

# Subconjuntos de diferentes estructuras de datos

Esta sección está basada en Wickham (2014, Subsetting) disponible en [línea](#).

Aprender a extraer subconjuntos de los datos es importante y permite realizar operaciones complejas con los mismos. De los conceptos importantes que se deben aprender son

- Los operadores para extraer subconjuntos (subsetting operators)
- Los 6 tipos de extracciones de subconjuntos
- Las diferencias a la hora de extraer subconjuntos de las diferentes estructuras de datos (factores, listas, matrices, dataframes)
- El uso de la extracción de subconjuntos junto a asignar variables.

Cuando tenemos que extraer pedazos de los datos (o analizar solamente parte de éstos), necesitamos complementar `str()` con `[[`, es decir, la estructura nos dirá cómo utilizar el operador subconjunto de manera que de hecho extraigamos lo que queremos.

## Operadores para extraer subconjuntos

Dependiendo la estructura de datos que tenemos, será la forma en la que extraemos elementos de ella. Hay dos operadores de subconjunto: `[[` y `$`. `[[` se parece a `[` pero regresa un solo valor y te permite sacar pedazos de una lista. `$` es un atajo útil para `[[`.

### Vectores atómicos

¿De qué formas puedo extraer elementos de un vector? Hay varias maneras **sin importar** la *clase* del vector.

- **Enteros positivos** regresan los elementos en las posiciones especificadas en el orden que especificamos.

```
x <- c(5.6, 7.8, 4.5, 3.3)
```

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

```
## [1] 4.5 5.6
```

```
## Si duplicamos posiciones, nos regresa resultados duplicados
```

```
x[c(1, 1, 1)]
```

```
## [1] 5.6 5.6 5.6
```

```
## Si usamos valores reales, se coercion a entero
```

```
x[c(1.1, 2.4)]
```

```
## [1] 5.6 7.8
```

```
x[order(x)]
```

```
## [1] 3.3 4.5 5.6 7.8
```

```
x[order(x, decreasing = T)]
```

```
## [1] 7.8 5.6 4.5 3.3
```

- **Enteros negativos** omiten los valores en las posiciones que se especifican.

```
x
```

```
## [1] 5.6 7.8 4.5 3.3
```

```
x[-c(3, 1)]
```

```
## [1] 7.8 3.3
```

*Mezclar* no funciona.

```
x[c(-3, 1)]
```

- **Vectores lógicos** selecciona los elementos cuyo valor correspondiente es **TRUE**. Esta es una de los tipos más útiles.

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

```
## [1] 5.6 7.8
```

```
x[c(TRUE, FALSE)] # Autocompleta el vector lógico al tamaño de x
```

```
## [1] 5.6 4.5
```

```
x[c(TRUE, TRUE, NA, FALSE)]
```

```
## [1] 5.6 7.8 NA
```

- **Nada** si no especifico nada, me regresa el vector original

```
x[]
```

```
## [1] 5.6 7.8 4.5 3.3
```

- **Cero** el índice cero no aplica en R, te regresa el vector vacío

```
x[0]
```

```
## numeric(0)
```

- Si el vector tiene **nombres** también los puedo usar.

```
names(x) <- c("a", "ab", "b", "c")
x["ab"]
```

```
## ab
## 7.8
```

```
x["d"]
```

```
## <NA>
## NA
```

```
x[grep("a", names(x))]
```

```
## a ab
## 5.6 7.8
```

Las **listas** operan básicamente igual a vectores recordando que si usamos `[` regresa una lista y tanto `[[` y `$` extrae componentes de la lista.

## Matrices y arreglos

Para estructuras de mayor dimensión se pueden extraer de tres maneras:

- Con vectores múltiples
- Con un solo vector
- Con una matriz

```
m <- matrix(1:12, nrow = 3)
colnames(m) <- LETTERS[1:4]
m[1:2, ]
```

```
##      A B C D
## [1,] 1 4 7 10
## [2,] 2 5 8 11
```

```
m[c(T, F, F), c("B", "C")]
```

```
## B C
## 4 7
```

```
m[1, 4]
```

```
## D
## 10
```

Como ven, es solamente generalizar lo que se hace en vectores replicándolo al número de dimensiones que se tiene.

```
m[c(T, F, F)]
```

```
## [1] 1 4 7 10
```

```
class(m[c(T, F, F)])
```

```
## [1] "integer"
```

[ simplifica al objeto. En matriz, me quita la dimensionalidad, en listas me da lo que esta dentro de esa celda.

