Primera tarea

Carbonera Francesco
Cesaro Giada

14/10/2019

Índice

1 Tarea 1

1.1 Error de muestreo y sesgo en la estimación de la varianza muestral

Creamos el conjunto de datos *input*

```
d <- c(0, 1, 3, 5, 8, 10, 20, 30, 32)
d
```

```
[1]  0  1  3  5  8 10 20 30 32
```

```
summary(d)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    3.00    8.00   12.11   20.00   32.00
```

Generamos las 9 muestras *jackknife* y para cada una calculamos la varianza de la muestra en el siguiente modo:

$$\hat{\theta}(i) = \frac{\sum_{j=1, j\neq i}^{n}(x_j - \bar{x}_{(i)})^2}{n}$$

donde $\bar{x}_i$ es la media de la muestra $\mathbf{x}_{(i)} = (x_i, ..., x_{i-1}, x_{i+1}, ..., x_n)$

```
library(bootstrap)
jacks = sapply(1:length(d), function(i) mean((d[-i]-mean(d[-i]))^2))
jacks
```

```
[1] 129.73438 133.00000 138.68750 143.25000 147.98438 149.73438 141.60938
[8] 105.35938  94.73438
```

La variable `jacks` contiene 9 valores del estadistico de interes que en este caso es la varianza muestral.

Calculamos la verdadera varianza de la muestra inicial ($\hat{\theta}$) que hemos nombrado `varcamp`, y la media de los valores del estadistico $\hat{\theta}(i)$ ($\hat{\theta}(.)$) que hemos nombrado `k`.

```
varcamp <- mean((d-mean(d))^2)
varcamp
```

```
[1] 133.6543
```

```
k <-mean(jacks)
k
```

```
[1] 131.566
```

Ahora calculamos el estimador *jacknife* del error estándar ($\hat{\sigma}_{jack}$) de nuestro estadístico de interés (la varianza muestral, $\hat{\theta}(i)$) como sigue:

$$\hat{\sigma}_{jack} = \sqrt{\frac{n-1}{n}\sum_{i=1}^{n-1}(\hat{\theta}_{(i)} - \hat{\theta}(.))^2}$$

```
n <-length(jacks)
```

```
sigmajack <- sqrt((n-1)*mean((jacks- k)^2))
sigmajack
```

```
[1] 51.09052
```

Y ahora aplicamos la fórmula para el sesgo

$$\hat{sesgo}_{jack} = (n-1)(\hat{\theta}(.) - \hat{\theta})$$

```
sesgo <-(n-1)*(k-varcamp)
sesgo
```

```
[1] -16.70679
```

1.2 Error de muestreo y sesgo en la estimación del estadistico 'media *trimmed*'

```
jacks = sapply(1:length(d), function(i) mean(d[-i],trim=0.2))
jacks
```

```
[1] 12.666667 12.666667 12.333333 12.000000 11.500000 11.166667  9.500000
[8]  7.833333  7.833333
```

Calculamos las medias recortadas (aquella de la muestra/población inicial y las de las muestras *jacknife*):

```
meantrim <- mean(d, trim = 0.2)
meantrim
```

```
[1] 11
```

```
## jacknife
k <-mean(jacks)
k
```

```
[1] 10.83333
```

Se calcula la desviación estándar:

```
sigmajack <- sqrt((n-1)*mean((jacks-k)^2))
sigmajack
```

```
[1] 5.22104
```

y el sesgo:

```
sesgo <-(n-1)*(k-meantrim)
sesgo
```

```
[1] -1.333333
```

2 Tarea 2

2.1 ANOVA y test de permutaciones

Se consideran dos variables: en la primera se recogen 4 tipos diferentes de tratamientos y en la segunda se recogen los valores obtenidos de productividad.

Creamos el conjunto de datos *input*

```
dati <- data.frame(trattamento= c("A","A","A","A","A","A","B","B","B","B",
                                  "B","B","C","C","C","C","C","C","D","D",
                                  "D","D","D","D" ),
                   risultato=c(4.6,5.5,3.4,5.0,3.9,4.5,3.6,4.5,2.4,4.0,
                               2.9,3.5,2.6,3.5,1.4,3.0,1.9,2.5,5.6,3.5,
                               5.1,4.0,4.6,4.7))
head(dati)
```

```
  trattamento risultato
1           A       4.6
2           A       5.5
3           A       3.4
4           A       5.0
5           A       3.9
6           A       4.5
```

```
summary(dati)
```

```
 trattamento   risultato
 A:6         Min.   :1.400
 B:6         1st Qu.:2.975
 C:6         Median :3.750
 D:6         Mean   :3.758
             3rd Qu.:4.600
             Max.   :5.600
```

Realizamos un análisis exploratorio de la media y la desviación estándar de los distintos grupos de tratamiento.

```
library(tidyverse)
dati%>%
group_by(trattamento) %>%
  summarise(count=n(), mean=mean(risultato), sd=sd(risultato))
```

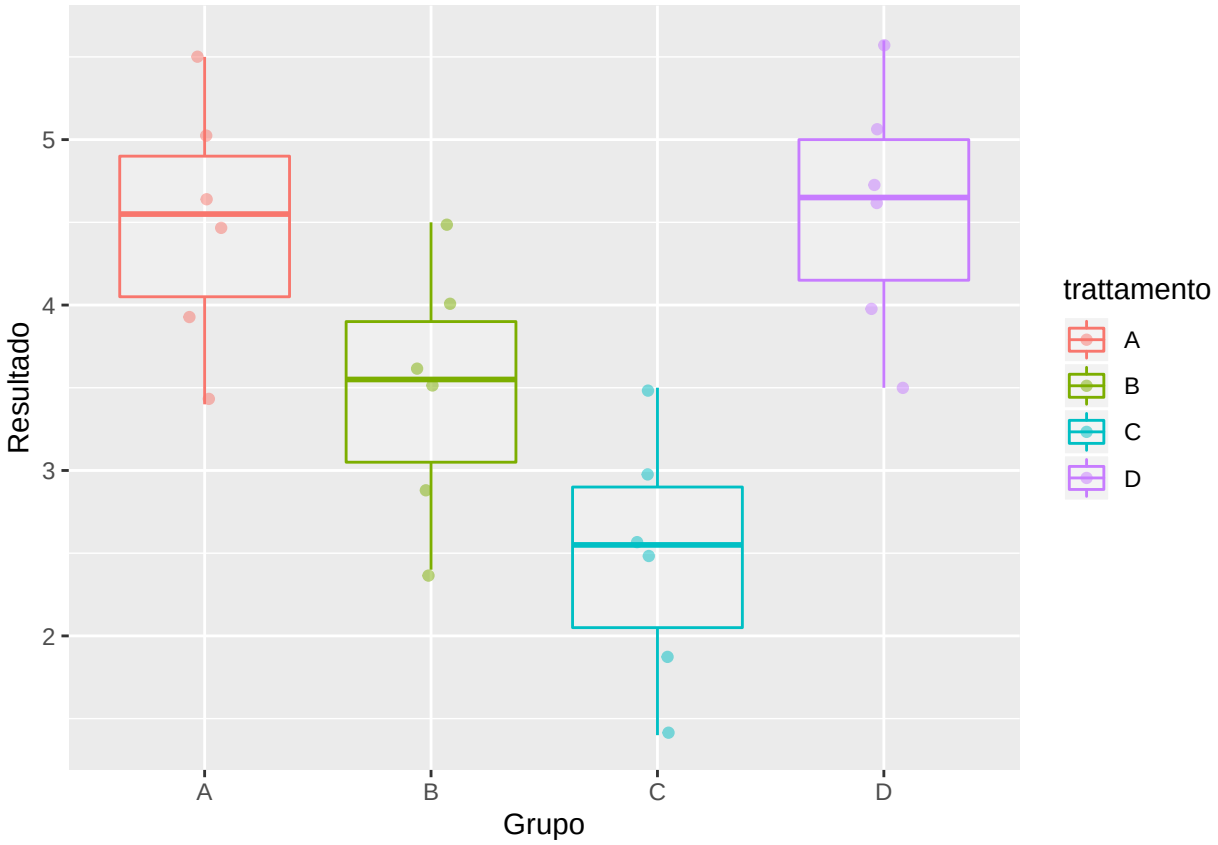
```
# A tibble: 4 x 4
  trattamento count  mean    sd
  <fct>       <int> <dbl> <dbl>
1 A               6  4.48 0.752
2 B               6  3.48 0.752
3 C               6  2.48 0.752
4 D               6  4.58 0.752
```

A priori no podemos decir si la diferencia entre los promedios es significativa o no, también teniendo en cuenta que las estimaciones de la desviación estándar son pequeñas.

Observamos a los *boxplot* [1]:

[1] Se utiliza un `jitter()` para que los puntos no se superpongan todos en la línea del *boxplot*

```
library(ggplot2)
ggplot(dati, aes(x=trattamento, y=risultato,
                 color=trattamento))+geom_boxplot(alpha=0.2)+
  geom_jitter(width=0.1, alpha=0.5)+ labs(y="Resultado", x= "Grupo")
```



Realizamos un test de análisis de varianza (`ANOVA`) con el comando `aov`.[2]

```
test.aov <- aov(risultato ~ trattamento, data = dati)
summary(test.aov)
```

```
            Df Sum Sq Mean Sq F value   Pr(>F)
trattamento  3  17.45   5.815   10.28 0.000263 ***
Residuals   20  11.31   0.566
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

El p-valor indica que hay una diferencia significativa en los promedios de los grupos.

Hacemos ahora un contraste para la misma hipótesis solo usando un test de permutación (que no necesita ninguna hipótesis de distribución):

```
library(coin)
oneway_test(risultato ~ trattamento, data = dati,distribution=approximate(nresample=1000))
```

```
    Approximative K-Sample Fisher-Pitman Permutation Test

data:  risultato by trattamento (A, B, C, D)
```

[2]La descripción que ofrece la documentación de R para el comando `aov(formula, data)` es "fit an analysis of variance model by a call to lm for each stratum".

