Análisis y Tratamiento de Datos con R

Con ejemplos e ilustraciones

Primera Edición

Diego Paul Huaraca S.
MS-PLUS, Inc.

MS+
Soluciones
Matemáticas

Un aporte de Source Stat Lab Ecuador a la sociedad.

Índice general

1

Estructura de datos

En las secciones anteriores mencionamos que R es un lenguaje de programación orientado a objetos, por lo que cualquier cosa que exista en él tales como: variables, bases de datos, funciones, etc. son objetos. En este capítulo abordaremos los diferentes tipos de objetos que podemos crear en el programa, además revisaremos estructuras de almacenamiento de datos más complejas construidas a partir de estructuras sencillas.

1.1. Workspace

Durante una sesión de trabajo los objetos creados se almacenan por nombre y tipo en el espacio de trabajo (*workspace*), R recurrirá siempre en primera instancia a buscar en este *ambiente* los objetos que se soliciten a través de la consola.

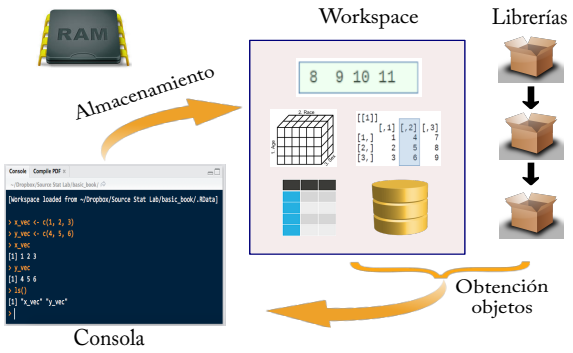


Figura 1.1: Funcionamiento espacio de trabajo

Para enlistar los objetos almacenados en el espacio de trabajo recurrimos a los comandos `ls()` o `objects()`.

```
# Inicialmente el espacio de trabajo se encuentra vacio
ls()

## character(0)

# Asignamos a x el valor 5
x <- 5
# Además asignamos a y el valor de x menos 2
```

```
y <- x-2
# Verificamos los objetos almacenados en el workspace
objects()

## [1] "x" "y"
```

Los objetos de R se muestran en la pestaña superior derecha de RStudio (*Environment*) a medida que son creados.

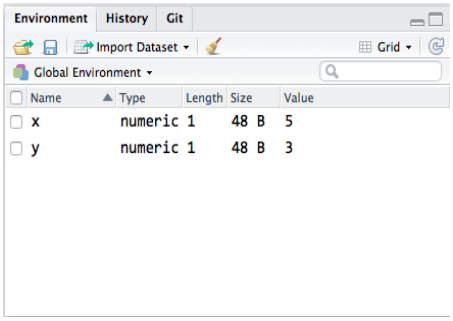


Figura 1.2: Workspace en RStudio

Debemos advertir que el operador de asignación ←, no es equivalente al operador habitual '='que se reserva para otro propósito, sino que consiste en dos caracteres '<'(*menor que*) y '-'(*guión*) que obligatoriamente deben ir unidos, además deben apuntar al objeto que recibe el valor de la expresión. Los usuarios principiantes de R suelen confundir estos dos conceptos por lo que en ciertas ocasiones presentan conflictos de interpretabilidad.

```
# Asignamos a w el valor de 8
x_new <- 8
# Tambien podemos asignar del siguiente modo
9 -> y_new
```

La última asignación pese a que es permitida por el lenguaje, no es muy recomendable debido que en la actualidad la tendencia es estandarizar la forma de escritura en R.

Una alternativa para asignar valores sin pasar por el operador (*flecha*), lo constituye la función `assign()`, como se muestra en el ejemplo siguiente:

```
# Asignamos a x_new el valor de 7
assign("z", 7)
# Visualizamos la asignacion
z

## [1] 7

# Mostramos los elementos creados
objects()

## [1] "x"     "x_new" "y"     "y_new" "z"
```

Los nombres que se les asigna a los objetos en R pueden ser de cualquier longitud, y a su vez pueden combinar letras, números y caracteres especiales (coma, punto, guión bajo, etc.), la única

exigencia al momento de asignar un nombre a un objeto es que el mismo inicie con una letra mayúscula o minúscula (vale la plena en este punto aclarar que R diferencia mayúsculas de minúsculas).

Recomendados	No permitidos
x, xVar, x.1	1x, $x, .xvar
x01, x_var, x_1	x___var, x#1, x-1

La construcción explícita de un objeto nos proporcionará un mejor entendimiento de su estructura, y nos permitirá ahondar en algunas nociones mencionadas previamente.

Las estructuras de datos en R pueden ser organizadas por su dimensionalidad (1 dimensión, 2 dimensiones o n-dimensiones), así como también por su tipo (homogéneo, heterogéneo), lo anterior da lugar a 6 tipos de estructuras que se resumen a continuación:

Dimensión	**1-d**	**2-d**	**n-d**
Homogéneo	Vector	Matriz	Array
Heterogéneo	Lista	Data Frame / Data Table	

Cuadro 1.1: Estructura de datos

En la actualidad ha tomado fuerza el uso de data tables a diferencias de data frames, esto debido a las diversas facilidades que poseen los primeros sobre la manipulación de grandes cantidades de información (incorporación del almacenamiento en disco).

1.2. Vectores

Es la estructura más simple de R que sirve para almacenar un conjunto de valores del mismo o diferente tipo, llamados *elementos*.

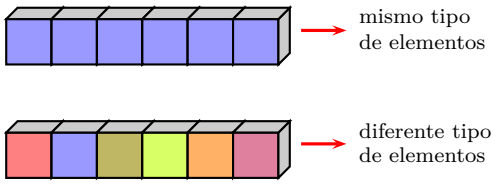


Figura 1.3: Vectores

Existen 6 tipos de elementos que R puede almacenar dentro de un vector:

- logical
- integer
- double
- complex
- character
- raw

1.2.1. Creación

Vectores homogéneos

Para la creación de vectores recurriremos a una función interna de R conocida como *concatenación* y es denotada por la letra `c()`. Iniciamos mostrando al lector la creación de vectores donde todos sus elementos son del mismo tipo.

```r
# Podemos crear vectores logicos tecleando TRUE y FALSE (o T & F).
logi_var <- c(TRUE, FALSE, TRUE, FALSE)
logi_var

## [1]  TRUE FALSE  TRUE FALSE

# Usamos el sufijo L, para crear un numero entero y diferenciarlo del resto
int_var <- c(2L, 4L, 7L, 5L)
int_var

## [1] 2 4 7 5

# Usamos el simbolo . para notar los decimales
dbl_var <- c(2.3, 6.8, 4.1)
dbl_var

## [1] 2.3 6.8 4.1

# Usamos el caracter i para notar la parte imaginaria
cpl_var <- c(3+6i, 5-2i, -8i)
cpl_var

## [1] 3+6i 5-2i 0-8i

# Los caracteres deben ir entre comillas
chr_var <- c("statistical", "model", "test")
chr_var

## [1] "statistical" "model"       "test"
```

La función `vector` permite crear vectores de un tipo y longitud determinada, dichos vectores se crean con elementos por defecto, como se muestra en los siguientes ejemplos:

```r
# creamos un vector logico de longitud 5
vector("logical", 5)

## [1] FALSE FALSE FALSE FALSE FALSE

# creamos un vector numerico de longitud 5
vector("numeric", 5)

## [1] 0 0 0 0 0

# creamos un vector complejo de longitud 5
vector("complex", 5)

## [1] 0+0i 0+0i 0+0i 0+0i 0+0i

# creamos un vector de caracteres de longitud 5
vector("character", 5)

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

Todo vector consta de dos argumentos: `mode` y `length`. El primero de ellos especifica el tipo de elementos que almacena el vector, mientras que el segundo argumento especifica la longitud o número de elementos que almacena dicho vector.

```
# creamos un vector numerico
var <- c(3, 6, 8, 9)
# verificamos el tipo de vector
mode(var)

## [1] "numeric"

# mostramos la longitud del vector
length(var)

## [1] 4
```

Vectores heterogéneos

El usuario tiene toda la libertad para crear un vector con diferentes tipos de elementos, sin embargo, debido al conflicto que se genera internamente por la *coerción*[1] de los distintos elementos tenemos la siguiente jerarquía:

−								+
logical	<	**integer**	<	**double**	<	**complex**	<	**character**

Cuadro 1.2: Flexibilidad de los elementos

La coerción se produce automáticamente, por tanto, al momento de crear un vector es importante revisar el tipo de elemento de mayor flexibilidad con el fin de conocer a que tipo de vector coercionará el resultado, revisamos algunos ejemplos:

```
var1 <- c(TRUE, 3L, FALSE, 5L)
mode(var1)

## [1] "numeric"

var2 <- c(2+5i, 6L, 8.14701)
mode(var2)

## [1] "complex"

var3 <- c(FALSE, 4L, 3.67012, -4+9i, "model")
mode(var3)

## [1] "character"
```

Posiblemente al lector le llamó la atención el primer ejemplo `var1`, debido que el mismo contiene elementos `logical` e `integer` y al momento de conocer el tipo de vector obtenemos `numeric` en lugar de `logical`. Esto se debe básicamente por la manera como R almacena y reconoce los diferentes elementos, la siguiente tabla resume lo expuesto.

[1] La coerción se produce cuando un objeto de un determinado tipo cambia a otro tipo diferente.

Tipo	Modo	Almacenamiento
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character
raw	raw	raw

Cuadro 1.3: Tipos de vectores

Dentro de la coerción un evento importante se produce cuando un vector logical coerciona a un vector integer, pues en estos casos el TRUE se convierte en 1 y el FALSE en 0.