## Dataframes

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
```

```
df[c(1, 2), ]
```

```
##   x y z  
## 1 1 3 a  
## 2 2 2 b
```

```
df[, c(1, 2)]
```

```
##   x y  
## 1 1 3  
## 2 2 2  
## 3 3 1
```

```
df[, c("z", "x")]
```

```
##   z x  
## 1 a 1  
## 2 b 2  
## 3 c 3
```

```
df[c("z", "x")]
```

```
##   z x  
## 1 a 1  
## 2 b 2  
## 3 c 3
```

```
class(df[, c("z", "x")])
```

```
## [1] "data.frame"
```

```
class(df[c("z", "x")])
```

```
## [1] "data.frame"
```

```
str(df["x"])
```

```
## 'data.frame':    3 obs. of  1 variable:  
## $ x: int  1 2 3
```

```
str(df[, "x"])
```

```
## int [1:3] 1 2 3
```

```
str(df$x)
```

```
## int [1:3] 1 2 3
```



## Ejercicios

1. Utiliza la base mtcars
2. Arregla los errores al extraer subconjuntos en dataframes

```
mtcars[mtcars$cyl == 4, ]  
mtcars[-1:4, ]  
mtcars[mtcars$cyl <= 5]  
mtcars[mtcars$cyl == 4 | 6, ]
```

3. ¿Por qué al correr  $x \leftarrow 1:5$ ;  $x[NA]$  obtengo valores perdidos?
4. Genera una matriz cuadrada tamaño 5 llamada m. ¿Qué te da correr  $m[\text{upper.tri}(m)]$ ?
5. ¿Por qué al realizar  $\text{mtcars}[1:20]$  me da un error? ¿Por qué  $\text{mtcars}[1:2]$  no me lo da? ¿Por qué  $\text{mtcars}[1:20, ]$  es distinto?
6. Haz una función que extraiga la diagonal de la matriz m que creaste antes. Debe dar el mismo resultado que ejecutar  $\text{diag}(m)$
7. ¿Qué hace  $\text{df}[\text{is.na}(df)] <- 0$ ?

## Asignar a un subconjunto

Muchas veces lo que necesitamos es encontrar ciertos valores para poder reemplazarlos con algo más. Por ejemplo, muchas veces queremos imputar valores perdidos con cierto valor.

```
# Variables continuas  
x <- c(1, 2, 3, NA, NaN, 7)  
media <- mean(x, na.rm = T)  
media
```

```
## [1] 3.25
```

```
x[is.na(x)] <- media
x
```

```
## [1] 1.00 2.00 3.00 3.25 3.25 7.00
```

```
# Variables discretas
x <- c(rep("azul", 3), "verde", NA, "verde", rep("rojo", 4))
x
```

```
## [1] "azul" "azul" "azul" "verde" NA "verde" "rojo" "rojo"
## [9] "rojo" "rojo"
```

```
moda <- names(table(x))[which(table(x) == max(table(x)))] # Engorroso, no?
x[is.na(x)] <- moda
x
```

```
## [1] "azul" "azul" "azul" "verde" "rojo" "verde" "rojo" "rojo"
## [9] "rojo" "rojo"
```

```
# Puedo reemplazar partes de un vector
x <- 1:5
x[c(1, 2)] <- 2:3
x
```

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

```
# Las longitudes de las asignaciones tienen que ser iguales
x[-1] <- 4:1
x
```

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

```
# No se revisan duplicados
x[c(1, 1)] <- 2:3
x
```

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

```
# Puedo sustituir valores considerando toda la logica
x <- c(1:10)
x[x > 5] <- 0
x
```

```
## [1] 1 2 3 4 5 0 0 0 0 0
```

Por último, es útil notar la utilidad de asignar utilizando la forma de asignar **nada** mencionada anteriormente.

