(group)]

## [1] NA NA

c("red", "blue")[factor(levels(group))]

## [1] "red"  "blue"
```

I.6.8. Matrices

Los vectores son unidimensionales, mientras que las matrices son bidimensionales, y los arrays pueden tener un número arbitrario de dimensiones. Aquí nos quedaremos en las matrices. Como en vectores, todos los elementos de una matriz o de un array son del mismo tipo.

Las matrices se pueden crear desde un vector:

```
matrix(1:10, ncol = 2)
```

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

```
matrix(1:10, nrow = 5)
```

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

```
matrix(1:10, ncol = 2, byrow = TRUE)
```

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

```
matrix(1:15, nrow = 5, ncol = 2)
```

```
## Warning in matrix(1:15, nrow = 5, ncol = 2): data length [15] is
not a sub-multiple or multiple of the number of columns [2]
```

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

Por defecto, R rellena la matriz por columnas, pero se puede especificar que sea por fila.

I.6.8.1. Combinar vectores para crear una matriz: `cbind`, `rbind`

Se pueden combinar vectores en horizontal o vertical para crear una matriz:

```
v1 <- 1:5
v2 <- 11:15
rbind(v1, v2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2   11   12   13   14   15
```

```
cbind(v1, v2)
```

```
##      v1 v2
## [1,]  1 11
## [2,]  2 12
## [3,]  3 13
## [4,]  4 14
## [5,]  5 15
```

También se puede hacer lo mismo con matrices siempre que tengan las dimensiones apropiadas:

```
A <- matrix(1:10, nrow = 5)
B <- matrix(11:20, nrow = 5)
cbind(A, B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
rbind(A, B)
```

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

```
##  [6,]   11   16
##  [7,]   12   17
##  [8,]   13   18
##  [9,]   14   19
## [10,]   15   20
```

I.6.8.2. Indexación y subsetting en matrices

Una matriz tiene dos dimensiones, pero por lo demás funciona de forma similar a vectores. La primera dimensión son filas, y la segunda son columnas. Si no se especifica nada para una dimensión, se devuelve en su totalidad.

```
A <- matrix(1:15, nrow = 5)
A[1, ] ## first row
```

```
## [1]  1  6 11
```

```
A[, 2] ## second column
```

```
## [1]  6  7  8  9 10
```

```
A[4, 2] ## fourth row, second column
```

```
## [1] 9
```

```
A[3, 2] <- 999
A[1, ] <- c(90, 91, 92)
A < 4
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,]  TRUE FALSE FALSE
## [3,]  TRUE FALSE FALSE
## [4,] FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE
```

El operador `which` puede no hacer lo que uno espera por defecto. Si se quieren los índices, se debe especificar.

```
which(A == 999)
```

```
## [1] 8
```

```
which(A == 999, arr.ind = TRUE)
```

```
##      row col
## [1,]   3   2
```

También se puede indexar mediante los nombres de filas y columnas:

```
B <- A
colnames(B) <- c("A", "E", "I")
rownames(B) <- letters[1:nrow(B)]
B[, "E"]

##   a   b   c   d   e
##  91   7 999   9  10

B["c", ]

##   A   E   I
##   3 999  13
```

Se puede utilizar una matriz para indexar otra. Esto es algo más avanzado, pero puede venir muy bien:

```
(m1 <- cbind(c(1, 3), c(2, 1)))

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    1

A[m1]

## [1] 91  3

## compare with
A[c(1, 3), c(2, 1)]

##      [,1] [,2]
## [1,]   91   90
## [2,]  999    3
```

Al indexar con una matriz, se devuelven tantos elementos como filas tiene la matriz.

Cuando se obtiene una sola columna, se pierde una dimensión y, en lugar de conseguir una matriz, el resultado es un vector. Para evitar esto, se puede emplear `drop = FALSE`

```
A[c(2, 4), 1, drop = FALSE]

##      [,1]
## [1,]    2
## [2,]    4
```

I.6.8.3. Operaciones con matrices

Hay muchas operaciones matriciales disponibles en R (abre tu libro de álgebra matricial e intenta encontrarlas, si quieres). Y muchas funciones operan directamente, por defecto, sobre toda la matriz, o sobre filas/columnas de la matriz:

```
sum(B)

## [1] 1366

mean(B)

## [1] 91.06667

colSums(B) #rowSums

##    A    E    I
##  104 1116  146

rowMeans(B) #colMeans

##        a        b        c        d        e
##  91.0000   7.0000 338.3333   9.0000  10.0000
```

También se pueden seleccionar filas y columnas utilizando esas operaciones:

```
B[rowMeans(B) > 9, ]

##    A   E  I
## a 90  91 92
## c  3 999 13
## e  5  10 15
```

I.6.9. Listas

Una lista es un contenedor general donde se pueden mezclar cosas de distintos tipos. De hecho, no debe por qué tener una estructura rectangular. Hay muchas formas de acceder a elementos de una lista.

```r
listA <- list(a = 1:5, b = letters[1:3])
listA[1]
```

```
## $a
## [1] 1 2 3 4 5
```

```r
listA[[1]]
```

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

```r
listA[["a"]]
```

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

```r
listA$a
```

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

```r
listA[[1]][2]
```

```
## [1] 2
```

Una lista más compleja sería la siguiente:

```r
(listB <- list(one.vector = 1:10,  hello = "Hola",
               one.matrix = matrix(rnorm(20), ncol = 5),
               another.list =
               list(a = 5,
                    b = factor(c("male",
                      "female", "female")))))
```

```
## $one.vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $hello
## [1] "Hola"
##
## $one.matrix
##            [,1]         [,2]       [,3]        [,4]       [,5]
## [1,]  0.7138685  0.006722624 -0.3526410  0.22530026  0.1095514
## [2,] -1.0418991  0.518587239  1.6283734  0.47995543 -1.3771762
## [3,]  1.0442272 -2.191620150  1.7483837  0.21057434 -1.2474583
## [4,]  1.1872855  0.996322371 -0.3794757 -0.05078153 -0.1741111
##
## $another.list
```

```
## $another.list$a
## [1] 5
##
## $another.list$b
## [1] male   female female
## Levels: female male

listB[[c(3, 11)]]

## [1] 1.748384

listB[[3]][11]

## [1] 1.748384

listB[[3]][3, 3]

## [1] 1.748384

listB[[3]][c(3, 3)]

## [1] 1.044227 1.044227

listB[c(3, 4)]

## $one.matrix
##            [,1]         [,2]       [,3]        [,4]       [,5]
## [1,]  0.7138685  0.006722624 -0.3526410  0.22530026  0.1095514
## [2,] -1.0418991  0.518587239  1.6283734  0.47995543 -1.3771762
## [3,]  1.0442272 -2.191620150  1.7483837  0.21057434 -1.2474583
## [4,]  1.1872855  0.996322371 -0.3794757 -0.05078153 -0.1741111
##
## $another.list
## $another.list$a
## [1] 5
##
## $another.list$b
## [1] male   female female
## Levels: female male
```

I.6.10. Dataframes

Un dataframe es una lista de vectores con la misma longitud y que pueden contener distintos tipos de objetos. La estructura es rectangular. Se pueden acceder

a los elementos como si fueran matrices y como si fueran listas. Además, se pueden convertir dataframes en matrices con `data.matrix(df)` y `as.matrix(df)`. Muchas operaciones de matrices, concretamente `rbind` y `cbind`, también funcionan con dataframes.

```
(AB <- data.frame(ID = c("a1", "a2", "a3", "a4", "a5"),
                  Age = c(12, 14, 12, 16, 19),
                  Sex = c("M", "F", "F", "M", "F"),
                  Y = c(11, 14, 15, 12, 19)))

##   ID Age Sex  Y
## 1 a1  12   M 11
## 2 a2  14   F 14
## 3 a3  12   F 15
## 4 a4  16   M 12
## 5 a5  19   F 19

(AC <- data.frame(ID = "a9", Age = 14, Sex = "M", Y = 17))

##   ID Age Sex  Y
## 1 a9  14   M 17

(AB2 <- rbind(AB, AC))

##   ID Age Sex  Y
## 1 a1  12   M 11
## 2 a2  14   F 14
## 3 a3  12   F 15
## 4 a4  16   M 12
## 5 a5  19   F 19
## 6 a9  14   M 17

as.matrix(AB) #convierte todo en strings

##      ID   Age  Sex Y
## [1,] "a1" "12" "M" "11"
## [2,] "a2" "14" "F" "14"
## [3,] "a3" "12" "F" "15"
## [4,] "a4" "16" "M" "12"
## [5,] "a5" "19" "F" "19"

data.matrix(AB) #convierte todo en números

##      ID Age Sex  Y
## [1,]  1  12   2 11
## [2,]  2  14   1 14
## [3,]  3  12   1 15
## [4,]  4  16   2 12
## [5,]  5  19   1 19
```

Es muy fácil añadir nuevas variables a los dataframes:

