Introducción a R y R Studio

Carlos Ignacio Patiño

Junio 3, 2015

Índice

1. Introducción

1.1. Objetivo

El objetivo fundamental del presente tutorial es guiar al estudiante en el proceso de descarga e instalación del paquete de análisis estadístico R y del paquete de ambiente de desarrollo R Studio, e introducir a éste a los comandos básicos y a la sintaxis de programación empleada por R.

1.2. El lenguaje R

R es un lenguaje de programación y un *software* para computación estadística y gráficos. El lenguaje R es ampliamente empleado por analistas estadísticos y mineros de datos para el desarrollo de programas estadísticos y análisis de datos. Una de las principales características de R es su naturaleza de código abierto (*open source*), lo cual permite la creación de un amplio ambiente de desarrolladores que colaboran constantemente con el desarrollo de paquetes de todo tipo. Lo anterior, permite mantener actualizado el *software* y garantiza las disponibilidad de paquetes (o librerías) para llevar a cabo o implementar casi cualquier método de análisis estadístico o de minería de datos.

Para mayor información sobre el lenguaje R, visitar las siguientes páginas:

- http://http://www.r-project.org/
- http://www.r-bloggers.com/
- http://www.rstudio.com/products/RStudio/

1.3. R Studio: Ambiente Integrado de Desarrollo

R Studio es un poderoso complemento para el lenguaje R ya que permite contar con un ambiente integrado para el desarrollo de análisis estadístico. R Studio cuenta con una consola de comandos, un editor de código que soporta ejecución directa de código, así como herramientas para gráficos, monitoreo de historia, *debugging* y gestión del espacio de trabajo. Otra característica importante de R Studio es que cuenta también con herramientas de edición de documentos tipo .Rmd (R markdown), los cuales permiten la fácil creación de documentos dinámicos (y reproducibles), presentaciones y reportes de análisis directamente desde R. Esta herramienta combina los comandos básicos del lenguaje markdown con la posibilidad de incluir porciones de código de R en el mismo documento, de forma que sus resultados se pueden presentar en el documento.

2. Instalación

Instalación de la versión base de R

- Dirigirse a la página de espejos de CRAN: http://cran.r-project.org/mirrors.html, y seleccione el espejo más cercano a su ubicación. En el caso de Colombia, el espejo se encuentra ubicado en la página de la Universidad ICESI.
- Dar click al espejo más cercano (en este caso http://www.icesi.edu.co/CRAN/).
- Una vez ubicado en la página de descargas, se puede seleccionar alguna de las versiones precompiladas del sistema base (usuarios de Mac o Windows). Dar click al link adecuado para su sistema operativo.
- Descargue la versión más reciente: En el caso de usuarios Mac con sistemas operativos Mac OS X 10.9 o superiores, la versión más reciente (al momento de escribir este tutorial) es R 3.2.0. El archivo que se debe descargar es R-3.2.0.pkg. Para usuarios de Windows que están instalando R por primera vez, se debe dar click al link denominado “base” y proceder a descargar la versión 3.2.0 para Windows (R-3.2.0.exe).
- Una vez descargada la versión adecuada para el sistema operativo, proceder a su instalación.

Instalación de R Studio

- Dirigirse a la página inicial de R Studio: http://www.rstudio.com/, y dar click al boton de “Download RStudio”.
- Seleccionar la versión adecuada entre la de escritorio y la de servidor. En este caso, seleccionar la versión de escritorio.
- Una vez se está en la página de la versión de escritorio, dar click al boton “Download RStudio Desktop”, que se encuentra en la última fila de la columna donde se muestran las características de la versión de código abierto.
- En esta página se encontrarán los instaladores para todas las plataformas que soporta R Studio. La versión más reciente de R Studio es la 0.98.1103.
- R Studio requiere R versión 2.11.1 (o más). Es necesario tener instalado primero R para que la instalación de R Studio funcione adecuadamente.
- Descargar el instalador adecuado para su sistema operativo.
- Proceder con la instalación.

3. Introducción a R

3.1. Comandos básicos

El comando fundamental de R es el comando de **asignación** `<-` el cual permite asignar valores a objetos:

```
# El siguiente comando asigna el valor de 10 al objeto "a"
a <- 10
```

Existen dos formas de **imprimir en pantalla** el valor de un objeto: la forma explícita empleando la función `print()`, y la forma implícita simplemente escribiendo el nombre del objeto que se quiere imprimir

```
# Imprimir en pantalla: version explícita
print(a)
```

```
[1] 10
```

```
# Version implícita para imprimir el valor asignado a un objeto (en pantalla)
a
```

```
[1] 10
```

Ambas formas generan el mismo resultado, imprimiendo un vector numerico de dimension 1. Este ejemplo sencillo muestra como en el lenguaje R **existen varias maneras de obtener el mismo resultado.**

Las funciones `c()`, `seq()` y `rep()` se emplean para generar vectores (detalles más adelante).

```
# ":" permite simplificar y en lugar de escribir 1,2,3 se puede escribir 1:3
a <- c(1:3,10,3)
print(a)
```

```
[1]  1  2  3 10  3
```

```
# Secuencia de 1 a 10
b <- seq(1,10,by=2)
b
```

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

```
# Repetir 2, 3 veces
c <- rep(2, times=3)
c
```

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

```
# Repetir 2 y 4, 3 veces cada uno
d <- rep(c(2,4), each=3)
d
```

```
[1] 2 2 2 4 4 4
```

Otros comandos y funciones fundamentales:

- `getwd()` permite imprimir el directorio actual de trabajo.
- `setwd()` permite cambiar el directorio de trabajo.
- `list.files()` permite mostrar el listado de archivos o carpetas presentes en el directorio actual de trabajo.
- `ls()` permite imprimir el listado de todos los objetos presentes en el ambiente de trabajo.
- `install.packages()` instala librerias (o paquetes) adicionales a los incluidos en la version base de R. El nombre del paquete debe ir como parametro entre comillas.
- Para cargar una libreria ya instalada se debe emplear la funcion `library()`. En este caso, el nombre de la libreria no debe ir rodeado de comillas.
- `rm()` permite eliminar el objeto u objetos incluidos como parametros.
- `class()` permite obtener el tipo del valor asignado al elemento incluido como parametro.

```
# Obtener directorio de trabajo y los componentes del espacio de trabajo global
getwd()
```

```
[1] "/Users/carlospatino/Google Drive/Metodos Cuantitativos InSoc/Intro R"
```

```
ls()
```

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

```
b <- c(1,2,3,4)
ls()
```

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

```
rm(b)

c <- "hola"
rm(list=ls())
ls()
```

```
character(0)
```

3.2. Clases de objetos en R

R emplea los siguientes tipos de objetos:

