Tercer Laboratorio - Introducción al R

Bioestadística 2020

Creación de Funciones

Volviendo al primer laboratorio, nos hemos encontrado en esta situación:

```
x = 0
if(x < 0){
  print("x es negativo")
} else if(x>0){
  # Podemos pensar que iniciamos una nueva discusión
  # para los casos que rechazamos en el primer 'if'.
  print("x es positivo")
} else{
  print("x es cero")
}
```

```
## [1] "x es cero"
```

```
x = 1
if(x < 0){
  print("x es negativo")
} else if(x > 0){
  print("x es positivo")
} else{
  print("x es cero")
}
```

```
## [1] "x es positivo"
```

Obervar que realizamos un código que identifica si un número x es positivo, negativo o cero. Sin embargo, si queremos aplicar ese código a un nuevo valor de x, tenemos que copiar y pegar el algoritmo para poder reutilizarlo. Para un algoritmo más complejo, que pueda depender de muchos más parámetros (y de maneras más complejas) que solo un número x, no solo hay que copiar y pegar el código: hay que también modificar cada parámetro utilizado. No hay que insistir en que eso es muy inconveniente.

Sin embargo, podemos crear funciones, que sirvan para invocar un código de manera eficiente. Veamos como hacerlo:

```
absoluto = function(x){
  if(x < 0){
    resultado = "x es negativo"
  } else if(x > 0){
    resultado = "x es positivo"
  } else{
    resultado = "x es cero"
  }
return(resultado)
}
absoluto(0)
```

```
## [1] "x es cero"
```

```
absoluto(2)
```

```
## [1] "x es positivo"
```

La sintaxis es *nombre_funcion = function(){}*. Podemos darle un nombre a nuestra función de la misma manera que le damos nombre a una variable, simplemente hay que tener cuidado de no sobreescribir funciones o variables ya existentes en nuestro código o en *R*. En los paréntesis curvos, escribimos una lista de todos los parámetros que tomará la función. En corchetes, está el código que se ejecutará, que depende de los parámetros que serán pasados. Es importante que piensen qué objeto va a devolver la función, lo guarden en una nueva variable, y lo retornen al final usando la función *return*.

Al invocar a la función, lo hacemos mediante *nombre_funcion(x)*, donde x tiene que ser un parámetro adecuado a la función.

A continuación crearemos una función que, dados n y k, devuelva

$$\sum_{i=1}^{n} i^k$$

```
sumas_potencias = function(n, k){
  suma = 0
  for(i in 1:n){
    suma = suma + i^k
  }
  return(suma)
}
sumas_potencias(1000000, 1)
```

```
## [1] 500000500000
```

```
sumas_potencias(100, 3)
```

```
## [1] 25502500
```

Es importante destacar que los argumentos de la función deben ser ingresados en el orden correcto: el primer argumento de *sumas_potencias* es el máximo entero que sumaremos (n), y el segundo argumento es la potencia a la que elevamos cada sumando (k).

Matrices

El uso de las matrices es de suma importancia en *R*. Sin ir más lejos, casi cualquier conjunto de datos que dispongan puede pensarse como un conjunto de n observaciones, y en cada observación se toman m mediciones (variables). Por lo general, las observaciones se piensan como las filas de una matriz, y las variables como las columnas. Eso quiere decir que, al manejar datos, estaremos lidiando con objetos matriciales. Disponemos de la función *matrix()* para crear matrices.

```
?matrix
```

```
## starting httpd help server ... done
```

```
M <- matrix(0, ncol = 2, nrow = 3)
# 'M' es una matrices con 3 filas y 2 columnas,
# y todas sus entradas son cero.
M
```

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

```r
M[1,2]
```

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

```r
# Accede al valor en la fila 1 y columna 2 de 'M'
M[1,]
```

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

```r
# Accede a la primera fila de 'M'.
# Observar que no se hacen restricciones sobre las columnas,
# y se restringe a la fila 1.
M[2,] <- c(1,2)
# La fila 2 de 'M' ahora es 1, 2.
M
```