```
AB2$status <- rep(c("Z", "V"), 3)
```

I.7. Números aleatorios y semillas

Los generadores de números aleatorios hacen lo que indica su nombre: generan números de forma aleatoria cada vez que se ejecutan. No obstante, hay veces en los que se buscan números aleatorios, pero también permitir la reproducción del código. En esos casos, se emplean semillas. En R, la forma más sencilla de fijar una semilla es con `set.seed()`.

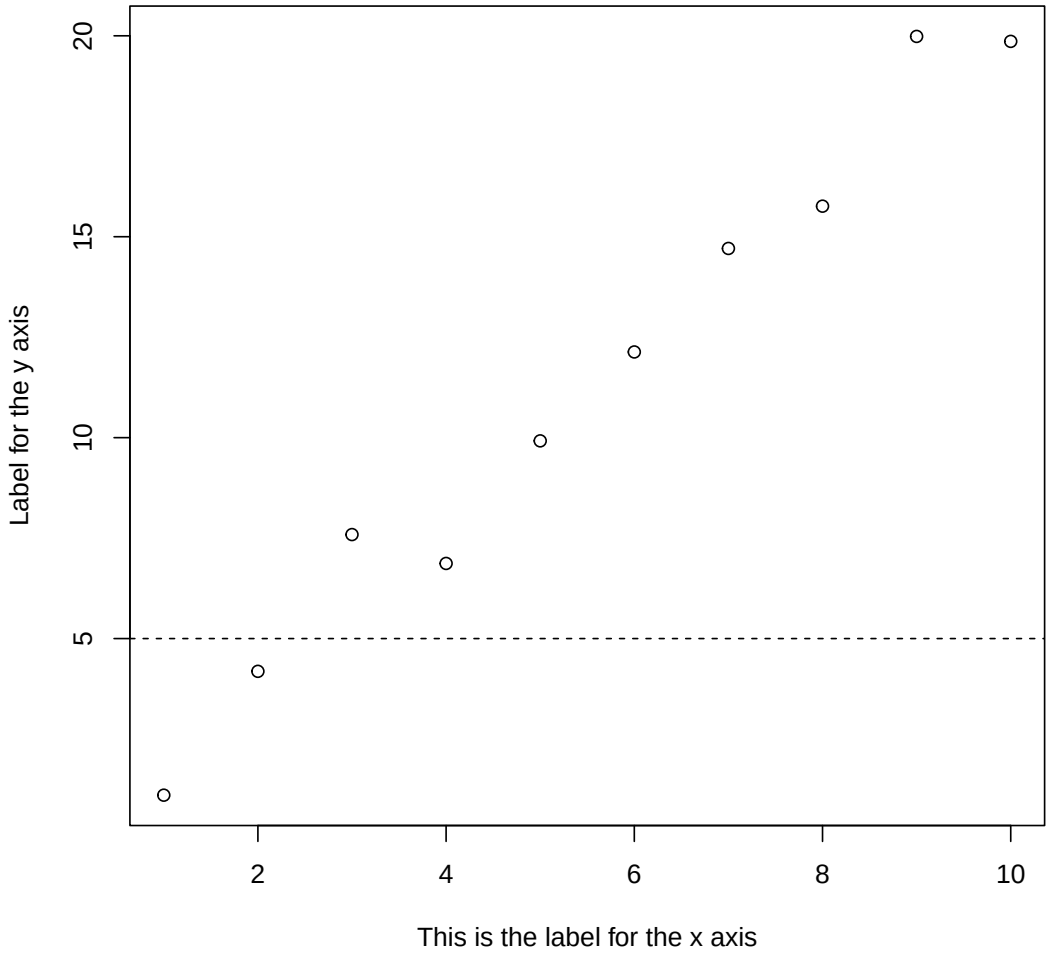
I.8. Plots (gráficos)

R puede producir una variedad de gráficos y se pueden modificar al gusto.

I.8.1. Lo más básico

La función de gráficos básica es `plot`. Su página de ayuda puede ser ligeramente engañosa y muchos argumentos adicionales se explican en `par`. Una buena analogía para empezar es la de un lienzo en el que se van añadiendo elementos. Veamos este sencillo ejemplo:

```
set.seed(2) ## for reproducibility
x <- 1:10
y <- 2 * x + rnorm(length(x))
plot(x, y, xlab = "This is the label for the x axis",
     ylab = "Label for the y axis")
## And now, we add a horizontal line:
abline(h = 5, lty = 2)
```



I.8.2. Personalización de plots: colores, tipos de línea y de puntos

Se pueden personalizar los gráficos añadiendo colores específicos, modificando el tipo de línea y de puntos.

```
plot(c(1, 21), c(1, 2.3),
     type = "n", axes = FALSE, ann = FALSE)
## show pch
points(1:20, rep(1, 20), pch = 1:20)
text(1:20, 1.2, labels = 1:20)
text(11, 1.5, "pch", cex = 1.3)

## show colors for rainbow palette
points(1:20, rep(2, 20), pch = 16, col = rainbow(20))
text(11, 2.2, "col", cex = 1.3)
```

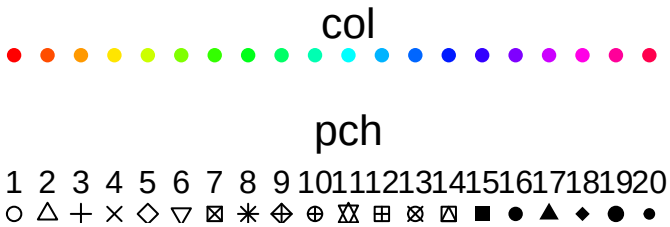


Figura I.1: *pch and col*

```
plot(c(0.2, 5), c(0.2, 5), type = "n", ann = FALSE, axes = FALSE)
for(i in 1:6) {
    abline(0, i/3, lty = i, lwd = 2)
    text(2, 2 * (i/3), labels = i, pos = 3)
}
```

I.8.2.1. Un ejemplo de cómo mejorar gráficos

El gráfico básico sería el siguiente:

```
hit <- read.table("data/hit-table-500-text.txt")
## We know, from the header of the file, that
## alignment length is the fifth column,
## score is the 13th and percent identity the 3rd
hist(hit[, 5]) ## the histogram
```

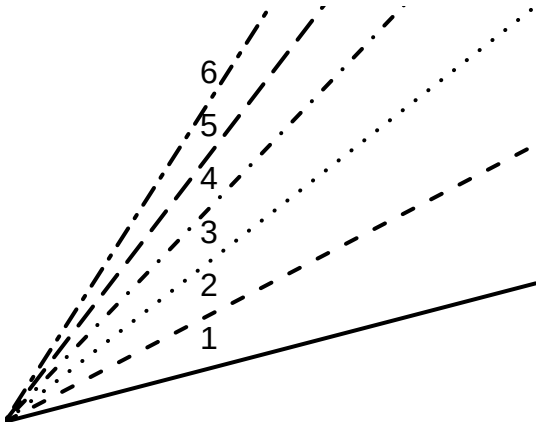
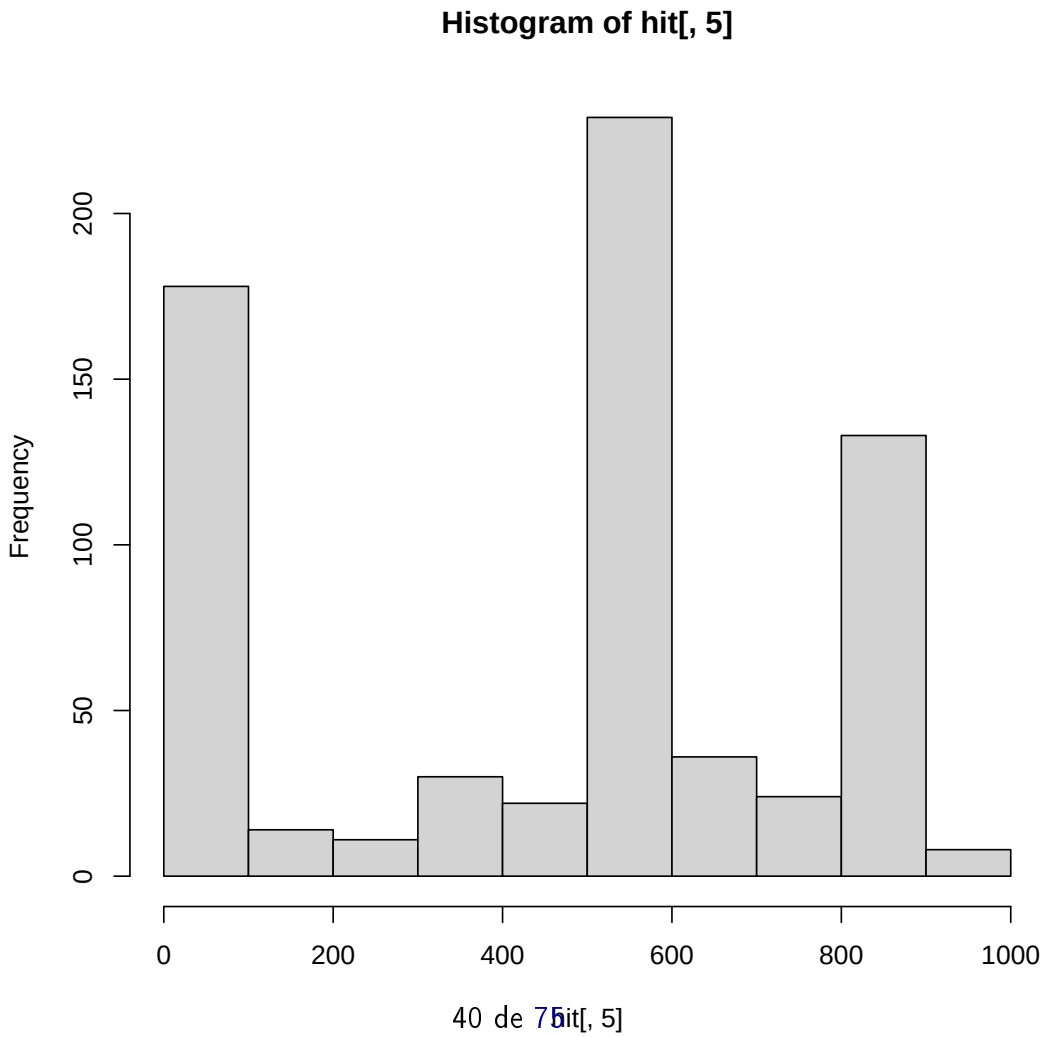
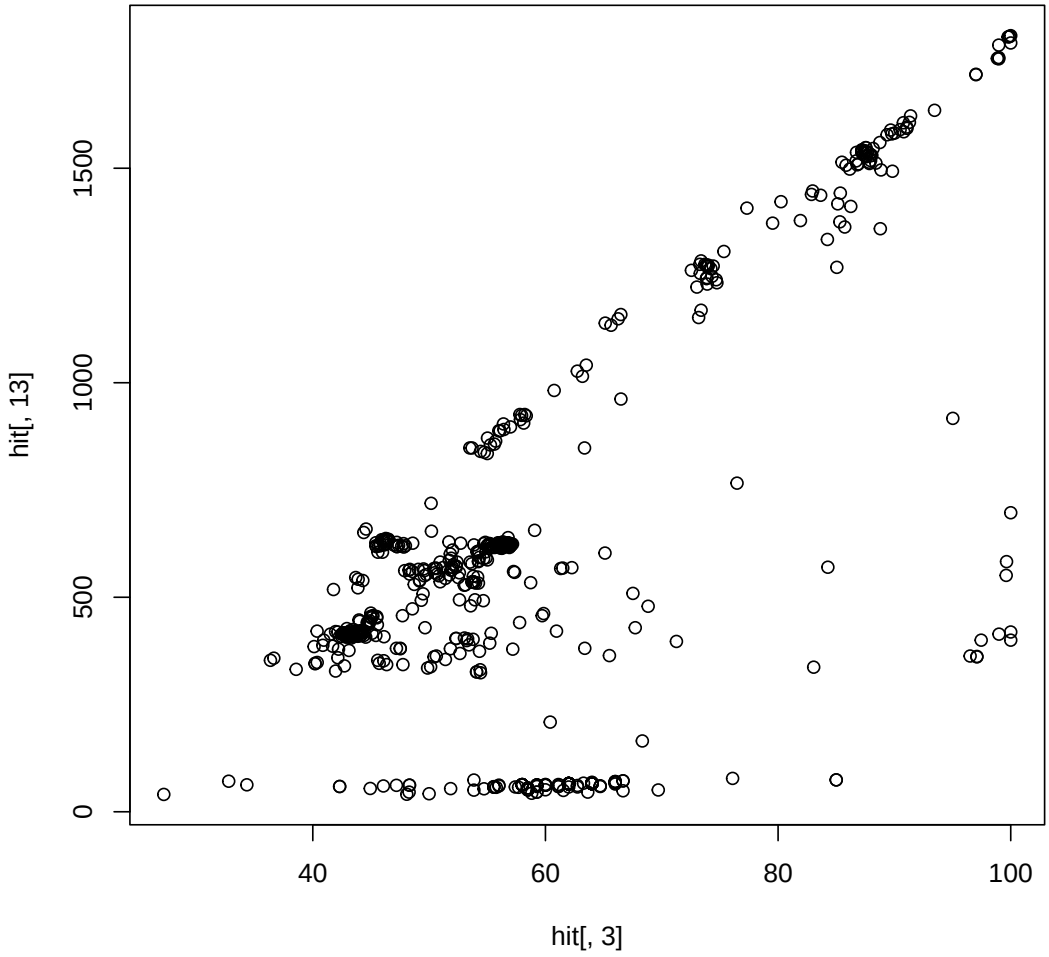


Figura I.2: *lty for values 1 to 6*



