Guía 2, ICO8306

Ricardo Mayer

2019-09-29

Contents

1 Segunda guía de práctica

Continuamos avanzando en la lista de cosas que deben saber. En esta guía veremos listas, indexación, manipulación de data frames y traer datos desde excel y un par de cosas más. Es el final de la lista de la guía 1 con unos cuantos agregados pensados especialemnte para este curso

13. Sabes como crear una lista?
14. Sabes como seleccionar partes de un vector? de un data.frame? de una lista?
15. Sabes como manipular un data.frame o un tibble?
16. Puedes crear un diagrama de dispersión, un diagrama de caja usando ggplot2?
17. Sabes cómo importar una hoja de Excel en R?
18. Has escrito un `for` loop?
19. Has escrito una función?

1.1 Crear listas

Las listas son un tipo de objecto muy versátil, porque dentro de una misma lista puedes incluir números, texto, data frames, tibbles, otras listas etc. Hay muchas funciones que retornan una gran lista como resultado y a menudo uno debe escoger qué elemento de la lista es el que nos interesa.

Veamos cómo crear manualmente una lista:

```r
# de la guía 1, pero  note el stringsAsFactors = FALSE
notas_curso_df <- read.csv2("notas_curso.csv", stringsAsFactors = FALSE)

# también de la guía 1
library(readxl) # instaló el package "readxl", cierto?
notas_curso_tbl <- read_excel("notas_curso.xlsx")

un_vector <- c(2,1,6,-8,9,-2)

otro_vector <- rnorm(50, mean = 0.5, sd = 2)
# Dado que rnorm genera números (pseudo)aleatorios, *no* van a ser los mismos
# que usted obtendrá (a menos que hagamos algo extra, pero
# eso lo discutiremos más adelante)

# a,b, un_df y un_tbl son los nombres que le daremos a los elementos
# de esta lista. Podríamos no usar nombres.
lista_ejemplo <- list(a = "Hola", b = 4,
                      un_v = un_vector, otro_v = otro_vector,
                      un_df = notas_curso_df,
                      un_tibble = notas_curso_tbl)

lista_ejemplo
```

```
## $a
## [1] "Hola"
##
## $b
## [1] 4
##
## $un_v
## [1]  2  1  6 -8  9 -2
##
## $otro_v
##  [1] -0.4394931  0.9771665  0.9915794  1.9392850  1.6618032 -1.6949656
##  [7] -2.4099963  3.1272810 -0.7545579  1.8231221  3.1247990 -2.7576515
## [13] -1.5064460  0.5975489 -0.8213598 -2.1179722  0.6695505  0.2543097
## [19] -2.4668815  1.2163213 -0.4029635  2.0344057 -0.4167141  2.4987348
## [25]  0.6143952 -0.0782180  2.0399871  1.9379280  2.7381044  2.1339604
## [31] -0.7988975  0.6179287  2.0566722 -1.3510451  2.0411126  0.4332649
## [37] -3.3263900 -2.1333297  2.5457382  1.1746684 -0.8087495  5.2098652
## [43] -1.0672129 -0.6750265  3.8421545 -1.0344389 -2.2817716 -1.9817625
## [49]  1.0286236  1.2086763
##
## $un_df
##   nombre solemne controles examen final
## 1  pedro     3.5       5.0    3.0   3.8
## 2   juan     4.0       3.0    4.0   3.7
## 3  diego     5.0       6.0    4.8   5.2
## 4  frida     6.0       6.5    5.5   6.0
## 5 simone     5.5       7.0    6.0   6.2
## 6 judith     7.0       6.0    6.0   6.3
##
## $un_tibble
```

```
## # A tibble: 6 x 5
##   nombre solemne controles examen final
##   <chr>    <dbl>     <dbl>  <dbl> <dbl>
## 1 pedro      3.5       5      3     3.8
## 2 juan       4         3      4     3.7
## 3 diego      5         6      4.8   5.2
## 4 frida      6         6.5    5.5   6
## 5 simone     5.5       7      6     6.2
## 6 judith     7         6      6     6.3
```

```
class(lista_ejemplo)
```

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

1.2 Seleccionar partes de una lista, vector o data frame

La manera más general de referirse a una parte de un objeto es usar `[]` o `[[]]`. Al comienzo, es completamente esperable equivacarse en cual de los dos usar, aunque hayamos leído las instrucciones recientemente ... pero para eso es la consola!! Para experimentar y *encontrar* la forma correcta de hacer las cosas, probando varias posibilidades y viendo los resultados.

