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 tal como: variables, bases de datos, funciones, etc. son objetos. En este capítulo trataremos los diferentes tipos de objetos que R crea y manipula, incluso revisaremos estructuras de almacenamiento de datos más complejas construidas a partir de otras más 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.

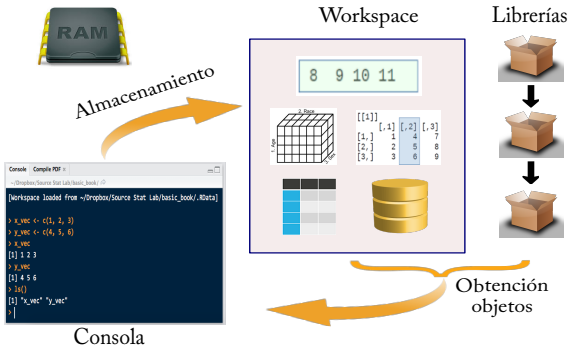


Figura 1.1: Funcionamiento espacio de trabajo

Para enlistar los objetos del área de trabajo recurrimos a los comandos `ls()` o `objects()`.

```
# Inicialmente el area de trabajo se encuentra vacia
ls()

## character(0)

# Generamos 2 vectores
x_vec <- c(1, 2, 3)
y_vec <- c(4, 5, 6)
# Los vectores creados ya se encuentran en el area de trabajo
objects()

## [1] "x_vec" "y_vec"
```

Entre los parámetros más utilizados del comando antes mencionado se encuentra `pattern`, mismo que facilita enlistar los objetos que cumplen con algún patrón (nombre, extensión, etc.).

```
# Objetos cuyo nombre tenga el caracter x
ls(pattern = "x")

## [1] "x_vec"
```

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.

La asignación puede realizarse también mediante la función `assign()`, la cual es una forma equivalente a la asignación anterior.

```
# Generamos 2 vectores nuevos
assign("x_new", c(2, 4, 6))
assign("y_new", c(3, 5, 7))
# Revisamos sus elementos
x_new

## [1] 2 4 6

y_new

## [1] 3 5 7

# Verificamos los objetos del espacio de trabajo
ls()

## [1] "x_new" "x_vec" "y_new" "y_vec"
```

En ocasiones los usuarios de R suelen realizar las asignaciones apuntando la flecha a la derecha '->', realizando el cambio obvio en la asignación. Esto no es muy recomendable debido que en la actualidad la tendencia es estandarizar la forma de escritura en R.

```
# Creamos un nuevo vector
c(5,10,15) -> x_mul
# Revisamos los elementos del vector
x_mul

## [1]  5 10 15
```

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 (R diferencia mayúsculas de minúsculas). 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 organizados 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 los data tables a diferencias de los data frames, esto debido a las diversas facilidades que poseen los primeras 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*. 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, dicha función es 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.

```
# 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 diferenciar un numero entero
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
```

```
# 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, a continuación mostramos unos ejemplos:

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

## [1] 0 0 0 0 0

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

## [1] FALSE FALSE FALSE FALSE FALSE

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

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

Observamos que la función `vector` genera vectores con sus valores por default, es decir, para el caso de vectores númericos los inicializa en 0, para el caso de vectores lógicos los inicializa en FALSE, mientras que para los vectores de caracteres los inicializa en vacíos.

Todo vector consta de dos argumentos: `mode` & `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 tiene 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 facilidad de crear un vector con diferentes tipos de elementos, sin embargo, debido al conflicto que se genera internamente por la *coerción* de los distintos elementos tenemos la siguiente jerarquía:

−						+
logical	<	**integer**	<	**double**	<	**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:

```r
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` & `integer` y al momento de conocer el tipo de vector obtenemos `numeric` en lugar de `logical`. Esto se debe básicamente a la manera como R almacena y reconoce los diferentes elementos, la siguiente tabla resume lo expuesto.

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 o double, pues en estos casos el TRUE se convierte en 1 y el FALSE en 0.

```r
# creamos un vector logico
vec <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
# coercionamos a numerico
as.numeric(vec)

## [1] 1 0 1 1 0
```

Para la coerción contamos con varias funciones que nos facilitan el trabajo tales como: `as.character()`, `as.double()`, `as.integer()`, `as.logical()`.

```r
# 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. Por esta razón hay que tener mucho cuidado al momento de forzar el coercionamiento de un objeto.

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

## [1] 2 4 6 8

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

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

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 perdidas usaremos la función `is.na( )`.

