Modelos de estadística clásica: muestreos aleatorios

Armando Ocampo

Librerías de trabajo

Para esta clase necesitamos cargar las siguientes librerias

```
library(dplyr)
library(ggplot2)
```

Dudas de la clase previa

Recodificar NA's

Los NA's son valores faltantes o no disponibles, los cuales suelen representar un problema durante la transformación y procesamiento de los datos. Por lo cual, previo a realizar cualquier análisis o función sobre el dataset es necesario conocer el comportamiento de nuestros datos. Para esto utilizaremos la función *summary()* de la paquetería *base* (Esta paquetería se encuentra instalada por defecto en el lenguaje de programación R, por lo cual no es necesario instalarla).

Generaremos un data frame con valores NA.

```
dummy_data <- data.frame(id=c(1,2,3,4,NA),
                         horas_estudio=c(2, 5, 4, 2, 1),
                         horas_recre=c(4, 2, 4, 3, 1),
                         edad = c(20, 15, 16, 22, 20))

summary(dummy_data)
```

```
##        id       horas_estudio  horas_recre       edad
##  Min.   :1.00   Min.   :1.0   Min.   :1.0   Min.   :15.0
##  1st Qu.:1.75   1st Qu.:2.0   1st Qu.:2.0   1st Qu.:16.0
##  Median :2.50   Median :2.0   Median :3.0   Median :20.0
##  Mean   :2.50   Mean   :2.8   Mean   :2.8   Mean   :18.6
##  3rd Qu.:3.25   3rd Qu.:4.0   3rd Qu.:4.0   3rd Qu.:20.0
##  Max.   :4.00   Max.   :5.0   Max.   :4.0   Max.   :22.0
##  NA's   :1
```

Además de algunas medidas de estadísitca descriptiva, esta función detalla si existen NA's en nuestro dataset. Otra manera de conocer si existen o no NA's es mediante la función *is.na()*. Esta generará como resultado TRUE, en el sitio donde encuentre un resultado faltante.

```
is.na(dummy_data)
```

```
##         id horas_estudio horas_recre  edad
## [1,] FALSE         FALSE       FALSE FALSE
## [2,] FALSE         FALSE       FALSE FALSE
## [3,] FALSE         FALSE       FALSE FALSE
## [4,] FALSE         FALSE       FALSE FALSE
## [5,]  TRUE         FALSE       FALSE FALSE
```

```
# podemos acompañarlo con la función sum(), para saber el total de NA's
sum(is.na(dummy_data))
```

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

```
# También, es posible utilizar la función which() para determinar el sitio
# en la columna donde se encuentra el NA. En este caso, conocemos que el NA
# se encuentra en la columna id, ahora identificaremos su posición

which(is.na(dummy_data$id))
```

```
## [1] 5
```

```
# Es así, como determinamos qué el valor faltante se encuentra en el quinto
# lugar de la columna id
```

Otra manera es utilizar la función *sapply()* acompañado de la función *sum()* para determinar el total de NA's por columna

```
sapply(dummy_data, function(y) sum(length(which(is.na(y)))))
```

```
##            id horas_estudio   horas_recre          edad
##             1             0             0             0
```

Estas son algunas maneras de identificar los valores faltantes. A continuación, se mostrarán algunos métodos para su eliminación.

El primer método es la recodificación, en el cual se le agrega un valor concreto al NA. Esto, al colocar la posición en la que se encuentra y renombrarla. En este caso, al ver la numeración, suponemos que el id faltante es el numero 5.

```
dummy_data$id[is.na(dummy_data$id)] = 5
dummy_data
```

```
##   id horas_estudio horas_recre edad
## 1  1             2           4   20
## 2  2             5           2   15
## 3  3             4           4   16
## 4  4             2           3   22
## 5  5             1           1   20
```

Vamos a generar de nuevo el valor NA. Sin embargo, en el siguiente ejemplo utilizaremos la función *na.omit()* para quitar el valor faltante. Esta función se caracteriza por eliminar la fila donde se encuentra el valor NA.

```
dummy_data <- data.frame(id=c(1,2,3,4,NA),
                         horas_estudio=c(2, 5, 4, 2, 1),
                         horas_recre=c(4, 2, 4, 3, 1),
                         edad = c(20, 15, 16, 22, 20))

na.omit(dummy_data)
```

```
##   id horas_estudio horas_recre edad
## 1  1             2           4   20
## 2  2             5           2   15
## 3  3             4           4   16
## 4  4             2           3   22
```