Veamos algunos ejemplos (como siempre, copie estos ejemplos en su sesión para probarlos y explorarlos por su cuenta)

```
un_vector
```

```
## [1]  2  1  6 -8  9 -2
```

```
un_vector[1:3]
```

```
## [1] 2 1 6
```

```
un_vector[c(1, 4, 6)]
```

```
## [1]  2 -8 -2
```

```
# identico resultado:
un_vector[c(TRUE, FALSE, FALSE, TRUE, FALSE, TRUE)]
```

```
## [1]  2 -8 -2
```

```
# qué ocurre aquí?:
un_vector[un_vector < 0]
```

```
## [1] -8 -2
```

```
# df, usando números
# recuerde: [filas, columnas]
notas_curso_df[1,1]
```

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

```r
notas_curso_df[1, ]
```

```
##   nombre solemne controles examen final
## 1  pedro     3.5         5      3   3.8
```

```r
notas_curso_df[, 2]
```

```
## [1] 3.5 4.0 5.0 6.0 5.5 7.0
```

```r
notas_curso_df[c(1,3), 1:3]
```

```
##   nombre solemne controles
## 1  pedro     3.5         5
## 3  diego     5.0         6
```

```r
# para seleccionar columnas usualmente podemos usar
# sus nombres --sin comillas-- y el signo peso
notas_curso_df$nombre
```

```
## [1] "pedro"  "juan"   "diego"  "frida"  "simone" "judith"
```

```r
notas_curso_df$solemne
```

```
## [1] 3.5 4.0 5.0 6.0 5.5 7.0
```

```r
# df, usando nombres o valores
# recuerde usar comillas para los nombres dentro de los
notas_curso_df[1, "nombre"]
```

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

```r
notas_curso_df[notas_curso_df$nombre == "pedro" , ] #que ocurrió aquí?
```

```
##   nombre solemne controles examen final
## 1  pedro     3.5         5      3   3.8
```

```r
notas_curso_df[, "solemne"]
```

```
## [1] 3.5 4.0 5.0 6.0 5.5 7.0
```

```r
notas_curso_df[c(1,3), c("nombre", "solemne", "controles")]
```

```
##   nombre solemne controles
## 1  pedro     3.5         5
## 3  diego     5.0         6
```

```r
# listas
# en el caso de las listas es mucho más usual seleccionar *un* elemento
# de la lista, que varios a la vez
lista_ejemplo[[1]]
```

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

```r
lista_ejemplo[[2]]
```

```
## [1] 4
```

```r
lista_ejemplo[[3]]
```

```
## [1]  2  1  6 -8  9 -2
```

```r
lista_ejemplo[["a"]]
```

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

```r
lista_ejemplo[["b"]]
```

```
## [1] 4
```

```r
lista_ejemplo[["un_v"]]
```

```
## [1]  2  1  6 -8  9 -2
```

```r
# el signo peso también funciona, pero para programar es más fácil usar [[]]
lista_ejemplo$a
```

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

```r
lista_ejemplo$otro_v
```

```
##  [1] -0.4394931  0.9771665  0.9915794  1.9392850  1.6618032 -1.6949656
##  [7] -2.4099963  3.1272810 -0.7545579  1.8231221  3.1247990 -2.7576515
## [13] -1.5064460  0.5975489 -0.8213598 -2.1179722  0.6695505  0.2543097
## [19] -2.4668815  1.2163213 -0.4029635  2.0344057 -0.4167141  2.4987348
## [25]  0.6143952 -0.0782180  2.0399871  1.9379280  2.7381044  2.1339604
## [31] -0.7988975  0.6179287  2.0566722 -1.3510451  2.0411126  0.4332649
## [37] -3.3263900 -2.1333297  2.5457382  1.1746684 -0.8087495  5.2098652
## [43] -1.0672129 -0.6750265  3.8421545 -1.0344389 -2.2817716 -1.9817625
## [49]  1.0286236  1.2086763
```

1.3 Sabes como manipular un data.frame o un tibble?

Este es un tópico muy largo, pero nos vamos a concentrar en tres comandos que nos entrega el paquete dplyr: filter, select y summarize. Los comandos funcionan de la misma manera un data frame tradiconal o un tibble (data frame más moderno)

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
select(notas_curso_df, c("nombre", "examen"))
```

```
##   nombre examen
## 1  pedro    3.0
## 2   juan    4.0
## 3  diego    4.8
## 4  frida    5.5
## 5 simone    6.0
## 6 judith    6.0
```

```r
filter(notas_curso_df, examen > 5)
```

```
##   nombre solemne controles examen final
## 1  frida     6.0       6.5    5.5   6.0
## 2 simone     5.5       7.0    6.0   6.2
## 3 judith     7.0       6.0    6.0   6.3
```