```
# 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()`.

```
# Mostramos los objetos actuales
ls()

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

# 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] "var"   "var1"  "var2"  "var3"  "vec"   "x"     "x_mul" "x_new"
##  [9] "x_vec" "xc"    "y"     "y_new" "y_vec"
```

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

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

## [1] "vec"   "x"     "x_mul" "x_new" "x_vec" "xc"    "y"     "y_new" "y_vec"
```

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

```
# 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

Ahora mostramos algunos operadores que son útiles para 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), 2)

## [1] 3 6 9 3 6 9

rep(1:2, 4)

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

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

## [1] 2 2 2 4 4 4 6 6 6

rep(c("Source", "Stat", "Ecuador"), 2)

## [1] "Source"  "Stat"    "Ecuador" "Source"  "Stat"    "Ecuador"
```

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

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
"+" (3, 5)

## [1] 8

"%/%" (8, 3)

## [1] 2
```

```
"log" (24, 3)

## [1] 2.892789

# creamos dos vectores
x <- c(1, 3, 5, 7, 9)
y <- c(2, 4, 6, 8, 10)
# sumamos los vectores
x + y

## [1]  3  7 11 15 19

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

## [1]  3  7 11 15 19
```

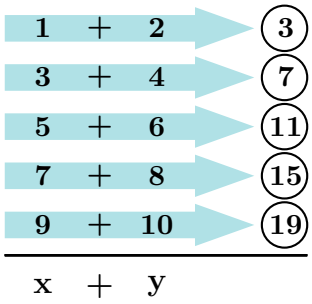


Figura 1.2: Suma de vectores de igual longitud

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

# 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
```

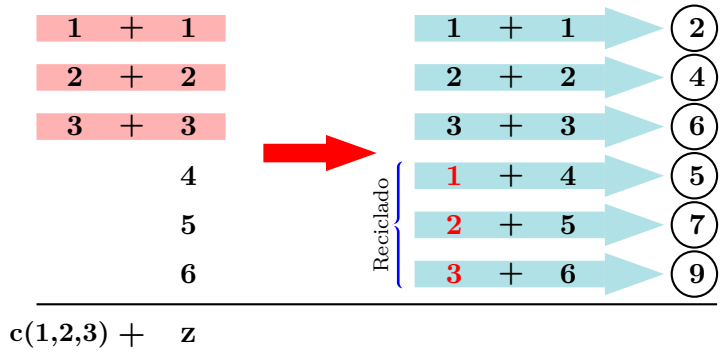


Figura 1.3: Suma de vectores de diferente longitud

El primer ejemplo muestra una advertencia debido que el vector más pequeño no es multiplo del vector más grande, lo cual no ocurre en el segundo ejemplo. 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
```

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[1], las cuales son utilizadas con frecuencia en distintos lenguajes de programación para verificar si un objeto cumplen ciertas condiciones deseadas.

```
# 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 cuando 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
```

1.2.6. Atributos

Todos los objetos en R tienen una lista de atributos, misma que es utilizada para almacenar la metadata[2] de los objetos. El acceso a los atributos de un objeto se lo puede realizar individualmente o a 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.

[1] George Boole (18151864)
[2] La Metadata es data que describe otra data. Es información que describe el contenido un archivo u 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("a", "b", "c", "d", "e", "f")
x

## a b c d e f
## 1 3 5 6 8 9

# 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] "a" "b" "c" "d" "e" "f"

# 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] "a" "b" "c" "d" "e" "f"
## attr(,"class")
## [1] "Mat23"

# listamos todos los atributos disponibles
attributes(x)

## $names
## [1] "a" "b" "c" "d" "e" "f"
##
## $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] "a" "b" "c" "d" ...
```

1.3. Factores

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

1.3.1. Creación

La función `factor()` tiene 3 argumentos de interés al momento de crear nuevas variables:

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

## [1] "character"

table(genero)

## genero
## f m
## 3 3

# Creacion del factor
genero <- factor(genero,
                 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
```

- El nombre del vector atómico a convertirse en factor "*genero*".
- Los niveles o valores que el factor puede recibir. En el ejemplo, el vector "*genero*" está limitado a los valores "`m`" y "`f`", sin embargo podemos incluir el valor "`o`" en 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.
- 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.

Los factores se almacenan como un vector de valores 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"

# Visualizamos los atributos
attributes(genero)

## $levels
## [1] "Masculino" "Femenino"  "Otro"
##
## $class
## [1] "factor"
```

Podemos acceder a los niveles permitidos por un factor mediante `levels()`, así como también a sus etiquetas por medio de `labels()`.