```
plot(hit[, 13] ~ hit[, 3]) ## the scatterplot
```




```
## plot(y ~ x) == plot(x, y)
```

Pero esto es fácilmente mejorable:

```
par(mfrow = c(1, 2)) ## two figures side by side
hist(hit[, 5], breaks = 50, xlab = "", main = "Alignment length")
plot(hit[, 13] ~ hit[, 3], xlab = "Percent. identity",
     ylab = "Bit score")
```

Por simetría, se podría añadir un título al segundo gráfico. También se pueden generar gráficos interactivos con la librería `car`.

```
library(car)
scatter3d(hit[, 13] ~ hit[, 3] + hit[, 5], xlab = "Ident",
          zlab = "Length", ylab = "Score")
```

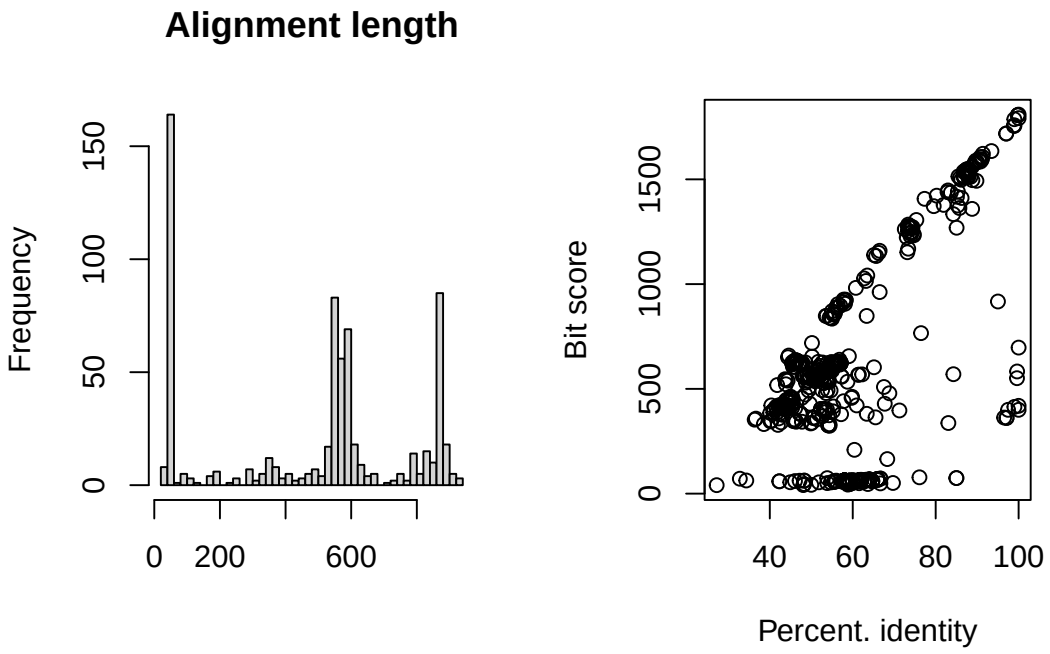


Figura I.3: *A quick look at the alignment results*

I.8.3. Guardar plots

Se pueden guardar las gráficas como PDF, png, etc. Desde RStudio hay una ventana de gráficos. Sin embargo, es mejor especificar y determinar unas características como tamaño, extensión, etc. Se pueden utilizar las funciones `pdf()` y `png()`: `pdf(file="plot.pdf")`. El paquete ggplot tiene la función `ggsave()`.

En el siguiente ejemplo se abre un PDF, se generan dos gráficos y hasta que no se ejecuta `dev.off()` no se guarda el contenido en el PDF. Además, cada gráfico se guarda en una página distinta del PDF.

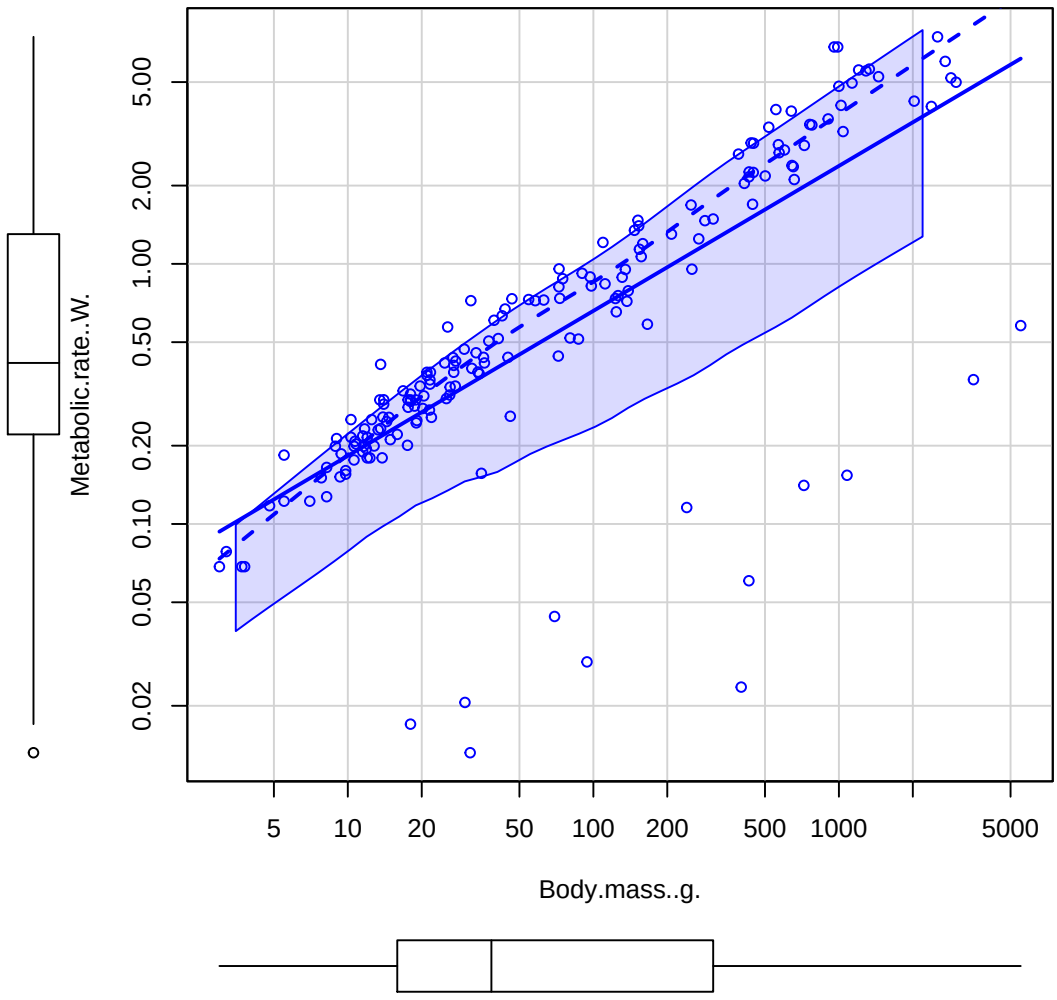
```
pdf(file = "file1.pdf", width = 2, heigh = 3)
plot(1:10)
abline(h = 4, lty = 2, col = "blue")
hist(rnorm(25))
dev.off()
```

I.8.4. Tipos de gráficos

Hemos visto que `plot` genera un gráfico simple de puntos, pero hay más tipos. Por ejemplo, `hist` genera un histograma. En el paquete de `ggplot2` hay más opciones de tipos de gráficos con una mayor posibilidad de personalización.

El paquete `car` también cuenta con varios tipos de gráficos, como scatter3d mencionado anteriormente. También tiene una función llamada `scatterplot`:

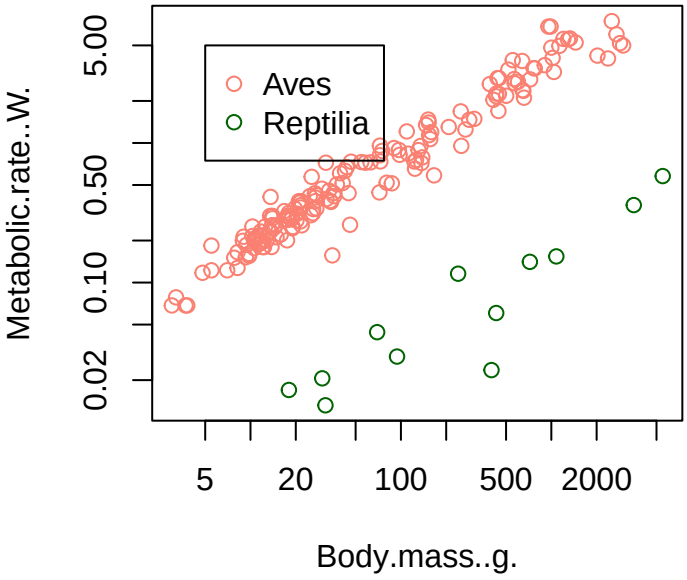
```
library(car)