Nota: *na.omit()* puede ser funcional si la recodificación no es posible. No obstante, hay que tener cuidado en no eliminar la mayor parte de la información. Esto lo veremos a continuación

Si tenemos varios NA a lo largo del dataset, *na.omit()* puede ser contraproducente. El siguiente ejemplo lo detalla.

```
dummy_data_2 <- data.frame(id=c(1,2,3,4,NA,6,7,8,9,10,11,12),
                           horas_estudio=c(NA,5,4,2,1,3,NA,2,4,3,5,NA),
                           horas_recre=c(9,NA,8,3,NA,8,5,6,8,NA,4,7),
                           edad = c(20,15,NA,22,NA,17,18,NA,18,20,18,15))

# contando NA's por columna
sapply(dummy_data_2, function(y) sum(length(which(is.na(y)))))
```

```
##            id horas_estudio   horas_recre          edad
##             1             3             3             3
```

Al aplicar la función *na.omit()* perdemos la mayor parte de la información

```
na.omit(dummy_data_2)
```

```
##    id horas_estudio horas_recre edad
## 4   4             2           3   22
## 6   6             3           8   17
## 9   9             4           8   18
## 11 11             5           4   18
```

Para este caso, podemos hacer dos maneras de recodificación. Colocar un 0 en todos los NA's, o definir un valor específico por columna.

```
dummy_data_2 <- data.frame(id=c(1,2,3,4,NA,6,7,8,9,10,11,12),
                           horas_estudio=c(NA,5,4,2,1,3,NA,2,4,3,5,NA),
                           horas_recre=c(9,NA,8,3,NA,8,5,6,8,NA,4,7),
                           edad = c(20,15,NA,22,NA,17,18,NA,18,20,18,15))

dummy_data_2[is.na(dummy_data_2)] = 0

dummy_data_2
```

```
##    id horas_estudio horas_recre edad
## 1   1             0           9   20
## 2   2             5           0   15
## 3   3             4           8    0
## 4   4             2           3   22
## 5   0             1           0    0
## 6   6             3           8   17
## 7   7             0           5   18
## 8   8             2           6    0
## 9   9             4           8   18
## 10 10             3           0   20
## 11 11             5           4   18
## 12 12             0           7   15
```

Para el segundo caso, colocaremos el valor de la media aritmética por columna para el proceso de recodificación.

```
dummy_data_2 <- data.frame(id=c(1,2,3,4,NA,6,7,8,9,10,11,12),
                           horas_estudio=c(NA,5,4,2,1,3,NA,2,4,3,5,NA),
                           horas_recre=c(9,NA,8,3,NA,8,5,6,8,NA,4,7),
                           edad = c(20,15,NA,22,NA,17,18,NA,18,20,18,15))

mean(dummy_data_2$horas_estudio, na.rm = TRUE)
```

```
## [1] 3.222222
```

```
dummy_data_2$horas_estudio[is.na(dummy_data_2$horas_estudio)]=2.4

mean(dummy_data_2$horas_recre, na.rm = TRUE)
```

```
## [1] 6.444444
```

```
dummy_data_2$horas_recre[is.na(dummy_data_2$horas_recre)] = 4.8

mean(dummy_data_2$edad, na.rm = TRUE)
```

```
## [1] 18.11111
```

```
dummy_data_2$edad[is.na(dummy_data_2$edad)] = 13

dummy_data_2
```

```
##    id horas_estudio horas_recre edad
## 1   1           2.4         9.0   20
## 2   2           5.0         4.8   15
## 3   3           4.0         8.0   13
## 4   4           2.0         3.0   22
## 5  NA           1.0         4.8   13
## 6   6           3.0         8.0   17
## 7   7           2.4         5.0   18
## 8   8           2.0         6.0   13
## 9   9           4.0         8.0   18
## 10 10           3.0         4.8   20
## 11 11           5.0         4.0   18
## 12 12           2.4         7.0   15
```

Nota: es posible combinar métodos de recodificación. Todo depende del objetivo de la limpieza del conjunto de datos

Identificando cuartiles

Los cuartiles dividen al conjunto de datos en 4 grupos, a partir de 3 puntos de corte. Primer cuartil (Q1, 25%), segundo cuartil (Q2, 50%), tercer cuartil (Q3, 75%). Derivado de esta información, podemos definir en que sitio se puede encontrar un dato nuevo. Los siguientes son solo ejemplo de cómo realizarlo, ya que pueden existir varias maneras de hacerlo.