```
chi-squared = 13.952, p-value < 0.001
```

También el test de permutación nos lleva a rechazar la hipótesis de la igualdad de los promedios entre los grupos.

2.2 Análisis *post-hoc*

Podemos pensar en hacer un análisis *a-posteriori* para ver si el hecho de que la hipótesis de igualdad entre los promedios sea rechazada se debe a que solo un grupo tiene una media significativamente diferente de los demás o no. Para ello, utilizamos el comando `pairwise.t.test` que realiza un análisis para cada pareja de grupos. Es necesario indicar qué corrección adoptar para el p-valor dado que de esta forma se realizan test múltiples. Para más detalles sobre la corrección de Bonferroni se puede hacer click aqui y consultar el pdf (en italiano).

```
pairwise.t.test(dati$risultato,dati$trattamento, p.adj= "bonferroni")
```

```
    Pairwise comparisons using t tests with pooled SD

data:  dati$risultato and dati$trattamento

  A      B      C
B 0.1929 -      -
C 0.0010 0.1929 -
D 1.0000 0.1187 0.0006

P value adjustment method: bonferroni
```

Resulta de esta última prueba que el grupo C es significativamente diferente en promedio de A y D, pero no de B. Por lo tanto, la hipótesis de igualdad de los promedios podría aceptarse si el grupo C fuera eliminado. Intentemos el análisis nuevamente:

```
test.aov <- aov(risultato ~ trattamento, data = subset(dati, dati$trattamento!= "C"))
summary(test.aov)
```

```
            Df Sum Sq Mean Sq F value Pr(>F)
trattamento  2  4.440  2.2200   3.925 0.0426 *
Residuals   15  8.485  0.5657
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

En este caso la hipótesis nula sería rechazada con nivel 5% pero no con nivel 4%. Repetimos también la prueba en parejas:

```
d2<- dati%>% filter(trattamento!= "C")
pairwise.t.test(d2$risultato,d2$trattamento, p.adj= "bonferroni")
```

```
    Pairwise comparisons using t tests with pooled SD

data:  d2$risultato and d2$trattamento

  A     B
B 0.108 -
D 1.000 0.069

P value adjustment method: bonferroni
```

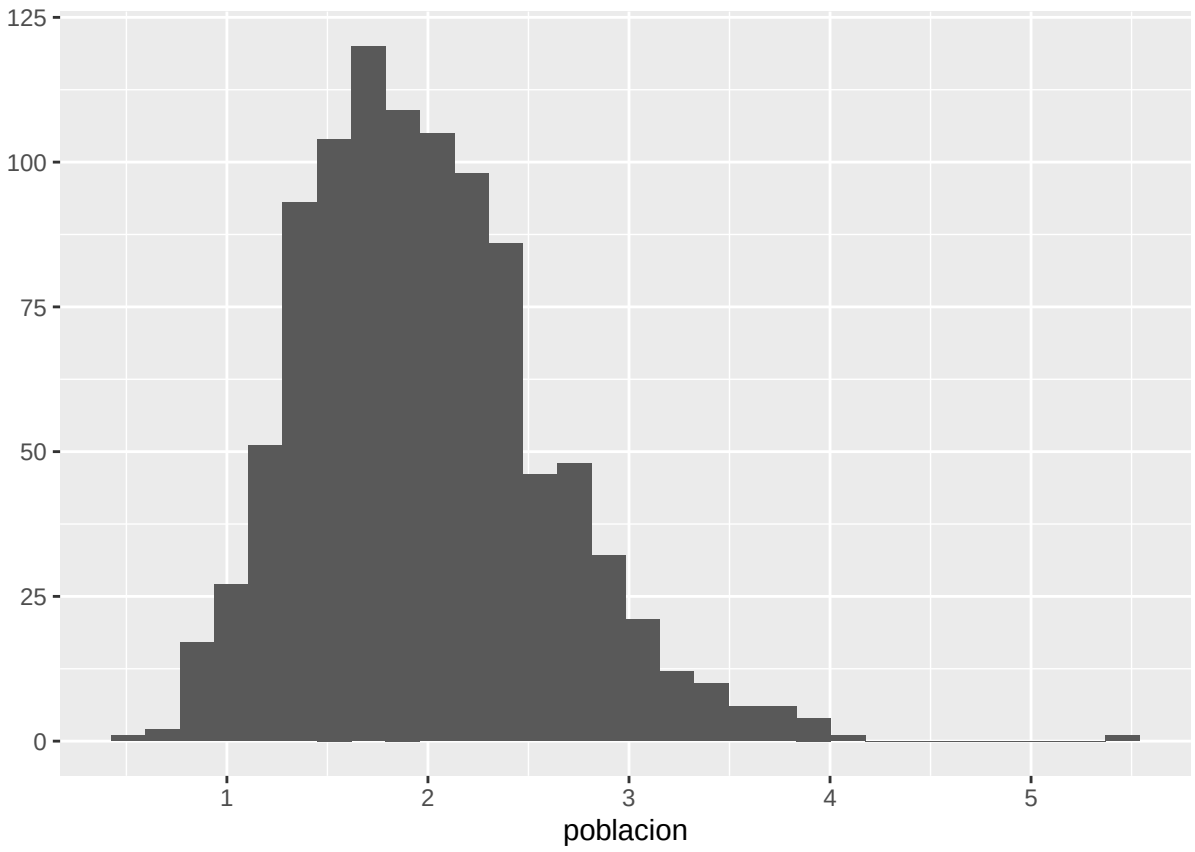
En este caso, la diferencia en promedio entre B y D es más pronunciada que antes, pero ya no es tan evidente (como en el caso anterior) que el grupo tiene un efecto significativo sobre la media del resultado.

3 Tarea 3

3.1 Bootstrap básico

Dado que es dificil encontrar poblaciones reales que se ajusten bien a un modelo dado, es mejor simular una población artificial. Simulamos 1000 observaciones de una v.a. X con distribución gamma de parametros a = 10 y b = 5.

```
library(tidyverse)

poblacion <- rgamma(1000,shape = 10, rate = 5)
qplot(poblacion)
```



A continuación, se toma una muestra aleatoria simple de tamano 100 de X para actuar como en un caso practico y se olvida la población inicial simulada.

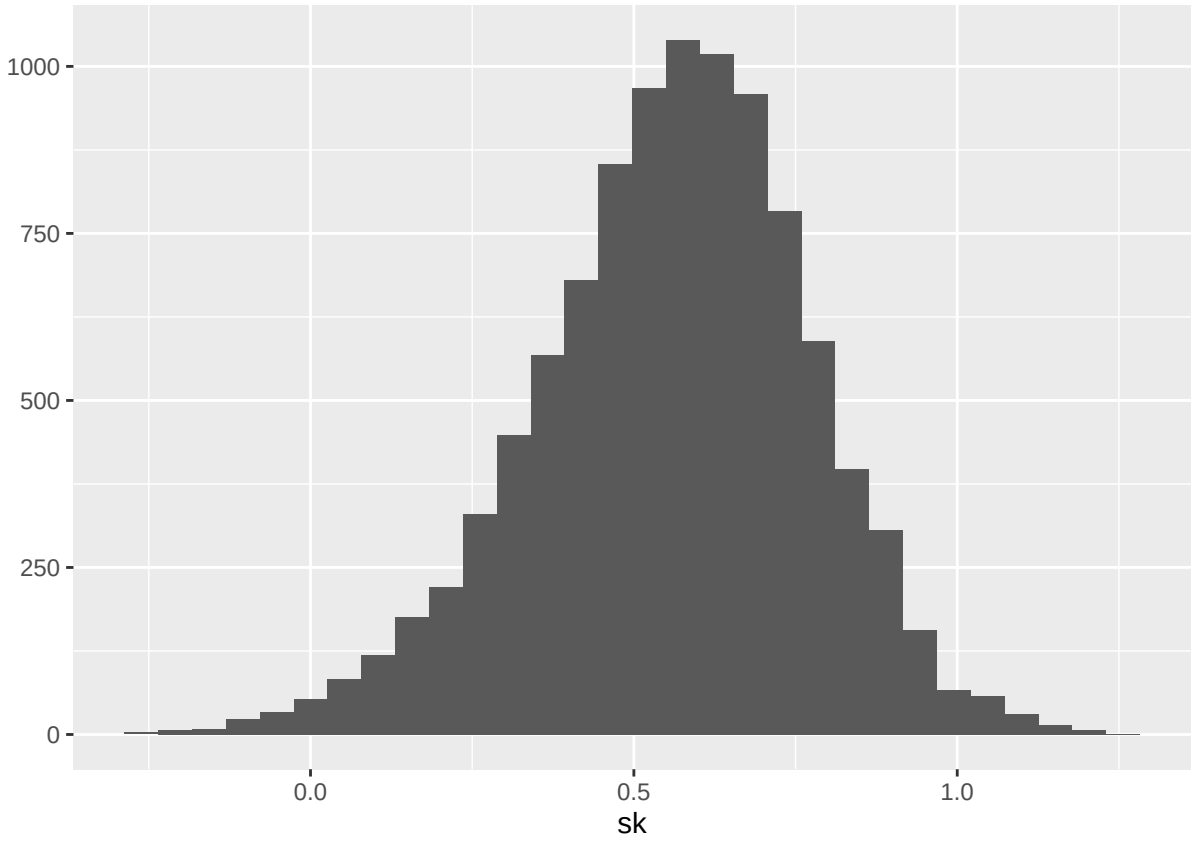
```
set.seed(0)
muestreo <- sample(poblacion,100, replace = F)

summary(muestreo)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.9093  1.5674  2.0181  2.0456  2.4662  3.9259 
```

Skewness

Ahora creamos las muestras *bootstrap* y calculamos la estadística de interés.

```
library(e1071)
sk=numeric(10000)
for (i in 1:10000) {
   sk[i]=skewness(sample(muestreo,length(muestreo), replace=TRUE), type=3)
   }
qplot(sk)
```



Calculamos la media de las 10000 muestras *bootstrap*

$$s(.) = \frac{\sum_{b=1}^{B} s(x^{*b})}{B}$$

```
skmean <- mean(sk)
skmean
```

```
[1] 0.5579708
```

