

Computación Estadística con R

Sesión 1

Programa del Curso

- Introducción a R.
- Instalación, entorno y estructura de datos.
- Lectura y Manipulación/Transformación de datos.
- Visualización con ggplot2.
- Loops, Funciones y familia Apply.
- Reportes (R Markdown).
- Extras dentro de cada clase.

Clases

- Cada clase tendrá una parte expositiva, ejercicios en conjunto y actividades.
- Si hay dudas, interrumpir. **No existen preguntas tontas!**
- **Todos** escribiremos código.
- Todo el material estará en la plataforma del curso.

Introducción a R

¿Por qué R?

- R es un lenguaje de programación para Computación Estadística.

¿ Existen programas similares, por qué usar R?

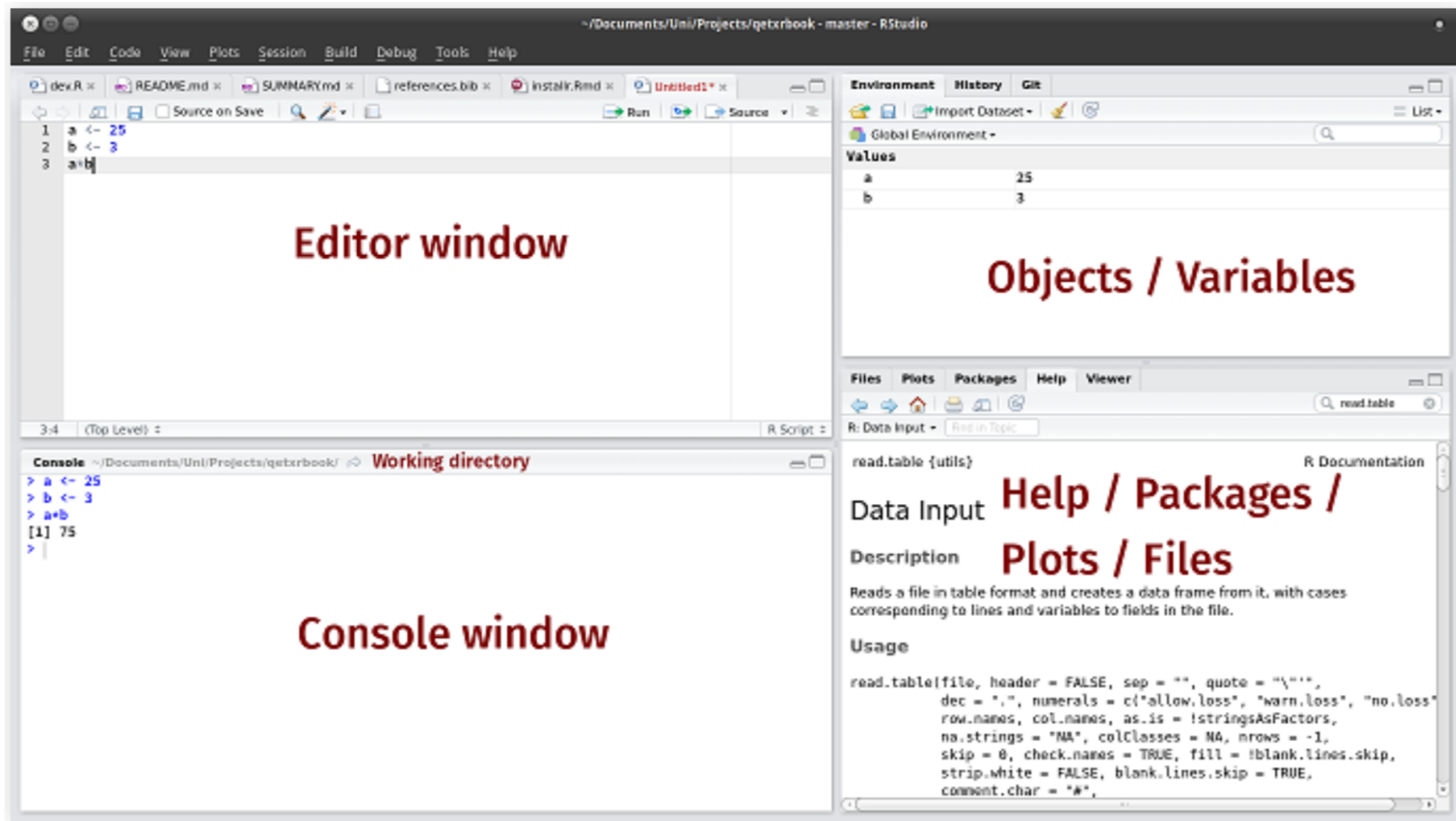
- R es **Gratis**.
- Existe una gran comunidad que lo utiliza.
- Puede manejar prácticamente cualquier formato de datos.
- Es un lenguaje, por lo que puede hacer todo.
- Es un buen trampolín para otros lenguajes como Python.