Primero identificaremos los cuartiles de la longitud del sépalo del dataset iris

```
quantile(iris$Sepal.Length)
```

```
##   0%  25%  50%  75% 100%
##  4.3  5.1  5.8  6.4  7.9
```

La siguiente función genera un mensaje dependiendo del sitio en el cual se encuentra el nuevo valor

```
identificando_cuartiles <- function(x){
  if(x <= 5.1) print('debajo Q1')
  if(x > 5.1 & x <= 5.8) print('entre Q1 y Q2')
  if(x >5.8 & x <= 6.4) print('entre Q2 y Q3')
  if(x > 6.4) print('superior a Q3')
}

identificando_cuartiles(6.5)
```

```
## [1] "superior a Q3"
```

La función *case_when()* de la paquetería *dplyr* permite generar condiciones similares a las utilizadas en la función *if()*. Cuando se utiliza en conjunto a la función *mutate()*, los resultados se guardan en una columna nueva.

```
iris_df <- iris

quantile(iris_df$Sepal.Length)
```

```
##   0%  25%  50%  75% 100%
##  4.3  5.1  5.8  6.4  7.9
```

```
iris_df_cuartil <- iris_df %>%
  mutate(cuartil = case_when(Sepal.Length <= 5.1 ~ '< Q1',
                             Sepal.Length > 5.1 & Sepal.Length <= 5.8 ~ 'Q1 & Q2',
                             Sepal.Length > 5.8 & Sepal.Length <= 6.4 ~ 'Q1 & Q3',
                             Sepal.Length > 6.4 ~ '> Q3'))

head(iris_df_cuartil)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species cuartil
## 1          5.1         3.5          1.4         0.2  setosa    < Q1
## 2          4.9         3.0          1.4         0.2  setosa    < Q1
## 3          4.7         3.2          1.3         0.2  setosa    < Q1
## 4          4.6         3.1          1.5         0.2  setosa    < Q1
## 5          5.0         3.6          1.4         0.2  setosa    < Q1
## 6          5.4         3.9          1.7         0.4  setosa Q1 & Q2
```

De esta manera, se puede colocar una variable nueva, pegarla al dataset y conocer en que cuartil se encuentra.

```
iris_df <- iris
iris_nueva <- c(5.222,NA,NA,NA,NA)

new_iris <- rbind(iris_nueva, iris_df)

head(new_iris %>%
       mutate(cuartil = case_when(Sepal.Length <= 5.1 ~ '< Q1',
                                  Sepal.Length > 5.1 & Sepal.Length <= 5.8 ~ 'Q1 & Q2',
                                  Sepal.Length > 5.8 & Sepal.Length <= 6.4 ~ 'Q1 & Q3',
                                  Sepal.Length > 6.4 ~ '> Q3')),1)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species cuartil
## 1        5.222          NA           NA          NA    <NA> Q1 & Q2
```

Muestreo

Cuando el conjunto de datos es grande y representa un elevado costo económico y de tiempo máquina se recomienda utilizar muestras. Estas deben de ser aleatorias y representativas. Los estadísticos obtenidos se pueden generalizar e inferir el comportamiento de la población. En R, existen dos funciones que permiten realizar este proceso, *sample()* y *slice_sample()*. La función *sample()* forma parte de la paquetería *base*, y se utiliza para encontrar muestras de vectores. Por su parte, la función *slice_sample()* pertenece a la paquetería *dplyr* y funciona para obtener muestras de data frames.