- Vectores
- Listas
- Arrays
- Matrices
- Tablas
- Data Frames

3.2.1. Vectores

El vector de dimensión 1x1 es el elemento atómico en R. Un vector puede contener objetos numéricos o de tipo caracter. Sin embargo, al crear un vector con esta mezcla de objetos, R forzará todos los elementos del vector a la clase de tipo caracter.

```
# Vector atómico
a <- "Universidad ICESI"

# Para corroborar que se trata de un vector de dimensión 1:
length(a)
```

```
[1] 1
```

```
# El valor asignado a "a" es de clase caracter
class(a)
```

```
[1] "character"
```

```
# R no arroja error cuando se asigna una mezcla de valores numéricos y
# valores de tipo caracter. Sin embargo, todos los elementos del vector se
# convierten a caracter:
mix <- c(20,2,3,NA,"icesi")

# Sin embargo, la cadena "NA" es empleada por R para asignar un valor no
# disponible a dicho objeto. En este caso, R mantiene esa asignacion y no
# obliga a la cadena "NA" a ser caracter
mix
```

```
[1] "20"    "2"     "3"     NA      "icesi"
```

También existen vectores de clase *booleana*. Por ejemplo, si se emplea la función `is.na()` en el vector `mix`, generado previamente, el resultado es un vector de tipo *booleano* con valores `FALSE` o `TRUE` dependiendo de si se cumple o no la condición evaluada. En el caso de la función en mención, la condición es si existe o no un valor no disponible, o `NA`. Cabe mencionar que detrás de los valores "FALSO" y "VERDADERO", en realidad se encuentran los valores numéricos 0 y 1. Por lo tanto, es posible "sumar" los valores al interior de un vector *booleano*, estrategia empleada usualmente para calcular el número de *missing values* existentes en una columna o base completa.

```
# is.na() resulta en un vector booleano
is.na(mix)
```

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

```
# En este caso, el cuarto elemento del vector mix es NA, por lo que el
# resultante vector muestra un TRUE como cuarto elemento

# Si sumamos los valores al interior de dicho resultado, podemos ver el
# número de valores missing presentes en el vector mix
sum(is.na(mix))
```

```
[1] 1
```

El comando anterior muestra cómo es posible encadenar funciones en R. El resultado de 1, señala el número de valores `NA` en el vector `mix`.

Si se asigna una mezcla de valores numéricos y *booleanos* a un vector dado, R obliga dicho vector a contener solamente elementos de clase numérica. La forma en que R hace esto es a partir de los valores detrás de los valores *booleanos*, 1 y 0 para verdadero y falso, respectivamente. También es posible convertir un vector *booleano* a uno numérico, empleando la función `as.numeric()`.

```
# R obliga a todos los elementos de esta lista a ser de clase numérica
mix <- c(1,0,FALSE,2,TRUE)
mix
```

```
[1] 1 0 0 2 1
```

```
# Es posible convertir un vector de una clase a otra.
mix2 <- c(3:5,NA,10,NA)
as.numeric(is.na(mix2))
```

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

Para acceder a los elementos de una lista se emplean los **índices**. En R, los índices, a diferencia de otros lenguajes como Python, empiezan desde 1. Por lo tanto, un vector de tamaño 10 tendrá índices que van de 1 a 10. Si se desea acceder al quinto elemento de dicho vector, se usará el índice 5. Es posible también acceder a múltiples elementos al mismo tiempo.

```
vec <- seq(1,20,by=2)
vec[5]
```

```
[1] 9
```

```
vec[5:7]
```

```
[1]  9 11 13
```

```
vec[c(1,5,10)]
```

```
[1]  1  9 19
```

Finalmente, es posible asignar **nombres** a los elementos de un vector.

```
vec.nombres <- rnorm(5) # rnorm genera datos aleat. usando una dist. normal
# La funcion names() permite asignar nombres a un vector
names(vec.nombres) # por si sola muestra los nombres (NULL en este caso)
```

```
NULL
```

```r
# Pero se puede asignar un vector con cadenas como nombres:
names(vec.nombres) <- c("Caso 1","Caso 2","Caso 3","Caso 4","Caso 5")
vec.nombres
```

```
    Caso 1     Caso 2     Caso 3     Caso 4     Caso 5
-0.9000920 -0.8371124 -1.0282418 -0.8818473  0.9108074
```

Lo anterior permite también acceder a los elementos de un vector, llamandolos por su nombre:

```r
vec.nombres["Caso 3"]
```

```
   Caso 3
-1.028242
```

```r
vec.nombres[c("Caso 1", "Caso 5")]
```

```
    Caso 1     Caso 5
-0.9000920  0.9108074
```

3.2.2. Listas

Las listas son vectores especiales que permiten no sólo contener elementos con nombres, sino también contener elementos de toda clase, sin la obligación de convertir todos sus elementos a una misma clase. La lista es una estructura de datos que permite una gestión más compleja de información. Adicionalmente, la estructura de listas permite incluir vectores como elementos dentro de la lista.

```r
# Una lista con elementos de clase caracter, numérico y booleano
l.simple <- list(nombre="Ignacio", apellido="Florez", edad=33, estudiante=FALSE)

# Una lista con vectores
l.comp <- list(nombre=c("Andres","Juan"),
               apellido=c("Montes","Castro"),
               edad=c(30,22),
               estudiante=c(FALSE,TRUE))
```

Ambas listas tienen 4 elementos. Sin embargo, los elementos de la segunda lista son vectores.

```r
length(l.simple); length(l.comp)
```

```
[1] 4
```

```
[1] 4
```

Para acceder a los elementos de una lista se pueden usar diversos métodos.

1. Empleando los índices. En este caso hay dos formas: usar un sólo *bracket* ("[]") o usar *bracket* doble ("[[]]").

```
# Usando el single bracket se obtiene:
l.simple[1] # valor y nombre del elemento
```

```
$nombre
[1] "Ignacio"
```

```
l.comp[1] # en este caso el nombre del elemento y el vector
```

```
$nombre
[1] "Andres" "Juan"
```

```
# Si se emplea el double bracket:
l.simple[[1]] # solo el valor
```

```
[1] "Ignacio"
```

```
l.comp[[1]] # solo el vector
```

```
[1] "Andres" "Juan"
```

2. Empleando los nombres de los valores.

```
# Tambien aplica el single o double bracket:
l.simple["nombre"]
```

```
$nombre
[1] "Ignacio"
```

```
l.comp[["edad"]]
```

```
[1] 30 22
```

3.2.3. Matrices

Las matrices tienen un comportamiento similar al de los vectores con la diferencia de que tienen dos dimensiones. Las matrices se generan empleando la función `matrix()`. La función `matrix()` recibe hasta 5 parámetros:

- `data`: valores a incluir en la matriz
- `nrow`: número de filas
- `ncol`: número de columnas
- `byrow`: forma en la que se rellena la matriz. Por default este parámetro está definido como `FALSE` por lo que la matriz se rellenará columna por columna
- `dimnames`: opcional, nombres para las dimensiones

```r
# Matriz cuadrada con valores aleatorios
vals <- runif(9,min=1,max=10)
mat <- matrix(vals,nrow=3,byrow=TRUE)
# Al fijar byrows=TRUE pedimos a R que construya la matriz tomando los datos
# de vals y asignándolos fila por fila:
vals; mat
```

```
[1] 4.416195 9.537839 6.347346 7.884254 7.234761 1.972107 1.443566 2.447796
[9] 8.097414
```

```
         [,1]     [,2]     [,3]
[1,] 4.416195 9.537839 6.347346
[2,] 7.884254 7.234761 1.972107
[3,] 1.443566 2.447796 8.097414
```

Note que al fijar el parámetro `nrow` igual a 3 no es necesario declarar un valor para el parámetro `ncol` ya que R, de acuerdo a la cantidad de elementos en el vector fuente (`vals`), construye la matrix empleando todos los valores del vector fuente. De faltar valores en el vector, R **reutiliza los valores** del vector para completar las dimensiones solicitadas en la función.

Los valores de una matriz se acceden de forma similar a los vectores, empleando sus índices (o nombres, si es el caso)

```r
mat[3,1] # primera dim es la fila, segunda es la columna
```

```
[1] 1.443566
```

```r
# Es posible agregar los nombres a las dimensiones luego de ser
# creada la matriz:
dimnames(mat) <- list(c("Fila1","Fila2","Fila3"),
                      c("Col1","Col2","Col3"))
mat
```

```
          Col1     Col2     Col3
Fila1 4.416195 9.537839 6.347346
Fila2 7.884254 7.234761 1.972107
Fila3 1.443566 2.447796 8.097414
```

```r
mat["Fila1",]
```

```
    Col1     Col2     Col3
4.416195 9.537839 6.347346
```

3.2.4. *Arrays*

Esta estructura de datos es similar a una matriz con la diferencia de que los *arrays* pueden tener más de dos dimensiones. En otras palabras, !las matrices son *arrays* de dos dimensiones!

```r
y <- array(1:16, dim=c(4,2,2)) # dim (filas, cols, capas)
y
```

```
, , 1

     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

, , 2

     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16
```

3.2.5. Tablas

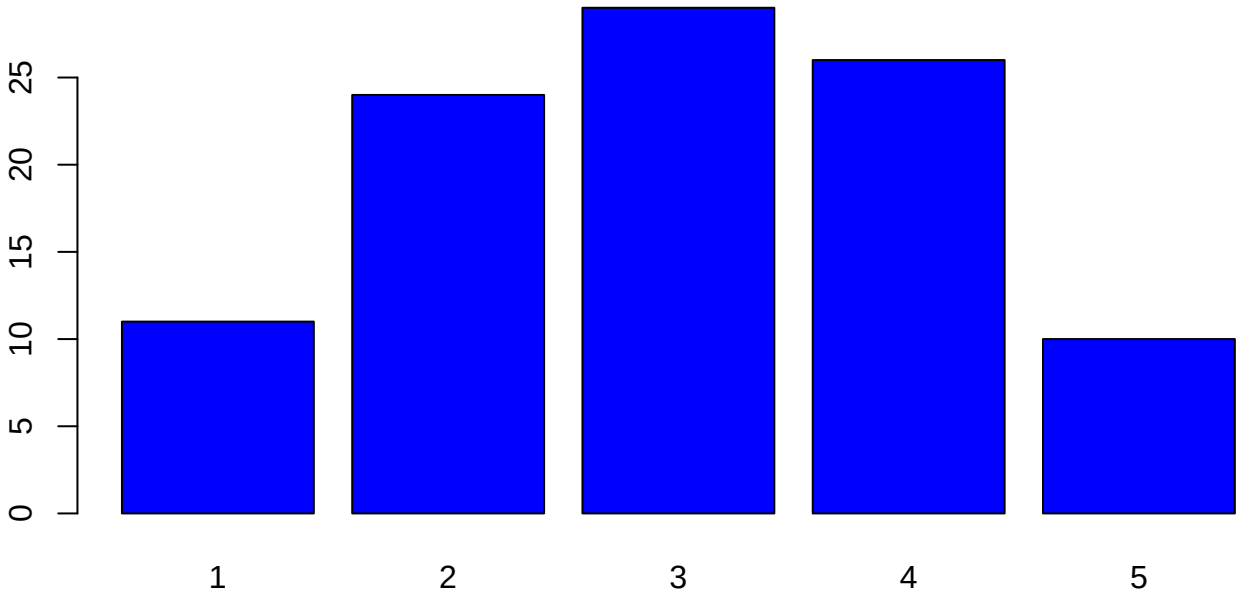
La función `table()` se emplea para crear tablas de frecuencias o tabulaciones cruzadas a partir de datos contenidos en un vector o *data frame*.

```
y <- round(runif(100,1,5),0)
table(y)
```

```
y
 1  2  3  4  5
11 24 29 26 10
```

Los dos comandos anteriores generan un vector de 100 elementos con números entre 1 y 5 (distribuidos uniformemente) y redondeados para que no tengan decimales; posteriormente se genera una tabulación de dicho vector, la cual muestra cuántas veces aparece cada número en el vector. Otra forma de ver estos resultados es generar un gráfico de barras (sobre gráficos se hablará más en detalle en su correspondiente tutorial, por ahora, no preste mucha atención a los comandos que vienen a continuación).

```
tbl <- table(y)
barplot(tbl, names.arg=names(tbl), col="blue")
```



3.2.6. *Data Frames*

Los *data frames* son, desde el punto de vista de un curso de análisis y visualización de conjuntos de datos, la estructura más importante y robusta al momento de llevar a cabo análisis de información empleando métodos de aprendizaje estadístico y minería de datos. Por este motivo, más adelante se verá un tutorial dedicado a esta estructura y a un paquete que complementa dicha estructura y que se denomina `dplyr`. La librería `dplyr` contiene un conjunto de funciones que facilitan la manipulación de grandes bases de datos estructuradas como *data frames*. Por ahora, miraremos los conceptos básicos alrededor de esta estructura. La función `data.frame()` se emplea para crear un *data frame* desde cero.

```
# Por estetica en el código, se construyen las columnas del data frame una
# por una, y luego se emplea la función para construir el data frame.
nombre <- c("tomate","cebolla","zanahoria","berenjena","lechuga")
vlr <- c(1000,1500,2500,3000,1500)
disp <- c(FALSE, TRUE, TRUE, TRUE, TRUE)

mercado <- data.frame(nombre=nombre,valor.kg=vlr,disponible=disp)

# Y para asegurarse que mercado es un data frame:
class(mercado)
```