```r
select(filter(notas_curso_df, examen > 5), c("nombre", "examen"))
```

```
##   nombre examen
## 1  frida    5.5
## 2 simone    6.0
## 3 judith    6.0
```

```r
# las pipas %>% hacen más legibles algunos programas
# todo se lee de izquierda a derecha
notas_curso_df %>% select(c("nombre", "examen"))
```

```
##   nombre examen
## 1  pedro    3.0
## 2   juan    4.0
## 3  diego    4.8
## 4  frida    5.5
## 5 simone    6.0
## 6 judith    6.0
```

```
notas_curso_df %>%  filter(examen > 5)
```

```
##   nombre solemne controles examen final
## 1  frida     6.0       6.5    5.5   6.0
## 2 simone     5.5       7.0    6.0   6.2
## 3 judith     7.0       6.0    6.0   6.3
```

```
notas_curso_df %>%
  filter(examen > 5) %>%
  select(c("nombre", "examen"))
```

```
##   nombre examen
## 1  frida    5.5
## 2 simone    6.0
## 3 judith    6.0
```

```
notas_curso_df %>%
  summarize(n_estudiantes = n(),
            media_final = mean(final),
            desv_est_final = sd(final))
```

```
##   n_estudiantes media_final desv_est_final
## 1             6         5.2       1.188276
```

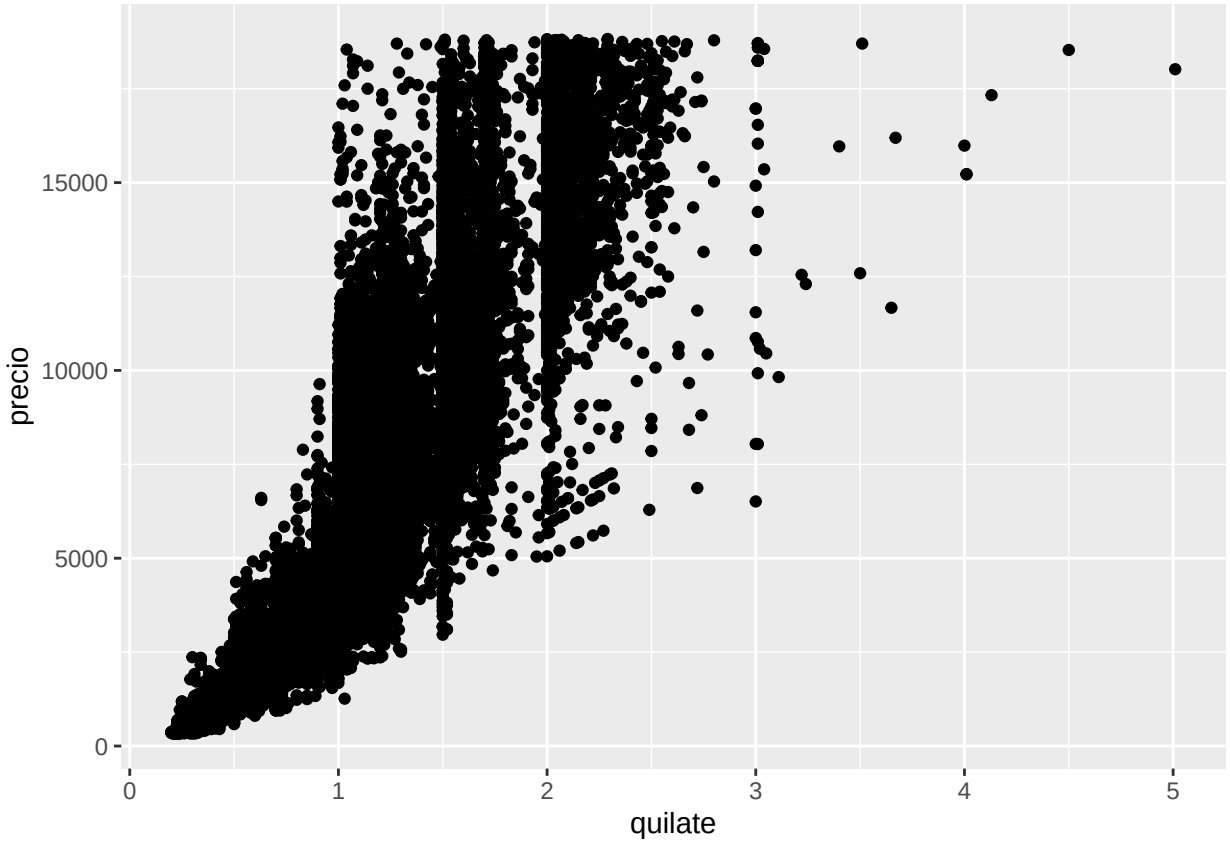
1.4 Puedes crear un diagrama de dispersión, un diagrama de caja usando ggplot2?