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

```r
M[,2] <- c(10, 11, 12)
# la columna 2 de 'M' ahora es 10, 11, 12.
# Observar que se sobreescribe la entrada 'M[2,2]'
M
```

```
##      [,1] [,2]
## [1,]    0   10
## [2,]    1   11
## [3,]    0   12
```

```r
M[c(TRUE, FALSE, TRUE), ]
```

```
##      [,1] [,2]
## [1,]    0   10
## [2,]    0   12
```

```r
# Accede a las filas 1 y 3
```

Si queremos inicializar la matriz con valores que ya tenemos, lo hacemos de la siguiente manera

```r
v <- c(2, 4, 6, 8)
N <- matrix(v, nrow = 2, byrow = TRUE)
N
```

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

Como v tiene 4 elementos, y especificamos la cantidad de filas, no hace falta especificar la cantidad de columnas. La parte $byrow = TRUE$ indica que la matriz se completa por filas (se llena la primera fila con los elementos de v en orden, y luego de completarla se pasa a la segunda fila). Por defecto, el parámetro *byrow* es *FALSE*.

Procedimientos gráficos

El lenguaje R dispone de varias funciones preparadas para la representación gráfica de datos y estas serán muy importantes a lo largo del curso:

Estas funciones se dividen en dos grandes grupos:

1. Gráficos de alto nivel: crean un nuevo gráfico en la ventana de gráficos.
2. Gráficos de bajo nivel: permiten añadir líneas, puntos, etiquetas, etc. a un gráfico ya existente.

Gráficos de alto nivel

De entre todos los gráficos de este tipo se destaca la función *plot()*, que tiene muchas variantes y dependiendo del tipo de datos que se le pasen como argumento actuará de modos distintos. Lo más común es *plot(x,y)* para representar un diagrama de puntos de y respecto a x, o *plot(f)* para representar el gráfico de la función f. Otras funciones de alto nivel importantes son, por ejemplo, *hist()* para dibujar histogramas o *barplot()* para dibujar gráficos de barras.

Algunos parámetros comunes a la mayoría de los gráficos de alto nivel:

- *add=TRUE* Fuerza a la función a actuar como si fuese de bajo nivel (intenta superponerse a un gráfico ya existente). Hay que tener cuidado, no sirve para todas las funciones.
- *type* Indica el tipo de gráfico a realizar, cabe destacar type="p" sirve para representar puntos (opción por defecto), type="l" para representar líneas, type="b" para representar los puntos unidos por líneas.