```
# creamos un vector logico
vec <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
# la coercion de vectores logicos se da al multiplicar por 1
1*vec

## [1] 1 0 1 1 0

# los vectores logicos coercionan automaticamente ante operaciones aritmeticas
sum(vec)

## [1] 3
```

R cuenta con un grupo de funciones que permiten comprobar el tipo de elementos que almacena un vector, así como también funciones que obligan a coercionar los elementos del vector.

Tipo	Comprobación	Cambio
logical	`is.logical( )`	`as.logical( )`
numeric	`is.numeric( )`	`as.numeric( )`
double	`is.double( )`	`as.double( )`
complex	`is.complex( )`	`as.complex( )`
character	`is.character( )`	`as.character( )`

Se recomienda tener mucho cuidado al momento de forzar el coercionamiento de vectores y posteriormente realizar operaciones.

```
# creamos un vector con la finalidad de coercionar a texto
x <- c(2,4,7,9)
xc <- as.character(x)
xc

## [1] "2" "4" "7" "9"

# calculemos la suma del vector coercionado
sum(xc)

## Error in sum(xc): invalid 'type'(character) of argument
```

El ejemplo anterior muestra que los números también pueden coercionar a texto de manera que las operaciones aritméticas aplicadas sobre estos no son válidas.

Almacenamiento de vectores

Una característica importante de los vectores es que se encuentran almacenados de forma plana incluso cuando se anidan vectores como se observa:

```
c(2L, 4L, 6L, 8L)

## [1] 2 4 6 8

# obtenemos el mismo resultado mediante
c(2L, c(4L, c(6L, 8L)))

## [1] 2 4 6 8
```

Observamos que la concatenación de vectores nos genera un nuevo vector.

1.2.2. Valores perdidos

En ocasiones puede que no todos los elementos de un vector sean conocidos por diferentes motivos, en este caso nos hacen falta dichos elementos, lo cuales se denominan *datos perdidos* o *faltantes*. R permite que el usuario establezca dichos valores perdidos a través del término NA.

```
#Generamos un vector cuyo tercer elemento es un dato perdido
vec <- c(2, 3, NA, 8)
vec

## [1]  2  3 NA  8
```

Debido a los distintos tipos de elementos que existen en R, se han creado terminologías para los datos perdidos de acuerdo al tipo de elemento, es por ello que tenemos:

```
#NA para elementos numericos
NA_real_

## [1] NA

#NA para elementos enteros
NA_integer_

## [1] NA

#NA para elementos caracteres
NA_character_

## [1] NA

#NA para elementos complejos
NA_complex_

## [1] NA
```

Lo anterior evitará que coercionen los elementos de un vector por algún motivo. Para conocer si un vector contiene valores perdidos usaremos la función `is.na( )`.

```r
# creamos dos vectores
x <- c(2, 4, 7, 9)
y <- c(3, NA, 8, 13)
# evaluamos si existen valores perdidos
is.na(x)

## [1] FALSE FALSE FALSE FALSE

is.na(y)

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

1.2.3. Eliminación

En el caso que se desee eliminar ciertas variables u objetos innecesarios del área de trabajo usamos los comandos `rm()` o `remove()`.

```r
# Mostramos los objetos actuales
ls()

##  [1] "chr_var"  "cpl_var"  "dbl_var"  "int_var"  "logi_var" "var"
##  [7] "var1"     "var2"     "var3"     "vec"      "x"        "x_new"
## [13] "xc"       "y"        "y_new"    "z"

# Iniciamos eliminando los vectores: chr_var, dbl_var, int_var, logi_var
remove(chr_var, dbl_var, int_var, logi_var)
# Mostramos los objetos restantes
ls()

##  [1] "cpl_var" "var"     "var1"    "var2"    "var3"    "vec"     "x"
##  [8] "x_new"   "xc"      "y"       "y_new"   "z"
```

Cuando el usuario desea eliminar los objetos cuyos nombres cumple con algún patrón en común, usamos lo siguiente:

```r
# Eliminamos las variables: var, var1, var2, var3
rm(list = ls(pattern = 'var'))
# Revisamos los objetos restantes
ls()

## [1] "vec"   "x"     "x_new" "xc"    "y"     "y_new" "z"
```

Finalmente, para eliminar todos los objetos del área de trabajo usaremos:

```r
# Elimina todos los objetos existentes
rm(list=ls())
# Verificamos el area de trabajo
ls()

## character(0)
```

1.2.4. Modificación

Los vectores en R son almacenados como arreglos lineales, la ventaja de aquello es que sus elementos se encuentran indexados.

```
# creamos un vector
vec <- c(2, 4, 6, 8, 10)
vec

## [1]  2  4  6  8 10

# accedemos al elemento de la posicion 3
vec[3]

## [1] 6

# visualizamos los elementos de la posicion 2 y 5
vec[c(2, 5)]

## [1]  4 10
```

Añadiendo elementos

Sabemos que el tamaño de un vector es determinado en su creación, por lo que si se desea añadir o eliminar elementos es necesario reasignar el vector.

```
# agregamos los elementos 15, 18 al final
vec <- c(vec, c(15, 18))
vec

## [1]  2  4  6  8 10 15 18

# agregamos el valor de -1 al inicio
vec <- c(-1, vec)
vec

## [1] -1  2  4  6  8 10 15 18
```

En el caso que se desee añadir elementos en posiciones intermedias del vector es necesario conocer las posiciones de sus elementos.

```
# agregamos el elemento -4 en la posicion 5
vec <- c(vec[1:4], -4, vec[5:8])
vec

## [1] -1  2  4  6 -4  8 10 15 18
```

Eliminando elementos

Para eliminar ciertos los elementos de un vector usaremos subíndices negativos, lo anterior nos permite excluir los elementos deseados como se muestra a continuación:

```
# eliminamos el valor de la posicion 2
vec <- vec[-2]
vec

## [1] -1  4  6 -4  8 10 15 18

# eliminamos los elementos de las posiciones 4, 7
vec <- vec[-c(4, 7)]
vec

## [1] -1  4  6  8 10 18
```

Modificando elementos

Para modificar los elementos de un vector es suficiente conocer la posición de los mismos como se muestra en el siguiente ejemplo:

```
# modificamos el primer elemento por 0
vec[1] <- 0
# visualizamos el vector modificado
vec

## [1]  0  4  6  8 10 18

# modificamos las posiciones 2 y 5
vec[c(2, 5)] <- c(5, 9)
vec

## [1]  0  5  6  8  9 18
```

Generación de secuencias

R posee operadores que son útiles a la hora de crear vectores, iniciamos con el operador ':', el cual genera un vector que abarca un rango de números con salto uniforme de amplitud 1.

```
1:12

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12

7:-5

##  [1]  7  6  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

El segundo operador que revisaremos es `seq( )`, mismo que tiene tres parámetros que controlan el inicio, final y salto de la secuencia.

```
seq(from = 1, to = 12, by = 1)

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12

seq(from = -2, to = 3, by = 0.7)
```

```
## [1] -2.0 -1.3 -0.6  0.1  0.8  1.5  2.2  2.9
```

```
seq(from = -4, to = 4, length = 12)
```

```
##  [1] -4.0000000 -3.2727273 -2.5454545 -1.8181818 -1.0909091 -0.3636364
##  [7]  0.3636364  1.0909091  1.8181818  2.5454545  3.2727273  4.0000000
```

Por último, la function `rep( )` nos permite repetir un objeto un número de veces especificado, a continuación mostramos algunos ejemplos:

```
rep(3, 8)
```

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

```
rep(c(3, 6, 9), times = 2)
```

```
## [1] 3 6 9 3 6 9
```

```
rep(c(2, 4, 6, 9), each = 3)
```

```
##  [1] 2 2 2 4 4 4 6 6 6 9 9 9
```

```
rep(c("MS", "PLUS", "EC"), times = 2)
```

```
## [1] "MS"   "PLUS" "EC"   "MS"   "PLUS" "EC"
```

1.2.5. Operaciones

Ahora revisaremos algunas operaciones comúnes relacionadas con los vectores. Cubriremos las operaciones aritméticas y lógicas más utilizadas.

Operación	Descripción
`a + b`	Suma
`a - b`	Resta
`a * b`	Multiplicación
`x / y`	División
`x ^ y`	Potencia
`sqrt(x)`	Raíz cuadrada
`abs(x)`	Valor absoluto
`exp(x)`	Exponencial
`log(x, base=n)`	Logaritmo en base n
`factorial(x)`	Factorial
`x%% y`	Módulo
`x%/% y`	División entera
`x == y`	Test de igualdad
`x != y`	Test de desigualdad
`x <= y`	Test menor o igual que
`x >= y`	Test mayor o igual que
`x && y`	Conjunción para escalares
`x || y`	Disyunción para escalares
`x & y`	Conjunción para vectores
`x | y`	Disyunción para vectores
`!x`	Negación
`sum(x)`	Suma todos los elementos de x
`prod(x)`	Multiplica todos los elementos de x

Cuadro 1.4: Operaciones básicas

Operaciones aritméticas

Recordemos que R es un lenguaje funcional, por lo que cualquier operador es una función. Veamos un ejemplo:

```
# suma de 2 elementos
3 + 5

## [1] 8

# Division entera
8 %/% 3

## [1] 2

# Logaritmo en base 3
log(24, base = 3)

## [1] 2.892789

# otra alternativa para operar
"+" (3, 5)

## [1] 8
```

```
"%/%" (8, 3)

## [1] 2

"log" (24, 3)