Descarga e Instalación



- R es el motor. (<https://cran.r-project.org>).
- RStudio es el panel de control. (<https://rstudio.com>).
- La instalación es como cualquier programa.
- Existe tambien <https://rstudio.cloud>.

Interfaz de RStudio



Estructura de Datos

Vectores

Los vectores los creamos con elementos del mismo tipo a través de la función **c()** (concatenar). En caso de mezclar caracteres con datos numéricos, el vector se considerará carácter.

```
vec <- c(1,4,3.14,-2) # <- es equivalente a usar =  
vec
```

```
## [1] 1.00 4.00 3.14 -2.00
```

Dimensión/largo del vector

```
length(vec)
```

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

Vectores (cont.)

Vectores reciclados

```
c(0.5,2) + c(1,2,3,4)
```

```
## [1] 1.5 4.0 3.5 6.0
```

```
c(0.5,2,0.5,2) + c(1,2,3,4)
```

```
## [1] 1.5 4.0 3.5 6.0
```

Vectores (cont.)

Un escalar es un caso especial

```
2 * c(1,-2,3) + 1
```

```
## [1] 3 -3 7
```

```
c(0.5,2) + c(4,-1,10)
```

```
## Warning in c(0.5, 2) + c(4, -1, 10): longer object length is not a multiple of  
## shorter object length
```

```
## [1] 4.5 1.0 10.5
```

Obs: No reciclar para no tener sorpresas en los cálculos.

Matemática en vectores

Algunas funciones operan sobre todo un vector y no sobre cada elemento.

```
#Sobre cada elemento  
c(1,2,3)^2
```

```
## [1] 1 4 9
```

```
# Sobre todo el vector  
sum(c(3,2,6))
```

```
## [1] 11
```

Algunas otras funciones: **mean()**, **max()**, **min()**, **median()**, **sd()**, **var()**.

Ejemplo: Estandarizando datos

Supongamos que obtuvimos algunos valores de mediciones de ph y queriamos ponerlos en una escala estandarizada:

$$ph^* = \frac{x_i - mean(x)}{sd(x)}$$

```
ph <- c(1.1,3.0,4.2,0.7,7.1,8,5.6,13.3,12.0)
z <- (ph - mean(ph)) / sd(ph) # z = ph estandarizado
round(z, 2)                  # redondeamos a 2 decimales

## [1] -1.12 -0.70 -0.43 -1.21  0.22  0.42 -0.11  1.61  1.32
```

Ejercicio: Realizar la función **scale()** sobre el vector de datos y comparar.

Tipos de vectores

Las funciones **class()** o **str()** nos indican que tipo de vectores tenemos.

- **Numérico:** `c(-2, 143, 1/2, -3.14)`, **Entero:* `0:10`
- **caracter:** `c("orange", "red", "yellow", "green")`
- **factor:** `factor(c("orange", "red", "yellow"))`
- **lógico:** `c(FALSE, FALSE, TRUE)`

Generando vectores numéricos

```
seq(-2, 5, by = 1.75) # secuencia desde -2 a 5, incrementos de 1.75
```

```
## [1] -2.00 -0.25  1.50  3.25  5.00
```

```
# seq(-2,5,length.out = 5) se obtiene lo mismo.
```

```
rep(c(-1, 0, 1), times = 3) # repetir c(-1,0,1) 3 veces
```