```
levels(genero)

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

labels(genero)

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

1.3.2. Modificación

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

```
# 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 una posibilidad nueva que no estuvo considerada
region[4] <- "Extranjero"

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

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=="C"

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

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

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

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

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

1.4. Matrices

Las matrices son arreglos bidimensionales empleados para almacenar elementos de un mismo tipo.

1.4.1. Creación

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

- Un vector con los elementos que formarán parte de la matriz.
- Un valor numérico `nrow` que define el número de filas.
- Un valor numérico `ncol` que define el número de columnas.
- 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 de la función `rbind()`:

```
# Generamos 3 vectores de igual longitud
vec1 <- c(1, 5, 8)
vec2 <- c(3, 6, 9)
vec3 <- c(2, 4, 7)
# Creamos la matriz usando los vectores como filas
Mat1 <- rbind(vec1, vec2, vec3)
Mat1

##      [,1] [,2] [,3]
## vec1    1    5    8
## vec2    3    6    9
## vec3    2    4    7

class(Mat1)

## [1] "matrix"
```

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

```
# Creamos la matriz usando los vectores como columnas
Mat2 <- cbind(vec1, vec2, vec3)
Mat2

##      vec1 vec2 vec3
## [1,]    1    3    2
## [2,]    5    6    4
## [3,]    8    9    7

class(Mat2)

## [1] "matrix"
```

1.4.2. Modificación

1.4.3. Operaciones

1.4.4. Atributos

Las matrices presentan un número mayor de atributos respecto a los vectores y factores tales como:

- `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.

```
# Numero de filas
nrow(Mat23)

## [1] 2

# Numero de columnas
ncol(Mat23)

## [1] 3

# Asignamos nombres a las columnas
colnames(Mat23) <- c("a", "b", "c")
# Asignamos nombres a las filas
rownames(Mat23) <- c("Diego", "Paul")
# Visualizamos los cambios
Mat23

##       a b  c
## Diego 2 6 10
## Paul  4 8 12
```

1.5. Listas

1.6. Arrays

Un arreglo (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 reglas que hemos descrito para las matrices. La sintaxis para definir un arreglo es

```
array(data, dim)
```

Las componentes data y dim deben presentarse como una sola expresión, por ejemplo

```
c(2,4,6,8,10)
c(2,3,2)
x <- array (1:24, c(3,4,2))
```

produce un arreglo tridimensional: la primera dimensión tiene tres niveles, la segunda tiene cuatro y la tercera tiene dos. Al imprimir el arreglo R comienza con la dimensión mayor y va bajando hacia la dimensión menor, imprimiendo matrices bidimensionales en cada etapa.

1.7. Data Frames

Tipo particular de listas de gran utilidad para el trabajo estadístico.

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
```

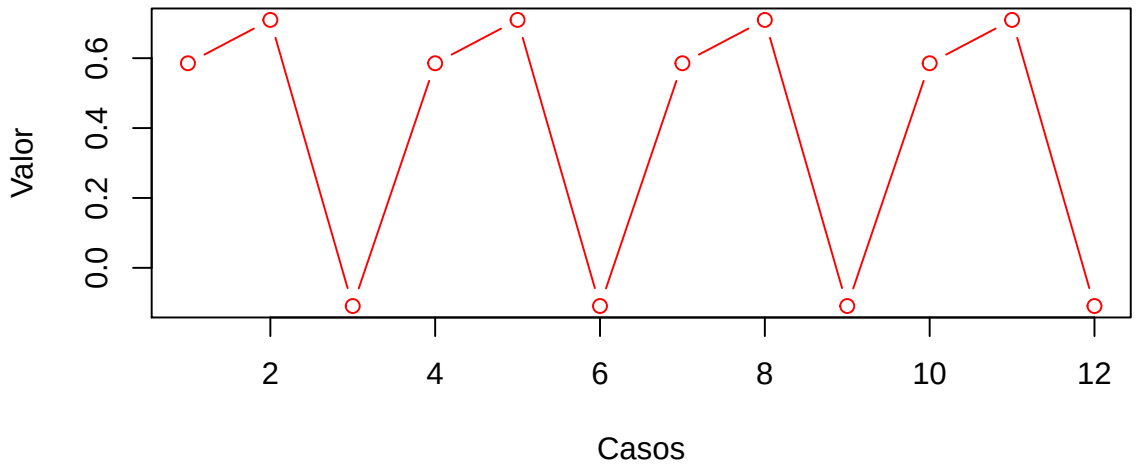
1.8.7. Eliminando variables

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
```