Es muy fácil, sólo necesitas que la información esté un data frame o en un tibble y usar las funciones geom_point y geom_box

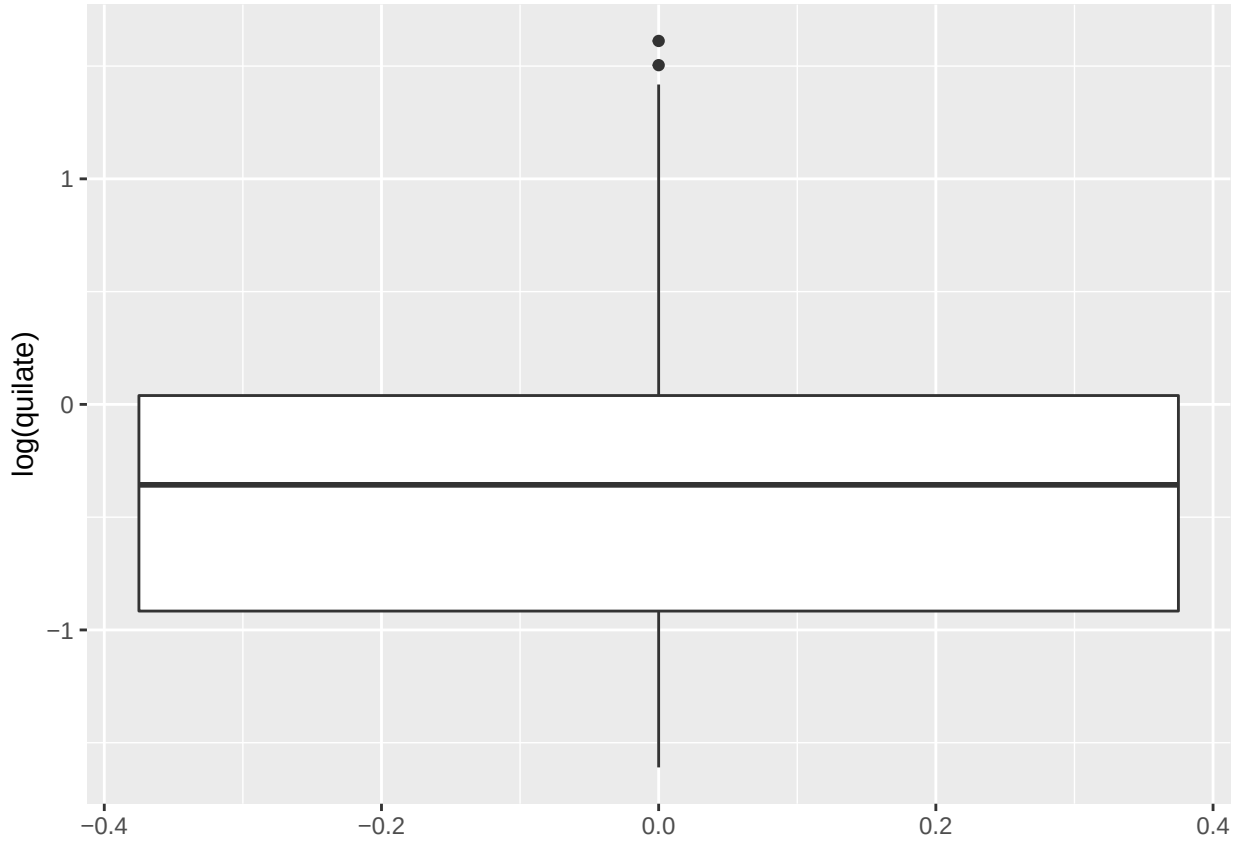
```
library(ggplot2)
library(datos)

diamantes
```

```
## # A tibble: 53,940 x 10
##    precio quilate corte  color claridad profundidad tabla     x     y     z
##     <int>   <dbl> <ord>  <ord> <ord>          <dbl> <dbl> <dbl> <dbl> <dbl>
##  1    326   0.23  Ideal  E     SI2             61.5    55  3.95  3.98  2.43
##  2    326   0.21  Premi~ E     SI1             59.8    61  3.89  3.84  2.31
##  3    327   0.23  Bueno  E     VS1             56.9    65  4.05  4.07  2.31
##  4    334   0.290 Premi~ I     VS2             62.4    58  4.2   4.23  2.63
##  5    335   0.31  Bueno  J     SI2             63.3    58  4.34  4.35  2.75
##  6    336   0.24  Muy b~ J     VVS2            62.8    57  3.94  3.96  2.48
##  7    336   0.24  Muy b~ I     VVS1            62.3    57  3.95  3.98  2.47
##  8    337   0.26  Muy b~ H     SI1             61.9    55  4.07  4.11  2.53
##  9    337   0.22  Regul~ E     VS2             65.1    61  3.87  3.78  2.49
## 10    338   0.23  Muy b~ H     VS1             59.4    61  4     4.05  2.39
## # ... with 53,930 more rows
```