## Cargando paquete requerido: carData

load("data/anage.RData")
scatterplot(Metabolic.rate..W. ~ Body.mass..g., log="xy",
            data = anage)
```



Dependiendo de la figura final que se quiera, se puede ir añadiendo elementos desde plot o utilizar scatterplot y eliminar cosas.

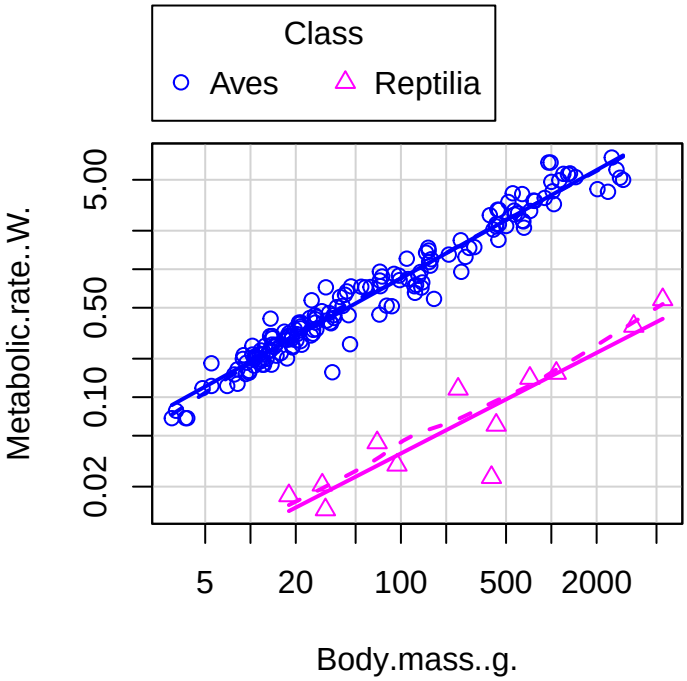
Las leyendas se introducen con la función `legend` en caso de utilizar plot.

```
plot(Metabolic.rate..W. ~ Body.mass..g., log="xy",
     col = c("salmon", "darkgreen")[Class], data = anage)
legend(5, 5, legend = levels(anage$Class),
       col = c("salmon", "darkgreen"),
       pch = 1)
```



Si se utiliza scatterplot, la sintaxis es diferente y añade directamente la línea de regresión:

```
scatterplot(Metabolic.rate..W. ~ Body.mass..g.|Class, log="xy",
            data = anage)
```



I.9. Tablas

Tabular datos es una operación muy común. Hay fundamentalmente dos formas de realizarlo con `table` (la más sencilla) y `xtabs` (con uso más genérico):

```
table(AB2$Sex, AB2$status)
```

```
##
##     V Z
##   F 1 2
##   M 2 1
```

```
with(AB2, table(Sex, status)) ## note "with"
```

```
##    status
## Sex V Z
##   F 1 2
##   M 2 1
```

```
xtabs( ~ Sex + status, data = AB2)
```

```
##    status
## Sex V Z
##   F 1 2
##   M 2 1
```

Tabular un dataframe completo saca varias tablas 2x2 en función de las combinaciones de los valores de las otras variables.

I.9.1. Más de dos dimensiones y `ftable`

Cuando hay más de dos dimensiones, utilizar las funciones anteriores saca el mismo resultado.

```
(x <-  data.frame(a = c(1,2,2,1,2,2,1),
                  b = c(1,2,2,1,1,2,1),
                  c = c(1,1,2,1,2,2,1)))
```

```
##   a b c
## 1 1 1 1
## 2 2 2 1
## 3 2 2 2
## 4 1 1 1
## 5 2 1 2
## 6 2 2 2
## 7 1 1 1
```

```
## Equivalent
table(x)

## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2

xtabs(~ a + b + c, data = x)

## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2
```

Sin embargo, hay veces en las que buscamos una tabla plana, es decir, encajar una de las variables dentro de otra:

```
ftable(xtabs(~ a + b + c, data = x))

##     c 1 2
## a b
## 1 1   3 0
##   2   0 0
## 2 1   0 1
##   2   1 2

## same as ftable(table(x))
```

I.9.2. Recuperar una tabla de un dataframe

Si una tabla se nos ha convertido en un dataframe, se puede volver a convertir en tabla:

```
## create a data frame with a "Freq" column:
## put the table in a data frame
(dfx <- as.data.frame(table(x)))

##   a b c Freq
## 1 1 1 1    3
## 2 2 1 1    0
## 3 1 2 1    0
## 4 2 2 1    1
## 5 1 1 2    0
## 6 2 1 2    1
## 7 1 2 2    0
## 8 2 2 2    2

## We can recover the table
xtabs(Freq ~ a + b + c, data = dfx)

## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2

## of course, this is the same as
## xtabs(~ a + b + c, data = x)
## or table(x)
```

I.10. La familia `apply`

Una de las grandes ventajas de R es poder operar sobre vectores, arrays, listas, etc enteros. Algunas funciones son `apply`, `lapply`, `sapply`, `tapply`, `mapply`.

I.10.1. apply

apply es una forma de utilizar una función sobre una cierta cantidad de elementos. Es una forma más elegante que realizando un bucle.

```
(Z <- matrix(c(1, 27, 23, 13), nrow = 2))

##      [,1] [,2]
## [1,]    1   23
## [2,]   27   13

apply(Z, 1, median)

## [1] 12 20

apply(Z, 2, median)

## [1] 14 18

apply(Z, 2, min)

## [1]  1 13
```

I.10.2. lapply

Con listas se utiliza `lapply`, y aplica una función definida a esa lista.

```
(listA <- list(one.vector = 1:10,  hello = "Hola",
               one.matrix = matrix(rnorm(20), ncol = 5),
               another.list = list(a = 5,
                 b = factor(c("male", "female", "female")))))

## $one.vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $hello
## [1] "Hola"
##
## $one.matrix
##            [,1]        [,2]       [,3]         [,4]       [,5]
## [1,]  0.4176508  1.78222896  1.0128287  1.589638200  0.4772373
## [2,]  0.9817528 -2.31106908  0.4322652  1.954651642 -0.5965582
## [3,] -0.3926954  0.87860458  2.0908192  0.004937777  0.7922033
## [4,] -1.0396690  0.03580672 -1.1999258 -2.451706388  0.2896367
```

```
##
## $another.list
## $another.list$a
## [1] 5
##
## $another.list$b
## [1] male   female female
## Levels: female male

lapply(listA, function(x) x[[1]])

## $one.vector
## [1] 1
##
## $hello
## [1] "Hola"
##
## $one.matrix
## [1] 0.4176508
##
## $another.list
## [1] 5
```

I.10.3. tapply y by

Se utiliza `tapply` cuando tenemos unos datos (una columna o varias) que se pueden utilizar para estratificar o seleccionar otros datos. Los dos vectores u objetos deben tener la misma longitud.