Calculamos la desviación estándar como

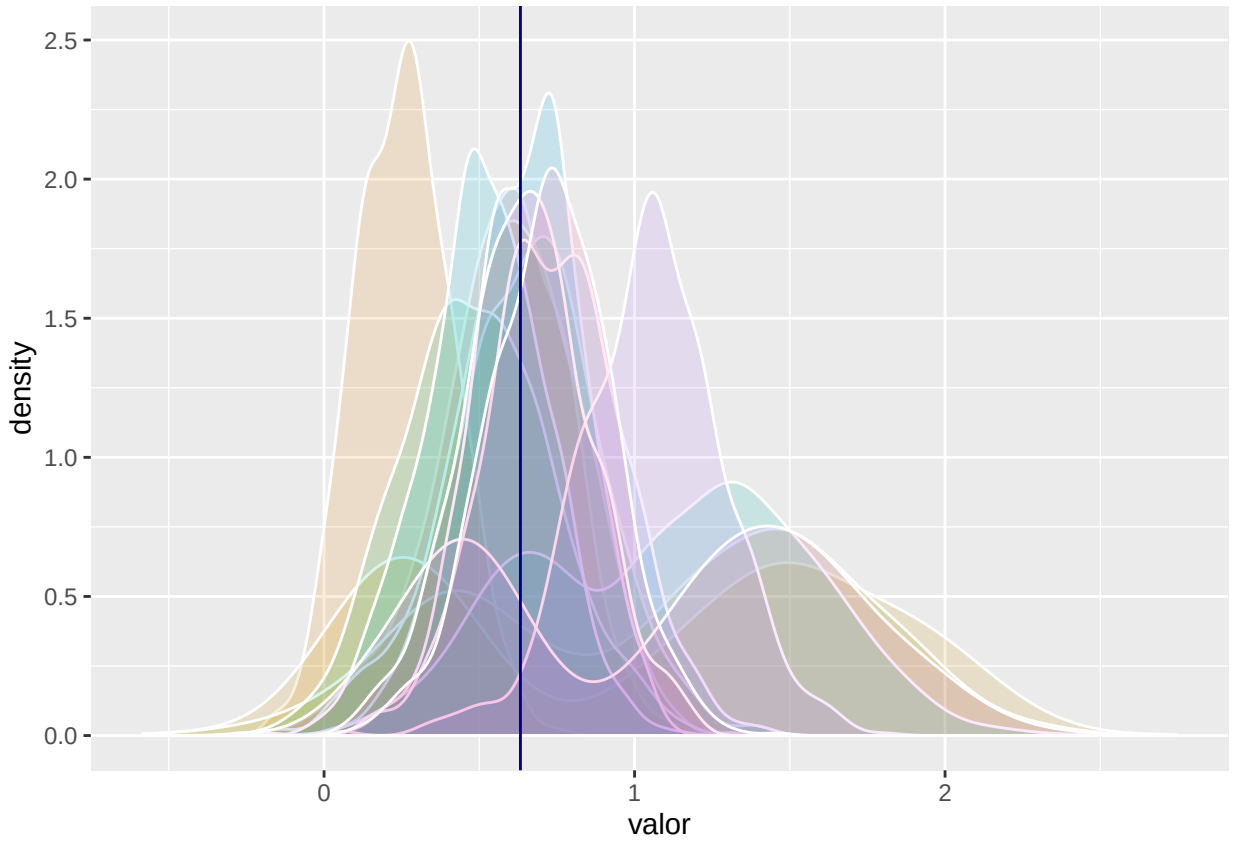
$$\hat{se}_{Boot} = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}(s(x^{*b}) - s(.))^2}$$

```
seboot= sqrt(sum((sk-skmean)^2)/(length(sk)-1))
seboot
```

```
[1] 0.2123349
```

Hacemos ahora el mismo procedimiento para el cálculo del *skewness*, solo usando muchas muestras distintas, y vemos como cambia la estimación de la densidad (con smoothing) del estadistico segun la muestra:

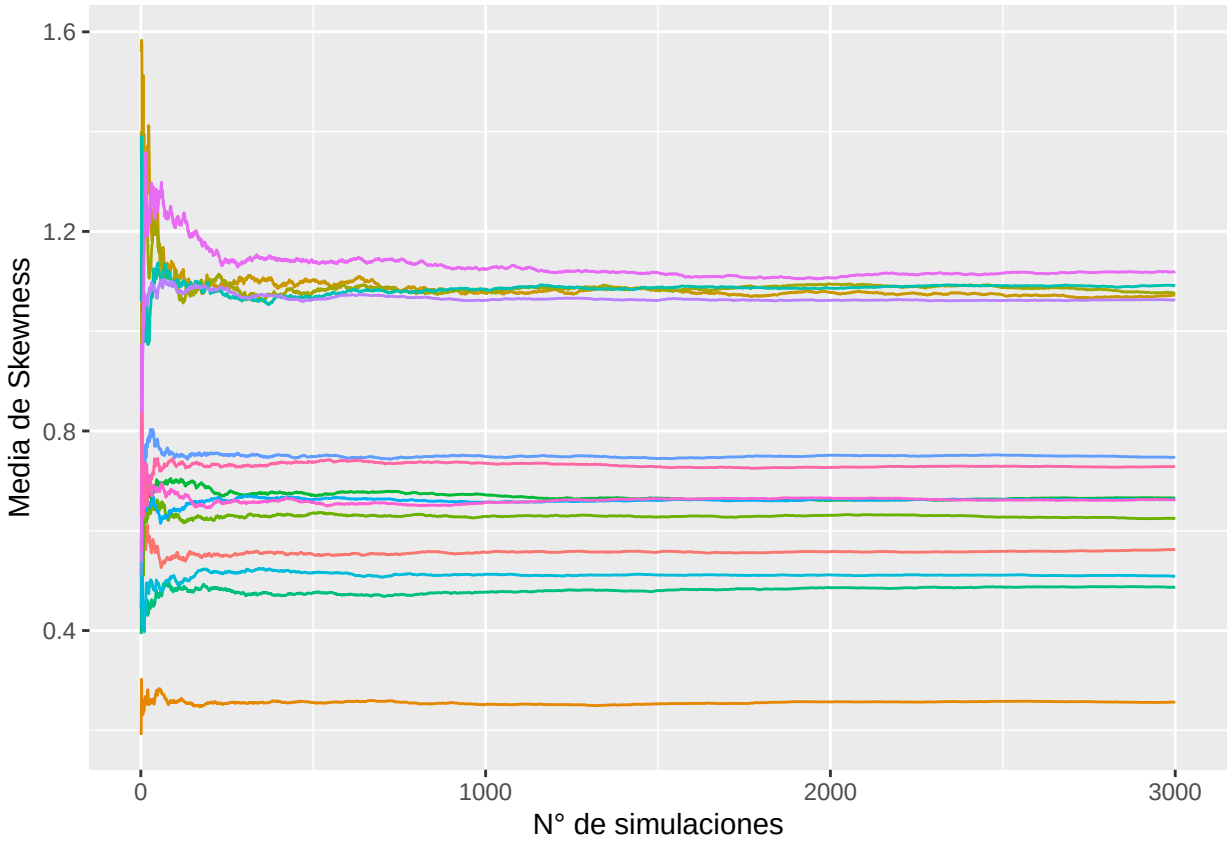
```r
set.seed(0)
a<- data.frame(replicate(15,sample(poblacion,100, replace = F)))

vect<- c(rep(0, 1500))

a2<- matrix(vect, ncol=15)
a2<- as.data.frame(a2)

for(i in 1:15)
for (j in 1:1000) {
   a2[j,i]=skewness(sample(a[,i] , length(muestreo), replace=TRUE), type=3)
}

muestraNum<- c(rep(1, 1000), rep(2,1000), rep(3,1000), rep(4,1000), rep(5,1000), rep(6,1000),
               rep(7,1000), rep(8,1000), rep(9,1000), rep(10,1000), rep(11,1000), rep(12,1000),
               rep(13,1000), rep(14,1000), rep(15,1000))


a3<- gather(data=a2, value=valor)
a3<- a3%>%mutate(muestra=as.factor(muestraNum))
realSkewness<- 2/sqrt(10)

p<- ggplot(data=a3, aes(x=valor))+ geom_density(aes(fill=muestra), alpha=0.15, col="white")+
  geom_vline (xintercept= realSkewness, col= "darkblue")

p + theme(legend.position = "none")
```



Se puede hacer un gráfico similar en significado al anterior, solo más claro. En lugar de ver la distribución del