```
ggplot(data = diamantes) +
  geom_point(aes(x = quilate, y = precio))
```




```
ggplot(data = diamantes) +
  geom_boxplot(aes(y = log(quilate)))
```



A modo de gentil recordatorio, como vimos en la guía 1, un gráfico básico de boxplot es aún más rápido de escribir: `boxplot(log(diamantes$quilate))`

1.5 Sabe cómo importar una hoja de Excel en R?

Sí, si sabe: :-), ya lo ha hecho. Para más detalles, consulte el help de read_excel, del paquete readxl

```
notas_curso_tbl <- read_excel("notas_curso.xlsx")
```

1.6 Ha escrito un `for` loop?

Bueno esto es un clásico de los lenguajes de programación. A veces se abusa de ellos, cuando existen alternativas que son más fáciles de leer. Pero otras los for loop presentan las cosas muy claramente y son muy muy flexibles.

Vamos a mostrar primero de forma muy sencilla como va cambiando el índice dento de del for loop y luego un ejemplo más interesante de inferencia estadística

Note que `{` y `}` marcan los límite del código (lo que calculamos) en un for loop.

```
# recuerde que 1:5 es lo mismo que c(1, 2, 3, 4, 5)
for (i in 1:5) {
  i_cuadrado <- i * i
  print(i_cuadrado)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

1.6.1 Ejercicio

Repita el loop anterior, pero ahora debe calcular los cuadrados desde 1 a 8.

1.6.2 Media muestral I

Para el siguiente ejemplo, calcularemos la media muestral de 4 muestras distintas, de una población cuyos valores siguen una distribución exponencial. Seguiremos los siguientes pasos

1. Generamos una muestra de 50 valores exponenciales
2. Calculamos el promedio de los 50 valores. Esa es la primera media muestral
3. Usamos print para que nos muestre el valor de esa media
4. Generamos la siguiente muestra de 50 valores exponenciales y seguimos los mismos pasos anteriores.
5. Hacemos lo mismo hasta completar cuatro medias muestrales.

Una nota: generar estos valores usando una función de números (pseudo) aleatorios es equivale a realizar un muestreo ("samplear") desde una población infinita.

```
# es una buena práctica dejar claro en otra línea, cuantas repeticiones pensamos hacer
muestras <- 4
n <- 50
for (i in 1:muestras) {
  esta_muestra <- rexp(n, rate = 0.5)
  media_muestral <- mean(esta_muestra)
  print(media_muestral)
}
```

```
## [1] 1.545009
## [1] 2.586509
## [1] 2.175179
## [1] 2.299209
```

1.6.3 Ejercicio

Repita el loop anterior pero cambie la cantidad de muestras de 4 a 6 cambie el tamaño de cada muestra de 50 a 100.

1.6.4 Media muestral II

Supongamos que tenemos un población que sigue siendo exponencial, pero esta vez en finita. En particular, que la población está compuesta de 10000 datos. Pero por razones de costo y tiempo sólo podemos tomar muestras de 50 observaciones. Cómo podemos hacer un muestreo ("sampleo") aletorio simple desde esa población? Fácil: primero guardamos la población en un vector y luego ocupamos la función `sample` para tomar una muestra aleatoria de 50 de esos 1000 componentes del vector.

Ocuparemos nuevamente un for loop para tomar 4 muestras y 4 medias muestrales

```r
muestras <- 4
n <- 50

# esta vez ocuparemos set.seed para que el vector población no cambie
# si corremos el código varias vaces
set.seed(12345)
poblacion <- rexp(10000, rate = 0.5)

for (i in 1:muestras) {
  esta_muestra <- sample(poblacion, size = n)
  media_muestral <- mean(esta_muestra)
  print(media_muestral)
}
```

```
## [1] 2.045064
## [1] 1.707899
## [1] 1.899274
## [1] 1.687065
```

1.6.5 Ejercicio

Cambie el loop anterior para que cada vez obtenga las muestras contengan 20 en vez de 50 observaciones.