## [1] 2.892789
```

Cuando usamos R como una calculadora es frecuente encontrarnos con la necesidad que evaluar expresiones aritméticas complicadas, por lo cual debemos tener muy en cuenta la jerarquía existente entre las operaciones:

$$\frac{5{,}5^{2+\ln(3)} - e^{-2}}{3!}$$

```
# Evaluación
(5.5^(2+log(3)) - exp(-2))/factorial(3)

## [1] 32.78287
```

$$10^2 + \frac{3 \times 60}{15 - \sqrt{6}} - 3^{2{,}25 - \frac{1}{4}}$$

```
# Evaluación
10^2 + 3*60/(15-sqrt(6)) - 3^(2.25-1/4)

## [1] 105.342
```

Las operaciones efectuadas entre vectores se llevan a cabo elemento a elemento como se muestra en los siguientes ejemplos:

```
# creamos un vector x
x <- c(1, 3, 5, 7, 9)
# raiz cuadrada de un vector
sqrt(x)

## [1] 1.000000 1.732051 2.236068 2.645751 3.000000

# logaritmo base 4 de un vector
log(x, base = 4)

## [1] 0.0000000 0.7924813 1.1609640 1.4036775 1.5849625

# creamos un vector y
y <- c(2, 4, 6, 8, 10)
# sumamos los vectores x e y
x + y

## [1]  3  7 11 15 19

# segunda alternativa para la suma de vectores
"+" (x, y)

## [1]  3  7 11 15 19

# suma de los elementos del vector x
sum(x)
```

```
## [1] 25

# multiplicación de los elementos del vector x
prod(x)

## [1] 945
```

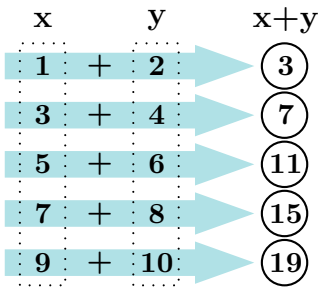


Figura 1.4: Suma de vectores de igual longitud

Cuando aplicamos una operación a dos vectores se requiere que tengan la misma longitud, caso contrario, R *recicla* o repite los elementos con el fin que los vectores tengan la misma longitud.

```
# suma de vectores longitud 2 y 3
c(2, 4) + c(3, 6, 9)

## Warning in c(2, 4) + c(3, 6, 9): longer object length is not a multiple of shorter
object length

## [1]  5 10 11
```

R muestra una advertencia debido que el vector más pequeño no es múltiplo del vector más grande, pese a eso recicla (*repite*) el primer elemento del vector más pequeño para operar.

```
# suma de vectores longitud 3 y 6
c(1, 2, 3) + c(1, 2, 3, 4, 5, 6)

## [1] 2 4 6 5 7 9
```

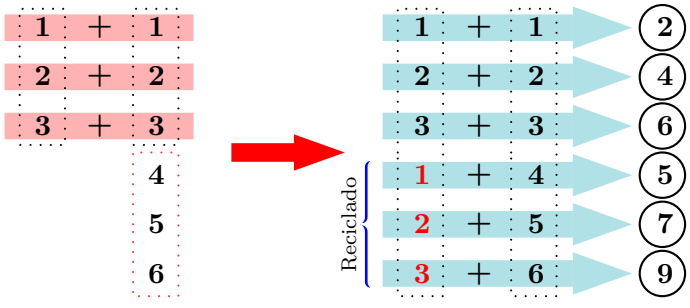


Figura 1.5: Suma de vectores de diferente longitud

Ahora, R no muestra ninguna advertencia dado que las longitudes de los vectores son múltiplos entre sí. Observamos que en ambos casos el vector más pequeño fue reciclado para que se realice la operación.

Si el usuario se encuentra familiarizado con el álgebra lineal se verá sorprendido al ver lo que sucede cuando se multiplica dos vectores.

```
# calculamos el producto de los vectores x, y
x*y

## [1]  2 12 30 56 90

# otra alternativa para la multiplicacion
"*" (x, y)

## [1]  2 12 30 56 90
```

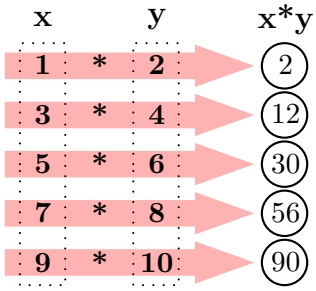


Figura 1.6: Producto de vectores de igual longitud

El ejemplo anterior muestra que el producto entre vectores de igual longitud se da entre elementos de la misma posición. En el caso que se multiplique un vector por un escalar el resultado es el siguiente:

```
# Producto
2 * c(1, 3, 5, 7)

## [1]  2  6 10 14

c(3, 4) * c(1, 3, 5, 7)

## [1]  3 12 15 28

# Division
c(2, 4, 6, 8) / 2

## [1] 1 2 3 4

c(24, 16, 15, 8) / c(4, 8, 3, 2)

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

Operaciones lógicas

Ahora revisaremos las expresiones lógicas o booleanas[2], las cuales son utilizadas con frecuencia en distintos lenguajes de programación para verificar si un objeto cumplen ciertas condiciones deseadas.

[2]George Boole (1815 1864)

```
# verificamos la siguientes expresiones
2 == 6

## [1] FALSE

7 <= 10

## [1] TRUE

(1 < 3) & (4 >= 2)

## [1] TRUE

(8 >= 5) != 1

## [1] FALSE

# generamos un vector entre 1 y 9
x <- seq(1,9)
# verificamos los elementos mayores a 5
x > 5

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

Las funciones `any( )` y `all( )` nos reportan si al menos uno o todos sus elementos son verdaderos.

```
# verificamos si algun elemento es mayor a 5
any(x>5)

## [1] TRUE

# verificamos si todos los elementos son mayores a 5
all(x>5)

## [1] FALSE
```

Las operaciones lógicas también pueden llevarse a cabo en vectores como se observa en el siguiente ejemplo:

p	q	r	$\sim$ q	$\sim$ q $\vee$ r	p $\wedge$ ($\sim$ q $\vee$ r)
V	V	V	F	V	V
V	V	F	F	F	F
V	F	V	V	V	V
V	F	F	V	V	V
F	V	V	F	V	F
F	V	F	F	F	F
F	F	V	V	V	F
F	F	F	V	V	F

```
# creación de vectores lógicos
p <- c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)
q <- c(TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE)
r <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
# evaluamos la expresión: p ^ (~q v r)
p & (!q | r)

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

Ordenando elementos

Los elementos de un vector pueden ser ordenados de forma ascendente o descendente empleando el comando `sort( )`:

```
# creamos un vector
x <- c(5, -3.2, 6.7, 0.21, 4.6, -1.23, 0, 9.2, -5.34)
# ordenamos de forma ascendente
sort(x, decreasing = FALSE)

## [1] -5.34 -3.20 -1.23  0.00  0.21  4.60  5.00  6.70  9.20

# ordenamos de forma descendente
sort(x, decreasing = TRUE)

## [1]  9.20  6.70  5.00  4.60  0.21  0.00 -1.23 -3.20 -5.34
```

1.2.6. Atributos

Todos los objetos en R tienen una lista de atributos, la misma que es utilizada para almacenar la metadata[3] de los objetos. El acceso a los atributos de un objeto se lo puede realizar individualmente o todos a la misma vez por medio de las funciones `attr( )` o `attributes( )` respectivamente.

Los tres atributos más importantes son:

- **Nombres**: Un vector de caracteres que provee un nombre a cada elemento del objeto.
- **Dimensiones**: Un vector numérico empleado para transformar vectores en matrices o arrays.
- **Clase**: Carácter que define el comportamiento y la apariencia de un objeto.

```
x <- c(1, 3, 5, 6, 8, 9)
# asignamos nombres a las posiciones del vector
attr(x, "names") <- c("uno", "tres", "cinco", "seis", "ocho", "nueve")
x

##   uno  tres cinco  seis  ocho nueve
##     1     3     5     6     8     9
```

[3]Es información que describe el contenido un archivo u objeto.

```
# los nombres se mantienen pese a realizar operaciones
x+3

##   uno  tres cinco  seis  ocho nueve
##     4     6     8     9    11    12

# convertimos el vector en una matriz 2 X 3
attr(x, "dim") <- c(2,3)
x

##      [,1] [,2] [,3]
## [1,]    1    5    8
## [2,]    3    6    9
## attr(,"names")
## [1] "uno"   "tres"  "cinco" "seis"  "ocho"  "nueve"

# asociamos el objeto a una nueva clases "Mat23"
attr(x, "class") <- "Mat23"
x

##      [,1] [,2] [,3]
## [1,]    1    5    8
## [2,]    3    6    9
## attr(,"names")
## [1] "uno"   "tres"  "cinco" "seis"  "ocho"  "nueve"
## attr(,"class")
## [1] "Mat23"

# listamos todos los atributos disponibles
attributes(x)

## $names
## [1] "uno"   "tres"  "cinco" "seis"  "ocho"  "nueve"
##
## $dim
## [1] 2 3
##
## $class
## [1] "Mat23"

# revisamos la estructura del objeto
str(x)

##  Mat23 [1:2, 1:3] 1 3 5 6 8 9
##  - attr(*, "names")= chr [1:6] "uno" "tres" "cinco" "seis" ...
```

La función `str( )` muestra un resumen de los atributos del objeto.

1.3. Factores

Los factores en R son vectores que tienen un número limitado de valores diferentes que pueden almacenar y son empleados generalmente para almacenar variables categóricas u ordinales.