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

```
rep(c(-1, 0, 1), each = 3) # repetir cada elemento 3 veces
```

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

```
n <- 12  
1:n
```

```
## [1]  1  2  3  4  5  6  7  8  9 10 11 12
```


Generando vectores no numéricos

```
# creamos un factor con tres niveles, a, b y c  
mi.factor <- factor( sample( letters[1:3], 20, replace = T ) )  
mi.factor
```

```
## [1] a a b c a c a a c a a b b b a a c b c a  
## Levels: a b c
```

```
levels( mi.factor )    # a, b, c
```

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

```
# preferimos la ordenación b, c, a
```

```
mi.factor <- factor( mi.factor,  
                     levels = levels( mi.factor )[ c( 2,3,1 ) ] )  
levels( mi.factor )    # b, c, a
```

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

Generando vectores lógicos

```
first_names <- c("Pedro", "Catalina", "Roberto", "Andrea")  
name_lengths <- nchar(first_names) # número de caracteres  
name_lengths
```

```
## [1] 5 8 7 6
```

```
name_lengths >= 4
```

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

Obs: Se puede realizar cálculos con vectores lógicos debido a que **R** considera **TRUE** = 1 y **FALSE** = 0

Combinando condiciones lógicas

Si quisieramos solo los nombres con más de 6 letras:

```
largo <- name_lengths < 6  
largo
```

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

Si quisieramos solo los nombres donde la cuanrta letra es "r":

```
# substr: substring (porción) de un char vector  
letra.4 <- substr(first_names, start=4, stop=4) == "r"  
letra.4
```

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

Operadores lógicos

- **&** es "AND" (Ambas condiciones deben ser **TRUE**):

```
largo & letra.4
```

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

- **|** es "OR" (Al menos una condicion debe ser **TRUE**):

```
largo | letra.4
```

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

Operadores lógicos (cont.)

- ! es "NOT" (niega las propociones logicas, cambiando **TRUE** por **FALSE** y viceversa)

```
!largo
```

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

```
!(largo & letra.4 )
```

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

Extra

```
# Obteniendo los nombres con largo mayor a 6  
first_names[largo]
```

```
## [1] "Pedro"
```

```
# Si tenemos un vector con los sexos  
sex <- c("M", "F", "M", "F")
```

```
first_names[sex=="M"] # para comparar usar doble signo = (==)
```

```
## [1] "Pedro" "Roberto"
```

Extra (cont.)

```
# Veamos si nuestra lista coincide en algun nombre con este vector  
first_names %in% c("Roberto", "Carla", "Daniel")
```

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

```
# Obtenemos la posición dentro de nuestro vector  
which(first_names %in% c("Roberto", "Carla", "Daniel"))
```

```
## [1] 3
```

```
# Veamos cual es el que coincide  
first_names[3]
```

```
## [1] "Roberto"
```

NA's, Inf y NaN

Un valor perdido es codificado como **NA** sin "".

```
# un vector con valores NA  
vector_conNA <- c(1,2,NA,4,-3,NA,NA,-4)  
mean(vector_conNA)
```

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

Problemas!... los NA no dejan hacer cálculos correctamente. que hacer? remover los valores uno por uno... que lata...

```
mean(vector_conNA, na.rm=TRUE)
```

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


NA's, Inf y NaN (cont.)

```
# Para ver la posición de los valores NA  
is.na(vector_conNA)
```

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

Como manejar los **NA**

```
vector_conNA == -3
```

```
## [1] FALSE FALSE NA FALSE TRUE NA NA FALSE
```

```
vector_conNA %in% -3
```

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

NA's, Inf y NaN (cont.)

```
vector_conNA %in% NA
```

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

```
# Como funcionan los Inf y NaN  
vec <- c(-2, -1, 0, 1, 2) / 0 ; vec
```

```
## [1] -Inf -Inf NaN Inf Inf
```

```
rbind(is.finite(vec),is.nan(vec))
```

```
##      [,1] [,2] [,3] [,4] [,5]  
## [1,] FALSE FALSE FALSE FALSE FALSE  
## [2,] FALSE FALSE TRUE FALSE FALSE
```