1.6.6 Media muestral III

OK, la gracia de nuestro for loop es poder generar tantas muestras como queramos y el código cambiará muy poco. Vamos a generar 200 muestras y calcularemos 200 medias muestrales. Por supuesto, no vamos a imprimir las 200 medias, si no que las vamos a guardar dentro de un vector, para usarlas después.

```r
muestras <- 400
n <- 50
set.seed(12345)
poblacion <- rexp(10000, rate = 0.5)

# este vector inicialmente tiene 200 componentes sin valores de verdad aún
vector_con_medias <- vector(mode = "numeric", length = muestras)

for (i in 1:muestras) {
  esta_muestra <- sample(poblacion, size = n)
  media_muestral <- mean(esta_muestra)
  # cuando i=1 cambiará el primer componente del vector
  # cuando i = 2 cambiará el segundo componente del vector
  vector_con_medias[i] <- media_muestral
}
```

1.6.7 Ejercicio

En el loop anterior, cambie el código para tomar 30 muestras en vez de 400

1.6.8 Media muestral IV: su distribucón empírica

En la sección anterior guardamos las 400 medias en un vector por un motivo: queremos examinar su distribución. Como vimos, hay azar involucrado en cuales 50 valores aparecen en cada muestra y por lo tanto la media muestral va cambiando con cada muestra, es efecto una variable aleatoria.

```
# primeras 10 medias muestrales
vector_con_medias[1:10]
```

```
##  [1] 2.045064 1.707899 1.899274 1.687065 2.013064 1.513769 1.693857
##  [8] 2.217416 1.677632 1.822376
```

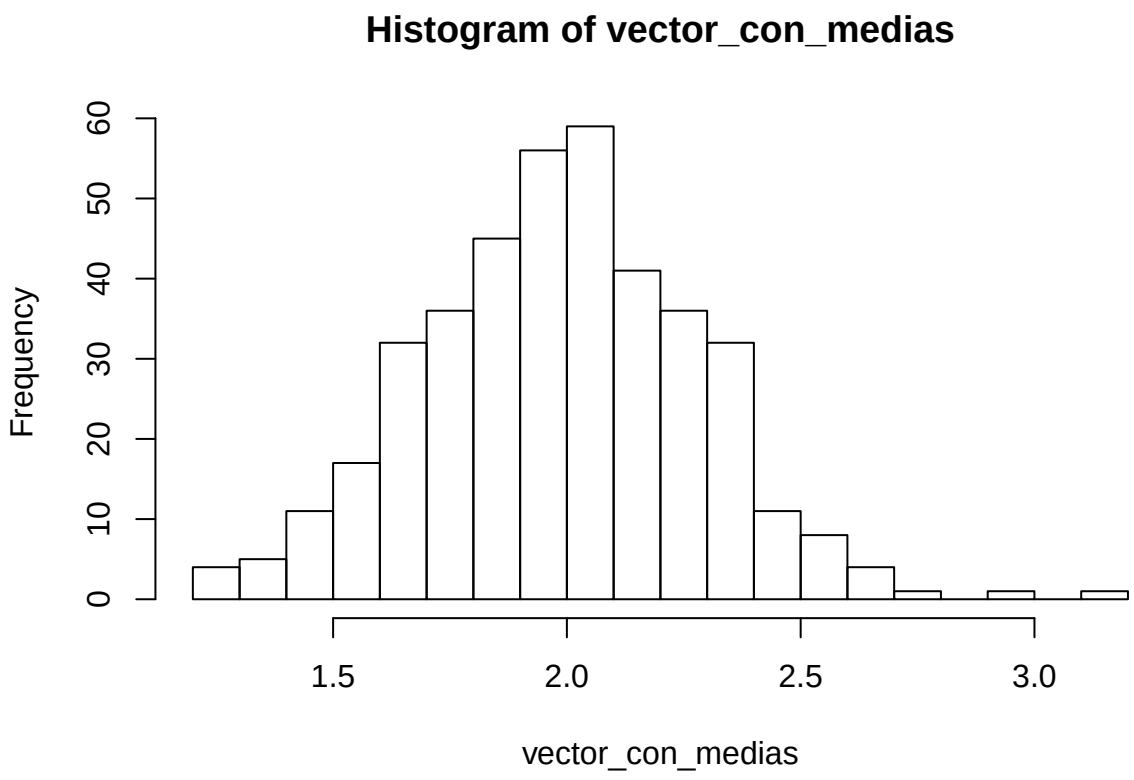
```
# promedio de las medias muestrales que obtuvimos
mean(vector_con_medias)
```

```
## [1] 1.985834
```

```
# desviación estándar de las medias muestrales que obtuvimos
sd(vector_con_medias)
```

```
## [1] 0.2947882
```

```
# histograma de las medias muestrales que obtuvimos
hist(vector_con_medias, breaks = 21)
```