1.3.1. Creación

La función `factor( )` tiene 3 argumentos de interés:

1. El *nombre* del vector atómico a convertirse en factor. En el ejemplo siguiente: **genero**.
2. Los *valores* que el factor puede recibir. En el ejemplo, el vector **vec_gen** está limitado a los valores "`m`" y "`f`", sin embargo podemos incluir el valor "`o`" dentro del factor para futuras expansiones. En el caso que los niveles presenten un orden natural por ejemplo: bajo, medio, alto se puede establecer el parámetro `ordered` en la función con la finalidad de resaltar dicho orden.
3. Por último, las etiquetas son opcionales y las mismas se cruzan en el mismo orden que los niveles. En el caso que no se liste las etiquetas el programa coloca los mismos valores de los niveles.

```
# Vector numerico
vec_gen <- c("m", "f", "f", "m", "m", "f")
class(vec_gen)

## [1] "character"

table(vec_gen)

## vec_gen
## f m
## 3 3

# Creacion del factor
genero <- factor(vec_gen,
                 levels = c("m", "f", "o"),
                 labels = c("Masculino", "Femenino", "Otro"), ordered = FALSE)
class(genero)

## [1] "factor"

table(genero)

## genero
## Masculino  Femenino      Otro
##         3         3         0
```

Los factores se almacenan como un vector de elementos enteros simultáneamente con un conjunto de caracteres a utilizar cuando se visualiza el factor.

```
# Visualizamos el almacenamiento de un factor
unclass(genero)

## [1] 1 2 2 1 1 2
## attr(,"levels")
## [1] "Masculino" "Femenino"  "Otro"
```

Podemos acceder a los niveles permitidos en un factor mediante el comando `levels( )`.

```r
levels(genero)

## [1] "Masculino" "Femenino"  "Otro"
```

1.3.2. Modificación

Algo importante de mencionar es que no se pueden ingresar valores no definidos previamente en sus niveles (*levels*).

```r
# resultados para el nuevo factor
vec <- c("S", "C", "A", "C", "S", "I", "S")
# creamos el factor con niveles y etiquetas
region <- factor(vec, levels = c("C", "S", "A", "I"),
                 labels = c("Costa", "Sierra", "Amazonía", "Insular"))
region

## [1] Sierra   Costa    Amazonía Costa    Sierra   Insular  Sierra
## Levels: Costa Sierra Amazonía Insular

# accedemos a la segunda posición
region[2]

## [1] Costa
## Levels: Costa Sierra Amazonía Insular

# modificamos la primera posición - Insular por Sierra
region[1] <- "Insular"
# visualizamos el factor modificado
region

## [1] Insular  Costa    Amazonía Costa    Sierra   Insular  Sierra
## Levels: Costa Sierra Amazonía Insular

# ingresamos un nuevo nivel que no estuvo considerado previamente
region[4] <- "Extranjero"

## Warning in `[<-.factor`(`*tmp*`, 4, value = "Extranjero"): invalid factor level,
NA generated
```

Al tratar de forzar el ingreso de un nivel que no estuvo considerado previamente se genera un conflicto interno, por lo cual R alerta al usuario, y además, reemplaza dicho elemento por un valor pérdido `NA`.

```r
# revisamos el factor region
region

## [1] Insular  Costa    Amazonía <NA>     Sierra   Insular  Sierra
## Levels: Costa Sierra Amazonía Insular
```

Una solución al problema anterior se logra al modificar los niveles del factor:

```
# Ampliamos los niveles del factor
levels(region) <- c("Costa", "Sierra", "Amazonía", "Insular", "Extranjero")
# Ingresamos el nivel nuevo
region[4] <- "Extranjero"
# Visualizamos el factor
region

## [1] Insular    Costa      Amazonía   Extranjero Sierra     Insular
## [7] Sierra
## Levels: Costa Sierra Amazonía Insular Extranjero
```

El filtrado de los elementos de un factor tiene un funcionamiento similar al filtrado de vectores, salvo que las comparaciones se realiza sobre el conjunto de etiquetas:

```
# comparacion sobre los niveles - modo incorrecto
region=="I"

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

# comparacion sobre las etiquetas - modo correcto
region=="Insular"

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

# filtrado de un factor
region[region=="Insular"]

## [1] Insular Insular
## Levels: Costa Sierra Amazonía Insular Extranjero
```

Los ejemplos antes vistos son de utilidad para la creación de variables nominales, ahora incorporaremos el argumento `ordered` para crear variables ordinales:

```
# creamos el factor para: bajo, medio y alto
etiquetas <- c("bajo", "medio", "alto")
opciones <- c("b", "m", "a")
valores <- c("m", "b", "m", "a", "m", "a", "b", "m", "a")
new_fac <- factor(x = valores, levels = opciones, labels = etiquetas,
                  ordered = TRUE)
# Visualizamos el nuevo factor "ordinal"
new_fac

## [1] medio bajo  medio alto  medio alto  bajo  medio alto
## Levels: bajo < medio < alto

table(new_fac)

## new_fac
##  bajo medio  alto
##     2     4     3
```

1.4. Matrices

Las matrices son arreglos bidimensionales empleados para almacenar elementos de un mismo tipo (logical, integer, double, complex, character, raw).

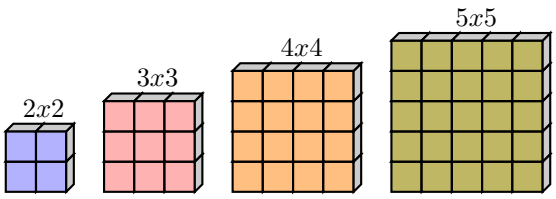


Figura 1.7: Matrices

1.4.1. Creación

Para crear una matriz empleamos la función `matrix( )` misma que posee 4 argumentos de interés:

1. Un vector con los elementos que formarán parte de la matriz.
2. Un valor numérico `nrow` que define el número de *filas*.
3. Un valor numérico `ncol` que define el número de *columnas*.
4. Un valor lógico que establece si el llenado de los elementos de la matriz se lo realiza por filas o por columnas.

```
# Creamos una vector con 6 elementos
vec <- c(2, 4, 6, 8, 10, 12)
# Creamos la matrix a partir del vector "vec"
Mat23 <- matrix(vec, nrow=2, ncol=3, byrow = FALSE)
Mat23

##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    4    8   12

# Verificamos el tipo de objeto
class(Mat23)

## [1] "matrix"
```

Si disponemos de varios vectores con una misma longitud podemos emplear estos como *filas* para formar una matriz por medio del comando `rbind( )`:

```
# Generamos 2 vectores de igual longitud
vec1 <- c(1, 5, 8)
vec2 <- c(3, 6, 9)
# Creamos la matriz usando los vectores como filas
Mat1 <- rbind(vec1, vec2)
Mat1

##      [,1] [,2] [,3]
## vec1    1    5    8
## vec2    3    6    9
```

```r
class(Mat1)

## [1] "matrix"
```

En el caso que se desee emplear los vectores como *columnas* para formar una matriz usamos el comando `cbind( )`:

```r
# Creamos la matriz usando los vectores como columnas
Mat2 <- cbind(vec1, vec2)
Mat2

##      vec1 vec2
## [1,]    1    3
## [2,]    5    6
## [3,]    8    9

class(Mat2)

## [1] "matrix"
```

1.4.2. Submatrices

Para acceder a un elemento de una matriz debemos especificar su posición en base al número de fila y columna que ocupa el mismo, es decir, especificaremos primero el nombre de la matriz seguido de la posición entre de corchetes separando el número de la fila y columna mediante una coma.

```r
# Creamos una nueva matriz a partir de un vector
Mat <- matrix(seq(2,32, by=2), ncol=4, byrow = TRUE)
Mat

##      [,1] [,2] [,3] [,4]
## [1,]    2    4    6    8
## [2,]   10   12   14   16
## [3,]   18   20   22   24
## [4,]   26   28   30   32

# Accedemos al elemento de la primera fila y segunda columna [1,2]
Mat[1,2]

## [1] 4

# Accedemos al elemento de la tercer fila y cuarta columna [3,4]
Mat[3,4]

## [1] 24

# Accedemos a los elementos de las posiciones [3,2] y [3,3]
Mat[3,c(2,3)]

## [1] 20 22

# Accedemos a los elementos de las posiciones [1,4], [2,4] y [3,4]
Mat[c(1,2,3),4]

## [1]  8 16 24
```

Si deseamos seleccionar todos los elementos de una fila o de una columna, basta con especificar el número de la fila o columna y dejar en blanco la otra posición:

```
# Seleccionamos la segunda fila
Mat[2,]

## [1] 10 12 14 16

# Seleccionamos la tercera columna
Mat[,3]

## [1]  6 14 22 30

# Extraemos una submatriz 2x2 del extremo derecho inferior
Mat[c(3,4),c(3,4)]

##      [,1] [,2]
## [1,]   22   24
## [2,]   30   32
```