```
class(mtcars)
```

```
## [1] "data.frame"
```

```
mtcars[] <- lapply(mtcars, as.integer)  
class(mtcars)
```

```
## [1] "data.frame"
```

```
dim(mtcars)
```

```
## [1] 32 11
```

```
mtcars <- lapply(mtcars, as.integer)  
class(mtcars)
```

```
## [1] "list"
```

```
dim(mtcars)
```

```
## NULL
```

Asignar utilizando el operador de suconjunto a nada nos permite preservar la estructura del objeto original así como su clase.

En el caso de listas, si combinamos un operador de suconjunto mas asignación a nulo, podemos remover objetos de ésta.

```
x <- list(a = 1, b = 2)  
x[[2]] <- NULL  
str(x)
```

```
## List of 1  
## $ a: num 1
```

```
x["b"] <- list(NULL)  
str(x)
```

```
## List of 2  
## $ a: num 1  
## $ b: NULL
```

## Operadores lógicos

**Ejemplo:** Supongamos que queremos saber qué elementos de  $x$  son menores que 5 y mayores que 8.

Operador	Descripción
<	menor que
<=	menor o igual que
>	mayor que
==	exactamente igual que
!=	diferente de
!x	no x
x   y	x O y
x & y	x Y y
isTRUE(x)	checa si x es verdadero

```
x <- c(1:10)
x[(x>8) | (x<5)]
```

```
## [1] 1 2 3 4 9 10
```

```
# ¿Cómo funciona?
x
```

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

```
x > 8
```

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

```
x < 5
```

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

```
x > 8 | x < 5
```

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

```
# x > 8 || x < 5
x[c(T,T,T,T,F,F,F,T,T)]
```

```
## [1] 1 2 3 4 9 10
```

|| vs. | y && vs. &

La diferencia entre & y && (o | y ||) es que el primero es vectorizado y el segundo no.

**Ejercicio** ¿Qué crees que pasa en las siguientes situaciones?

```
rm(list = ls())
TRUE || a
FALSE && a
TRUE && a
TRUE | a
FALSE & a
```



La forma larga (la versión doble) no parece ser muy útil. El propósito de ésta es que es más apropiado cuando se programa usando estructuras de control, por ejemplo, en `ifs`.\*

```
if( c(T, F) ) print("Hola")
```

```
## [1] "Hola"
```

Poner el `&&` me garantiza que la condicional será evaluado sobre un único valor falso/verdadero.

```
(-2:2) >= 0
```

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

```
(-2:2) <= 0
```

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

```
((-2:2) >= 0) && ((-2:2) <= 0)
```

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

**Ejercicio** Explora los siguientes comandos

```
impares <- 1:10 %% 2 == 1
mult.3 <- 1:10 %% 3 == 0

impares & mult.3
impares | mult.3

xor(impares, mult.3)
```

## ¿Por qué tanto detalle? Aplicaciones

Una de las formas más fáciles de frustrarse con R (y con cualquier otro lenguaje) es no saber decirle al lenguaje lo que se desea hacer. Entender cómo manipular las estructuras de datos y la lógica detrás de su comportamiento ahorra mucho sufrimiento y permite adaptarse ante cosas que necesitamos que aún no se encuentran implementadas por alguien más de una manera más sencilla.

Con saber de subconjuntos podemos realizar varias tareas indispensables.

## Buscar o buscarh

Excel es excelente haciendo estas tareas. Lo malo de excel es que no es **reproducible**. Es muy común que resulte imposible llegar de los datos originales al resultado final pues muchos pasos intermedios de limpieza no están documentados de forma alguna. Un *script* de limpieza nos permite no solamente ir del *raw* a la estructura de datos limpia y analizable sino que permite que alguien más verifique las operaciones que se están realizando, se identifiquen errores y que, cuando nos llega un nuevo mes, sea trivial incluir estos datos al resultado final.

```
rm(list = ls())
x <- c("m", "f", "u", "f", "f", "m", "m")
busca <- c(m = "Male", f = "Female", u = NA)
busca[x]
```

```
##           m           f           u           f           f           m           m
##  "Male" "Female"      NA "Female" "Female"  "Male"  "Male"
```

```
unname(busca[x])
```

```
## [1] "Male"  "Female" NA      "Female" "Female" "Male"  "Male"
```

```
c(m = "humano", f = "humano", u = "desconocido")[x]
```

```
##           m           f           u           f           f
##  "humano"  "humano" "desconocido"  "humano"  "humano"
##           m           m
##  "humano"  "humano"
```

Esto nos permite pegar un vector a una base de datos de acuerdo a una condicion.