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

```
# La función str es muy usada cuando se está explorando por primera vez un
# data frame desconocido ya que permite dar un vistazo a las columnas y las
# clases a la que pertenecen, así como algunos de sus valores (muy util
# cuando se trata de un data frame de cientos de miles de registros)
str(mercado)
```

```
'data.frame':   5 obs. of  3 variables:
 $ nombre    : Factor w/ 5 levels "berenjena","cebolla",..: 4 2 5 1 3
 $ valor.kg  : num  1000 1500 2500 3000 1500
 $ disponible: logi  FALSE TRUE TRUE TRUE TRUE
```

Note que si bien el nombre de la verdura inicialmente se creó como un caracter, al generar el *data frame* la función convierte esta columna al tipo "factor". Las columnas de este tipo son clave en la implementación de algoritmos para la exploración y modelado de bases de datos. Este tipo de valor representa valores cualitativos no ordinales. Los factores se guardan en R como enteros con su correspondiente conjunto de caracteres que se usan cuando el factor es mostrado en pantalla (output). Los factores representan una manera muy eficiente de almacenar caracteres, ya que cada valor único se guarda sólo una vez y los datos como tal, como se mencionó inicialmente, son almacenados como un vector de enteros. De esta manera, si bien en pantalla aparecen las palabras "tomate", "cebolla", etc, en realidad, los datos están almacenados como enteros (1, 2, etc).

```
# Para revisar los niveles de un factor:
levels(mercado$nombre) # Note el uso de "$" para llamar la columma de interés
```

```
[1] "berenjena" "cebolla"   "lechuga"   "tomate"    "zanahoria"
```

Sin embargo, rara vez estaremos escribiendo directamente los valores para las columnas de los *data frames* que necesitamos para llevar a cabo algún tipo de análisis. En la siguiente sección veremos de manera breve como cargar datos desde diferentes fuentes como Excel (csv, xlsx, etc) o archivos de texto (txt, tab, etc). Por ahora, usaremos un *data frame* de ejemplo que viene con la librería `ggplot2`, una librería que emplearemos mucho en próximos tutoriales, y que se presenta como una alternativa al sistema base de gráficos en R y al sistema más avanzado de gráficos `lattice`.

```
# Usamos la librería ggplot2
# si no la tiene instalada, correr install.packages("ggplot2") para instalar
library(ggplot2)

# ggplot2 contiene el data frame mpg, el cual incluye información de
# consumo de combustible (millas por galón) para algo más de 200 vehículos
str(mpg)
```

```
'data.frame':   234 obs. of  11 variables:
 $ manufacturer: Factor w/ 15 levels "audi","chevrolet",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ model       : Factor w/ 38 levels "4runner 4wd",..: 2 2 2 2 2 2 2 3 3 3 ...
 $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
 $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
 $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
 $ trans       : Factor w/ 10 levels "auto(av)","auto(l3)",..: 4 9 10 1 4 9 1 9 4 10 ...
 $ drv         : Factor w/ 3 levels "4","f","r": 2 2 2 2 2 2 2 1 1 1 ...
 $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
 $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
 $ fl          : Factor w/ 5 levels "c","d","e","p",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ class       : Factor w/ 7 levels "2seater","compact",..: 2 2 2 2 2 2 2 2 2 2 ...
```

```
# Usamos head() para revisar los primeros registros del data frame
head(mpg)
```

```
  manufacturer model displ year cyl      trans drv cty hwy fl   class
1         audi    a4   1.8 1999   4   auto(l5)   f  18  29  p compact
2         audi    a4   1.8 1999   4 manual(m5)   f  21  29  p compact
3         audi    a4   2.0 2008   4 manual(m6)   f  20  31  p compact
4         audi    a4   2.0 2008   4   auto(av)   f  21  30  p compact
5         audi    a4   2.8 1999   6   auto(l5)   f  16  26  p compact
6         audi    a4   2.8 1999   6 manual(m5)   f  18  26  p compact
```

```r
# tail() permite explorar los últimos datos en nuestro data frame
tail(mpg, n=3)
```

```
    manufacturer  model displ year cyl      trans drv cty hwy fl   class
232   volkswagen passat   2.8 1999   6   auto(l5)   f  16  26  p midsize
233   volkswagen passat   2.8 1999   6 manual(m5)   f  18  26  p midsize
234   volkswagen passat   3.6 2008   6   auto(s6)   f  17  26  p midsize
```

```r
# Para revisar una fila particular:
mpg[123,] # fila 123
```

```
    manufacturer              model displ year cyl    trans drv cty hwy fl
123         jeep grand cherokee 4wd     3 2008   6 auto(l5)   4  17  22  d
    class
123   suv
```

```r
# O un grupo de filas para una columna particular:
mpg[10:12,"hwy"]
```

```
[1] 28 27 25
```

3.3. Cargue de datos (introducción)

Durante este curso lo más común va a ser encontrarse con bases de datos que se encuentran almacenadas como archivos planos o archivos de Excel (usualmente xls, preferiblemente csv). R base viene cargado con funciones que permiten cargar datos o tablas en estos tipos de archivos comunes. Para archivos en otros formatos, como SAS, SPSS, Stata, Matlab, etc, existen funciones que el estudiante podrá explorar en internet, y que podrán ser descargadas para su posterior uso al cargar información expresada en tales formatos.

- `read.table()` es la función genérica que se emplea en R para leer y cargar datos en archivos en formato de tablas. Esta función crea un *data frame* por lo que no es necesario emplear la función `data.frame()` sobre el elemento cargado.
- `read.csv()` es una extensión de la función anterior, que está diseñada para leer archivos en formato csv.

Para revisar los parámetros que pide una función en R, sólo es necesario emplear la función `args()`:

```r
args(read.table)
```

```
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
    col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown", text, skipNul = FALSE)
NULL
```

Como se ve en el resultado anterior, la función `read.table()` acepta una gran variedad de argumentos opcionales, los cuales permiten adaptar la función a diferentes archivos tabulares.