1.4.3. Modificación

La modificación de los elementos de una matriz se realiza especificando las posiciones a ser intercambiadas, como se muestra en el siguiente ejemplo:

```
Mat

##      [,1] [,2] [,3] [,4]
## [1,]    2    4    6    8
## [2,]   10   12   14   16
## [3,]   18   20   22   24
## [4,]   26   28   30   32

# Modificamos la posicion [2,2] por el valor de -12
Mat[2,2] <- -12
Mat

##      [,1] [,2] [,3] [,4]
## [1,]    2    4    6    8
## [2,]   10  -12   14   16
## [3,]   18   20   22   24
## [4,]   26   28   30   32

# Modificamos la primera fila por c(-2, 5, -6, 9)
Mat[1,] <- c(-2, 5, -6, 9)
Mat

##      [,1] [,2] [,3] [,4]
## [1,]   -2    5   -6    9
## [2,]   10  -12   14   16
## [3,]   18   20   22   24
## [4,]   26   28   30   32
```

```
# Modificados las posiciones [2,1] y [3,1] por c(-10, -15)
Mat[c(2,3),1] <- c(-10, -15)
Mat

##      [,1] [,2] [,3] [,4]
## [1,]   -2    5   -6    9
## [2,]  -10  -12   14   16
## [3,]  -15   20   22   24
## [4,]   26   28   30   32

# Modificamos ahora una submatriz por su identidad
Mat[c(3,4),c(2,3)] <- matrix(c(1,0,0,1), ncol=2, byrow = TRUE)
Mat

##      [,1] [,2] [,3] [,4]
## [1,]   -2    5   -6    9
## [2,]  -10  -12   14   16
## [3,]  -15    1    0   24
## [4,]   26    0    1   32
```

1.4.4. Operaciones

Las principales operaciones con matrices se resumen en el siguiente cuadro:

Operación	Descripción
`A + B`	Suma
`A%*% B`	Multiplicación
`t(A)`	Tranpuesta
`solve(A)`	Inversa
`det(A)`	Determinante
`diag(A)`	Diagonal
`eigen(A)`	Autovalores, autovectores

Cuadro 1.5: Operaciones entre matrices

```
# creamos dos matrices A y B de 2 x 3 con numeros aleatorios entre 1 y 20
A <- matrix(sample(seq(1,20), size=6), nrow = 2, byrow = FALSE)
A

##      [,1] [,2] [,3]
## [1,]   15    9    2
## [2,]   13   14   20

B <- matrix(sample(seq(1,20), size=6), nrow = 2, byrow = FALSE)
B

##      [,1] [,2] [,3]
## [1,]   14    5   11
## [2,]    7   12   15
```

```r
# Suma de matrices
A+B
```

```
##      [,1] [,2] [,3]
## [1,]   29   14   13
## [2,]   20   26   35
```

```r
# Multiplicamos A x B, transponemos la matriz B
A %*% t(B)
```

```
##      [,1] [,2]
## [1,]  277  243
## [2,]  472  559
```

```r
# Un resultado diferente se obtiene al transponer la matriz A
t(A) %*% B
```

```
##      [,1] [,2] [,3]
## [1,]  301  231  360
## [2,]  224  213  309
## [3,]  168  250  322
```

```r
# Creamos la matriz C
C <- matrix(sample(seq(1,20), size=9), nrow = 3, byrow = FALSE)
C
```

```
##      [,1] [,2] [,3]
## [1,]    4   13    6
## [2,]   20    8    2
## [3,]   14    1   11
```

```r
# Obtenemos su determinante
det(C)
```

```
## [1] -2704
```

```r
# Dado que el determinante es diferente de cero, obtenemos la inversa
solve(C)
```

```
##             [,1]        [,2]         [,3]
## [1,] -0.03180473  0.05066568  0.008136095
## [2,]  0.07100592  0.01479290 -0.041420118
## [3,]  0.03402367 -0.06582840  0.084319527
```

```r
# Obtenemos los valores y vectores propios
eigen(C)
```

```
## eigen() decomposition
## $values
## [1]  26.000000 -11.807764   8.807764
##
## $vectors
##            [,1]       [,2]       [,3]
## [1,] -0.5324004  0.6691281 -0.1068467
## [2,] -0.6515945 -0.6369705 -0.4489636
## [3,] -0.5403466 -0.3828005  0.8871389
```

Al igual que en el Álgebra Lineal el producto de un escalar por una matriz genera una matriz, cuyo resultado se obtiene al multiplicar el escalar por cada uno de sus elementos:

```
D <- matrix(sample(seq(-10,10), size=4), nrow = 2, byrow = FALSE)
D

##      [,1] [,2]
## [1,]   -7   -6
## [2,]   10    3

# Producto de un escalar por una matriz
3*D

##      [,1] [,2]
## [1,]  -21  -18
## [2,]   30    9
```

En el caso de la suma de un escalar y una matriz, ocurre que el escalar se recicla (se genera una matriz de la misma dimensión con todos sus elementos igual al escalar) para lograr sumarse con todos los elementos de la matriz

```
# Suma de un escalar y una matriz
2+D

##      [,1] [,2]
## [1,]   -5   -4
## [2,]   12    5
```

Una *matriz identidad* representada por I_n, es una matriz cuadrada de dimensión $n \times n$ que contiene 1's en la diagonal y 0's en el resto de posiciones, por ejemplo:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

La creación de este tipo de matrices se facilita al emplear el comando `diag( )`:

```
# creacion de una matriz I3
I3 <- diag(3)
I3

##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

El comando `apply( )` da la facilidad al usuario de realizar operaciones sobre las filas o columnas de una matriz, él mismo cuenta con tres argumentos:

1. Una matriz sobre la cual se realizará la operació indicada.
2. Un valor entero denominado *margin*, mismo que indica si la operació se realiza sobre las filas (*valor igual a 1*), o sobre las columnas (*valor igual a 2*).

3. Una función u operación que se aplicará a las filas o columnas.

```
# Creamos una matriz de dimensión 4x4
matriz <- matrix(sample(-10:10, size = 16, replace = TRUE), ncol=4, nrow=4)
matriz

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

# Suma de los elementos por fila
apply(matriz, 1, sum)

## [1]   4 -10 -27   0

# Obtención del mínimo valor por columna
apply(matriz, 2, min)

## [1]  -8  -9 -10  -6

# Calculo del promedio de cada fila
apply(matriz, 1, mean)

## [1]  1.00 -2.50 -6.75  0.00
```

En el capítulo siguiente ampliaremos el uso del comando `apply( )`, creando funciones propias del usuario.

1.4.5. Atributos

Las matrices presentan un mayor número de atributos respecto a los vectores y factores, éstos son:

- `nrow`: Número de filas.
- `ncol`: Número de columnas.
- `dim`: Dimensión del arreglo bidimensional.
- `length`: Número de elementos de la matriz, es el resultado del producto entre `nrow` y `ncol`.
- `rownames`: Vector de caracteres que define el nombre de cada una de las filas.
- `colnames`: Vector de caracteres que define el nombre de cada una de las columnas.

```
Mat <- matrix(c(2+3i, -4i, 5-1i, 1+6i, 4, 6-7i), nrow=2)
# Numero de filas
nrow(Mat)

## [1] 2

# Numero de columnas
ncol(Mat)
```

```
## [1] 3

# Asignamos nombres a las columnas
colnames(Mat) <- c("a", "b", "c")
# Asignamos nombres a las filas
rownames(Mat) <- c("X", "Y")
# Visualizamos los cambios
Mat

##      a    b    c
## X 2+3i 5-1i 4+0i
## Y 0-4i 1+6i 6-7i
```

1.5. Listas

Las listas son colecciones ordenadas de objetos que pueden ser de distintos tipos y dimensiones que generalmente están identificados por un nombre.

1.5.1. Creación

La creación de listas se realizan por medio del comando `list( )`:

```
# creamos una lista que consta de: vector, matriz, vector
lista <- list(vec1=c(2L, 5L, 8L), mat=diag(4), vec2=c("a", "b", "c", "d", "e"))
lista

## $vec1
## [1] 2 5 8
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
##
## $vec2
## [1] "a" "b" "c" "d" "e"
```

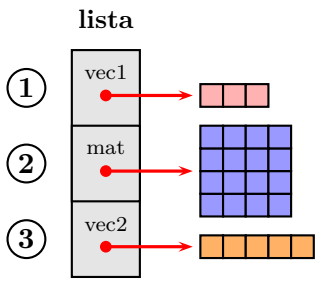


Figura 1.8: Estructura de una lista

Para obtener un *componente* de una lista se puede emplear el operador $ seguido del nombre.

```
# extraemos vec1 usando el operador $
lista$mat

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1

# exploramos el tipo de objeto que almacena
class(lista$mat)

## [1] "matrix"
```

Los componentes de una lista siempre están enumerados, por lo cual pueden ser referidos además por dicho número. En este caso, usaremos índices entre *doble corchetes*.

```
# extraemos la segunda componente de lista
lista[[3]]

## [1] "a" "b" "c" "d" "e"

# exploramos la dimensión de la componente
length(lista[[3]])

## [1] 5
```

Como, además `lista[[3]]` es un vector podemos ingresar a sus *elementos* empleando corchetes simples y especificando sus posiciones.

```
# extraemos los tres primeros elementos de la tercera componente de lista
lista[[3]][1:3]

## [1] "a" "b" "c"
```

1.5.2. Modificación

Dentro de una lista se pueden modificar tanto las componentes como sus elementos accediendo a las mismas y reemplazando sus valores.

```
# empleamos la lista antes creada
lista

## $vec1
## [1] 2 5 8
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
##
## $vec2
## [1] "a" "b" "c" "d" "e"

# modificamos el primer componente
lista$vec1 <- c(-3, 6.1, 8.7)
# modificamos los elemento [3,1] y [3,2] de la matriz
lista$mat[3,c(1,2)] <- c(-1, 4)
# modificamos los elementos [1,4], [2,4] y [3,4] de la matriz
lista[[2]][c(1,2,3),4] <- c(2, -3, 5)
# visualizamos los cambios
lista

## $vec1
## [1] -3.0  6.1  8.7
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    2
## [2,]    0    1    0   -3
## [3,]   -1    4    1    5
## [4,]    0    0    0    1
##
## $vec2
## [1] "a" "b" "c" "d" "e"
```

Las listas son flexibles a la hora de modificar sus componentes, cosa que no sucede en el resto de estructuras.