```
(one.dataframe <- data.frame(age = c(12, 13, 16, 25, 28),
                             sex = factor(c("male", "female",
                                            "female", "male", "male")))
)

##   age    sex
## 1  12   male
## 2  13 female
## 3  16 female
## 4  25   male
## 5  28   male

one.dataframe <- rbind(one.dataframe, one.dataframe)
one.dataframe$age[6:10] <- one.dataframe$age[6:10] + 2
one.dataframe$country <- rep(c("A", "B"), c(5, 5))
one.dataframe$Y <- rnorm(10)
one.dataframe
```

```
##    age    sex country          Y
## 1   12   male       A  0.7389386
## 2   13 female       A  0.3189604
## 3   16 female       A  1.0761644
## 4   25   male       A -0.2841577
## 5   28   male       A -0.7766753
## 6   14   male       B -0.5956605
## 7   15 female       B -1.7259798
## 8   18 female       B -0.9025845
## 9   27   male       B -0.5590619
## 10  30   male       B -0.2465126

tapply(one.dataframe$age, one.dataframe$sex, mean)

##   female     male
## 15.50000 22.66667
```

Se pueden formar grupos con dos o más variables siempre y cuando se proporcionen en forma de lista:

```
tapply(one.dataframe$age,
       list(one.dataframe$sex, one.dataframe$country),
       mean)

##                A        B
## female 14.50000 16.50000
## male   21.66667 23.66667
```

De igual forma, se puede utilizar una función que devuelva más que un solo valor:

```
tapply(one.dataframe$age,
       one.dataframe$sex,
       function(x) c(Mean = mean(x), Var = var(x)))

## $female
##      Mean       Var
## 15.500000  4.333333
##
## $male
##     Mean      Var
## 22.66667 59.06667
```

El problema con esto viene cuando se quiere realizar lo anterior en base a dos o más variables. Para estos casos, se debería emplear `aggregate`.

La función `by` es similar a `tapply`, pero para dataframes. Las salidas de ambas funciones también son diferentes:

```r
by(one.dataframe,
   list(one.dataframe$sex, one.dataframe$country),
   function(x) c(Mean_Age = mean(x$age), SD_Age = sd(x$age),
                 Median_Y = median(x$Y)))
```

```
## : female
## : A
##   Mean_Age     SD_Age   Median_Y
## 14.5000000  2.1213203  0.6975624
## ------------------------------------------------
## : male
## : A
##   Mean_Age     SD_Age   Median_Y
## 21.6666667  8.5049005 -0.2841577
## ------------------------------------------------
## : female
## : B
##  Mean_Age    SD_Age  Median_Y
## 16.500000  2.121320 -1.314282
## ------------------------------------------------
## : male
## : B
##   Mean_Age     SD_Age   Median_Y
## 23.6666667  8.5049005 -0.5590619
```

I.10.4. aggregate

La función aggregate suele devolver la salida en un formato más conveniente. En este caso, el segundo argumento debe ser siempre una lista:

```r
aggregate(one.dataframe$age, list(one.dataframe$sex), mean)
```

```
##   Group.1        x
## 1  female 15.50000
## 2    male 22.66667
```

```r
## make the aggregating variable explicit,
## and give it another name
aggregate(one.dataframe$age,
          list(Sexo = one.dataframe$sex), mean)
```

```
##     Sexo        x
## 1 female 15.50000
## 2   male 22.66667
```

```r
## or use the name of the column/variable
aggregate(one.dataframe$age,
          one.dataframe[2], mean)

##      sex        x
## 1 female 15.50000
## 2   male 22.66667
```

Se puede utilizar con dos o más variables:

```r
aggregate(one.dataframe$age,
          list(Sex = one.dataframe$sex,
               Country = one.dataframe$country), mean)

##      Sex Country        x
## 1 female       A 14.50000
## 2   male       A 21.66667
## 3 female       B 16.50000
## 4   male       B 23.66667
```

También se puede utilizar para devolver varios valores:

```r
aggregate(one.dataframe$age,
          list(Sex = one.dataframe$sex,
               Country = one.dataframe$country),
          function(x) c(Mean = mean(x), SD = sd(x))
          )

##      Sex Country    x.Mean      x.SD
## 1 female       A 14.500000  2.121320
## 2   male       A 21.666667  8.504901
## 3 female       B 16.500000  2.121320
## 4   male       B 23.666667  8.504901
```

aggregate también se puede llamar con una sintaxis de tipo fórmula, que puede ser más intuitiva:

```r
aggregate(age ~ sex + country, data = one.dataframe,
          function(x) c(Mean = mean(x), SD = sd(x)))

##      sex country  age.Mean    age.SD
## 1 female       A 14.500000  2.121320
## 2   male       A 21.666667  8.504901
## 3 female       B 16.500000  2.121320
## 4   male       B 23.666667  8.504901
```

Esto también funciona para funciones con múltiples columnas o vectores:

```
(ag1 <- aggregate(cbind(age, Y) ~ sex + country,
                  data = one.dataframe,
                  function(x) c(Mean = mean(x), SD = sd(x))))

##      sex country  age.Mean    age.SD     Y.Mean       Y.SD
## 1 female       A 14.500000  2.121320  0.6975624  0.5354240
## 2   male       A 21.666667  8.504901 -0.1072981  0.7731305
## 3 female       B 16.500000  2.121320 -1.3142821  0.5822284
## 4   male       B 23.666667  8.504901 -0.4670783  0.1918901

aggregate(one.dataframe[, c("age", "Y")],
          list(Sex = one.dataframe$sex,
               Country = one.dataframe$country),
          function(x) c(Mean = mean(x), SD = sd(x)))

##      Sex Country  age.Mean    age.SD     Y.Mean       Y.SD
## 1 female       A 14.500000  2.121320  0.6975624  0.5354240
## 2   male       A 21.666667  8.504901 -0.1072981  0.7731305
## 3 female       B 16.500000  2.121320 -1.3142821  0.5822284
## 4   male       B 23.666667  8.504901 -0.4670783  0.1918901
```

Es importante mencionar que el resultado no es un dataframe de 6 columnas, si no de 4: Mean y SD forman una matriz de dos columnas dentro de una misma columna. Para que el dataframe de salida sí tenga las 6 columnas, se puede utilizar `do.call`:

```
do.call(data.frame,
        aggregate(cbind(age, Y) ~ sex + country,
                  data = one.dataframe,
                  function(x) c(Mean = mean(x), SD = sd(x)))
        )

##      sex country age.Mean   age.SD     Y.Mean      Y.SD
## 1 female       A 14.50000 2.121320  0.6975624 0.5354240
## 2   male       A 21.66667 8.504901 -0.1072981 0.7731305
## 3 female       B 16.50000 2.121320 -1.3142821 0.5822284
## 4   male       B 23.66667 8.504901 -0.4670783 0.1918901
```

I.10.5. split

La función `split` sirve para dividir un dataframe en varios en función de una variable

```
split(one.dataframe, one.dataframe$sex)

## $female
```

```
##   age    sex country          Y
## 2  13 female       A  0.3189604
## 3  16 female       A  1.0761644
## 7  15 female       B -1.7259798
## 8  18 female       B -0.9025845
##
## $male
##    age  sex country          Y
## 1   12 male       A  0.7389386
## 4   25 male       A -0.2841577
## 5   28 male       A -0.7766753
## 6   14 male       B -0.5956605
## 9   27 male       B -0.5590619
## 10  30 male       B -0.2465126
```

```
split(one.dataframe, c(one.dataframe$sex, one.dataframe$country))
```

```
## Warning in split.default(x = seq_len(nrow(x)), f = f, drop = drop,
...): largo de datos no es múltiplo de la variable de separación
```

```
## $`1`
##   age    sex country          Y
## 2  13 female       A  0.3189604
## 3  16 female       A  1.0761644
## 7  15 female       B -1.7259798
## 8  18 female       B -0.9025845
##
## $`2`
##    age  sex country          Y
## 1   12 male       A  0.7389386
## 4   25 male       A -0.2841577
## 5   28 male       A -0.7766753
## 6   14 male       B -0.5956605
## 9   27 male       B -0.5590619
## 10  30 male       B -0.2465126
##
## $A
## [1] age     sex     country Y
## <0 rows> (o 0- extensión row.names)
##
## $B
## [1] age     sex     country Y
## <0 rows> (o 0- extensión row.names)
```

Esto se puede combinar con *apply:

```r
lapply(split(one.dataframe,
             list(one.dataframe$sex,
                  one.dataframe$country)),
       function(x) lm(Y ~ age, data = x)) #or lm(x$Y ~ x$age)

## $female.A
##
## Call:
## lm(formula = Y ~ age, data = x)
##
## Coefficients:
## (Intercept)          age
##     -2.9623       0.2524
##
##
## $male.A
##
## Call:
## lm(formula = Y ~ age, data = x)
##
## Coefficients:
## (Intercept)          age
##     1.84109     -0.08993
##
##
## $female.B
##
## Call:
## lm(formula = Y ~ age, data = x)
##
## Coefficients:
## (Intercept)          age
##     -5.8430       0.2745
##
##
## $male.B
##
## Call:
## lm(formula = Y ~ age, data = x)
##
## Coefficients:
## (Intercept)          age
##    -0.84879      0.01613
```

El procedimiento anterior está relacionado con los enfoques split-apply-combine y map-reduce. Y `by`, `aggregate`, y amigos pueden ser considerados como formas especialmente prácticas de hacer la combinación anterior de `split` con `*apply` y alguna(s) función(es) de resumen particular(es).

I.10.6. apply y dejar caer dimensiones en matrices

A menos que utilicemos `drop = FALSE`, si seleccionamos sólo una fila o una columna, el resultado no es una matriz, sino un vector. Pero a veces necesitamos que permanezcan como matrices. Ese es a menudo el caso en muchas operaciones matriciales, y también cuando se utiliza `apply` y afines.