A continuación, se emplea la función `download.file()` para descargar información sobre accidentes de transito ocurridos en el municipio de Hato Corozal Casanare durante el año 2013. Esta información se encuentra disponible en el Catálogo de Datos Abiertos del Gobierno de Colombia (http://www.datos.gov.co/frm/catalogo/frmCatalogo.aspx?dsId=62290). El código a continuación muestra la serie de comandos necesaria para descargar directamente el archivo csv de la página de Datos Abiertos y para leer el archivo a un *data frame*, empleando la función auxiliar `read.csv()`. Primero se asigna la cadena de texto con el *path* directo al archivo csv. Posteriormente se descarga el archivo a la carpeta establecida como directorio de trabajo. Finalmente, se lee el archivo. Una inspección inicial al archivo publicado en la página señala que algunos campos vienen en blanco. Por lo tanto, al emplear la función de cargue del archivo, se especifica "" en el argumento `na.strings` de la función. El archivo en formato csv se puede descargar directamente siguiendo el siguiente link:

http://www.datos.gov.co/frm/catalogo/frmDescargaArchivo.aspx?FILENAME=Alcaldia_de_Hato_Corozal.accidentetransito.csv

```
url <- "http://www.datos.gov.co/frm/catalogo/frmDescargaArchivo.aspx?FILENAME=Alcaldia_de_Hato_Corozal.a
download.file(url, destfile="accidentes.csv", method="curl")
accidentes <- read.csv("accidentes.csv", header=TRUE, na.strings="")
dim(accidentes)
```

```
[1] 50  6
```

Todas las columnas de este *data frame* son automaticamente generadas como factores. En próximos tutoriales veremos como convertir columnas de fechas al formato adecuado para su adecuado análisis. Por el momento, podemos revisitar la función para la generación de tablas y así hacer una exploración rápida de los datos presentes en la base descargada.

```
# Conteo según tipo de accidente
table(accidentes$tipo_accidente)
```

```
        Atropello             Caida            Choque
                4                19                 8
    Choque Mortal   Cierre Vehiculo      Falla Humana
                1                 5                 5
   Falla Mecanica Perdida de Control        Salida Via
                2                 4                 1
      Volcamiento
                1
```

```
# Conteo de casos según tipo de vehículo
table(accidentes$tipo_vehiculo)
```

```
                Camion              Camioneta             Motocicleta
                     1                      1                      46
Motocicleta y Automovil
                     1
```

```
# Otra forma de usar la función table, para ver crosstabs
table(accidentes$tipo_vehiculo,accidentes$lesiones)
```

```
                         No Si
  Camion                  0  1
  Camioneta               1  0
  Motocicleta             0 46
  Motocicleta y Automovil  1  0
```

Se observa como en el municipio de Hato Corozal 47 de los 49 accidentes reportados durante 2013 involucraron una motocicleta. De los casos que involucraron sólo motocicletas, todos presentaron lesiones.

3.4. Operaciones

En esta sección se revisa brevemente las operaciones más comunes entre vectores y matrices, así como algunos métodos para la manipulación de *data frames*

- Concatenando vectores
- Suma
- Resta
- Multiplicación
- División
- Subsetting *data frames*

3.4.1. Concatenando vectores

Concatenar vectores es una forma alternativa de construir vectores:

```
x <- round(runif(5,1,20),0)
y <- round(runif(5,50,70),0)
z <- c(x,y)
z
```

```
 [1] 10  1 15  2 12 50 52 66 54 67
```

3.4.2. Suma (y resta)

En el caso de la suma de vectores, R cuenta con la ventaja de que no es necesario que las dimensiones concuerden. En otras palabras, si se intenta sumar sumar un vector de dimensiones 1x3 a uno de dimensiones 1x1, lo que hace R es artificialmente replicar el unico elemento del segundo vector para que éste concuerde en dimensiones y se pueda sumar al primer vector.

```
x <- c(1,2,3)
y <- 2
x + y # en este caso "y" es empleado como [2,2,2]
```

```
[1] 3 4 5
```

```
y <- c(2,1)
x + y # en este caso "y" se convierte en [2,1,2] antes de ser usado
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
[1] 3 3 5
```

En la última suma, R arroja un *Warning* que nos dice que el tamaño del objeto más grande (x en este caso) no es míltiplo del tamaño del mas corto (y).

En cuanto a la suma de matrices, la lógica es la misma:

```
x <- matrix(c(1,2,3,4), nrow=2)
y <- matrix(c(2,2,2,2), nrow=2)
x; y
```

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

```
     [,1] [,2]
[1,]    2    2
[2,]    2    2
```

```
x - y
```

```
     [,1] [,2]
[1,]   -1    1
[2,]    0    2
```

Sin embargo, para sumar o restar matriz con matriz, las dimensiones deben coincidir. De no ser así, R arroja un error y no permite la operación. En cuanto a la operación entre matrices y escalares, ésta sí es permitida por R.

```
x + 10
```

```
     [,1] [,2]
[1,]   11   13
[2,]   12   14
```

3.4.3. Multiplicación (y división)

Para multiplicación y división elemento por elemento, se deben emplear los operadores "*ז "/"

```
x; y
```

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

```
     [,1] [,2]
[1,]    2    2
[2,]    2    2
```

```
x * y
```

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

```
x / y
```

```
     [,1] [,2]
[1,]  0.5  1.5
[2,]  1.0  2.0
```

Para multiplicación matricial, se debe tener en cuenta la regla de multiplicación matricial (si A y B son matrices con dimensiones kxn y qxj, respectivamente, para poder multiplicar AxB las dimensiones n y q deben ser iguales) y se debe usar el operador " %*%".

```
x <- matrix(c(1,2,3,4), nrow=2)
y <- matrix(c(2,2), nrow=2)
x; y
```

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

```
     [,1]
[1,]    2
[2,]    2
```

```
x %*% y
```

```
     [,1]
[1,]    8
[2,]   12
```

3.4.4. Subsetting (filtrado en *data frames*)

Como es el caso en todo lo que se puede hacer en R, existen diversas maneras de seleccionar columnas o filas en un *data frame*. La manera fundamental de seleccionar columnas o filas en una estructura de este tipo es empleando los índices. Por ejemplo, si queremos explorar solamente las filas 10 y 11, y las columnas 2 y 3 del conjunto de datos sobre accidentes de tránsito, el siguiente código nos permite llevar a cabo esa selección:

```
accidentes[10:11,c(2,3)]
```

```
          lugar_accidente tipo_accidente
10  Km 3 Vereda El Cedral Falla Mecanica
11 Km 4 Vereda El Control   Falla Humana
```

```
# Ó, de manera alternativa:
accidentes[10:11,c("lugar_accidente","tipo_accidente")]
```

```
          lugar_accidente tipo_accidente
10  Km 3 Vereda El Cedral Falla Mecanica
11 Km 4 Vereda El Control   Falla Humana
```