```
# reemplazamos el primer componente por otra lista
lista$vec1 <- list(c(TRUE, FALSE), diag(2), c("HOLA", "MUNDO"))
lista

## $vec1
## $vec1[[1]]
## [1]  TRUE FALSE
##
## $vec1[[2]]
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $vec1[[3]]
## [1] "HOLA"  "MUNDO"
##
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    2
## [2,]    0    1    0   -3
## [3,]   -1    4    1    5
## [4,]    0    0    0    1
```

```
##
## $vec2
## [1] "a" "b" "c" "d" "e"
```

El ejemplo anterior nos muestra claramente, cómo una componente de una lista puede ser a su vez una lista, cosa que no sucedía en las estructuras de almacenamiento de datos mostradas hasta el momento. Ahora revisemos la forma de extracción empleada para éstas *listas anidadas*.

```
# acceso al primer componente de la nueva lista
lista$vec1[[1]]

## [1]  TRUE FALSE

# acceso al tercer componente de la nueva lista
lista[[1]][[3]]

## [1] "HOLA"  "MUNDO"
```

Se recomienda al lector etiquetar las componentes de una lista de manera que se facilite su acceso por medio del nombre y no de la posición que ocupa.

```
# etiquetamos los nuevos componentes
names(lista$vec1) <- c("vec_log", "mat_I2", "vec_chr")
lista

## $vec1
## $vec1$vec_log
## [1]  TRUE FALSE
##
## $vec1$mat_I2
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $vec1$vec_chr
## [1] "HOLA"  "MUNDO"
##
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    2
## [2,]    0    1    0   -3
## [3,]   -1    4    1    5
## [4,]    0    0    0    1
##
## $vec2
## [1] "a" "b" "c" "d" "e"

# accedemos a la nueva matriz I2
lista$vec1$mat_I2
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

La eliminación de una componente se da tras la asignación de `NULL` a la componente que se desea eliminar.

```
# eliminamos el tercer componente inicial: vec2
lista$vec2 <- NULL
# eliminamos el primer componente de la nueva lista: vec_log
lista$vec1$vec_log <- NULL
# visualizamos el resultado
lista

## $vec1
## $vec1$mat_I2
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $vec1$vec_chr
## [1] "HOLA"  "MUNDO"
##
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    2
## [2,]    0    1    0   -3
## [3,]   -1    4    1    5
## [4,]    0    0    0    1
```

1.5.3. Concatenación

En ocasiones trabajar con varias listas y da manera separada no es aconsejable, por lo cual recomendamos al usuario hacer uso del comando *concatenar* `c( )`, para unir las listas.

```
# lista de vectores
lista1 <- list(impar=c(1,3,5,7,9), par=c(2,4,6,8,10))
str(lista1)

## List of 2
##  $ impar: num [1:5] 1 3 5 7 9
##  $ par  : num [1:5] 2 4 6 8 10

# lista de matrices
lista2 <- list(I2=diag(2), I3=diag(3))
str(lista2)

## List of 2
##  $ I2: num [1:2, 1:2] 1 0 0 1
##  $ I3: num [1:3, 1:3] 1 0 0 0 1 0 0 0 1
```

```r
# lista de listas
lista3 <- list(LST1=list(a=1, b=c(2,2)), LST2=list(txt="Ecuador", dia="Lunes"))
str(lista3)

## List of 2
##  $ LST1:List of 2
##   ..$ a: num 1
##   ..$ b: num [1:2] 2 2
##  $ LST2:List of 2
##   ..$ txt: chr "Ecuador"
##   ..$ dia: chr "Lunes"

# concatenacioón de listas
lista_completa <- c(lista1, lista2, lista3)
# número de componentes de la lista resultante
length(lista_completa)

## [1] 6

# visualizamos la estructura
str(lista_completa)

## List of 6
##  $ impar: num [1:5] 1 3 5 7 9
##  $ par  : num [1:5] 2 4 6 8 10
##  $ I2   : num [1:2, 1:2] 1 0 0 1
##  $ I3   : num [1:3, 1:3] 1 0 0 0 1 0 0 0 1
##  $ LST1 :List of 2
##   ..$ a: num 1
##   ..$ b: num [1:2] 2 2
##  $ LST2 :List of 2
##   ..$ txt: chr "Ecuador"
##   ..$ dia: chr "Lunes"

# visualizamos el resultado
lista_completa

## $impar
## [1] 1 3 5 7 9
##
## $par
## [1]  2  4  6  8 10
##
## $I2
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $I3
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
```

```
## [3,]    0    0    1
##
## $LST1
## $LST1$a
## [1] 1
##
## $LST1$b
## [1] 2 2
##
##
## $LST2
## $LST2$txt
## [1] "Ecuador"
##
## $LST2$dia
## [1] "Lunes"
```

Como se observa la concatenación da como resultado una lista, cuyos componentes son todos los de los argumentos unidos sucesivamente.

1.6. Arrays

Un arreglo o *array* de datos es un objeto que puede ser concebido como una matriz multidimensional (hasta 8 dimensiones). Una ventaja de este tipo de objeto es que sigue las mismas reglas que hemos descrito para las matrices.

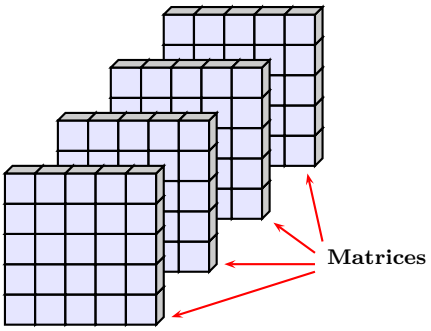


Figura 1.9: Arreglo tridimensional

1.6.1. Creación

La creación de arrays se realiza por medio del comando `array( )`, el cual posee 3 argumentos de interés:

1. Un vector de elementos que formarán parte de cada una de las matrices a crearse dentro del array.
2. Un vector numérico `dim` que especifica la *dimensión* del objeto.
3. Una lista de vectores `dimnames` que define el nombre de cada una de las dimensiones, así como también de las filas y columnas.

```r
# Vector de elementos
valores <- c(1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16,18)

# Nombres para filas, columnas y dimensiones
filas <- c("F1", "F2", "F3")
columnas <- c("C1", "C2", "C3")
dimensiones <- c("Dimensión 1", "Dimensión 2")

# Creamos el arreglo
arreglo <- array(data = valores, dim = c(3,3,2),
                 dimnames=list(filas, columnas, dimensiones))
arreglo
```

```
## , , Dimensión 1
##
##    C1 C2 C3
## F1  1  7 13
## F2  3  9 15
## F3  5 11 17
##
## , , Dimensión 2
##
##    C1 C2 C3
## F1  2  8 14
## F2  4 10 16
## F3  6 12 18
```

Hay que evidenciar que el relleno de los elementos en cada una de las matrices se realiza por columnas, a diferencia de las matrices no tenemos la opción de rellenar dichos elementos por filas.

El acceso a los elementos de un array se realiza de manera similar al de las matrices por medio de *corchetes*, en este caso debemos tener muy en cuenta el número de dimensiones sobre el cual estamos trabajando.

```r
# Usaremos el array antes creado con tres dimensiones
arreglo
```

```
## , , Dimensión 1
##
##    C1 C2 C3
## F1  1  7 13
## F2  3  9 15
## F3  5 11 17
##
## , , Dimensión 2
##
##    C1 C2 C3
## F1  2  8 14
## F2  4 10 16
## F3  6 12 18
```

```r
# seleccionamos la primera matriz
arreglo[,,1]
```

```
##    C1 C2 C3
## F1  1  7 13
## F2  3  9 15
## F3  5 11 17

# extraemos la tercera columna de la segunda matriz
arreglo[,3,2]

## F1 F2 F3
## 14 16 18

# obtenemos las dos primeras filas de la primera matriz
arreglo[c(1,2),,1]

##    C1 C2 C3
## F1  1  7 13
## F2  3  9 15

# seleccionamos la primera columna de ambas matrices
arreglo[,1,]

##    Dimensión 1 Dimensión 2
## F1           1           2
## F2           3           4
## F3           5           6
```

1.6.2. Modificación

La modificación de los elementos de una array se realiza asignado nuevos valores a las posiciones especificadas de antemano, miremos un par de ejemplos:

```
# Usaremos el array antes creado con tres dimensiones
arreglo

## , , Dimensión 1
##
##    C1 C2 C3
## F1  1  7 13
## F2  3  9 15
## F3  5 11 17
##
## , , Dimensión 2
##
##    C1 C2 C3
## F1  2  8 14
## F2  4 10 16
## F3  6 12 18

# reemplazamos la posición [3,1,1] por -2
arreglo[3,1,1] <- -2
# reemplazamos la posición [2,3,2] por -5
arreglo[2,3,2] <- -5
```

```
# modificamos las posiciones [1,1,] por 0's
arreglo[1,1,] <- c(0,0)
# modificamos la segunda columna de ambas dimensiones por 2'
arreglo[,2,] <- matrix(rep(2,6), nrow=3, ncol=2)
# visualizamos el resultado
arreglo

## , , Dimensión 1
##
##    C1 C2 C3
## F1  0  2 13
## F2  3  2 15
## F3 -2  2 17
##
## , , Dimensión 2
##
##    C1 C2 C3
## F1  0  2 14
## F2  4  2 -5
## F3  6  2 18
```

La modificación en los nombres de las dimensiones, filas y columnas se realiza de la siguiente manera:

```
# modificamos los nombres de las filas y columnas del array anterior
rownames(arreglo) <- c("fila_1", "fila_2", "fila_3")
colnames(arreglo) <- c("col_1", "col_2", "col_3")

# modificamos el nombre de las dimensiones
dimnames(arreglo)[[3]] <- c("dim_1", "dim_2")

# visualizamos resultados
arreglo

## , , dim_1
##
##        col_1 col_2 col_3
## fila_1     0     2    13
## fila_2     3     2    15
## fila_3    -2     2    17
##
## , , dim_2
##
##        col_1 col_2 col_3
## fila_1     0     2    14
## fila_2     4     2    -5
## fila_3     6     2    18
```

Para constrastar la dimensión del arreglo usamos el comando `dim( )`:

```
# obtención de la dimensión del arreglo
dim(arreglo)

## [1] 3 3 2
```

1.7. Data Frame

Un Data Frame es una estructura de datos bidimensional que generaliza a las matrices, en el sentido que las columnas pueden almacenar diferentes tipos de elementos entre sí, sin embargo, todos los elementos de una misma columna deben ser del mismo tipo.