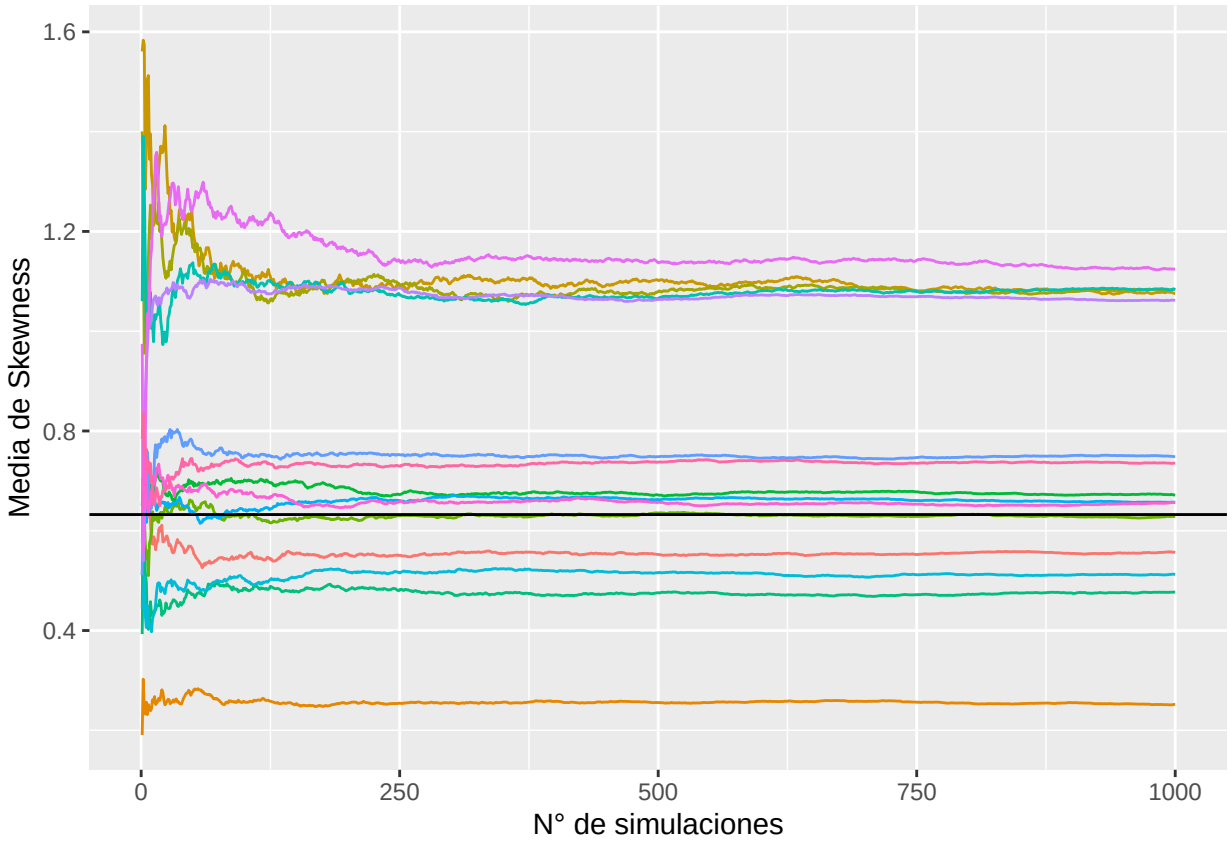
estadístico de interés, se puede ver cómo su promedio acumulativo varía con el número de simulaciones, para cada muestra aleatoria seleccionada. Vemos que el promedio tiende a estabilizarse en torno al valor real de el estadístico en la población (aquì las poblaciones son cada una de las muestras), y también vemos cómo este valor real varía de acuerdo con la muestra inicial de 100 unidades que se tome.

```
vect<- c(rep(0, 45000))

a2<- matrix(vect, ncol=15)
a2<- as.data.frame(a2)

for(i in 1:15)
  for (j in 1:3000) {
    a2[j,i]=skewness(sample(a[,i] , length(muestreo), replace=TRUE))
  }


muestraNum<- c(rep(1, 3000), rep(2,3000), rep(3,3000), rep(4,3000), rep(5,3000), rep(6,3000),
               rep(7,3000), rep(8,3000), rep(9,3000), rep(10,3000), rep(11,3000), rep(12,3000),
               rep(13,3000), rep(14,3000), rep(15,3000))

numeroSim= c(rep(seq(1,3000, by=1), 15))


a3<- gather(data=a2, value=valor)
a3<- a3%>%mutate(muestra=as.factor(muestraNum), numSim= numeroSim)



p<- a3 %>% group_by(muestra)%>% mutate(SumaCum=cumsum(valor)/numSim) %>%
                      ggplot(aes(x=numSim, y=SumaCum, col=muestra)) + geom_line() +
                          theme(legend.position = "none")


p+ labs(y= "Media de Skewness", x= "N° de simulaciones")
```



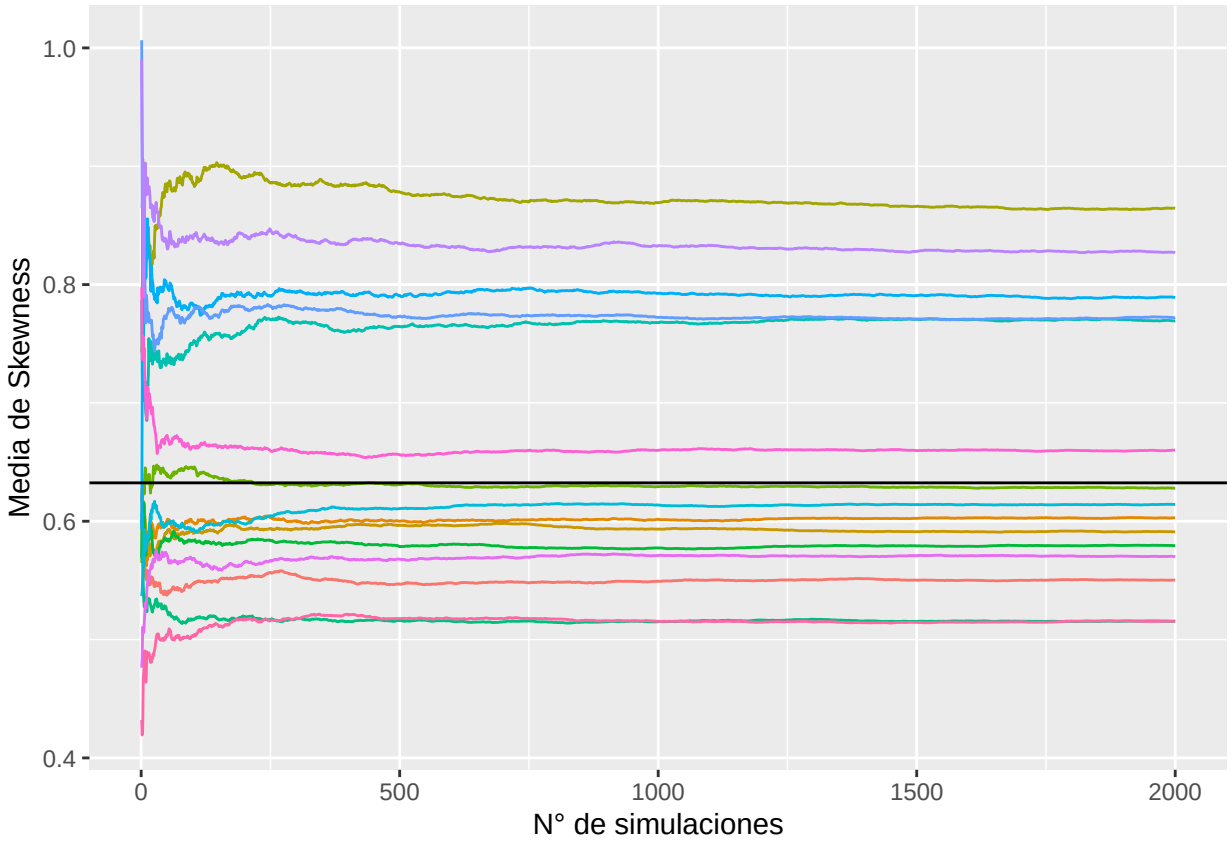
Si nos enfocamos en las primeras 1000 simulaciones se puede comparar más fácilmente este gráfico con el de las densidades visto antes, que se basaban en 1000 réplicas *bootstrap* de cada "población" (cada "población" es a su vez una muestra de la verdadera población que es una gama, aunque a su vez el vector población se ha obtenido simulando de una gama). Si las densidades de antes se hubieran calculado con 3000 réplicas bootstrap, habrían estado más concentradas (más puntiagudas).

```
p + xlim(c(0,1000))+ labs(y= "Media de Skewness", x= "N° de simulaciones") +
  geom_hline(yintercept= realSkewness)
```



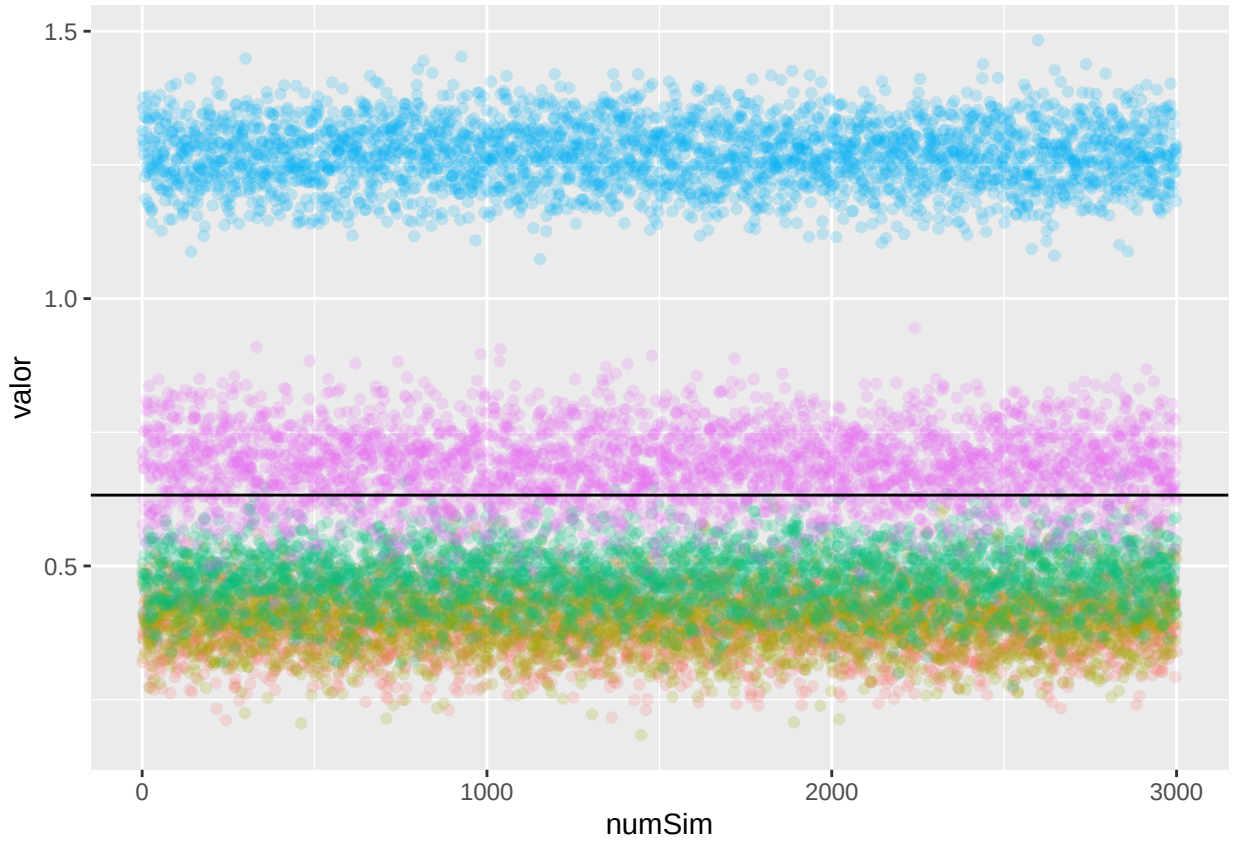
Finalmente se podría ver si al aumentar el tamaño de las muestras tomadas de la gama (nuestras “poblaciones”), el valor al que tienden las sumas acumuladas està más cerca de lo real. Entonces en lugar de tomar muestras iniciales de tamaño 100 de la gama, las tomamos de tamaño 1000. Por lo tanto, también cambiamos el tamaño del vector “población” originario, dado que inicialmente consistía en 1000 observaciones, y ahora queremos tomar muestras de tamaño 1000 de la población sin remplazo.

```
p2+ xlim(c(0,2000)) + labs(y= "Media de Skewness", x= "N° de simulaciones")+
  geom_hline(yintercept= realSkewness)
```



Aquì debajo se ven los valores del *skewness* para cada replica y para cada muestra (aqui se han reducido las muestras a 5, de lo contrario, el gráfico habría tenido muchos puntos superpuestos).