```
(E <- matrix(1:9, nrow = 3))
```

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

```
E[, 1]
```

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

```
E[, 1, drop = FALSE]
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

```
E[1, ]
```

```
## [1] 1 4 7
```

```
E[1, , drop = FALSE]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
```

Esto suele ser importante cuando se escribe código genérico y una variable solo tenga una dimensión.

I.10.7. Algunas apreciaciones

Hay otros tipos de `apply` que veremos más adelante, tales como `vapply`, `sapply`, `mapply`. Además, las operaciones con `apply` son fácilmente paralelizables (librería `parallel`).

I.11. Programación en R

I.11.1. Flow control

R tiene las típicas construcciones condicionales y estructuras de control: `if`, `ifelse`, `for`, `while`, `repeat`, `switch`, `break`. Un `for` loop rara vez es la opción adecuada, normalmente es mejor utilizar `apply`.

```
names.of.friends <- c("Ana", "Rebeca", "Marta",
                      "Quique", "Virgilio")
for(friend in names.of.friends) {
  cat("\n I should call", friend, "\n")
}
```

```
##
##  I should call Ana
##
##  I should call Rebeca
##
##  I should call Marta
##
##  I should call Quique
##
##  I should call Virgilio
```

```
x <- y <- 0
iteration <- 1
while( (x < 10) && (y < 2)) {
  cat(" ... iteration", iteration, "\n")
  x <- x + runif(1)
  y <- y + rnorm(1)
  iteration <- iteration + 1
}
```

```
##  ... iteration 1
##  ... iteration 2
##  ... iteration 3
##  ... iteration 4
##  ... iteration 5
##  ... iteration 6
##  ... iteration 7
##  ... iteration 8
```

```
x
```

```
## [1] 3.183051
```

```
y

## [1] 3.020543
```

`while` normalmente se combina con `break` para salidr del bucle en cuanto pasa algo (normalmente detectado mediante `if`). Break sirve para salir del bloque de llaves del bucle en el que está metido, no para todo.

```
iteration <- 0
while(TRUE) {
  iteration <- iteration + 1
  cat(" ... iteration", iteration, "\n")
  x <- rnorm(1, mean = 5)
  y <- rnorm(1, mean = 7)
  z <- x * y
  if (z < 15) break
}
```

```
aa <- 9

if (aa < 95) {
  cat("\n aa is < 95\n")
} else if (aa > 100) {
  cat("\n hummm.... larger than a 100\n")
} else {
  cat("\n between 95 and a 100\n")
}

##
##  aa is < 95
```

I.11.2. Definir funciones

Se pueden crear funciones en R mediante `function`:

```
multByTwo <- function(x) {
  z <- 2 * x
  return(z)
}

a <- 3
multByTwo(a)

## [1] 6
```

```
multByTwo(45)

## [1] 90
```

Si no se incluye `return`, la función devuelve el último valor generado, pero es recomendable añadirlo para facilitar la lectura.

Las funciones pueden tener varios argumentos, y es posible que tengan valores por defecto:

```
plotAndLm <- function(x, y, title = "A figure") {
  lm1 <- lm(y ~ x)
  cat("\n Printing the summary of x\n")
  print(summary(x))
  cat("\n Printing the summary of y\n")
  print(summary(y))
  cat("\n Printing the summary of the linear regression\n")
  print(summary(lm1))
  plot(y ~ x, main = title)
  abline(lm1)
  return(lm1)
}

x <- 1:20
y <- 5 + 3 *x + rnorm(20, sd = 3)
plotAndLm(x, y)
plotAndLm(x, y, title = "A user specified title")
```

I.11.3. Orden de los argumentos, argumentos con y sin nombre

R es bastante flexible a la hora de llamar a una función y el orden en el que se pasan los argumentos, pero hay formas mejores y peores de hacerlo. En general, se utiliza la posición de llamada solo para los primeros dos argumentos, y se recomienda evitar pasar argumentos sin nombre después de haber nombrado a algunos:

```
f1 <- function(one, two, three) {
    cat("one = ", one,
        " two = ", two,
        " three = ", three, "\n")}

## We are OK
f1(1, 2, 3)

## one =  1  two =  2  three =  3
```

```
## We are OK, but this is getting risky
f1(two = 2, three = 3, 1)

## one =  1  two =  2  three =  3

## We are no longer OK. Nothing "strange" happened
## but we would need to be very careful. So avoid it.
f1(two = 2, 3, 1)

## one =  3  two =  2  three =  1
```

I.11.4. Scoping, frames y entornos

R puede tener variables globales y locales.

```
f1 <- function(x) {
    x + z
}

z <- -100 #variable global

f11 <- function(y) {
    z <- 10 #variable local
    f1(y)
}

f11(4)

## [1] -96
```

En este caso, z podría adquirir el valor donde se definió f1 (el entorno global) o usando el valor del entorno local en el que se llamó a f1. R utiliza la primera opción: resuelve donde se definió f1, tomando el valor de z de ese entorno. Esto es igual en otros lenguajes como Python.

Este es otro ejemplo en el que, como se define una función dentro de otra, al llamarla hereda los valores de las variables del entorno local.

```
v <- 1000
f3 <- function(x, y) {
    v <- 3 * x
    f2 <- function(u) {
        u + v
    }
    f2(y)
}
```

```
f3(2, 9)
```

binding En `y <- 9`, y está unida al valor 9.

free variable z es una variable libre en la función f1 de arriba. No está unida a nada (al menos en ese frame)

frame Una serie de bindings (y a 9, x a 77, etc.).

environment Puedes pensar en ello como una secuencia de frames Cuando f2 (bueno, R) busque el valor de v lo hará a través de una secuencia de frames De hecho, un entorno tiene dos componentes: un frame y una referencia a otro entorno, su entorno padre (o su entorno adyacente); puesto que cada entorno tiene una referencia a otro entorno, ahora puedes entender fácilmente la idea de un entorno como una secuencia de frames

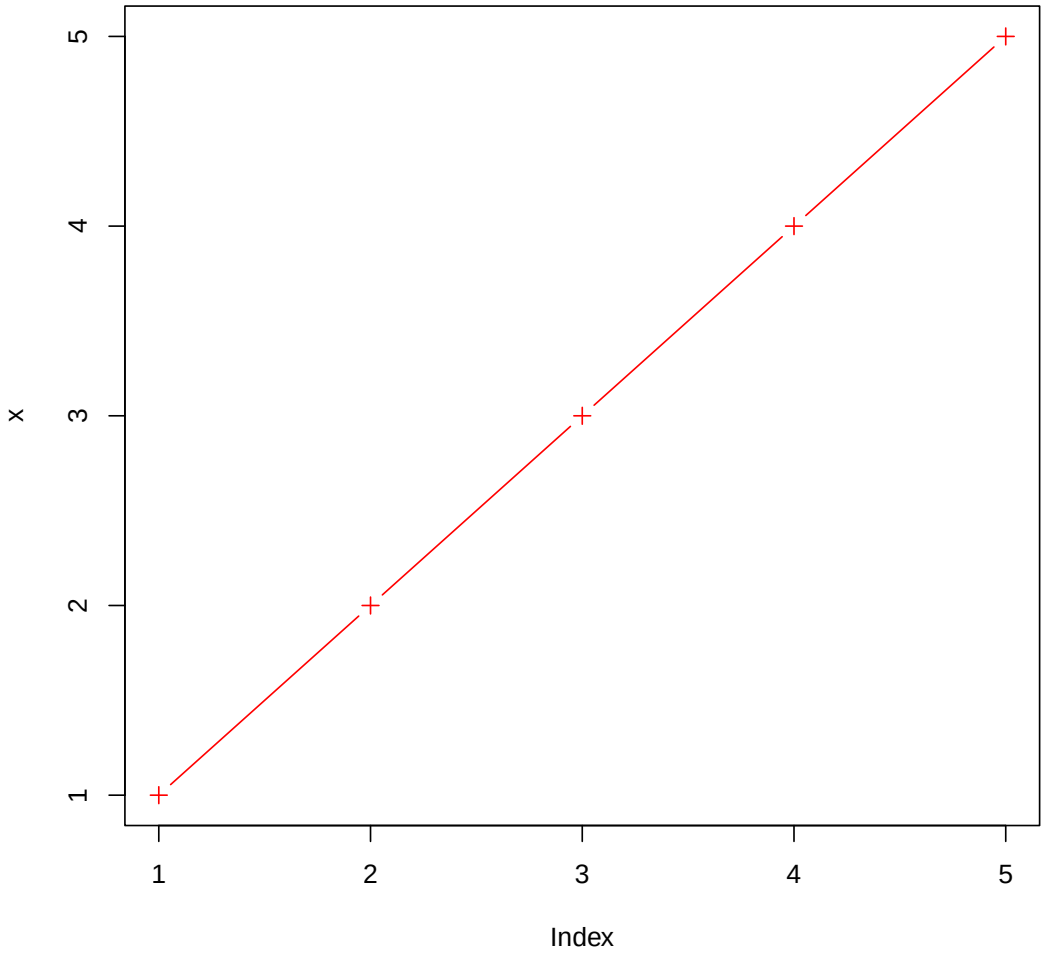
```
search()
```

Esto se utiliza implícitamente o explícitamente en gran parte del código. Lo que hace es listar los distintos entornos que hay y su orden. Así, cuando se cargan librerías o se utilizan variables, se utiliza `search` para localizar lo que se está pidiendo.