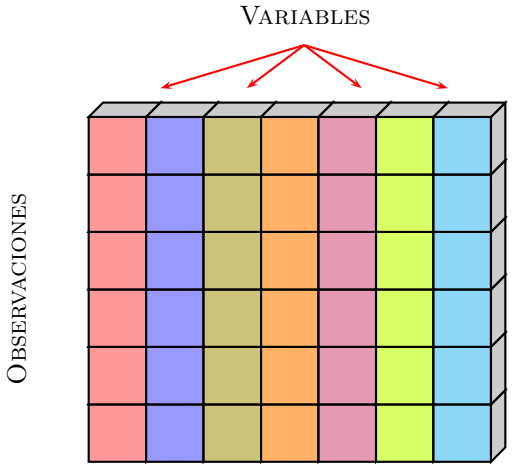


Figura 1.10: Estructura Data Frame

1.7.1. Creación

Para crear un data frame usamos el comando `data.frame( )`, dentro del cual colocamos los vectores de igual longitud que formarán parte de la estructura separados por coma.

```
datos <- data.frame(vec1=seq(1,4), vec2=c(T,F,T,F), vec3=LETTERS[1:4])
datos

##   vec1  vec2 vec3
## 1    1  TRUE    A
## 2    2 FALSE    B
## 3    3  TRUE    C
## 4    4 FALSE    D
```

En el caso que los vectores no tengan la misma longitud, R emplea las mismas reglas de reciclado correspondiente a los vectores.

```
datos <- data.frame(vec1=seq(1,4), vec2=c(T,F,T,F), vec3=LETTERS[1:4], vec4=c(-1,-3),
                    vec5=c("EC"))
str(datos)

## 'data.frame': 4 obs. of  5 variables:
##  $ vec1: int  1 2 3 4
##  $ vec2: logi  TRUE FALSE TRUE FALSE
```

```
##  $ vec3: Factor w/ 4 levels "A","B","C","D": 1 2 3 4
##  $ vec4: num  -1 -3 -1 -3
##  $ vec5: Factor w/ 1 level "EC": 1 1 1 1
```

Por defecto, R transforma los vectores de *caracteres* en *factores*, para evitar esta conversión automática colocamos el argumento `stringsAsFactors=FALSE`:

```
datos <- data.frame(vec1=seq(1,4), vec2=c(T,F,T,F), vec3=LETTERS[1:4], vec4=c(-1,-3),
                    vec5=c("EC"), stringsAsFactors = FALSE)
str(datos)

## 'data.frame': 4 obs. of  5 variables:
##  $ vec1: int  1 2 3 4
##  $ vec2: logi  TRUE FALSE TRUE FALSE
##  $ vec3: chr  "A" "B" "C" "D"
##  $ vec4: num  -1 -3 -1 -3
##  $ vec5: chr  "EC" "EC" "EC" "EC"
```

1.7.2. Modificación

1.8. Data Table

El paquete `data.table` fue creado por Matt Dowle conjuntamente con grupo de contribuidores y fue publicado en el año 2015. El paquete ofrece una versión mejorada de los `data.frame`, a continuación enumeramos las mejoras:

1. Rápida agregación para datos de gran tamaño (por ejemplo: 100 GB en RAM).
2. Lectura rápida y amigable para archivos a través de la función `fread`.
3. Añade, modifica y elimina variables sin utilizar copias en absoluto.
4. Rápido ordenamiento de variables: hacia adelante, hacia atrás.

La syntaxis básica de data.table no es difícil de dominar debido que los autores se preocuparon en reducir el tiempo que le toma al usuario programar:

$$\text{DT[where, select | group by]}$$

Existe una similitud entre la syntaxis de SQL y R, misma que se resume a continuación:

SQL:	where	select	group by
R:	i	j	by

1.8.1. Creación

En las siguientes líneas revisamos como crear un objeto `data.table`:

```
library(data.table)
set.seed(12345)
base <- data.table(B1=1:12, B2=LETTERS[1:4], B3=round(rnorm(3), 4),
                   B4=c(2L, 5L, 8L))
class(base)
```

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

dim(base)

## [1] 12  4
```

1.8.2. Filtrado de filas

Para filtrar las filas de acuerdo a la posición realizamos lo siguiente:

```
# Filtramos la tercera fila
base[3]

##    B1 B2      B3 B4
## 1:  3  C -0.1093  8

# Filtramos desde la novena a la decimo primera fila
base[9:11]

##    B1 B2      B3 B4
## 1:  9  A -0.1093  8
## 2: 10  B  0.5855  2
## 3: 11  C  0.7095  5

# Filtramos la tercera, novena y decimo segunda fila
base[c(3, 9, 12)]

##    B1 B2      B3 B4
## 1:  3  C -0.1093  8
## 2:  9  A -0.1093  8
## 3: 12  D -0.1093  8
```

La estructura tabular de `data.table` cuenta con un símbolo especial `.N`, el cual contiene el número total de filas.

```
# Imprimimos la última fila de la base empleando '.N'
base[.N]

##    B1 B2      B3 B4
## 1: 12  D -0.1093  8

# Imprimimos la antepenúltima fila
base[.N-2]

##    B1 B2     B3 B4
## 1: 10  B 0.5855  2
```

Para filtrar las filas de acuerdo a una condición realizamos lo siguiente:

```
# Filtramos filas que presentan un valor de 8 en la variable B4
base[B4==8L]
```

```
##    B1 B2      B3 B4
## 1:  3  C -0.1093  8
## 2:  6  B -0.1093  8
## 3:  9  A -0.1093  8
## 4: 12  D -0.1093  8

# Filtramos filas que contiene la letra D en la variable B2
base[B2=="D"]

##    B1 B2      B3 B4
## 1:  4  D  0.5855  2
## 2:  8  D  0.7095  5
## 3: 12  D -0.1093  8

# Filtramos filas que contiene la letra A o B en la variable B2
base[B2 %in% c("A","B")]

##    B1 B2      B3 B4
## 1:  1  A  0.5855  2
## 2:  2  B  0.7095  5
## 3:  5  A  0.7095  5
## 4:  6  B -0.1093  8
## 5:  9  A -0.1093  8
## 6: 10  B  0.5855  2

# Filtramos filas con valor superior a 9 en la variable B1
base[B1>9]

##    B1 B2      B3 B4
## 1: 10  B  0.5855  2
## 2: 11  C  0.7095  5
## 3: 12  D -0.1093  8
```

El total de filas en una base puede ser obtenido por medio del símbolo `.N` o `nrow`.

```
base[,.N]

## [1] 12

nrow(base)

## [1] 12
```

1.8.3. Selección de variables

Considerando el hecho de que un data.table hereda el comportamiento de un data.frame, la selección de variables se puede realizar de las siguientes maneras:

```
# Extracción de la primera variable
base$B1

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Este primer método de extracción de variables emplea el operador $ para ingresar a la base y seleccionar la variable "B1". El principal inconveniente de este método es la extracción múltiple de variables.

```
# Extracción de la tercera variable
base[["B3"]]

##  [1]  0.5855  0.7095 -0.1093  0.5855  0.7095 -0.1093  0.5855  0.7095
##  [9] -0.1093  0.5855  0.7095 -0.1093
```

El segundo método emplea doble corchete y especifica el nombre de la variable. Posee el mismo inconveniente del método anterior al no poder extraer dos o más variables al mismo instante.

```
# Extracción de la segunda variable
base[,B2]

##  [1] "A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"
```

Este último método emplea el nombre de la variable para la extracción de la misma, tiene la ventaja de extraer dos o más variables bajo la siguiente sintaxis:

```
# Extracción de dos variables "B2" y "B3"
base[,.(B2,B3)]

##     B2      B3
##  1:  A  0.5855
##  2:  B  0.7095
##  3:  C -0.1093
##  4:  D  0.5855
##  5:  A  0.7095
##  6:  B -0.1093
##  7:  C  0.5855
##  8:  D  0.7095
##  9:  A -0.1093
## 10:  B  0.5855
## 11:  C  0.7095
## 12:  D -0.1093
```

Un método equivalente para la extracción múltiple de variables se logra con el comando `list()`:

```
# Extracción de dos variables "B2" y "B3"
base[,list(B2,B3)]

##     B2      B3
##  1:  A  0.5855
##  2:  B  0.7095
##  3:  C -0.1093
##  4:  D  0.5855
##  5:  A  0.7095
##  6:  B -0.1093
##  7:  C  0.5855
##  8:  D  0.7095
```

```
##  9:  A -0.1093
## 10:  B  0.5855
## 11:  C  0.7095
## 12:  D -0.1093
```