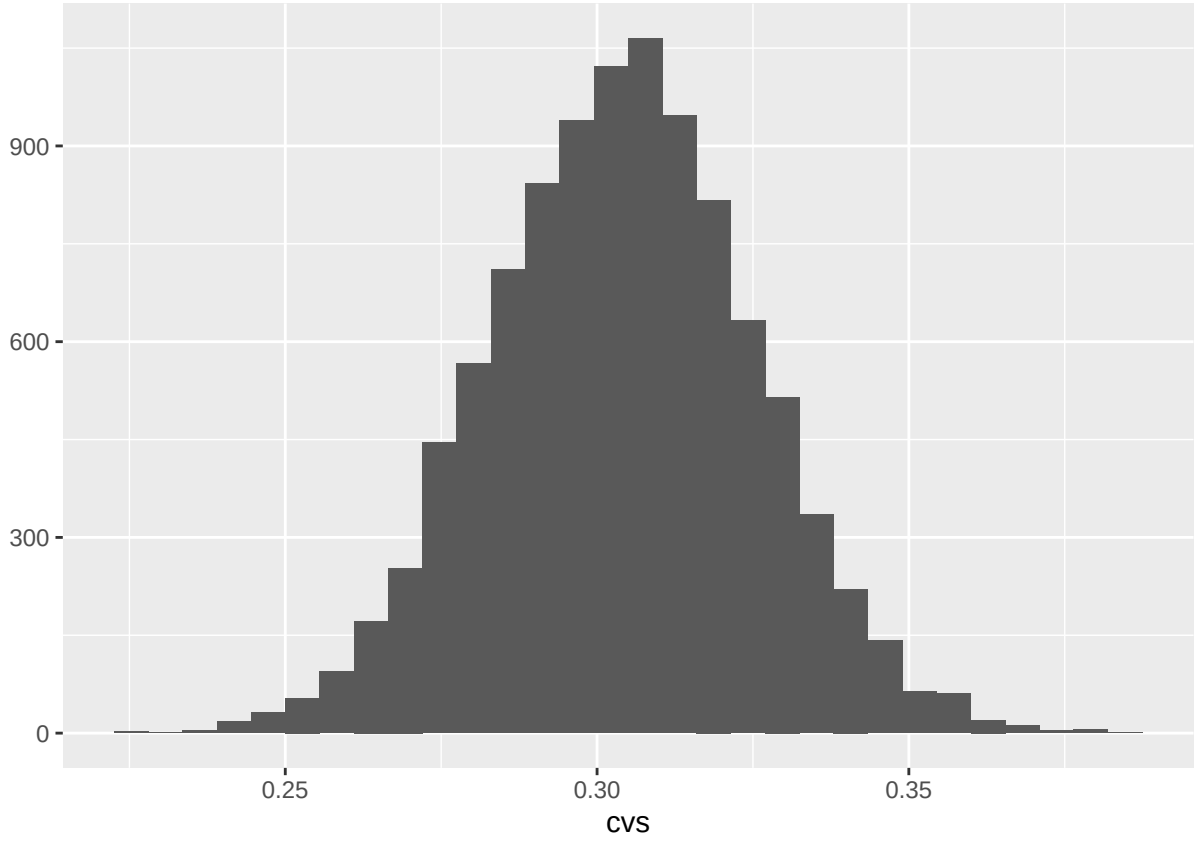
```
p22
```


1.5
1.0
0.5
valor
0
1000
2000
3000
numSim

Coeficiente de variación

Calculamos los coeficientes de variación de las muestras bootstrap.

```
library(labstatR)

cvs=c(rep(-1,10000))
for (i in 1:10000) {
   cvs[i]=cv(sample(muestreo,length(muestreo), replace=TRUE))
}


qplot(cvs)
```



Calculamos la media

```
cvmean<- mean(cvs)
cvmean
```

```
[1] 0.3039345
```

y la desviación tipica:

```
sebootcv= sqrt(sum((cvs-cvmean)^2)/(length(cvs)-1))
sebootcv
```

```
[1] 0.0211881
```

3.2 Rcpp

3.2.1 Skewness con Rcpp

Implementamos ahora el *bootstrap* escribiendo un programa en C ++ , compilándolo y ejecutándolo a través de la llamada a `sourceCpp()` desde R. Los resultados del valor de asimetría pueden ser un poco diferentes de los anteriores porque depende de cómo se implemente el cálculo de asimetría en R.

```
library(Rcpp)

sourceCpp(code='
#include <cmath>
#include <Rcpp.h>
using namespace Rcpp ;
// [[Rcpp::export]]



List boot_Sk(NumericVector poblacion, int replicas=1000) {


    int n1 = poblacion.size();


    NumericVector samplesk(replicas);
    NumericVector sampl(replicas);
    NumericVector n_sampl(replicas);

    bool replace = true;

    for(int i=0; i<replicas; i++) {

        sampl = sample(poblacion, n1, replace);

        for(int j = 0 ; j < n1; j++){
            n_sampl[j] = pow(sampl[j] - mean(sampl),3);
        }

     samplesk[i] = mean(n_sampl)/pow(sd(sampl),3);

    }


    double sk = mean(samplesk);
    double sale = sd(samplesk);

    List saletodo;
    saletodo["t"] = samplesk;
    saletodo["mean"] = sk;
    saletodo["sd"] = sale;
    return saletodo;
    }')
```

Después se llama la función que acabamos de crear en C ++ desde R. Guardamos el tiempo de ejecución de 100000 réplicas *bootstrap* para poder comparar con lo que veremos en la siguiente sección.

```
s<- system.time ({ sale <- boot_Sk(muestreo, 100000) })
s
```

```
   user  system elapsed
 19.803   0.170  21.164
```

Mostramos los elementos de la lista que devuelve la función `boot_Sk`. Con respecto a los valores de la estadística para cada réplica *bootstrap* (guardadas en el elemento `"t"` de la lista `sale`) solo mostramos los primeros 5.

```
sale$t [1:5]
```

```
[1] 0.0007785597 0.0006088189 0.0007278805 0.0008269717 0.0006635817
```

```
sale$mean
```

```
[1] 0.0005571727
```

```
sale$sd
```

```
[1] 0.0002142381
```

3.2.2 CV con Rcpp

Repetimos los mismos pasos solo que esta vez calculamos el coeficiente de variación.

```
sourceCpp(code='
#include <cmath>
#include <Rcpp.h>
using namespace Rcpp ;
// [[Rcpp::export]]


List boot_CV(NumericVector poblacion, int replicas=1000) {


    int n1 = poblacion.size();


    NumericVector sampleCV(replicas);
    NumericVector sampl(replicas);

    bool replace = true;

    for(int i=0; i<replicas; i++) {

        sampl = sample(poblacion, n1, replace);
        sampleCV[i] = sd(sampl)/abs(mean(sampl));

    }


    double cv = mean(sampleCV);
    double sdev = sd(sampleCV);

    List saletodo;
    saletodo["t"] = sampleCV;
    saletodo["mean"] = cv;
    saletodo["sd"] = sdev;
```

```
    return saletodo;
    }')
```

```
s<- system.time ({ sale <- boot_CV(muestreo, 100000) })
s
```

```
   user  system elapsed
  0.160   0.020   0.181
```

```
sale$t [1:10]
```

```
 [1] 0.2752565 0.3083763 0.2951395 0.2905756 0.2932847 0.3005201 0.3309818
 [8] 0.3024247 0.2942413 0.2747238
```

```
sale$mean
```

```
[1] 0.3054884
```

```
sale$sd
```

```
[1] 0.02160893
```

3.3 Bootstrap en paralelo

Tener acceso a un *"cluster"* de CPU todo integrado en la misma computadora es mucho más fácil de lo que solía ser y ha permitido que muchas personas se puedan acercar a la computación paralela. Incluso puede que el ordenador esté calculando en paralelo sin que no os damos cuenta. De hecho, muchas librerías implementan un paralelismo incorporado que se ejecuta detrás de escena. Por lo general, este tipo de "paralelismo oculto" generalmente mejora la eficiencia computacional.