Gráficos de bajo nivel

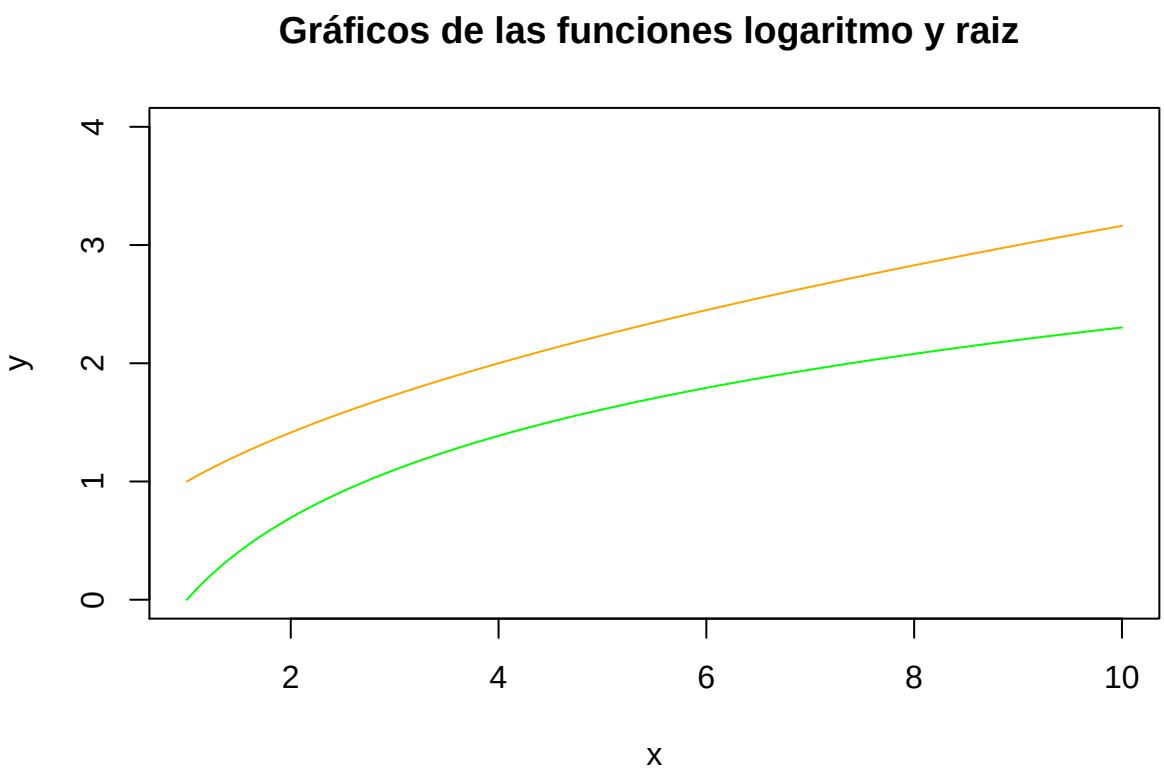
Son de gran utilidad para completar un gráfico, compararlo con otro superponiendo ambos, etc. Destacan los siguientes (help(par) para una descripción completa de estos y otros parámetros):

- lines: Permite superponer nuevas funciones en una gráfica ya existente.
- points: Permite añadir puntos.
- legend: Para añadir una nueva leyenda.
- text: Añade texto.
- pch: Indica la forma en que se dibujarán los puntos.
- lty: Indica la forma en que se dibujaran las líneas. (Line TYpe)
- lwd: Ancho de las líneas. (Line WiDth)
- col: Color usado para el gráfico (ya sea para puntos, líneas...)
- font: Fuente a usar en el texto.

Muchos de éstos parámetros se pueden ingresar como un único valor, o como un vector de valores. En este caso el gráfico irá alternando dicho parámetro en cada punto/línea de la gráfica.

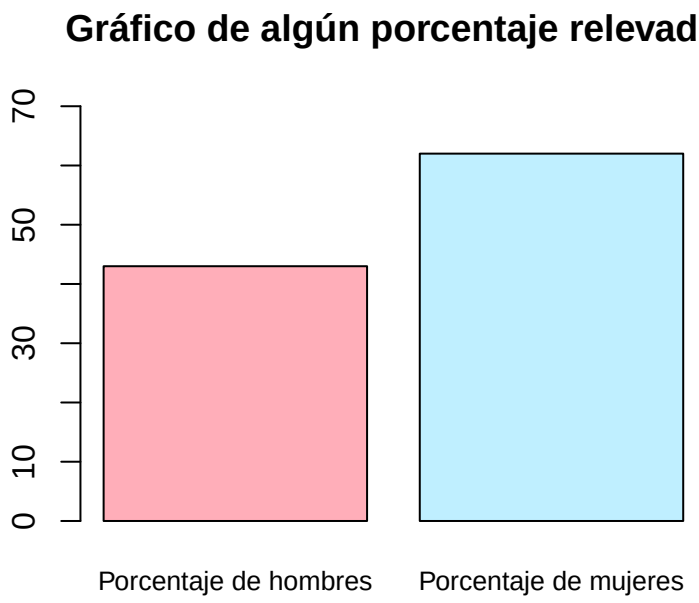
El siguiente ejemplo grafica la función logaritmo para los enteros de 1 a 10. Luego, graficamos sobre la misma, la función raiz cuadrada en la misma región usando *lines*.

```
x = seq(1,10,0.1)
y = log(x) #evalúa la función logaritmo en cada entrada del vector x
z  = sqrt(x) #idem, con la función raiz
plot(x,y, type = 'l', main = 'Gráficos de las funciones logaritmo y raiz ',
col = 'GREEN', ylim = c(0,4) )
lines(x,z, col = 'ORANGE')
```