Una vez seleccionadas las columnas deseadas podemos calcular estadísticos de interés por medio de una sintaxis sencilla:

```
# Obtenemos el promedio de la variable B3
base[,mean(B3)]

## [1] 0.3952333
```

```
# Obtenemos la suma y la desviación de las variables B4 y B3 respectivamente
base[,list(SUMA=sum(B4), DESVIACION=sd(B3))]

##    SUMA DESVIACION
## 1:   60  0.3763552
```

Por último, podemos reciclar los resultados obtenidos:

```
base[,.(B2, DESVIACION = sd(B3))]

##     B2 DESVIACION
##  1:  A  0.3763552
##  2:  B  0.3763552
##  3:  C  0.3763552
##  4:  D  0.3763552
##  5:  A  0.3763552
##  6:  B  0.3763552
##  7:  C  0.3763552
##  8:  D  0.3763552
##  9:  A  0.3763552
## 10:  B  0.3763552
## 11:  C  0.3763552
## 12:  D  0.3763552
```

1.8.4. Agrupación

Una actividad importante en el manejo de datos es la agrupación de información en base a una o más variables por medio del parámetro `by`.

```
# Creamos un nuevo data.table
set.seed(123)
datos <- data.table(V1=LETTERS[1:2], V2=c(1,2,3,4), V3=seq(1,8),
                    V4= abs(rnorm(8, 5, 2)))
datos

##    V1 V2 V3       V4
## 1:  A  1  1 3.879049
## 2:  B  2  2 4.539645
```

```
## 3:  A  3  3 8.117417
## 4:  B  4  4 5.141017
## 5:  A  1  5 5.258575
## 6:  B  2  6 8.430130
## 7:  A  3  7 5.921832
## 8:  B  4  8 2.469878

# Calculamos la suma de V3 para cada grupo de V1
datos[,.(Suma=sum(V3)), by=V1]

##    V1 Suma
## 1:  A   16
## 2:  B   20

# Calculamos la suma de V3 para cada grupo formado por V1 y V2
datos[,.(Suma=sum(V3)), by=.(V1,V2)]

##    V1 V2 Suma
## 1:  A  1    6
## 2:  B  2    8
## 3:  A  3   10
## 4:  B  4   12

# Numero de filas por cada grupo
datos[, .N, by=V1]

##    V1 N
## 1:  A 4
## 2:  B 4
```

1.8.5. Actualizando variables

Para actualizar una o más variables emplearemos el operador de referencia `:=`, como se muestra a continuación:

```
# Actualizamos la variable V4, redondemos la variables a 2 decimales
datos[, V4:=round(V4,2)]
# Actualizamos las variables V2 y V4
datos[, c("V2", "V4") := list(V2-1, round(sqrt(V4), 3))]
```

Una forma alternativa para la actualización de dos o más variables es la siguiente:

```
datos[,':=' (V2 = V2-1, V4 = round(sqrt(V4), 3))][]
```

1.8.6. Añadiendo variables

Para añadir variables emplearemos el operador de referencia `:=`, como se muestra en los siguientes ejemplos:

```
# Creamos la variable V5 como la suma de V2 y V3
datos[, V5 := V2+V3]
# Visualizamos la variable creada
datos
```

```
##    V1 V2 V3    V4 V5
## 1:  A  0  1 1.970  1
## 2:  B  1  2 2.131  3
## 3:  A  2  3 2.850  5
## 4:  B  3  4 2.267  7
## 5:  A  0  5 2.293  5
## 6:  B  1  6 2.903  7
## 7:  A  2  7 2.433  9
## 8:  B  3  8 1.572 11
```

```
datos[,':=' (V6 = 2*V2-V3, V7 = ifelse(V5-2*V4>2,1,0))]
datos
```

```
##    V1 V2 V3    V4 V5 V6 V7
## 1:  A  0  1 1.970  1 -1  0
## 2:  B  1  2 2.131  3  0  0
## 3:  A  2  3 2.850  5  1  0
## 4:  B  3  4 2.267  7  2  1
## 5:  A  0  5 2.293  5 -5  0
## 6:  B  1  6 2.903  7 -4  0
## 7:  A  2  7 2.433  9 -3  1
## 8:  B  3  8 1.572 11 -2  1
```

1.8.7. Eliminando variables

La eliminación de variables se realiza de una manera sencilla empleando el operador de referencia `:=`.

```
# Eliminamos la variable V7
datos[, V7 := NULL]
datos
```

```
##    V1 V2 V3    V4 V5 V6
## 1:  A  0  1 1.970  1 -1
## 2:  B  1  2 2.131  3  0
## 3:  A  2  3 2.850  5  1
## 4:  B  3  4 2.267  7  2
## 5:  A  0  5 2.293  5 -5
## 6:  B  1  6 2.903  7 -4
## 7:  A  2  7 2.433  9 -3
## 8:  B  3  8 1.572 11 -2
```

```
# Eliminamos las variables V4 y V5
datos[, c("V4", "V5") := NULL]
datos
```

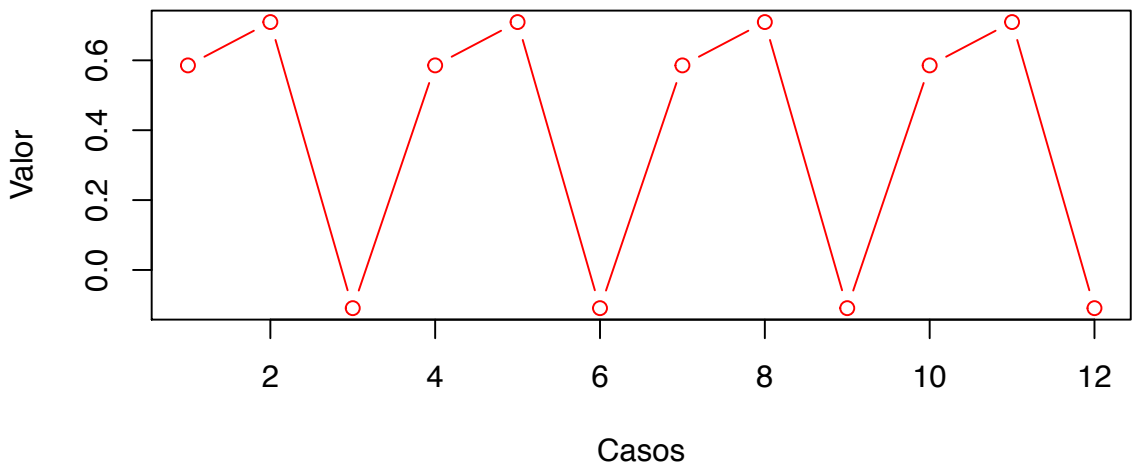
```
##    V1 V2 V3 V6
## 1:  A  0  1 -1
## 2:  B  1  2  0
## 3:  A  2  3  1
## 4:  B  3  4  2
## 5:  A  0  5 -5
## 6:  B  1  6 -4
## 7:  A  2  7 -3
## 8:  B  3  8 -2
```

```
# vector con los nombres de las variables a eliminarse
eliminar <- c("V3", "V6")
datos[, (eliminar) := NULL]
datos
```

```
##    V1 V2
## 1:  A  0
## 2:  B  1
## 3:  A  2
## 4:  B  3
## 5:  A  0
## 6:  B  1
## 7:  A  2
## 8:  B  3
```

```
base[,{print(B1)
     plot(B3, xlab='Casos', ylab='Valor', col='red', type='b')
     NULL}]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```



```
## NULL
```

1.9. Consultas mediante sentencias SQL

`sqldf` es una librería creada para ejecutar sentencias SQL desde R, el paquete soporta bases de datos SQLite, PostgreSQL y MySQL.

```
## Loading required package: gsubfn
## Loading required package: proto
## Warning in doTryCatch(return(expr), name, parentenv, handler): unable to load
shared object '/Library/Frameworks/R.framework/Resources/modules//R_X11.so':
##  dlopen(/Library/Frameworks/R.framework/Resources/modules//R_X11.so, 6): Library
not loaded: /opt/X11/lib/libfontconfig.1.dylib
##  Referenced from: /Library/Frameworks/R.framework/Resources/modules//R_X11.so
##  Reason: Incompatible library version: R_X11.so requires version 11.0.0 or later,
but libfontconfig.1.dylib provides version 10.0.0
## Loading required package: RSQLite
```

```
# Creamos un data frame sobre el cual realizaremos consultas
database <- data.frame(codigo=paste("ASR", sample(100:999, 20), sep=""),
                       pais=sample(c("AR", "BR", "EC", "CO", "ES", "US"), 20, replace=TRUE),
                       sueldo=sample(366:3600, 20),
                       estcivil=sample(c("S", "C", "V", "U"), 20, replace=TRUE))
database
```

```
##    codigo pais sueldo estcivil
## 1  ASR321   ES    778        C
## 2  ASR137   BR   2802        V
## 3  ASR394   BR   3259        C
## 4  ASR956   BR   1576        S
## 5  ASR897   AR   2514        S
## 6  ASR720   EC    672        V
## 7  ASR672   EC   1605        C
## 8  ASR987   EC   1251        U
## 9  ASR684   AR   2994        S
## 10 ASR731   AR   1812        C
## 11 ASR584   BR   2978        U
## 12 ASR628   EC   2985        U
## 13 ASR356   BR   2926        U
## 14 ASR230   US   1783        S
## 15 ASR953   AR   2796        S
## 16 ASR898   EC   2392        V
## 17 ASR710   ES   2652        C
## 18 ASR802   AR    368        V
## 19 ASR121   CO   1895        C
## 20 ASR520   BR   1073        S
```