1.6.9 Ejercicio

Compare la distribución anterior, con la que obtendría si el tamaño de cada muestra hubiese sido de 20 y no de 50

1.6.10 Media muestral V: error de estimación de β

En una distribución exponencia el parámetro clave es la media poblacional, que se denota comúnmente como β. Si lee la documentación de `rexp` se dará cuenta que el parámetro `rate` que usamos es el recíproco de la media. Como escribimos `rate = 0.5`, implícitamente usamos una media poblacional igual a 2. Si se fija, nuestras media muestrales son cercanas pero no idénticas a 2, lo que implica que estiman relativamente bien la media poblacional pero con algún error de estimación. Cambiemos levemente el código anterior para examinar la distribución del *error de estimación* y del error de muestreo (en valor absoluto, como lo define el libro).

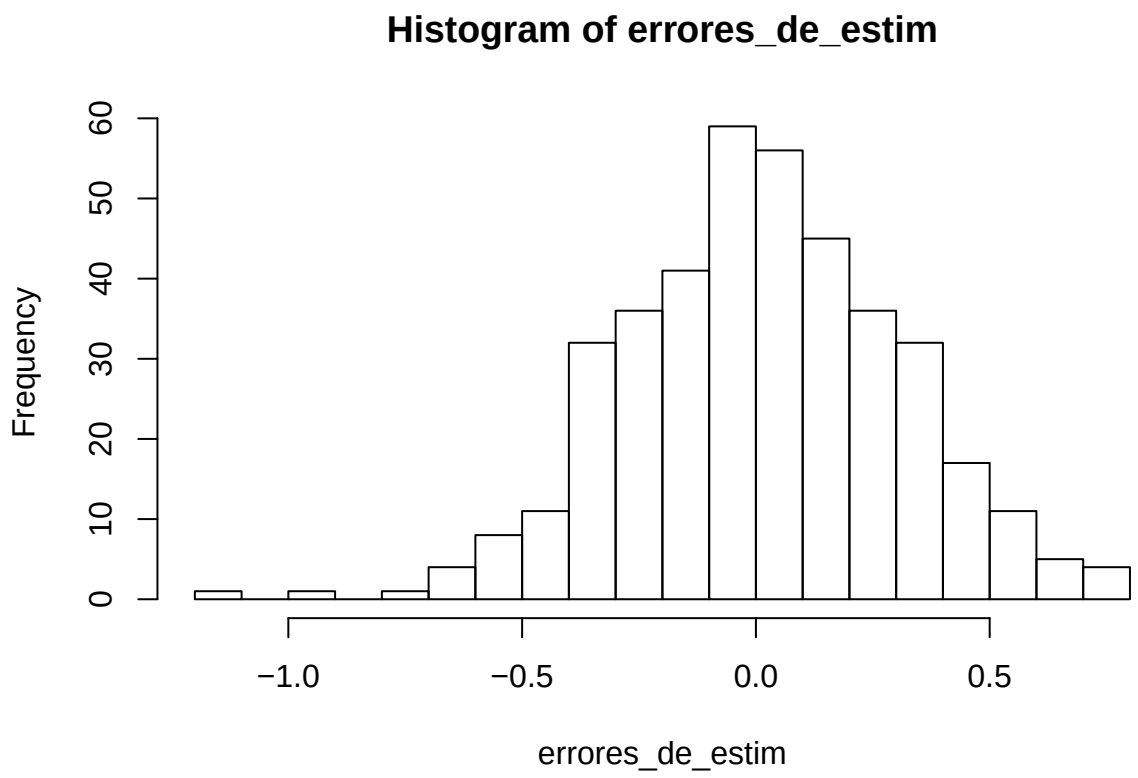
```
muestras <- 400
n <- 50
set.seed(12345)
poblacion <- rexp(10000, rate = 0.5)
media_poblacional <- 1/0.5  # es 2

# este vector inicialmente tiene 200 componentes sin valores de verdad aún
errores_de_estim <- vector(mode = "numeric", length = muestras)
errores_de_muestreo <- vector(mode = "numeric", length = muestras)

for (i in 1:muestras) {
  esta_muestra <- sample(poblacion, size = n)
  media_muestral <- mean(esta_muestra)

  error_de_estim <- media_poblacional - media_muestral
  error_de_muest <- abs(media_poblacional - media_muestral)

  errores_de_muestreo[i] <- error_de_muest
  errores_de_estim[i] <- error_de_estim
}

mean(errores_de_estim)
```

```
## [1] 0.01416574
```

```
sd(errores_de_estim)
```

```
## [1] 0.2947882
```

```
hist(errores_de_estim, breaks = 20)
```