La función *barplot* recibe un vector y grafica en barras sus valores, en el orden que los recibe. Como se verá en el ejemplo, resulta adecuada para graficar información por categorías

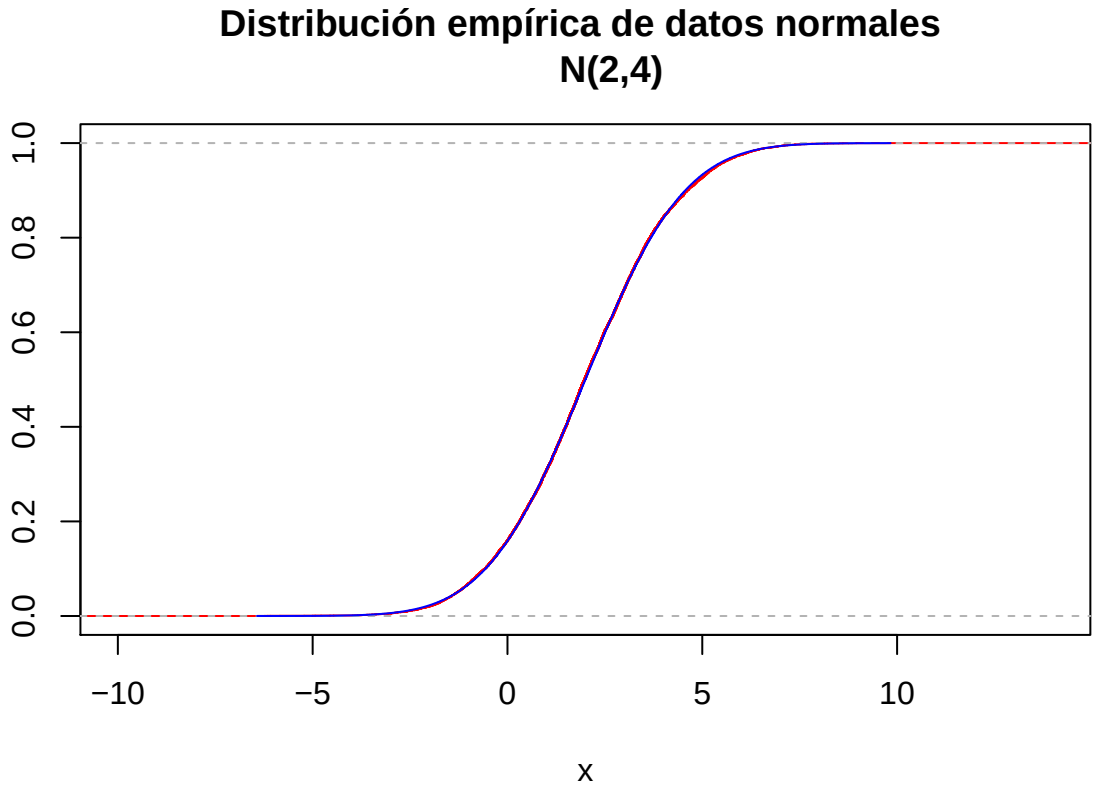
```
datos_porc = c(43, 62) #por ejemplo, podemos suponer que
#son porcentajes tomados en categorías hombre/mujer
barplot(datos_porc, main = "Gráfico de algún porcentaje relevado",
        names.arg = c('Porcentaje de hombres', 'Porcentaje de mujeres'),
        cex.names=0.8, col = c('lightpink1', 'lightblue1'), ylim=c(0,70))
```



Nota: si escribimos en la consola colours() obtenemos los nombres preestablecidos de colores en R.