Para hacer filtrado condicionado a los valores de una columna, se puede implementar el siguiente código:

```
# Se quiere ver todos los registros relacionados con choques que generaron
# lesiones, sin ver las dos primeras columnas (fecha y dirección)
accidentes[accidentes$tipo_accidente=="Choque" & accidentes$lesiones=="Si",-c(1,2)]
```

```
   tipo_accidente     tipo_accidentado tipo_vehiculo lesiones
6          Choque            Conductor   Motocicleta       Si
8          Choque            Conductor   Motocicleta       Si
9          Choque            Conductor   Motocicleta       Si
32         Choque            Conductor   Motocicleta       Si
36         Choque Conductor y Ciclista   Motocicleta       Si
48         Choque            Conductor   Motocicleta       Si
50         Choque            Conductor        Camion       Si
```

```
# Ó, de manera alternativa, empleando la función subset():
subset(accidentes, tipo_accidente=="Choque" & lesiones=="Si")[,-c(1,2)]
```

```
   tipo_accidente     tipo_accidentado tipo_vehiculo lesiones
6          Choque            Conductor   Motocicleta       Si
8          Choque            Conductor   Motocicleta       Si
9          Choque            Conductor   Motocicleta       Si
32         Choque            Conductor   Motocicleta       Si
36         Choque Conductor y Ciclista   Motocicleta       Si
48         Choque            Conductor   Motocicleta       Si
50         Choque            Conductor        Camion       Si
```

Nótese en las dos líneas de código anteriores el uso de "&" para señalar que se trata de dos criterios que se deben cumplir. En otras palabras, el filtrado anterior le pide a R que muestre las filas donde la variable `tipo_accidente` es igual ("==") a "Choque" **y** ("&") donde la variable `lesiones` es igual a "Si". Todos los operadores lógicos pueden ser empleados para filtrar filas en un *data frame*: Mayor (">"), mayor o igual (">="), menor ("<"), menor o igual ("<="), diferente de ("!=") o igual ("=="). Cuando se quiera incorporar varios criterios opcionales, el operador "ó" (`|`) puede ser empleado. Por ejemplo:

```
subset(accidentes, (tipo_accidente=="Choque" | tipo_accidente=="Volcamiento") &
               lesiones=="No")[,-c(1,2)]
```

```
  tipo_accidente tipo_accidentado           tipo_vehiculo lesiones
2    Volcamiento        Conductor               Camioneta       No
7         Choque      Conductores Motocicleta y Automovil       No
```

3.5. Control de flujo

Las estructuras de control son empleadas para controlar el flujo en un programa. La mayoría de las estructuras de control no son muy usadas cuando se trabaja en una sesión de R de manera interactiva. Éstas son más útiles cuando se están escribiendo funciones o expresiones muy largas.

3.5.1. If

Condicional para ejecutar una serie de comandos siempre y cuando se cumpla una condición dada.

```
# Ejemplo de un "if"
a <- 1
if (a==1) {
        print("a es igual a 1")
} else if (a==2) {
        print("a es igual a 2")
} else {
        print("a es diferente de 1 o 2")
}
```

```
[1] "a es igual a 1"
```

3.5.2. for

El *for loop* toma una variable iteradora y asigna sus valores sucesivos de una secuencia o vector. Esta estructura de control es muy empleada para iterar sobre los elementos de una lista o vector.

```
# Por ejemplo, una forma de sumar un escalar a un vector es la siguiente:
a <- 2
vector.base <- c(1:10)
vector.res <- vector.base
for (i in 1:length(vector.base)) {
        vector.res[i] <- vector.base[i] + a
}
vector.res
```

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

```
# Una forma muy larga y compleja para:
vector.base + a
```

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

3.5.3. while

Los *while loops* inician evaluando una condición. Si ésta es verdadera, entonces el cuerpo del *loop* es ejecutado por el programa. Una vez ejecutado el cuerpo del *loop*, la condición se evalua de nuevo. Así, hasta que la evaluación de la condición arroja un valor falso. Es de suma importancia tener mucho cuidado al escribir este tipo de estructuras ya que un *loop* infinito puede desencadenar en el cierre de la sesión actual y posible pérdida de información no guardada y que esté en el ambiente global de trabajo de R.

```
# Una manera de imprimir los 5 primeros elementos de un vector o lista:
a <- c("uno","dos","tres","cuatro","cinco","seis","siete","ocho","nueve","diez")
count <- 1
while (count <= 5) {
        print(a[count])
        count <- count + 1
}
```

```
[1] "uno"
[1] "dos"
[1] "tres"
[1] "cuatro"
[1] "cinco"
```

```
# De nuevo, una alternativa diferente (y compleja) para:
a[1:5]
```

```
[1] "uno"    "dos"    "tres"   "cuatro" "cinco"
```

Recomendación: Si un *while* puede ser reemplazado por un *for*, es más eficiente y seguro emplear el *for*.

3.6. Funciones

En R, las funciones pueden ser creadas por el usuario empleando la función `function()`. Éstas son almacenadas como objetos de primera clase en R.

- Pueden ser pasadas como argumentos para otras funciones en R
- Pueden ser anidadas, es decir, el usuario puede definir una función al interior de otra función.
- El valor a retornar es la última expresión a ser evaluada en el cuerpo de una función

Las funciones contienen argumentos (con posibles valores *default*).

La evaluación de los argumentos que son pasados a una función en R, se hace de forma *lazy* (simple). Por ejemplo, la siguiente función acepta dos argumentos, "a" y "b". Si bien nunca emplea a "b", el cuerpo de la función toma el valor de "a" y lo eleva al cuadrado. Por lo tanto, existen varias formas de llamar dicha función:

```
prueba <- function(a,b) {
        a^2
}
```

```
# Se puede asignar el valor de 2 directamente al argumento "a"
prueba(a=2)
```

```
[1] 4
```

```
# También se puede llamar de forma implícita
prueba(2)
```

```
[1] 4
```

Dado que R emplea una evaluación *lazy*, el valor de "2" queda asignado posicionalmente al argumento a. ¿Qué sucederá si llamamos `prueba(2,3)`? ¿Y si llamamos `prueba(b=2)`?

Creemos ahora, a manera de ejemplo, una función sencilla que nos permita tomar un vector numérico y retornar un vector, de igual dimensión, pero con los valores normalizados, es decir, valores a los que se les resta la media y se les divide por la desviación estándar de todo el conjunto de datos (vector). Para efectos de simplificar el cuerpo de la función, lo que hacemos es incluir funciones de R para calcular tanto la media como la desviación estándar a una colección de datos.

```
normaliza <- function(vector) {
        # Primero, chequea vector
        if (class(vector) %in% c("numeric","integer")) {
                print("Normalizando vector")
                # calcula media y st dev.
                mu <- mean(vector, na.rm=T)
                std <- sd(vector, na.rm=T)
                # normaliza vector
                return((vector-mu)/std)
        } else {stop("Vector input debe ser numérico")}
}
```

En la función anterior, primero inciamos con una estructura de control. El condicional empleado verifica que efectivamente el vector pasado como argumento de la función contenga valores numéricos (o enteros). Si es el caso, el flujo continúa y se calculan las variables temporales necesarias para normalizar los componentes del vector: media y desviación estándar. Una vez calculados estos valores, se procede a normalizar, restando la media a cada componente del vector y dividiendo el resultante valor por la desviación estándar. Esta última expresión se antecede por la función primitiva `return`, que en este caso le "dice" a R que esa expresión (o más bien su resultado) debe ser el valor retornado por la función. Volviendo a la condición inicial (vector numérico), si ésta no se cumple, el flujo pasa al "else" *statement* y por lo tanto la función retorna en este caso el mensaje de error estipulado dentro de la función `stop()`. En este caso no se hace uso de `return`. Como se observa, el uso de esa función no es necesario ya que R automaticamente retorna el último valor o expresión en el cuerpo de la función, o de cada uno de sus flujos, si se está empleando una estructura de control como la registrada en este caso.