Matrices

Una **Matriz** extiende los vectores a dos dimensiones. Veamos algunos comandos útiles

```
(a_matrix <- matrix(letters[1:8], nrow=2, ncol=4))
```

```
##      [,1] [,2] [,3] [,4]  
## [1,] "a"  "c"  "e"  "g"  
## [2,] "b"  "d"  "f"  "h"
```

```
(b_matrix <- matrix(letters[1:8], nrow=2, ncol=4, byrow=TRUE))
```

```
##      [,1] [,2] [,3] [,4]  
## [1,] "a"  "b"  "c"  "d"  
## [2,] "e"  "f"  "g"  "h"
```

Obs: Usando parentesis en la misma linea muestra los valores.

Matrices (cont.)

```
# empalmando vectores por fila  
(c_matrix <- rbind(c(2,4,6),c(-1,0,-3)))
```

```
##      [,1] [,2] [,3]  
## [1,]    2    4    6  
## [2,]   -1    0   -3
```

```
# empalmando vectores por columna  
(c_matrix <- cbind(c(1, 2,5,3), c(3, 4), c(5, 6)))
```

```
##      [,1] [,2] [,3]  
## [1,]    1    3    5  
## [2,]    2    4    6  
## [3,]    5    3    5  
## [4,]    3    4    6
```

Aquí tenemos un problema...

Matrices (cont.)

```
##      [,1] [,2] [,3] [,4]  
## [1,] "a"  "c"  "e"  "g"  
## [2,] "b"  "d"  "f"  "h"
```

```
# extraer la fila 2  
a_matrix[2,]
```

```
## [1] "b" "d" "f" "h"
```

```
# extraer la columna 3  
a_matrix[,3]
```

```
## [1] "e" "f"
```

```
# extraer el elemento 2,4  
a_matrix[2,4]
```

```
## [1] "h"
```

Matrices (cont.)

Ahora el comando para ver el tamaño es **dim()**

```
# primer valor es la cantidad de filas y el otro valor las columnas.  
dim(a_matrix)
```

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

```
matrix <- cbind(c("SI", "NO"), c(18, 31))  
# nombramos las filas y columnas  
rownames(matrix) <- c("Pedro", "Carla")  
colnames(matrix) <- c("Fuma", "Edad")  
matrix
```

```
##           Fuma Edad  
## Pedro "SI"  "18"  
## Carla "NO"  "31"
```

¿Algún problema?

```
typeof(matrix)
```

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

Listas

Una **Lista** es un objeto que puede almacenar muchos tipos de datos

```
(mi.lista <- list("letras"      = letters[1:4],  
                 "matriz"      = matrix(8:11, nrow = 2),  
                 "resultados" = lm(dist ~ speed, data = cars)))
```

```
## $letras  
## [1] "a" "b" "c" "d"  
##  
## $matriz  
##      [,1] [,2]  
## [1,]    8  10  
## [2,]    9  11  
##  
## $resultados  
##  
## Call:  
## lm(formula = dist ~ speed, data = cars)  
##  
## Coefficients:  
## (Intercept)      speed  
##    -17.579      3.932
```

Listas (cont.)

Como acceder a cada conjunto de datos en una lista?

```
mi.lista[["letras"]]
```

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

```
mi.lista$letras
```

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

```
mi.lista[[1]]
```

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


Renombrar elementos

Con el siguiente comando podemos renombrar tanto una columna o algún elemento de una lista.

```
require(reshape)
```

```
datos = rename(datos, c(nommbre_a_cambiar="newname"))  
names(datos)
```

```
lista = mi.lista[1:2]  
datos = rename(lista, c(letras="LETTERS"))  
datos
```

```
## $LETTERS  
## [1] "a" "b" "c" "d"  
##  
## $matriz  
##      [,1] [,2]  
## [1,]    8   10  
## [2,]    9   11
```