3.3.1 La opción `parallel="multicore"` de `boot()`

En primer lugar, cargamos la librería `parallel` que nos permite reescribir nuestras funciones para que se ejecuten en paralelo.

```
library(parallel)
```

Para correr el código en paralelo se puede usar la opción `parallel="multicore"` de la función `boot`. DetectCores() detecta el numero de *cores* CPU disponibles. Por defecto el comando detecta el numero de *cores* virtuales (`logic=TRUE`). En este caso estamos trabajando con un MacBook Pro 2017 (sin Touch Bar) que tiene 4 *cores* virtuales y `detectCores(logical = FALSE)` reales. Esto significa que tiene 2 CPUs físicos, pero, ya que cada *core* admite *hyperthreading*, cada *core* se presenta como 2 *cores* separados. En este ejemplo ponemos `ncpus` $= 4$, que es el número de procesos que se usaran en las operaciones paralelas.

```
library(boot)

lafuncion = function(x, i) {
  skewness(sample(x,length(x),TRUE))
 }
set.seed(0)
a<- system.time(boot(data=muestreo, lafuncion, R=100000, parallel = "multicore", ncpus=4))
a
```

```
   user  system elapsed
  4.985   1.444   2.525
```

```
# parallel multicore no funciona en Windows porque se basa en el Forking;
# hay que poner parallel = "snow". Parallel= "snow" es cross-platform, o
# sea funciona independientemente del sistema operativo.

set.seed(0)
b <- system.time(boot(data=muestreo, lafuncion, R=100000))
b
```

```
   user  system elapsed
  2.333   0.117   2.463
```

Con 100000 simulaciones se ve (arriba) que el tiempo de ejecución (`elapsed`) en paralelo es menor que el tiempo que tarda la funcion `boot()` sin opción para ejecución paralela.

Se nota que, en general, el tiempo transcurrido (`elapsed`) no será 1/4 del tiempo del `user`, que es lo que podríamos esperar con 4 *cores* si hubiera una ganancia de rendimiento perfecta en la paralelización.

Se puede ver la diferencia en los tiempos de ejecución, que es aún más evidente cuando aumenta el número de simulaciones. Con `n=500000` son (respectivamente con y sin paralelización):

```
d
```

```
   user  system elapsed
 18.774   1.868   6.529
```

```
c
```

```
   user  system elapsed
 12.551   0.659  13.272
```

Para más detalles sobre como se calculan los tiempos de ejecución, ver el [**capítulo 19** ***Profiling R code***] (https://bookdown.org/rdpeng/rprogdatascience/profiling-r-code.html)

3.3.2 Usando `mclapply`

Otra manera de paralelizar es usar `mclapply`, que se encuentra siempre en la librería `parallel` (que se desarrolla alrededor de los paquetes `multicore` y `snow` del CRAN). De la version 2.14.0 de R està preinstalada, o sea pertenece al *R-core*. `Mclapply` es la versión en paralelo de `lapply`, que es disponible solo en Mac (porque esta función también se basa en el mecanismo del *Fork*[3], típico de los sistemas operativos *Unix-style*). Basicamente distribuye los calculos los cálculos secuenciales de la función `lapply`. También hay otra funcion similar que es `parLapply` (a continuacion).

`Mclapply` tiene los mismos parametros que `lapply` pero necesita que se especifiquen otros valores como el argumento `mc.cores` con el que se especifica la cantidad de procesaos paralelos entre los que se desea dividir la computación.

```
funcion2<- function(x, i){

  x<-muestreo
  sk<-skewness(sample(x,length(x), replace=TRUE))

  # hay que definir una funcion de x y i

}


  s <- system.time({
          r <- mclapply(1:100000, funcion2, mc.cores = 4)})

  # r es una lista de 100 000 elementos (cada elemento tiene un unico valor que es el resultado
  # de la función skewness que se aplica 100000 a distintas réplicas bootstrap).
```

Aquì abajo vemos los primeros 5 elementos de la lista `r` y el tiempo necesario para la ejecución.

```
r[1:5]
```

```
[[1]]
[1] 0.5867948

[[2]]
[1] 0.6363734

[[3]]
[1] 0.6363734

[[4]]
[1] 0.6437906

[[5]]
[1] 0.7639253
```

```
s
```

```
   user  system elapsed
  2.727   0.578   1.145
```

Se observa que este método es más rápido que el `boot` en paralelo.

Comprobamos si incluso con 500000 simulaciones (las mismas que habíamos hecho antes con `parallel= TRUE` de la función `boot()`) los resultados de mclapply son mejores.

[3]El mecanismo del *Forking* copia toda la versión actual de R y la mueve a un nuevo *core*.

El tiempo de ejecución sube a 1.145, que sigue siendo mas rápido del tiempo correspondiente con `boot()`, que era de 6.529.

3.3.3 Usando `foreach` (con `registerDoMC()` o `registerDoparallel()`)

Las que vimos hasta ahora (basadas en el *Forking*) no son las unicas formas de ejecutar computación en paralelo que el paquete `parallel` ofrece. Otra forma es construir un *Cluster* utilizando múltiples *cores* a través de *Sockets*[4]

Escribimos un programa en paralelo con la librería `foreach` que puede implementar ambos mecanismos, dependiendo del *backend*. Se pueden especificar diferentes *backends* y, según lo que se especifique, se obtendran diferentes formas de implementar la paralelización. Si se usa `registerDoMC()`, este trabajará con el *Forking*; alternativamente puede usar `registerDoparallel()` que funciona con *Sockets*, pero además del comando `registerDoparallel()` en este segundo caso hay que hacer algo mas.

Para más detalles sobre la diferencia entre los mecanismos del *Forking* y el del *Socket* se vea el [**capítulo 2 *Parallel Computation***] (https://bookdown.org/rdpeng/rprogdatascience/parallel-computation.html)

```
library(foreach)
library(doParallel)
library(doMC)
library(moments)

registerDoMC(4)
# parallel backend (sino va a simular de manera secuencial)

X<-list(replicate(100000,sample(muestreo, 100, replace = T)))

functSkew<- function(x) {skewness(x)}


s<- system.time(
    {result<-foreach(x = X) %dopar% functSkew(x)})
s
```

```
   user  system elapsed
  1.639   0.129   1.781
```

```
# Ejecutado el comando registerDoMC() se especifica el numero de *cores* que
# se emplearan en el calculo distribuido.
```

Ahora veamos cómo crear un *cluster* que comunica a través de *Sockets*. El comando `makeCluster()` sirve para crear este *cluster* virtual.

```
cl <- makeCluster(4)
cl
```

```
socket cluster with 4 nodes on host 'localhost'
```

```
summary(cl)
```

```
     Length Class    Mode
[1,] 3      SOCKnode list
[2,] 3      SOCKnode list
[3,] 3      SOCKnode list
[4,] 3      SOCKnode list
```

El objecto `cl` que acabamos de crear es una abstracción de todo el *cluster* y es el objeto que indica que queremos hacer computación paralela.

[4]El mecanismo de *Sockets* lanza una nueva versión de R en cada *core*. Técnicamente, esta conexión se realiza a través de la red, por ejemplo, lo mismo que si estuviera conectado a un servidor remoto, pero la conexión se realiza en tu propia computadora.

```
registerDoParallel(cl) #se registra un backend apropriado para este segundo mecanismo
```

Si se ejecuta la misma línea de código que llamaba a la función `foreach` (justo antes de crear el objeto `cl`) se obtiene el siguiente error:

Error in functSkew(x) : task 1 failed - "non trovo la funzione"skewness""

La razón es que los datos, las funciones y los parametros que hemos definido o cargado en nuestra sesión R, no están disponibles para los procesos secundarios que han sido generados por la función `makeCluster()`. Los datos, y cualquier otra información que el proceso secundario necesitará para ejecutar su código debe exportarse al proceso hijo desde el proceso padre a través de la función `clusterExport()`. El segundo argumento de `clusterExport()`es un vector de caracteres con los nombres de los objetos (datos, funciones, variables...) que deben estar disponibles para los otros procesos. Además, para pasar comandos como `library()` se debe usar la función `clusterEvalQ()`.

La necesidad de exportar datos y librerías es una diferencia clave en el comportamiento entre el mecanismo `multicore` y el de *Socket*.

```
clusterEvalQ(cl, {library(moments)})
```

```
[[1]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[2]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[3]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[4]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

```
clusterExport(cl,varlist = c("muestreo", "functSkew", "X"))

s<- system.time(
    {result<-foreach(x = X) %dopar% functSkew(x)})
s
```

```
   user  system elapsed
  0.204   0.017   2.006
```

3.3.4 Usando parLapply

Para ejecutar la operación `lapply()` en este caso sobre un *cluster de Sockets* (antes lo vimos con el mecanismo del *Forking* con `mclapply()`) podemos usar la función `parLapply()`.

```
library(tidyverse)
clusterEvalQ(cl, {library(moments)})
```

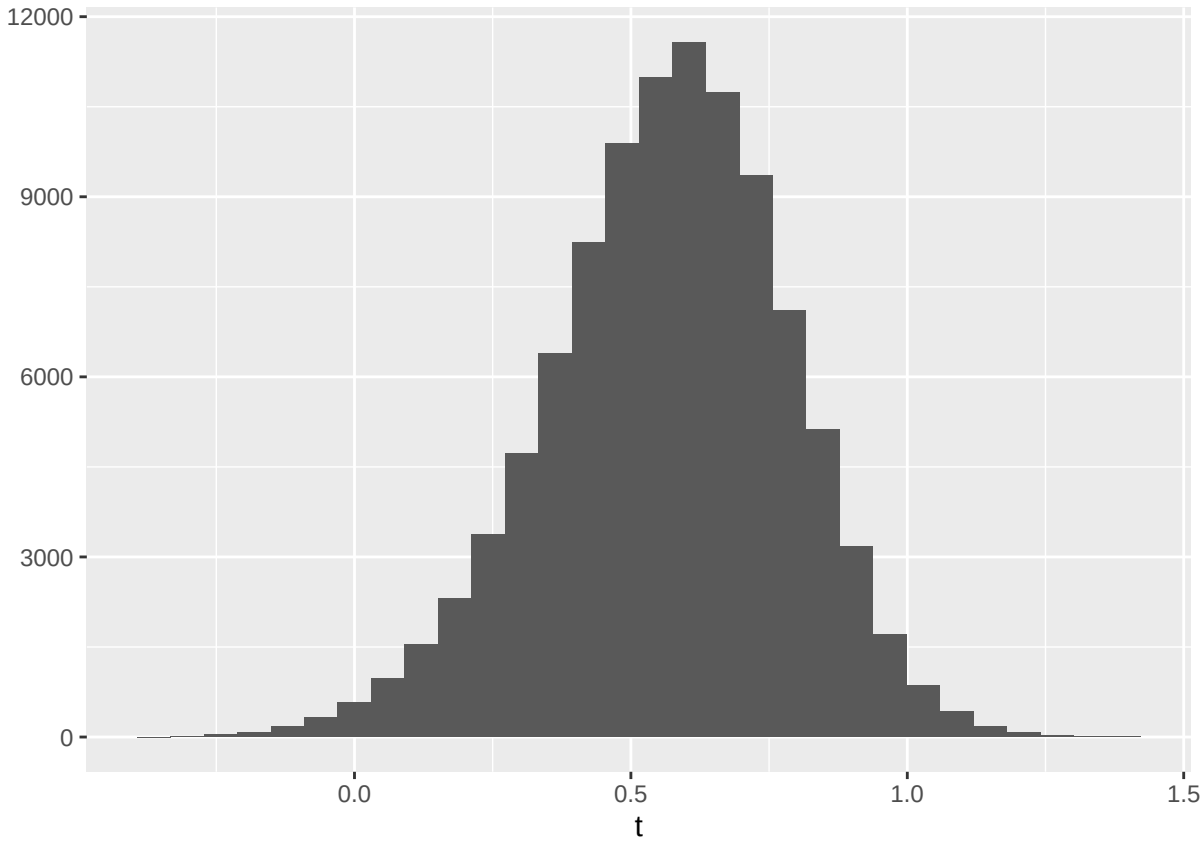
```
[[1]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[2]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

```
[[3]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[4]]
[1] "moments"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