Evaluemos entonces el comportamiento de la función:

```
normaliza(runif(5,1,10))
```

```
[1] "Normalizando vector"
```

```
[1] -0.02822149 -0.41131217  1.72828310 -0.72382848 -0.56492097
```

```
normaliza(c(1,2,3,4,5))
```

```
[1] "Normalizando vector"
```

```
[1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
```

```
normaliza(c("1","2","3"))
```

```
Error in normaliza(c("1", "2", "3")): Vector input debe ser numérico
```

Los datos normalizados tienen una media de 0 y una desviación estándar de 1.

```
vector <- rnorm(10000,50,10) # media 50 y St. Dev. 10
mean(normaliza(vector)); sd(normaliza(vector))
```

```
[1] "Normalizando vector"
```

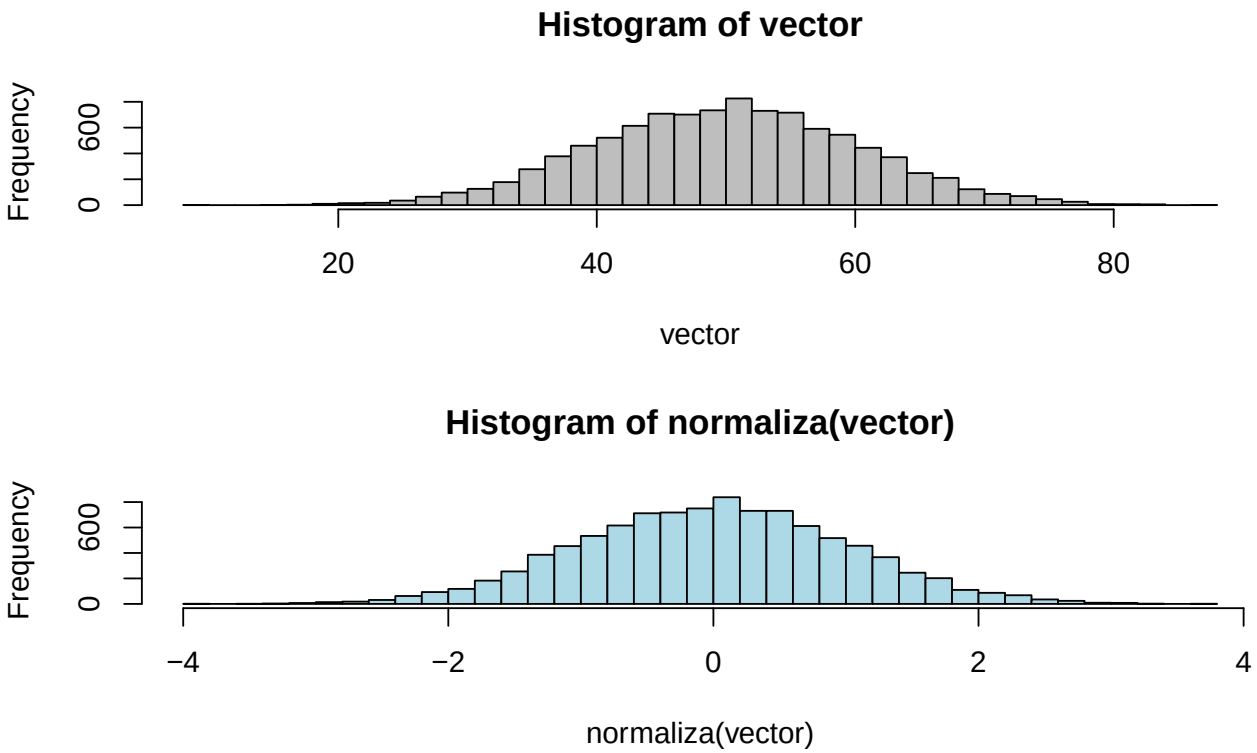
```
[1] -2.920355e-16
```

```
[1] "Normalizando vector"
```

```
[1] 1
```

Comparemos ahora la distribución del vector original (que se distribuye normalmente con media 50 y desviación estándar 10) y la distribución del vector normalizado, es decir, ajustado para tener media 0 y desviación estándar 1.

```
[1] "Normalizando vector"
```