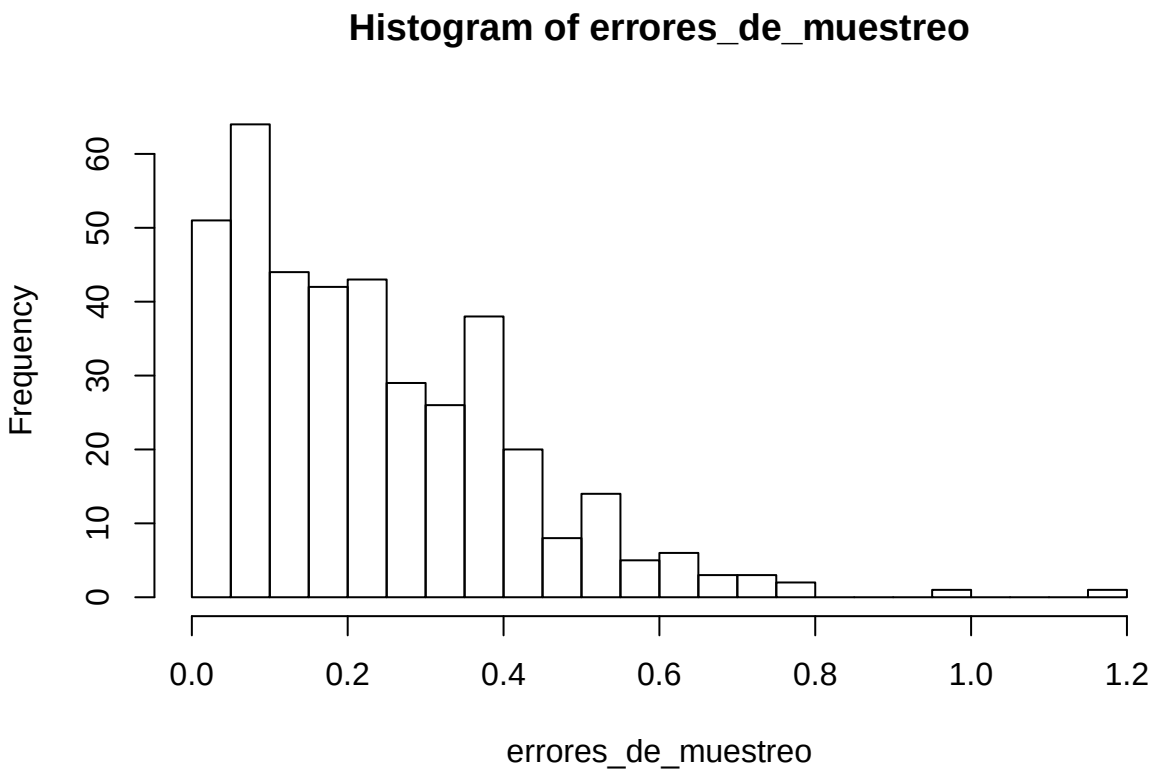

```
mean(errores_de_muestreo)
```

```
## [1] 0.2335694
```

```
sd(errores_de_muestreo)
```

```
## [1] 0.1800274
```

```
hist(errores_de_muestreo, breaks = 20)
```



1.6.11 Ejercicio

Compare la media, la desviación estándar y el histograma de los errores que obtuvo en el código anterior, con los que resultan en estos dos casos: + muestras de tamaño 20 (en vez de 50) + muestras de tamaño 200 (en vez de 50)

1.7 Ha escrito una función?

Es muy sencillo (y además rápido si usan el autocompletar en RStudio después de escribir `fun`).

Recordemos que estandarizar una variable significa restar la media y dividir por la desviación estándar. Supongamos que tenemos un vector con 10 observaciones de una distribución normal com media igual 2 y desviación estándar igual a 3. La estandarizacipon es muy sencilla:

```
mi_media <- 2
mi_sd <- 3

x <- rnorm(10, mean = mi_media, sd = mi_sd)

z <- (x - mi_media)/mi_sd # a cada elemento le sustrae 2 y luego a cada elemento lo divide por 3
```

Cómo podríamos usar esos mismos calculos en una función? Definiendo el vector x, la media y la desviación estándar como los argumentos de la función y el vector z como el producto de la función:

```r
#definimos la función (recuerden usar el snippet que ofrece despues de fun)
estandarizar <- function(x, media, desv) {
  z <- (x-media)/desv
}

# y ahora la probamos
estandarizar(x, mi_media, mi_sd)
```