Nota: debido a que las muestras se realizan "al azar", es necesario definir un parámetro que permita replicar los resultados cuando se trabaja en grupos o para verificar la información. Para esto, se utiliza la función *set.seed()* antes de comenzar a trabajar

En el siguiente ejemplo, se generará un vector con numeros del 1 al 100000. Posteriormente, se tomará una muestra de 20000 numeros. Dentro de la función *sample()*, se coloca como primer argumento el vector del cual se va a extraer la información y el segundo argumento indica el tamaño de la muestra.

```
set.seed(123)
vectorcito <- seq(1, 100000, by = 1)

vectorcito_muestra <- sample(vectorcito, size = 20000)

length(vectorcito_muestra)
```

```
## [1] 20000
```

Por su parte, *slice_sample()*, permite obtener una muestra de un data frame. El siguiente ejemplo extraerá 100 elementos al azar del conjunto de datos iris.

```
set.seed(123)
iris_muestra <- slice_sample(iris, n = 100)

summary(iris_muestra)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.200   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.500   Median :1.400
##  Mean   :5.847   Mean   :3.045   Mean   :3.796   Mean   :1.214
##  3rd Qu.:6.400   3rd Qu.:3.400   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :34
##  versicolor:29
##  virginica :37
##
##
##
```

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Muestreo con reemplazo

Una variante del muestreo aleatorio es el muestreo con reemplazo o *bootstrapping*, en este cada elemento puede ser seleccionado más de una vez. Se suele utilizar cuando nuestra proporción de datos es pequeña, por

lo cual se busca enriquecer la información. Sin embargo, tiende a presentar un sesgo, ya que solo depende de nuestros datos de inicio. Para el muestreo con reemplazo se utilizan las funciones *sample()* y *slice_sample()*, solo que se agrega el argumento *replace = TRUE*. En el primer ejemplo generaremos un vector con numeros del 1 al 100 y se realizará un muestreo con reemplazo de 300 números

```
set.seed(123)
vector_2 <- seq(1,100, by = 1)
vector_boot <- sample(vector_2, size = 300, replace = TRUE)
length(vector_boot)
```

```
## [1] 300
```

Para el ejemplo de muestreo con reemplazo utilizando *slice_sample()*, se tomará de nuevo el dataset iris, generando una muestra de 300 elementos.

```
set.seed(123)

iris_boot <- slice_sample(iris, n = 300, replace = TRUE)

summary(iris_boot)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.075   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.400   Median :1.300
##  Mean   :5.836   Mean   :3.045   Mean   :3.786   Mean   :1.208
##  3rd Qu.:6.500   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.700   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    : 98
##  versicolor:102
##  virginica :100
##
##
##
```

Nota: en el muestreo sin reemplazo se genera una muestra a partir de la población. En el muestreo con reemplazo se desarrolla una población teórica derivado de la información de una muestra

Distribución probabilística

Define la probabilidad de que un vento ocurra. La suma de las posibilidades de un evento debe ser igual a 1. Se habla de distribución discreta cuando los eventos son números enteros (1,2,3,4). Por su parte, la distribución contínua hace referencia a valore infinitos (1.1, 1.2, 1.3). Asimismo, en este proyecto hablaremos sobre dos tipos de distribuciones; uniforme y normal.

Distribución uniforme

En este tipo de probabilidad, todos los eventos tienen las mismas posibilidades de ocurrir, pongamos como ejemplo el lanzar un dado. Cada número tiene 1/6 (~0.16) de probabilidad de ocurrir. Para conocer la probabilidad de cada evento dentro de la distribución uniforme se utiliza la función *punif()* de la paquetería *stats*. Esta paquetería se incluye dentro de los datos por defecto del lenguaje de programación, por lo que no se debe descragar nada.

En el siguiente ejemplo, se calculará cual es la probabilidad de que al lanzar un dado se obtenga un número menor a 5. Los argumentos son lo siguientes. El primero describe el valor del evento que queremos conocer,

min se utiliza para definir el evento mínimo que puede ocurrir, en este caso, no lanzar el dado. Max, por su parte, es el valor máximo que puede presentarse, siendo 6.

```
punif(5, min = 0, max = 6)
```

```
## [1] 0.8333333
```

Si queremos conocer la probabilidad de que un evento ocurra cercano a la unidad se agrega el argumento lower.tail = FALSE. En el siguiente ejemplo calcularemos la probabilidad de obtener un numero mayor a 5 al lanzar un dado.

```
punif(5, min = 0, max = 6, lower.tail = FALSE)
```

```
## [1] 0.1666667
```

Para rangos, se resta la probabilidad menor a la probabilidad mayor. En el siguiente ejemplo obtendremos la probabilidad de obtener un número entre 3 y 5 al lanzar un dado.

```
punif(5, min = 0, max = 6) - punif(3, min = 0, max = 6)
```

```
## [1] 0.3333333
```

Distribución normal