```
calificaciones <- c(10, 9, 5, 5, 6)
aprueba <- data.frame(
  calificacion = 10:1,
  descripcion = c(rep("excelente", 2), "bueno", rep("aceptable", 2), rep("no satisfactorio", 5)),
  aprobatorio = c(rep(T, 5), rep(F, 5))
)
id <- match(calificaciones, aprueba$calificacion)
aprueba[id, ]
```

```
##      calificacion      descripcion aprobatorio
## 1             10      excelente          TRUE
## 2              9      excelente          TRUE
## 6              5 no satisfactorio          FALSE
## 6.1            5 no satisfactorio          FALSE
## 5              6      aceptable          TRUE
```



## Ejercicios

1. Realiza la misma operación con las calificaciones pero utilizando los nombres de las filas, es decir, los rownames(aprueba)
2. Carga la libreria ggplot2 y utiliza la base de datos diamonds
3. Utiliza el comando match para quedarte con las variables cut y x
4. Genera la variable categórica tal que, si el precio es mayor que 5,000 el valor de price.cat es cara, si es mayor que 2,000 es normal y barata en otro caso.

## Muestras aleatorias

Podemos utilizar índices enteros para generar muestras aleatorias de nuestras bases de datos o de nuestros vectores.

```
set.seed(102030)
aprueba[sample(nrow(aprueba)), ]
```

```
##      calificacion      descripcion aprobatorio
## 10             1 no satisfactorio      FALSE
##  2             9      excelente      TRUE
##  8             3 no satisfactorio      FALSE
##  7             4 no satisfactorio      FALSE
##  9             2 no satisfactorio      FALSE
##  3             8      bueno      TRUE
##  1            10      excelente      TRUE
##  6             5 no satisfactorio      FALSE
##  4             7      aceptable      TRUE
##  5             6      aceptable      TRUE
```

```
aprueba[sample(nrow(aprueba), replace = T, size = 5), ]
```

```
##      calificacion      descripcion aprobatorio
## 4             7      aceptable      TRUE
## 1            10      excelente      TRUE
## 4.1          7      aceptable      TRUE
## 7             4 no satisfactorio      FALSE
## 2             9      excelente      TRUE
```



### Ejercicios

1. Utiliza la base de datos de iris y genera un conjunto de prueba y uno de entrenamiento correspondientes al 20 y 80 % de los datos, respectivamente.
2. Genera un vector `x` de tamaño 1000 con realizaciones de una normal media 10, varianza 3.
3. Crea 100 muestras bootstrap del vector `x`.
4. Calcula la media para cada una de tus muestras.
5. Grafica con la función `hist()` el vector de medias de tus muestras.
6. Genera un vector `l` de letras, tamaño 10 y ordénalo. (Usa `letters` y `order`).
7. Ordena la base `cars` de acuerdo a distancia, en forma descendiente (muestra la cola -usa `tail`- de la base ordenada).

## Expande bases

Ahora, a veces tenemos tablas de resumen pero quisieramos extraer los datos originales. Combinamos `rep` con subconjuntos de enteros para expandir.

```
df <- data.frame(
  color = c("azul", "verde", "amarillo"),
  n = c(4, 3, 5)
)
df
```

```
##      color n
## 1     azul 4
## 2    verde 3
## 3 amarillo 5
```

```
df[rep(1:nrow(df), df$n), ]
```

```
##      color n
## 1     azul 4
## 1.1    azul 4
## 1.2    azul 4
## 1.3    azul 4
## 2     verde 3
## 2.1    verde 3
## 2.2    verde 3
## 3  amarillo 5
## 3.1  amarillo 5
## 3.2  amarillo 5
## 3.3  amarillo 5
## 3.4  amarillo 5
```

## Otras

Ya estuvimos utilizando otras aplicaciones de estos comandos: ordenamientos, selección de filas o columnas según una condición lógica.

También utilizamos un comando muy útil llamado `which`.

```
set.seed(45)
x <- sample(letters, 10)
x <= "e"
```

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

```
which(x <= "e")
```

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

Junto con `which`, puedes usar `intersect` y `union`.