Los gráficos son una buena manera de visualizar un conjunto de datos:

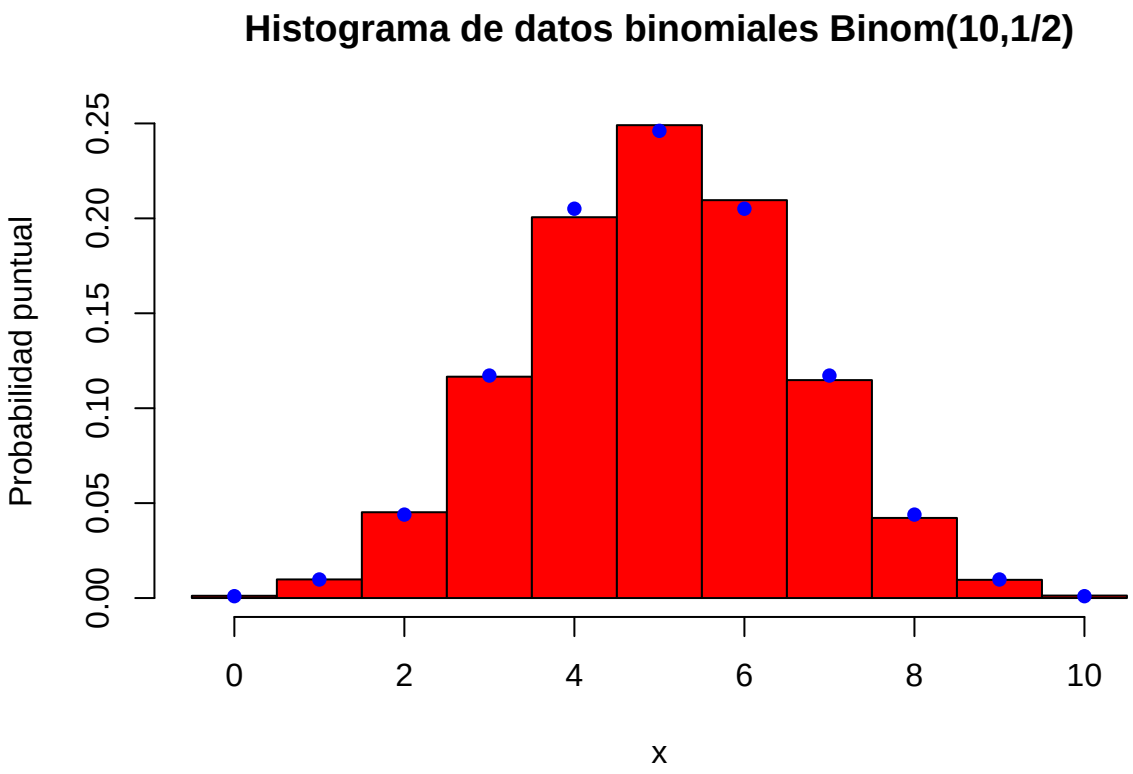
```
datos_norm = rnorm(10000, 2, 2)
dist_emp = ecdf(datos_norm)
# Función de distribución empírica
# para  los datos simulados 'datos_norm'
plot(dist_emp, xlim = c(-10, 14),
     main = "Distribución empírica de datos normales
     N(2,4)", col = "red", ylab = "")
lines(sort(datos_norm), pnorm(sort(datos_norm), mean = 2, sd = 2),
      col = "blue")
```




```
# Evaluamos la distribución teórica en los datos
# simulados (ordenados de menor a mayor), y comparamos
```

En este ejemplo graficamos en rojo la función de distribución empírica para nuestros datos y la comparamos con la distribución teórica.