```
clusterEvalQ(cl, {library(parallel)})
```

```
[[1]]
[1] "parallel"  "moments"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[2]]
[1] "parallel"  "moments"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[3]]
[1] "parallel"  "moments"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"

[[4]]
[1] "parallel"  "moments"   "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"
```

```
 q <- system.time({
         t <- parLapply(cl, X = X, functSkew)})

t<-unlist(t)
qplot(t)
```




```
q
```

```
   user  system elapsed 
  0.192   0.018   1.886 
```

Esta simulación ha tardado 1.886.

Hay que hacer un numero de simulaciones bastante elevado para empezar a ver los efectos de la ejecución en paralelo cuando se implementa el mecanismo del *Socket*: eso pasa porque algunas veces en *overhead* que conlleva la distribución de la tarea (enviar los datos para que sean accesibles en todos los procesos, hacer que las funciones estén disponibles para todos los *threads*, comunicar los resultados etc. . .) es mas costoso de la ventaja que se consigue con la parallelización del código. En este caso, entonces, con un numero pequeño de simulaciones incluso puede pasar que la función `lapply()` podría ser más rápida que su versión paralela.

Una vez que se haya terminado de trabajar con un *cluster*, es bueno limpiar los procesos secundarios de los objetos que han sido creados en la sesión (salir de R también hace que terminen todos los procesos secundarios).

```
stopCluster(cl)
```

4 Tarea 4

En estadística la **distribución de Cauchy** (a veces también **distribución de Lorentz**) es una distribución de probabilidad continua cuya función de densidad es

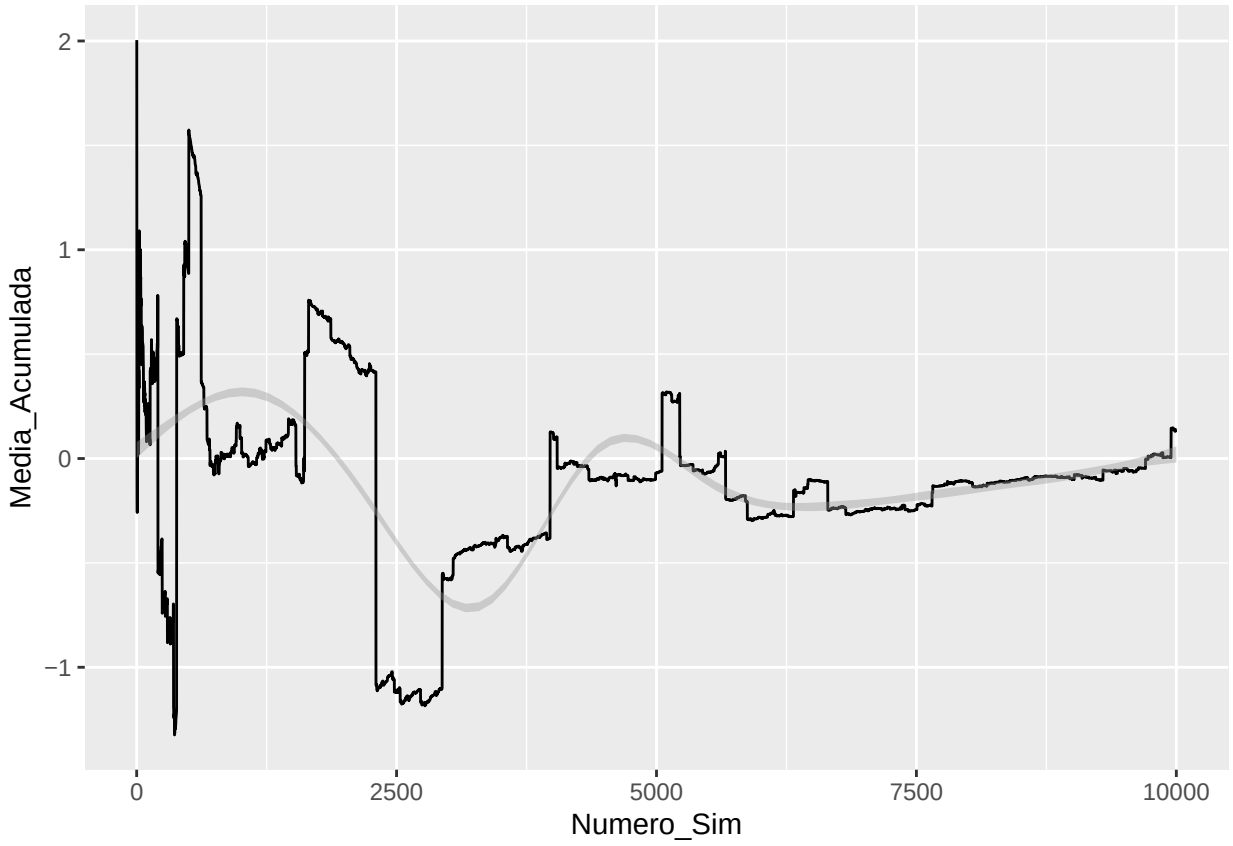
$$f(x; x_o, \gamma) = \frac{1}{\pi}[\frac{\gamma}{(x - x_o)^2 + \gamma^2}]$$

donde x_o es el parámetro de corrimiento que específica la ubicación del pico de la distribución, y γ es el parámetro de escala. En el caso especial donde $x_o = 0$ y $\gamma = 1$ es denominado la distribución estándar Cauchy con la función de densidad de probabilidad

$$f(x; 0, 1) = \frac{1}{\pi}\frac{1}{(x^2 + 1)}$$

.La distribución de Cauchy coíncide con la distribución t de Student con un grado de libertad. En general la distribución de Cauchy no tiene valor esperado ni varianza, mientras que su moda y mediana son iguales a x_o. De hecho se puede ver en la siguiente figura que la **ley de los grandes números** no se puede aplicar en este caso, ya que la media no es finita:

```
simul<- rcauchy(10000,0,1)
num<- seq(1, 10000, by=1)
data<- data.frame(Simulaciones= simul, Numero_Sim= num)
data%>% mutate(Media_Acumulada= cummean(Simulaciones))%>% ggplot(aes(x=Numero_Sim,
                                                    y=Media_Acumulada))+ geom_line()+
geom_smooth(size=NA)
```



Vemos que el promedio, cuando n aumenta, no converge a ningún valor. Esto pasa porque, al no tener una varianza finita, a veces se genera un valor extremo que hace que el promedio acumulado cambie repentinamente y no se establezca. La distribución de Cauchy con parámetros $(0, 1)$ se puede usar para definir todas las otras distribuciones de Cauchy: si la variable aleatoria T sigue esta distribución, entonces la variable aleatoria $x_o + y_o \cdot T$ sigue la distribución de Cauchy con parámetros (x_o, y_o).

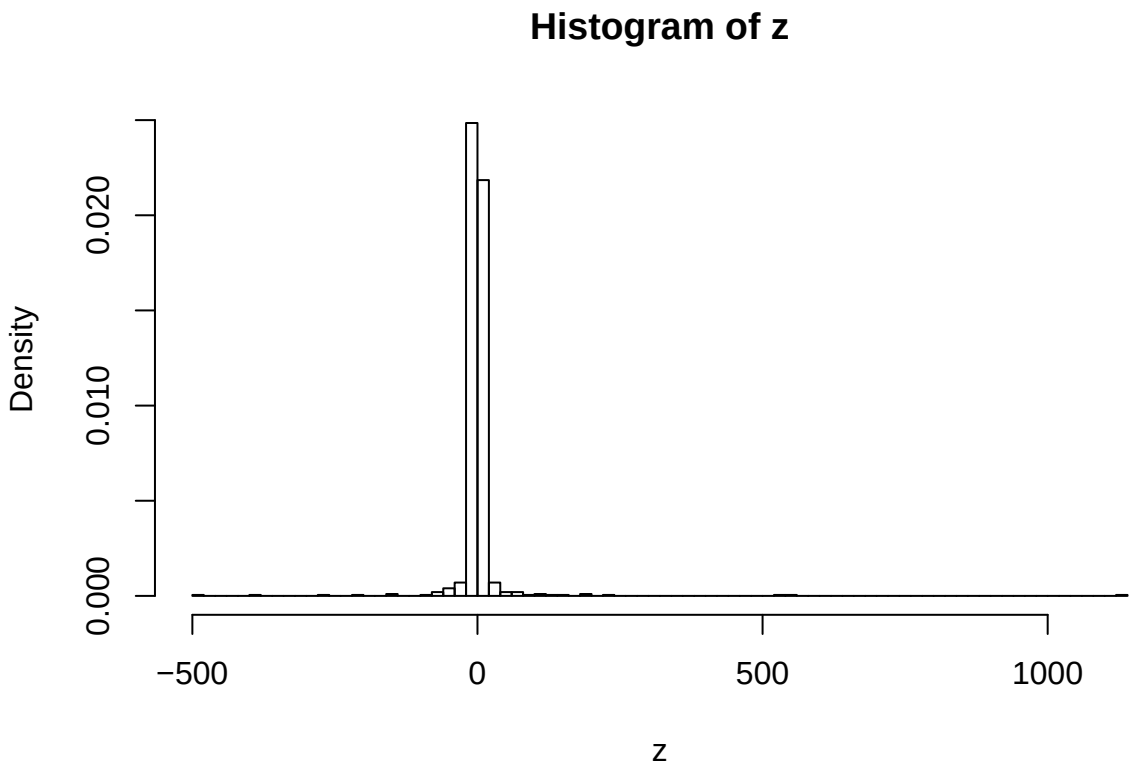
Hay un resultado que dice que cuando U y V son dos variables aleatorias independendientes y normalmente distribuidas con valor esperado 0 y varianza 1, luego la tasa $\frac{U}{V}$ tiene la distribución estándar de Cauchy.

Sin embargo, a continuación crearemos valores a partir de una Cauchy definida a partir del ratio de dos normales con media 0 pero con diferente varianza (respectivamente σ_U^2 es igual a 10 y σ_V^2) es igual a 2. Se puede demostrar que la distribución resultante es precisamente un cauchy con la siguiente función de densidad:

$$f_X(x) = \frac{1}{\pi} \frac{\frac{\sigma_U}{\sigma_V}}{x^2 + (\frac{\sigma_U}{\sigma_V})^2}$$

donde $X = \frac{U}{V}$. Haga clic aquí para ver los detalles de la demostración.