```
pares <- 1:10 %% 2 == 0
m.5 <- 1:10 %% 5 == 0

c(1:10)[union(which(pares), which(m.5))]
c(1:10)[intersect(which(pares), which(m.5))]
c(1:10)[which(xor(pares, m.5))]
```

## Split-apply-combine

El paradigma split-apply-combine se resume en la figura 1.

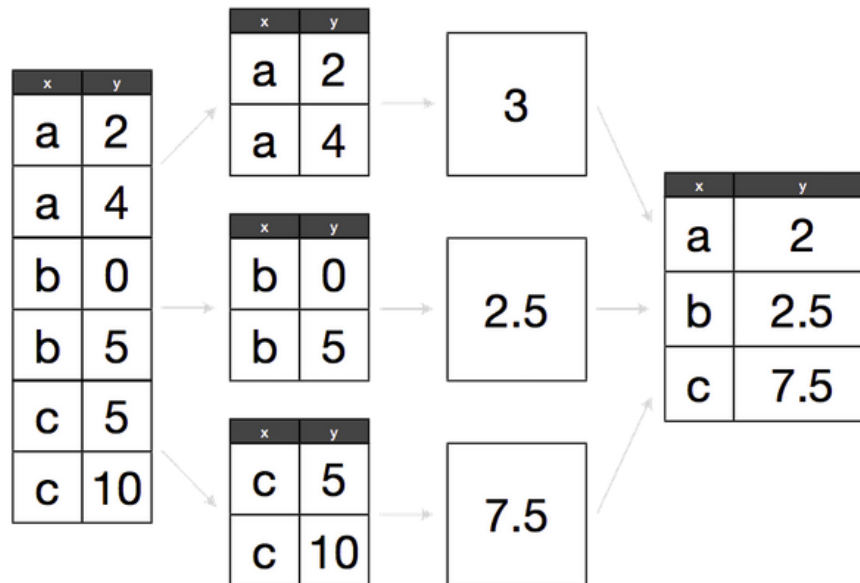


Figura 1: Ejemplificación del split-apply-combine Vaidyanathan (2014, Split-Apply-Combine).

Entendamos mejor:

```
letras <- sample(letters, 3)
df <- data.frame(
  letra = letras[rep(seq(letras), 4)],
  valor = sample(1:10, size = 12, replace = T)
)
df <- df[order(df$letra), ]
df
```

```
##      letra valor
## 3         h     5
## 6         h     5
## 9         h     4
## 12        h     9
## 1         j     5
## 4         j    10
## 7         j     3
## 10        j     6
## 2         w     5
## 5         w     2
## 8         w     2
## 11        w     7
```

Queremos estimar la media del valor para cada tipo de letra.

```
# Dividimos
for (l in unique(df$letra) ){
  print(df[l == df$letra, ])
}
```

```
##      letra valor
## 3         h     5
## 6         h     5
## 9         h     4
## 12        h     9
##      letra valor
## 1         j     5
## 4         j    10
## 7         j     3
## 10        j     6
##      letra valor
## 2         w     5
## 5         w     2
## 8         w     2
## 11        w     7
```

```
# Aplicamos
for (l in unique(df$letra) ){
  print(mean(df[l == df$letra, ]$valor))
}
```

```
## [1] 5.75
## [1] 6
## [1] 4
```

```
# Combinamos
medias <- list()
for (l in unique(df$letra) ){
  medias[[l]] <- mean(df[l == df$letra, ]$valor)
}
as.data.frame(list(letras = names(medias), medias = unname(unlist(medias))))
```

```
##      letras medias
## 1         h   5.75
## 2         j   6.00
## 3         w   4.00
```

R tiene muchas funciones que facilitan realizar este tipo de operaciones. En particular, la familia `apply` fue pensada para realizar ese tipo de operaciones.

## apply

`apply` aplica una función a cada fila o columna en una matriz.