I.11.5. Los ...

Los ... permiten pasar argumentos adicionales en funciones que deben manejarlas.

```
f0 <- function(x, pch = 3, ...) {plot(x, pch = pch, ...)}

f0(1:5, col = "red", type = "b")
```



Aquí, col y type que no se han especificado en f0, se pasan directamente a la función. Como plot acepta muchos argumentos adicionales, no es necesario especificar todos, si no que se pueden poner ....

```
fa <- function(x, col = "red", ...) {plot(x, col = col, ...)}
fa(1:5, "blue", pch = 8)

fb <- function(x, col = "red", ...) {plot(x, col = col)}
fb(1:5, "blue", pch = 8)

fc <- function(x, col = "red") {plot(x, col = col, ...)}
fc(1:5, "blue", pch = 8)
```

La función fa recoge pch dentro de los tres puntos. Además, el color se sobrescribe (el plot resultante tiene los puntos en azul). Tanto fb como fc no hacen lo que se espera, ya que les falta ...en la función y en el plot respectivamente. Esa ausencia hace que las funciones no hagan nada con ese argumento.

I.11.6. local

La función `local` permite crear un entorno local en el que trabajar y luego, al salir, que no tenga guardado el valor de las variables.

```
i <- 2
local({cat("i ", i); i <- 99; cat(";  i = ", i)})

## i  2;  i =  99

i

## [1] 2

try(rm(vv))
local({vv <- 99; cat("vv = ", vv)})

## vv =  99

try(vv)

## Error in eval(expr, envir) : objeto 'vv' no encontrado
```

I.11.7. Evaluación vaga

La siguiente función toma 2 argumentos, pero solo utiliza 1. Por ello, cuando solo se le pasa un argumento, no se produce ningún error.

```
flazy <- function(x, y) {return(2 * x)}
flazy(4)

## [1] 8
```

Esto no es algo a hacer de forma habitual, pero es importante entender por qué el código no se rompe. En otras palabras, la evaluación vaga es la evaluación de los valores cuando se van a utilizar, no cuando se definen.

I.12. Debugging y capturar excepciones

Debugging consiste en recorrer el código para comprobar que funciona como se espera y no se estropee.

I.12.1. `traceback`

`traceback` muestra la última llamada y ayuda a identificar dónde se rompió el código para ver qué función tiene un problema.

```
f1 <- function(x) 3 * x

f2 <- function(x) 5 + f1(x)

f3 <- function(z, u) {
    v <- runif(z)
    a <- f2(u)
    b <- f2(3 * v)
    return(a + b)
}

f3(3, 7)

## [1] 36.97035 35.67900 33.98585

f3(-5, 6)

## Error in runif(z): invalid arguments

traceback()

## No traceback available

f3(5, "a")

## Error in 3 * x: argumento no-numérico para operador binario

traceback()

## No traceback available
```

I.12.2. `debug` **and** `browser`

El comando debug permite ir paso a paso ejecutando cada línea del código. Cuando se quiera parar, hay que poner simplemente undebug.

```
debug(f3)
f3(3, 5)
undebug(f3) ## stop debugging
f3(3, 5)
```

El comando `browser` para la ejecución de la expresión actual y permite acceder al intérprete de R. También se puede realizar de forma condicional.

```
## just browser
f3 <- function(z, u) {
    v <- runif(z)
    a <- f2(u)
    browser()
    b <- f2(3 * v)
    return(a + b)
}

## with conditional browser
f3 <- function(z, u) {
    v <- runif(z)
    if (z > 5) browser()
    a <- f2(u)
    b <- f2(3 * v)
    return(a + b)
}
```

Desde `browser`, hay una serie de expresiones:

- n o enter permite ejecutar la siguiente línea.
- c: salir del browser y continuar la ejecución del siguiente statement.
- s: evalúa el siguiente statement entrando en las siguientes funciones.
- Q: salir de la evaluación actual e ir al sitio desde donde se llamó.

`debug` es como poner `browser` al inicio del código.

I.12.3. `trace` para ver funciones arbitrarias en sitios arbitrarios

Se puede utilizar `debug` con funciones que no hayamos escrito nosotros (por ejemplo, `debug(lm)`). Sin embargo, la función lm es muy larga y quizás no queremos empezar por arriba, si no que sospechamos que nuestros problemas están por el medio. Para eso, podemos utilizar `trace`.

```
trace("lm", edit = TRUE)
```

```
as.list(body(lm))
trace(lm, tracer = browser, at = 5)
y <- runif(100)
x <- 1:100
```

```
lm(y ~ x)
## stop tracing
untrace(lm)
```

I.12.4. Warnings

En R puede haber algunas funciones que no den error, pero muestren warnings. En algunos casos, los warnings pueden indicar que algo no esté funcionando, por lo que se pueden convertir los warnings en errores para que la función no se ejecute.

```
opt <- options(warn = 2)
```

Este código hace que los warnings se comporten como errores. Una vez terminado, se puede reestablecer a los valores predeterminados mediante:

```
options(opt)
```

I.12.5. `where` para cuando uno está perdido en dónde está

A veces, cuando se hace debugging, especialmente cuando se está dentro de varias funciones que se llaman unas a otras, uno puede perderse y no saber dónde está. En estos casos, se utiliza `where`, que devuelve la función en la que nos encontramos.

```
debug(f1); debug(f2); debug(f3)
f3(4, 5) ## now, keeping pressing enter or n
         ## and you'll get deeper and deeper
         ## while in browser mode, type where

#Return things to normal
undebug(f3)
undebug(f2)
undebug(f1)
```

I.12.6. Protección frente a posibles fallos

Hay un manejo excepcional en R mediante `try`. Esto permite evitar que el código no falle. Cuando se va a dar un error, la variable adquiere la clase try-error.

```
ft <- function(x) {
    tmp <- try(log(x), silent = TRUE)
    if(inherits(tmp, "try-error")) {
        warning(paste("It looks like something did not work:\n",
```

```
                        "    ", tmp))
    } else{
        return(tmp)
    }
}

ft(9)

## [1] 2.197225

ft("a")

## Warning in ft("a"): It looks like something did not work:
##      Error in log(x) : Argumento no numérico para una función matemática
```

I.12.7. Funciones de debugging que no son exportadas

Si cargamos un paquete, sirve con `library(paquete)`. No obstante, puede haber algunas funciones no exportadas (no se ve directamente al teclear el nombre).

```
trace(randomForest:::predict.randomForest, edit = TRUE)
```

Las **funciones exportadas** son aquellas que el paquete pone a disposición del usuario final. Esto significa que cualquier persona que cargue el paquete puede llamar a estas funciones directamente. Así, cuando la función está exportada, el usuario solo necesita escribir el nombre de la función para ejecutarla, siempre que el paquete esté cargado.

Las **funciones no exportadas** son aquellas que están presentes en el paquete pero no están pensadas para ser utilizadas por el usuario final. Estas funciones suelen ser de uso interno, y los desarrolladores del paquete las utilizan para realizar tareas auxiliares o para construir las funciones exportadas de manera modular. No aparecen en la lista de funciones del paquete y no son accesibles directamente.

I.13. Programación orientada a objetos: clases S3 y S4

En R hay varios sistemas de programación orientada a objetos. Los sistemas originales en R son los sistemas S3 y S4, siendo los más extendidos.

I.13.1. `methods`

```
methods('plot')
getAnywhere(plot.TukeyHSD)
#stats:::plot.TukeyHSD
```

`getAnywhere` permite obtener las funciones no exportadas. `plot` realmente no hace nada, solo determina el tipo de objeto que se le ha pasado y llama a la función específica para ese objeto.

Lo que se ve con `methods` depende de los paquetes que haya cargados y, por tanto, lo que haya en nuestro espacio de búsqueda.

En POO en R, no se define una clase dentro de la que definir métodos (como en Python). Los métodos no pertenecen a la clase, si no que hay que definirlos por separado.

Se pueden buscar todos los métodos de una clase con:

```
methods(class = 'lm')
methods(class = 'lm', byclass = FALSE)
```

El argumento `byclass` muestra el nombre completo de los métodos y si están o no exportados.