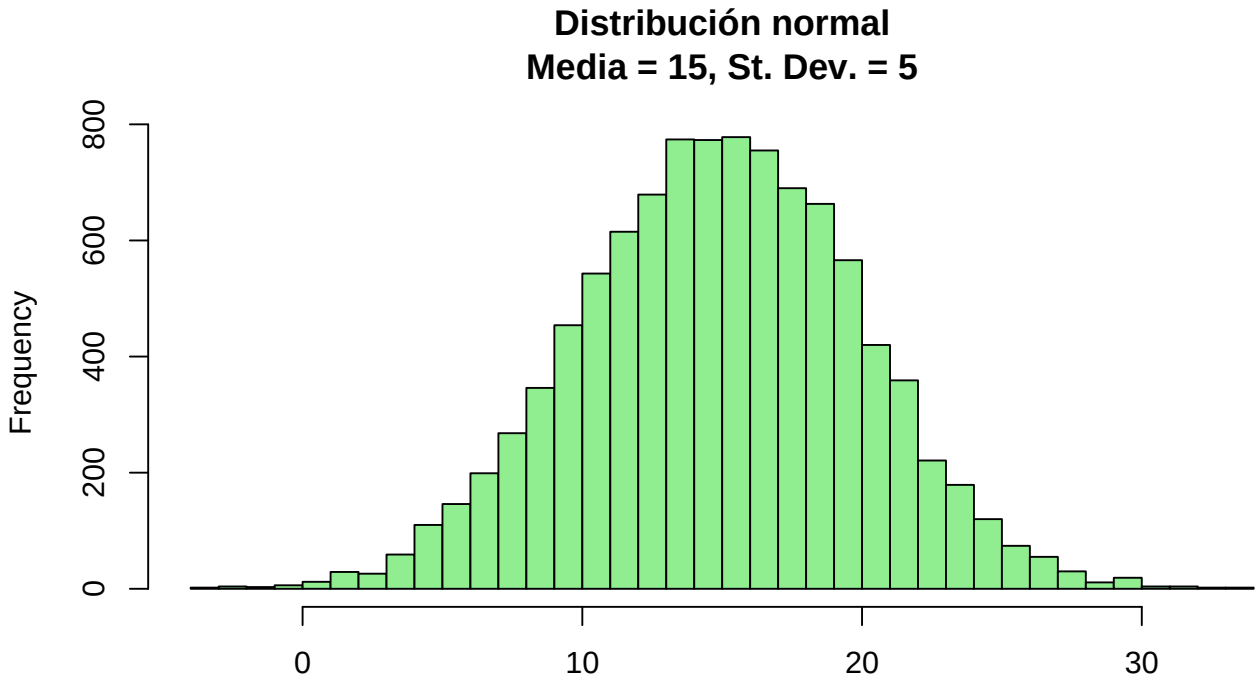


Para finalizar este tutorial, a continuación creamos una función algo más compleja que incorpora algunos de los elementos vistos en este tutorial: i) el uso de funciones al interior de funciones; ii) el uso de argumentos con valores predefinidos (o default); iii) el uso del argumento "...", que permite pasar argumentos adicionales no preestablecidos en la función. Veamos:

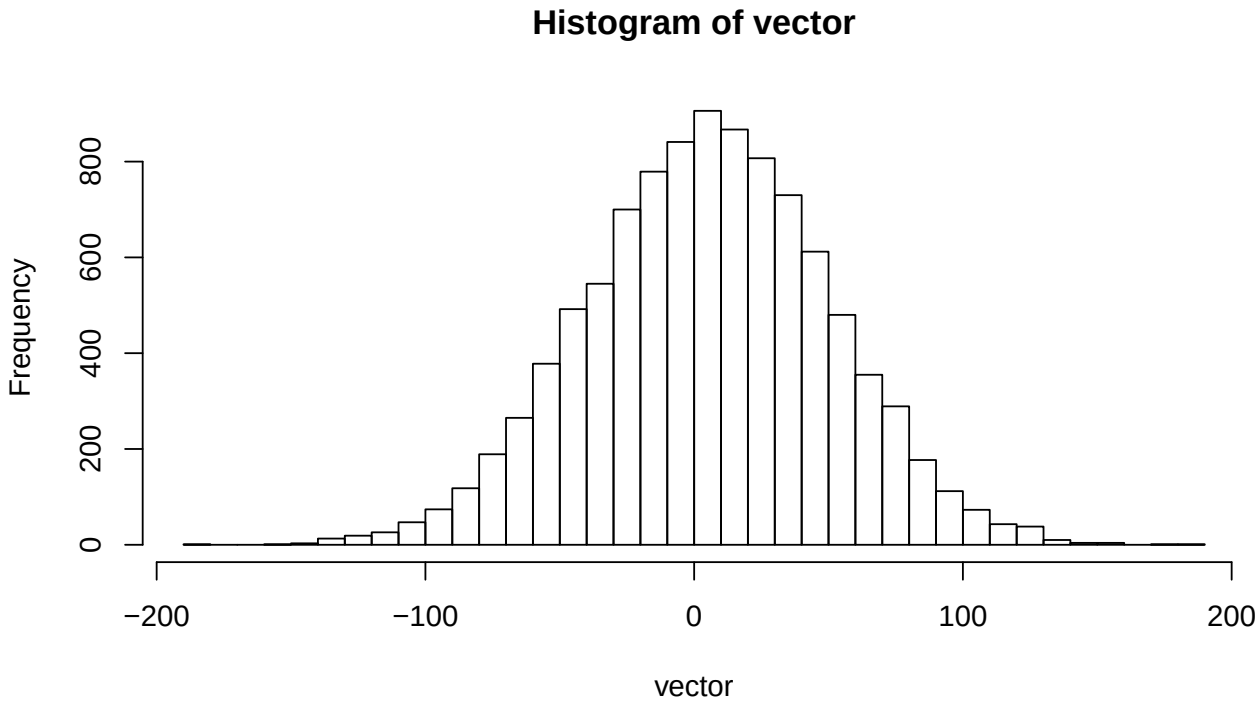
```
# Se construye una función que genera un vector de valores aleatorios
# a partir de una distribución normal y que se grafican usando el
# sistema de gráficos base de R.

grafica.dist <- function(n,media=0,std=1,...) {
        vector <- rnorm(n,media,std)
        hist(vector,breaks=50,...)
}
```

En la función anterior, los argumentos `media` y `std` cuentan con valores predefinidos de 0 y 1, respectivamente, por lo que el usuario puede optar por no llamar tales argumentos explícitamente al llamar la función, siempre y cuando desee generar "n" valores aleatorios, distribuidos normalmente, con media 0 y desviación estándar 1. Por otro lado, la expresión "..." permite pasar argumentos adicionales a la función de gráficos `hist`, de la siguiente manera:

```r
# Usando argumentos opcionales y adicionales para hist
grafica.dist(10000,media=15,std=5,
             main="Distribución normal\nMedia = 15, St. Dev. = 5",
             xlab="",col="lightgreen")
```




```r
# Sin argumentos adicionales para hist
grafica.dist(10000,media=5,std=45)
```



4. Taller

1. Muestreo. Genere un vector con 10000 valores aleatorios distribuidos uniformemente entre 5 y 10. Posteriormente, obtenga una muestra aleatoria de 100 elementos del vector generado inicialmente. Compare la media y la desviación estandar de ambos vectores. Muestre el código empleado pero no los vectores (sólo estadísticas descriptivas). ¿Es la muestra una muestra representativa del vector inicial? (Ayuda: existe una función base en R que permite obtener muestras aleatorias)
2. Funciones: Escriba una función que toma como argumentos dos valores numéricos y arroja como resultado el mayor de éstos.
3. Funciones: Escriba una función que toma como argumentos dos valores numéricos y un valor de tipo caracter el cual puede ser "mayor", "mayor igual", "menor", "menor igual" o "igual" (el cual al mismo tiempo debe ser asignado como default), y que evalua la condición asignada en dicho argumento para "A" y "B" y arroja como resultado un valor booleano. Por ejemplo, si la función ha sido nombrada como "evalua.argumentos", entonces `evalua.argumentos(5,2,"mayor")` debe arrojar como resultado `TRUE` ya que 5>2 es verdadero.
4. Funciones (avanzado, para extra puntos): Escriba una función que recibe un argumento y que imprime como resultado el número primo ubicado en dicha posición. Por ejemplo, los 10 primeros números primos son: 2,3,5,7,11,13,17,19,23,29; por lo que `primos(10)` debe arrojar como resultado el número 29. ¿Cuál es el número primo ubicado en la posición 12345678?
5. Explique por qué `c(1,2,"3") + 2` arroja como resultado el error `non-numeric argument to binary operator`.