```
m <- matrix(c(1:5, 6:10), nrow = 5, ncol = 2)
# 1 is the row index 2 is the column index
m
```

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

```
apply(m, 1, sum)
```

```
## [1]  7  9 11 13 15
```

```
apply(m, 2, sum)
```

```
## [1] 15 40
```

**Ejercicio** Haz una función que reciba un vector y devuelva la suma de la posición  $v_i + v_{i+1}$ . Para el n-esimo elemento, suma el primero. Aplica esa función a las columnas y filas de la matriz m.

## lapply

`lapply` aplica una función a cada elemento en una lista. Como sabemos, un `data.frame` es únicamente un estilo particular de lista tal que todos sus elementos tienen el mismo tamaño. Por ende, también podemos utilizar `lapply` para iterar sobre las columnas de un `data.frame`.

```
lista <- list(a = 1:10, b = 2:20)
lapply(lista, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 11
```

```
df <- data.frame(a = 1:10, b = 11:20)
lapply(df, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 15.5
```

El `summary` de un `data.frame` genera un resumen para los vectores que la conforman de acuerdo a la clase de la misma. Genera una función que regrese una tabla de frecuencias para factores y caracteres o una lista con media, desviación estándar para vectores numéricos o enteros. Aplícalo a la base `diamonds` usando `lapply`.

## sapply

sapply es otra versión de lapply que regresa una lista de una matriz cuando es apropiado.

```
x <- sapply(lista, mean, simplify = F)
x
```

```
## $a
## [1] 5.5
##
## $b
## [1] 11
```

```
x <- sapply(lista, mean, simplify = T)
x
```

```
##      a      b
## 5.5 11.0
```

**Ejercicio** Obtén un vector tipo caracter con los nombres de las clases de las columnas de iris.

**Ejercicio** Repite el ejercicio de la suma rara pero usa sapply.

*Recuerda la instrucción:* Haz una función que reciba un vector y devuelva la suma de la posición  $v_i + v_{i+1}$ . Para el n-esimo elemento, suma el primero. Utiliza sapply para realizar esta operacion.

## mapply

mapply es como la versión multivariada de sapply. Le aplica una función a todos los elementos correspondientes de un argumento.

```
l1 <- list(a = c(1:5), b = c(6:10))
l2 <- list(c = c(11:15), d = c(16:20))
l1
```

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

```
l2
```

```
## $c
## [1] 11 12 13 14 15
##
## $d
## [1] 16 17 18 19 20
```

```
mapply(sum, l1$a, l1$b, l2$c, l2$d)
```

```
## [1] 34 38 42 46 50
```



```
11[["a"]][1] + 11[["b"]][1] + 12[["c"]][1] + 12[["d"]][1]
```

```
## [1] 34
```

## tapply

tapply le aplica una función a subconjuntos de un vector.

```
head(warpbreaks)
```

```
##   breaks wool tension
## 1     26    A        L
## 2     30    A        L
## 3     54    A        L
## 4     25    A        L
## 5     70    A        L
## 6     52    A        L
```

```
with(warpbreaks, tapply(breaks, list(wool, tension), mean))
```

```
##           L           M           H
## A 44.55556 24.00000 24.55556
## B 28.22222 28.77778 18.77778
```

```
tapply(warpbreaks$breaks,
       list(wool = warpbreaks$wool, tension = warpbreaks$tension),
       mean)
```

```
##      tension
## wool      L      M      H
##   A 44.55556 24.00000 24.55556
##   B 28.22222 28.77778 18.77778
```

## Ejercicio

### by

by le aplica una función a subconjuntos de un `data.frame`. Se divide un `data.frame` según los valores de uno o más factores. Se aplica la función FUN a cada subconjunto.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1         5.1         3.5         1.4         0.2   setosa
## 2         4.9         3.0         1.4         0.2   setosa
## 3         4.7         3.2         1.3         0.2   setosa
## 4         4.6         3.1         1.5         0.2   setosa
## 5         5.0         3.6         1.4         0.2   setosa
## 6         5.4         3.9         1.7         0.4   setosa
```

```
by(data = iris[, 1:2], INDICES = iris[, "Species"], FUN = summary)
```

```
## iris[, "Species"]: setosa
##   Sepal.Length   Sepal.Width
##   Min.    :4.300   Min.    :2.300
##   1st Qu.:4.800   1st Qu.:3.200
##   Median :5.000   Median :3.400
##   Mean    :5.006   Mean    :3.428
##   3rd Qu.:5.200   3rd Qu.:3.675
##   Max.    :5.800   Max.    :4.400
## -----
## iris[, "Species"]: versicolor
##   Sepal.Length   Sepal.Width
##   Min.    :4.900   Min.    :2.000
##   1st Qu.:5.600   1st Qu.:2.525
##   Median :5.900   Median :2.800
##   Mean    :5.936   Mean    :2.770
##   3rd Qu.:6.300   3rd Qu.:3.000
##   Max.    :7.000   Max.    :3.400
## -----
## iris[, "Species"]: virginica
##   Sepal.Length   Sepal.Width
##   Min.    :4.900   Min.    :2.200
##   1st Qu.:6.225   1st Qu.:2.800
##   Median :6.500   Median :3.000
##   Mean    :6.588   Mean    :2.974
##   3rd Qu.:6.900   3rd Qu.:3.175
##   Max.    :7.900   Max.    :3.800
```