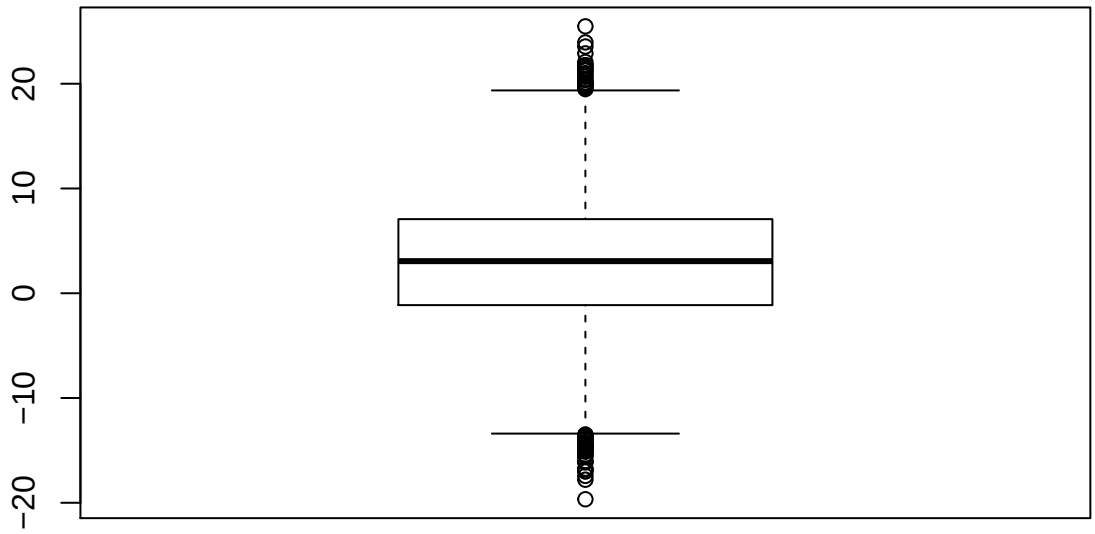
```r
datos_binom = rbinom(10000, 10, 1/2)
hist(datos_binom, freq = FALSE, breaks = 0:11 - 0.5,
     ylab = "Probabilidad puntual",
     xlab = "x",
     main = "Histograma de datos binomiales Binom(10,1/2)",
     col = "red")
points(0:10, dbinom(0:10, 10, 1/2), col = "blue", pch = 16)
```



Boxplots

Los *boxplot* son una manera muy conveniente de visualizar datos, basada en distintos cuantiles de la muestra. Si bien muestra menos información que un histograma (por ejemplo), su simplicidad lo vuelve conveniente a la hora de comparar un número grande de muestras simultaneamente. Además, la información que muestra un boxplot suele ser suficiente para sacar algunas conjeturas acerca de la muestra (simetría respecto a la mediana, desvío estándar, outliers).