El código fuente de todas las funciones está disponible. Para todas las funciones S3 exportadas desde el namespace, se puede escribir el nombre del método en la línea de comando como generic.class. Para las funciones no exportadas, se puede utilizar `getAnywhere` o `getS3method`. Sabiendo el namespace, también se puede utilizar :::.

```
add1.lm
getAnywhere('add1.lm')
stats:::add1.lm
getS3method('add1', 'lm')
```

I.13.2. Creación de clases y métodos

Vamos a suponer que queremos trabajar con unos data frames especiales con información sobre colesterol, la expresión de un gen y el tipo de experimento que lo midió. Lo primero que se quiere es convertir data frames en objetos de mi clase (y más adelante convertir matrices o vectores a objetos), crear un summary de los objetos y ajustar funciones de plot. Finalmente, hay que testear el código mediante la librería `testthat`.

Empezamos con una función genérica de conversión al objeto y luego un método que funcione para los data frames. El objeto será Cholest_Gene object, por lo que un conversor genérico (y sus comentarios) sería:

```
# object -> Cholest_Gene object
# General converter to Cholest_Gene object.
```

```r
to_CG <- function(x, ...) {
  UseMethod("to_CG")
}
```

El primer método será convertir un data frame al objeto:

```r
# data.frame -> Cholest_Gene object
# Take a data frame and return (if possible) a Cholest_Gene object.
to_CG.data.frame <- function(x) {
  cns <- c("Cholesterol", "Gene", "Kind")
  if (!(all(colnames(x) %in% cns)))
        stop(paste("Column names are not ", cns))
    tmp <- x[, cns]
    ## Notice I do not set this to be of data.frame class
    class(tmp) <- c("Cholest_Gene")
    return(tmp)
}
```

Y se debe probar:

```r
uu <- to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20, Kind = "Cl1"))
uu
```

El resultado de la visualización es muy feo y se debe mejorar. Esto se debe a que la clase es Cholest_Gene y, por ello, utiliza print default. Por ello, se debe cambiar la clase del objeto a la clase Cholest_Gene, adjuntando la clase que tenía previamente:

```r
# data.frame -> Cholest_Gene object
# Take a data frame and return (if possible) a Cholest_Gene object.
to_CG.data.frame <- function(x) {
    cns <- c("Cholesterol", "Gene", "Kind")
    if (!(all(colnames(x) %in% cns)))
        stop(paste("Column names are not ", cns))
    tmp <- x[, cns]
    tmp$Kind <- factor(tmp$Kind)
    class(tmp) <- c("Cholest_Gene", class(tmp)) ## "data.frame")
    return(tmp)
}
```

De esta forma, la visualización del objeto está bien, al igual que la salida de summary, reutilizando así las funciones existentes.

```r
uu <- to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20, Kind = "Cl1"))
summary(uu)
print(uu)
uu
```

La siguiente función es más sofisticada:

```
# Cholest_Gene object -> printed Cholest_Gene object
# Print a Cholest_Gene object.
print.Cholest_Gene <- function(x) {
    u <- x[, c(1, 2)]
    class(u) <- "data.frame"
    print(u)
    cat("\n Printing summary of first column \n")
    print(summary(x[, 1]))
}
```

En este caso, se asigna la clase data.frame (y solo esa clase) para evitar que se llame a sí mismo.

Como por el momento solo se ha creado el método para convertir data frames a nuestro objeto, se debe comprobar que el objeto que se pase sea de una clase soportada (data frame) y no se ejecute cuando la clase (por ejemplo, matriz) no está soportada. Además, esto muestra un mensaje de error personalizado.

```
# arbitrary object -> failure message if no method
# Return error message if there is no specific method to convert
# from that class to Cholest_Gene class
to_CG.default <- function(x) {
    stop("For now, only methods for data.frame are available.")
}
```

I.13.3. Testeo y test-driven development

El último paso es el testeo, y es algo fundamental. En los tests se van poniendo casos en los que se encuentran bugs y se arreglan. Como mínimo, se debe comprobar que se puede crear un objeto legítimo de un data frame, que falla (como esperamos) cuando al data frame le faltan columnas y que falla (como esperamos) cuando no se proporciona un data frame. El testeo se puede llevar a cabo con el paquete `testthat`, el cual tiene varios bloques de salidas que se pueden esperar (`expect_s3_class`, `expect_error`, `expect_equal`, `...`)

```
library(testthat)
test_that("minimal conversions and failures", {

    expect_s3_class(to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20,
                                     Kind = "Cl1")), "Cholest_Gene")


    expect_error(to_CG(cbind(Cholesterol = 1:10, Gene = 11:20)),
                 "For now, only methods for data.frame are available",
                 fixed = TRUE)
```

```
    expect_error(to_CG(data.frame(Cholesterol = 1:10, Geni = 11:20,
                                  Kind = "Cl1")),
                 "Column names are not ",
                 fixed = TRUE)

    expect_s3_class(to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20,
                                  Kind = "Cl1",
                                  whatever = "abcd")),
                    "Cholest_Gene")

})
```

El último bloque de test ha fallado, por lo que hay que hacer debugging.

```
debugonce(to_CG.data.frame)

dummy <- to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20,
                                Kind = "Cl1",
                                whatever = "abcd"))
```

En este entorno se va viendo qué va pasando en cada línea de código, y se verifica dónde está el problema. En este caso, el if comprueba si todos los nombres de columnas están en cns, cuando debería ser al revés: que todos los nombres de cns estén en el data frame (aunque haya otras columnas adicionales). Por tanto, hay que reescribir la función invirtiendo eso:

```
to_CG.data.frame <- function(x) {
    cns <- c("Cholesterol", "Gene", "Kind")
    if (!(all(cns %in% colnames(x))))
        stop(paste("Column names are not ",
                   paste(cns, collapse = " ")))
    tmp <- x[, cns]
    tmp$Kind <- factor(tmp$Kind)
    class(tmp) <- c("Cholest_Gene", class(x))
    return(tmp)
}
```

Y volvemos a ejecutar el bloque de comprobaciones:

```
test_that("minimal conversions and failures", {
    expect_s3_class(to_CG(data.frame(Cholesterol = 1:10,
                                  Gene = 11:20,
                                  Kind = "Cl1")),
                    "Cholest_Gene")
    expect_error(to_CG(cbind(Cholesterol = 1:10, Gene = 11:20)),
                 "For now, only methods for data.frame are available",
```

```
                  fixed = TRUE)
    expect_error(to_CG(data.frame(Cholesterol = 1:10, Geni = 11:20,
                                  Kind = "Cl1")),
                  "Column names are not",
                  fixed = TRUE)
    expect_s3_class(to_CG(data.frame(Cholesterol = 1:10, Gene = 11:20,
                                     Kind = "Cl1",
                                     whatever = "abcd")), "Cholest_Gene")
})
```

I.13.4. Creación de función de plot

```
## Cholest_Gene object -> ggplot object
## Produce a ggplot of a Cholest_Gene object.
plot.Cholest_Gene <- function(x, ...) {
  class(x) <- "data.frame"
  require(ggplot2)
  ## FIXME: should I explicitly print? Hummm.. return, as orthodox?
  if (nlevels(x$Kind) >= 2)
    p1 <- ggplot(aes(y = Cholesterol, x = Gene, col = Kind),
                 data = x) +
      facet_grid(~ Kind)
  else
    p1 <- ggplot(aes(y = Cholesterol, x = Gene), data = x)
  p1 <- p1 + geom_point()
  return(p1)
}
```

En R, cuando se pasa un argumento, tan pronto como se utiliza en el interior de la función, se hace una copia. Así, cuando se modifica la clase dentro de una función, no se altera el argumento original, solo la copia interna.

I.13.5. Clases S4

Las clases S4 se utilizan en algunos paquetes de BioConductor. Funcionan de forma similar a las clases S3, pero son más formales y rigurosas.

```
library(Matrix)
m1 <- Matrix(1:9, nrow = 3)
m2 <- Diagonal(5)

x <- 0:10
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
fit1 <- lm(y ~ x)
```

```r
class(fit1)
```

```
## [1] "lm"
```

```r
is.list(fit1)
```

```
## [1] TRUE
```

```r
isS4(fit1)
```

```
## [1] FALSE
```

```r
print(fit1)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##      19.955       -1.682
```

```r
stats:::print.lm(fit1)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##      19.955       -1.682
```

```r
fit1
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##      19.955       -1.682
```

```r
names(fit1)
```

```
##  [1] "coefficients"  "residuals"     "effects"
##  [4] "rank"          "fitted.values" "assign"
##  [7] "qr"            "df.residual"   "xlevels"
## [10] "call"          "terms"         "model"

fit1$coefficients

## (Intercept)           x
##   19.954545   -1.681818

## don't do that for real. Use coefficients
coefficients(fit1)

## (Intercept)           x
##   19.954545   -1.681818

isS4(m1)

## [1] TRUE

is.list(m1)

## [1] FALSE

class(m1)

## [1] "dgeMatrix"
## attr(,"package")
## [1] "Matrix"

slotNames(m1)

## [1] "Dim"      "Dimnames" "x"        "factors"

slotNames(m2)

## [1] "diag"     "Dim"      "Dimnames" "x"

m1

## 3 x 3 Matrix of class "dgeMatrix"
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
m1@Dim

## [1] 3 3

m1@x

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

m2@Dim

## [1] 5 5
```

I.13.6. Resumen sobre la programación orientada a objetos en R

Es recomendable familiarizarse con las clases S3 en R. En BioConductor, es posible encontrarse con las clases S4, pero en general se puede ejecutar todo con clases S3. Hay otras clases, como las R6, pero tienen un uso muy concreto en situaciones muy específicas.