Puedo calcular, por ejemplo, la suma de los valores del largo y ancho de los sépalos en la base de datos iris según la especie.

```
res <- by(iris[, c("Sepal.Length", "Sepal.Width")], iris[, "Species"], sum)
```

Posteriormente, se pueden combinar los elementos.

```
as.data.frame(list(
  "species" = names(res),
  "suma" = sapply(seq(length(res)), FUN = function(i) res[[i]])
))
```

```
##      species suma
## 1      setosa 421.7
## 2 versicolor 435.3
## 3  virginica 478.1
```

**Ejercicio** Vuelve a utilizar la base de diamonds para calcular el promedio de carat según cut y color.

```
## # A tibble: 6 × 10
##   carat      cut color clarity depth table price      x      y      z
##   <dbl>    <ord> <ord>    <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
```

```
## 1 0.23      Ideal      E      SI2 61.5    55    326 3.95 3.98 2.43
## 2 0.21    Premium      E      SI1 59.8    61    326 3.89 3.84 2.31
## 3 0.23      Good      E      VS1 56.9    65    327 4.05 4.07 2.31
## 4 0.29    Premium      I      VS2 62.4    58    334 4.20 4.23 2.63
## 5 0.31      Good      J      SI2 63.3    58    335 4.34 4.35 2.75
## 6 0.24 Very Good      J      VVS2 62.8    57    336 3.94 3.96 2.48
```

## replicate

`replicate` es una función muy útil sobretodo en el contexto de simulación.

```
replicate(5, rnorm(6), simplify = F)
```

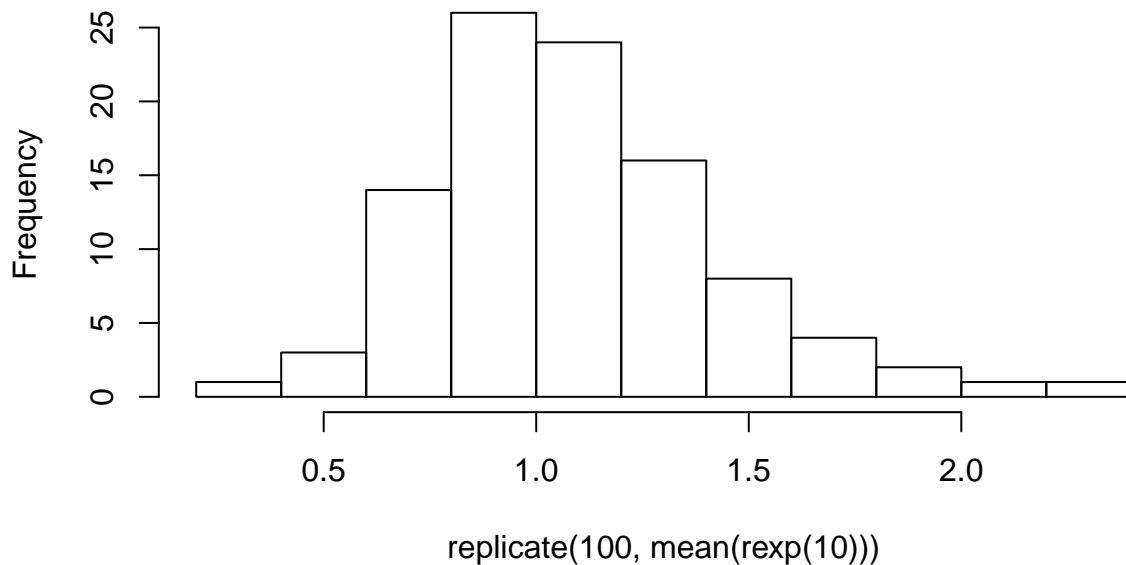
```
## [[1]]
## [1] -1.48174362 0.66978164 -0.78426278 -0.45197092 1.02655102 0.02008344
##
## [[2]]
## [1] -0.5126506 -0.4285076 0.5632417 0.1028410 0.1923036 1.1650204
##
## [[3]]
## [1] -0.18038327 0.33794788 0.08232277 -0.10389078 -1.14011737 -0.61650967
##
## [[4]]
## [1] 0.7039886 -2.2066521 -0.8376586 1.7216593 0.4360063 0.7571990
##
## [[5]]
## [1] 1.5398858 1.5393945 2.1829996 -0.9953758 -0.3013422 0.6005107
```

```
replicate(6, rnorm(4), simplify = T)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] -0.3633049 0.8675148 -1.3168208 1.5543102 0.6130339 0.5737528
## [2,] -0.2211350 -0.1700282 -0.6075526 0.3306465 0.7158021 0.2225771
## [3,] -0.1981113 1.1900491 -0.9454914 0.2031200 0.6501763 1.1414076
## [4,] 0.6676305 0.9188715 1.2493954 -2.5829970 0.7201334 -0.6722725
```

```
hist(replicate(100, mean(rexp(10))))
```