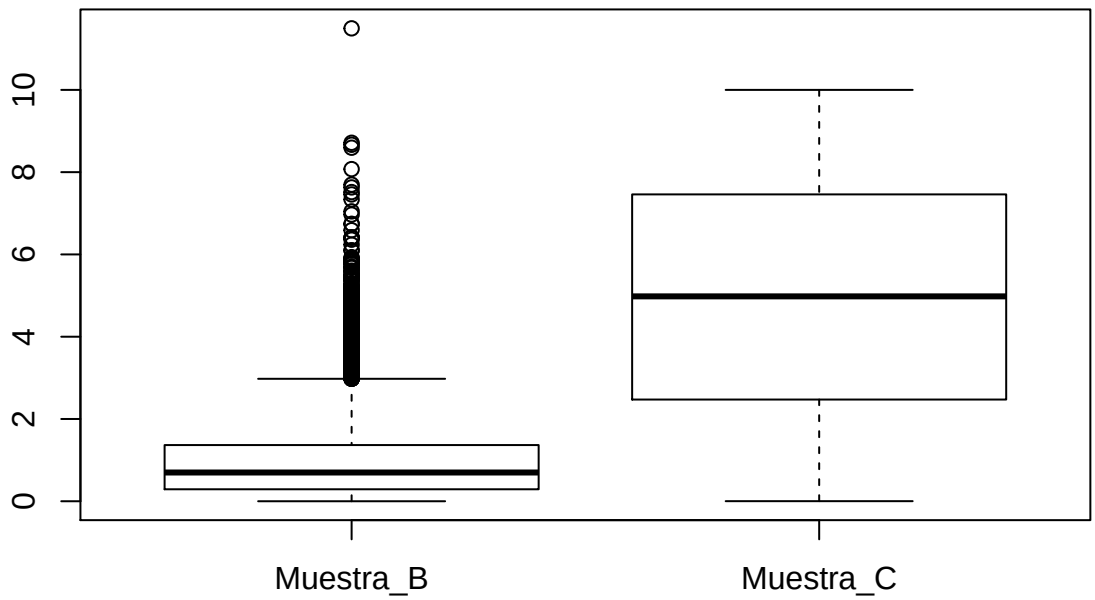
```r
?boxplot
Muestra_A = rnorm(10^4, 3, 6)
boxplot(Muestra_A)
```



Observamos que que el boxplot consiste en

- Una caja, definida por el primer y tercer cuartil de la muestra.
- La línea en el medio de la caja es la mediana
- Los bigotes, que se extienden desde la caja hasta el mínimo o máximo de la muestra (según corresponda), o hasta alcanzar el rango intercuartílico multiplicado por 1.5 (lo que suceda primero).
- Todos los datos de la muestra que caen fuera de los bigotes y la caja.

```
datos = data.frame(Muestra_B = rexp(10^4), Muestra_C = runif(10^4, 0, 10))
boxplot(datos)
```



En este ejemplo, podemos ver que la forma final del boxplot depende de la distribución de la cual los datos provienen. Eso quiere decir que el boxplot es mucho más informativo de lo que parece a simple vista, siempre y cuando sepamos interpretarlo.

Podemos ver que incluso una muestra uniforme se ve distinta a una normal (menos datos fuera de los bigotes). Por lo tanto, es importante saber qué esperar de un boxplot (en función del tamaño muestral y de la distribución que suponemos que tiene la muestra), y comparar con lo obtenido. De todas formas, el boxplot está para complementar al resto de las posibles visualizaciones de datos, no para sustituir.

Ejercicio 1 Crear una función que, dado un x real positivo, retorne *log(xˆ2)*. Si $x \leq 0$, retornar un mensaje que diga "Error, x es menor o igual a 0".

Ejercicio 2 Crear una función que, dado un vector v de datos con media (empírica) $\hat{\mu}$ y varianza (empírica) $\hat{\sigma}^2$, retorne el vector de datos estandarizados $w = (v - \hat{\mu})/\hat{\sigma}$. La única entrada de la función debe ser el vector v.

Ejercicio 3 Crear una matriz M de tamaño 100×1000, cuyas entradas sean simulaciones de variables uniformes $Unif(0, 1)$.

Ejercicio 4 Dada la matriz M creada en el ejercicio 3, crear un vector v cuyas entradas sean las sumas de los valores en cada columna de M.

Ejercicio 5 Crear una matriz N de tamaño 100×100 que cumpla $N[i, j] = i^2 - j^3$.

Ejercicio 6 Dada la matriz N del ejercicio 5, retornar una matriz N^* que cumpla $N^*[i,] = f(N[i,])$, donde f es la función del ejercicio 2.

Ejercicio 7 Graficar en un histograma los resultados obtenidos en el ejercicio 4.

Ejercicio 8 Simular 10000 datos normales $N(0, 1)$. Realizar el histograma de los datos, y comparar con la densidad teórica.