```
x1 <- rnorm(1000, mean = 0, sd=sqrt(10))
x2 <- rnorm(1000, mean=0, sd=sqrt(2))
z <- x1/x2
hist(z, breaks = 100, freq = F)
```



Se puede ver cómo del histograma del cociente (nuestra muestra de la Cauchy) obtenemos valores que son extremos con respecto a dónde se concentra la masa.

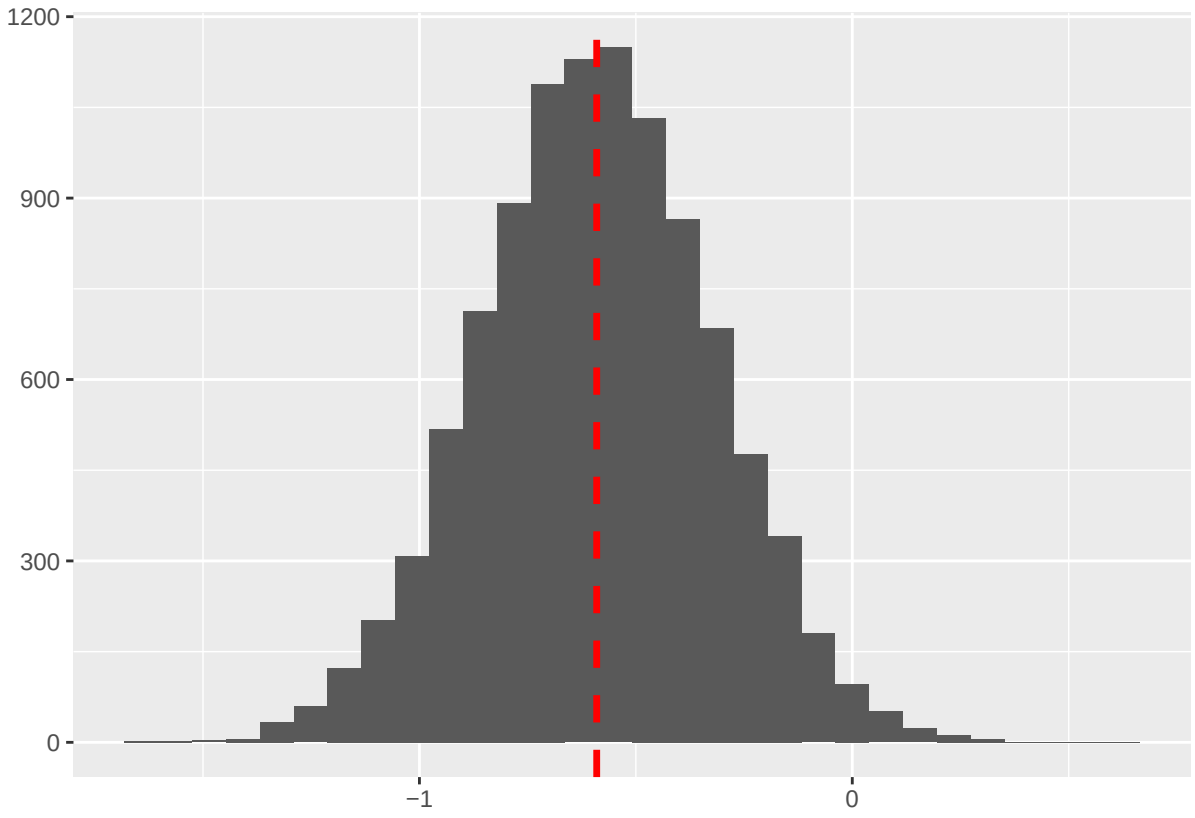
Se utiliza ahora un procedimiento de *bootstrap* paramétrico para calcular el error estándar de la mediana. Basicamente, en lugar de muestrear con reemplazamiento a partir de los datos originales (la muestra `z`), se sacan `B` muestras de tamaño `n` desde el estimador paramétrico de la población (al que se denota con $\hat{F}_{par}$). El estimador paramétrico se obtiene suponiendo saber cuál es el modelo probabilístico que generó los datos, y luego utilizando ese modelo con parámetros estimados en base a la muestra. Posteriormente se siguen los mismos pasos del algoritmo general del bootstrap no paramétrico: se calcula el correspondiente estadístico en cada muestra bootstrap y luego se calcula la desviación estándar de las `B` réplicas. En nuestro caso, necesitamos estimar los parámetros de la Cauchy a partir de la muestra, y luego usar $\hat{F}_{par}$ para obtener `B` réplicas de nuestro estadístico de interés (la mediana). Debido a que los parámetros de la distribución de Cauchy no corresponden a media y varianza, intentar estimar los parámetros utilizando la media y la varianza muestrales no es correcto. Aunque los valores de la muestra se concentrarán sobre el valor central x_o, la media de la muestra será cada vez más variable a medida que se tomen más observaciones, debido a la mayor probabilidad de encontrar puntos de muestra con un valor absoluto grande. De hecho, la distribución de la media muestral será igual a la distribución de las observaciones mismas; es decir, la media muestral de una

muestra grande no es un estimador mejor (o peor) de x_o que cualquier observación individual de la muestra. Del mismo modo, calcular la varianza de la muestra dará como resultado valores que aumentarán a medida que se tomen más observaciones. Por eso se utilizan medidas mas robustas para la estimación. Un método simple es tomar la mediana de la muestra como un estimador de x_o y la mitad del rango intercuartil de la muestra como un estimador de γ. Se han desarrollado otros métodos más precisos y robustos, que son los que vamos a usar en los siguientes pasos. Por ejemplo, la media truncada del 24% de las observaciones observaciones mas centrales de la muestra ordenada (`mean(muestra,trim = 0.76)`) produce una estimación para x_0 que es más eficiente que usar la mediana de la muestra o la media de la muestra completa. Sin embargo, debido a las colas de la distribución de Cauchy, la eficiencia del estimador disminuye si se usa más del 24% de la muestra.

```
zr <- sample(z, 200, replace=T)
xk=mean(zr,trim = 0.76)  #estimar x_o
realgama= sqrt(10)/sqrt(2)

z2=IQR(zr)
g=z2/2 # estimar gamma
```

Las estimaciones de x_o y γ son respectivamente -0.5904207 y 2.387511. Sabemos, conociendo los parámetros de U y V (las normales sa partir de las cuales hemos obtenido una muestra de la Cauchy), que los parámetros reales de la Cauchy $X = \frac{U}{V}$ deberían ser 0 y 2.236068 (que corresponde a $\frac{\sigma_U}{\sigma_V}$).

```
medians=numeric(10000)

for (i in 1:10000) {
sample=rcauchy(200,xk,g)
medians[i]=median(sample)
}

tt=data.frame(x=medians)
ggplot(tt,aes(x=tt$x))+geom_histogram()+xlab("")+ylab("")+geom_vline(xintercept=xk,
                                                    linetype="dashed",col="red", size=1.2)
```



El histograma muestra 10000 valores de la mediana (calculada como promedio recortado) calculada para 10,000 muestras, cada una obtenida muestreando de la Cauchy con parámetros estimados ($\hat{F}_{par}$). A continuación, utilizamos el procedimiento ya descrito en el capítulo 1 para calcular el error estándar de nuestra estimación (que aqui se denota con $se_{\hat{F}_{par}}(\hat{\theta}^*)$ para subrayar el hecho de que las muestras se obtuvieron de manera diferente).

```
medmean<- mean(medians)
sebootmedians= sqrt(sum((medians-medmean)^2)/(length(medians)-1))
```

Se obtiene que $se_{\hat{F}_{par}}(\hat{\theta}^*)$ es 0.2691118.

5 Bibliografía y recursos interactivos

- "Getting Started with doParallel and foreach", *Steve Weston and Rich Calaway* (2019) https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf;
- "Package 'parallel' R-core" (2018) https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf;
- "Getting Started with doMC and foreach", *Steve Weston* (July 16, 2019) https://cran.r-project.org/web/packages/doMC/vignettes/gettingstartedMC.pdf;
- "R Programming for Data Science", *Roger D.Peng* 2019-09, https://bookdown.org/rdpeng/rprogdatascience/;
- http://people.unica.it/musio/files/2008/10/Contrasti.pdf
- useR! International R User 2017 Conference Introduction to parallel computing with R https://www.youtube.com/watch?v=rlbeZv_cxJw&t=1290s.