## Histogram of replicate(100, mean(rexp(10)))



**Ejercicio** Replica el ejercicio de muestras bootstrap utilizando replicate.

Recordando:

1. Genera un vector `x` de tamaño 1000 con realizaciones de una normal media 10, varianza 3.
2. Crea 100 muestras bootstrap del vector `x`.
3. Calcula la media para cada una de tus muestras.
4. Grafica con la función `hist()` el vector de medias de tus muestras.

### ¿Puede ser más fácil?

La familia `apply` viene con R básico. Sin embargo, hay 3 implementaciones excelentes del paradigma split-apply-combine: `plyr`, `dplyr` y `data.table`.

Si la familia `apply` es poderosa, se queda corta comparada con estos tres. `plyr` es la primera versión de s-a-c de Wickham. Posteriormente, mejoró muchas de las funciones en `dplyr` sobretodo entorno a velocidad y facilidad de uso. `plyr` no termina de ser relevante pues varias de sus funciones aun no están en `dplyr`.

`data.table`, es una implementación con una tradición muy diferente y tiene también funciones muy poderosas aunque con una sintaxis muy distinta a `dplyr`. Es absurdamente eficiente y tiene múltiples aplicaciones.

Muchas de las funciones en `dplyr` también están implementadas en `data.table`. Cuál usar es cuestión de gustos. Depende de con qué se acomoda cada quién pero, para algunas cosas uno es superior al otro y viceversa.

## Referencias

- [Vai14] Ramnath Vaidyanathan. *Introduction to R*. Github books, en pycon 2014, 2014. URL: <https://ramnathv.github.io/pycon2014-r>.
- [Wic14] Hadley Wickham. *Advanced R*. CRC Press, 2014.

```
sessionInfo()
```

```
## R version 3.3.1 (2016-06-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 15.04
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
##  [3] LC_TIME=es_MX.UTF-8      LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=es_MX.UTF-8  LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=es_MX.UTF-8     LC_NAME=C
##  [9] LC_ADDRESS=C             LC_TELEPHONE=C
## [11] LC_MEASUREMENT=es_MX.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets  base
##
## other attached packages:
## [1] ggplot2_2.1.0      xtable_1.8-2      rformat_0.1
## [4] rmarkdown_1.0.9016
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.7      digest_0.6.10    assertthat_0.1   plyr_1.8.4
##  [5] grid_3.3.1      gtable_0.2.0     formatR_1.4      magrittr_1.5
##  [9] scales_0.4.0     evaluate_0.9     stringi_1.1.1    tools_3.3.1
## [13] stringr_1.1.0    munsell_0.4.3    yaml_2.1.13      colorspace_1.2-6
## [17] htmltools_0.3.5  knitr_1.14       methods_3.3.1    tibble_1.2
```