Renombrar elementos (cont.)

```
head(mtcars,3)
```

```
##           mpg cyl  disp  hp  drat    wt  qsec vs  am  gear  carb
## Mazda RX4      21.0   6  160  110 3.90  2.620  16.46  0   1    4    4
## Mazda RX4 Wag  21.0   6  160  110 3.90  2.875  17.02  0   1    4    4
## Datsun 710     22.8   4  108   93 3.85  2.320  18.61  1   1    4    1
```

```
mtcars2 = rename(mtcars, c(cyl="cilindrada"))
head(mtcars2,3)
```

```
##           mpg cilindrada disp  hp  drat    wt  qsec vs  am  gear  carb
## Mazda RX4      21.0         6  160  110 3.90  2.620  16.46  0   1    4    4
## Mazda RX4 Wag  21.0         6  160  110 3.90  2.875  17.02  0   1    4    4
## Datsun 710     22.8         4  108   93 3.85  2.320  18.61  1   1    4    1
```

Vamos a RStudio

Actividad 1

Resolvamos los ejercicios propuestos con lo que hemos aprendido hasta el momento.

- Descargar el **script1_clase1.R**

Expresiones Regulares

Detectando Patrones

- **str_which(x,pattern)** : encuentra las posiciones de las frases que contienen el patrón.
- **str_detect(x,pattern)**: Detecta si hay coincidencias con el patrón (valores lógicos).
- **str_c(x, collapse = ", ")** _ para colapsar vectores en un elemento.
- **strsplit()** , **str_split()**: dividen un string usando un patrón.
- **str_locate()** , **str_locate_all()**: localiza el patrón.
- **str_extract()** , **str_extract_all()**: Extraen las coincidencias (agregando simplify = T, se obtiene una matriz con los resultados).
- **str_replace()** reemplaza las coincidencias (**str_replace_all()** reemplaza todas).

Obs: todos los **str_** son cargando la librería **stringr** (es parte de la librería **Tidyverse**) .

Ejemplos

```
frases <- c("Chile se divide en regiones",  
            "La región del Biobío esta en Chile",  
            "El Sur de chile es lo mejor")  
patron <- '(en|Chile)' # o "en|Chile"
```

```
library(stringr)  
# extraemos posiciones  
str_which(frases, pattern = patron)
```

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

```
str_which(frases, pattern = patron, negate = T) # entrega el complemento
```

```
## [1] 3
```

```
str_subset(frases, pattern = patron) # obtiene las frases con el patrón
```

```
## [1] "Chile se divide en regiones"      "La región del Biobío esta en Chile"
```

Ejemplos (cont.)

```
str_split(frases, " ")
```

```
## [[1]]  
## [1] "Chile"      "se"         "divide"     "en"         "regiones"  
##  
## [[2]]  
## [1] "La"         "región"     "del"        "Biobío"     "esta"       "en"         "Chile"  
##  
## [[3]]  
## [1] "El"         "Sur"        "de"         "chile"      "es"         "lo"         "mejor"
```


Ejemplos (cont.)

```
stringr::str_locate_all(frases , patron)
```

```
## [[1]]  
##      start end  
## [1,]     1  5  
## [2,]    17 18  
##  
## [[2]]  
##      start end  
## [1,]    27 28  
## [2,]    30 34  
##  
## [[3]]  
##      start end
```

Ejemplos (cont.)

```
stringr::str_extract_all(frases , patron)
```

```
## [[1]]  
## [1] "Chile" "en"  
##  
## [[2]]  
## [1] "en"    "Chile"  
##  
## [[3]]  
## character(0)
```

```
stringr::str_extract_all(frases , patron, simplify = T)
```

```
##      [,1]      [,2]  
## [1,] "Chile"  "en"  
## [2,] "en"     "Chile"  
## [3,] ""      ""
```

Extra. paste()

Realize un vector que diga "iter 1", "iter 2",..., "iter 5".

```
iteraciones <- c("iter 1","iter 2","iter 3","iter 4","iter 5")
iteraciones
```

```
## [1] "iter 1" "iter 2" "iter 3" "iter 4" "iter 5"
```

Fácil!... pero si necesitamos que sea un vector de la forma "iter 1",..., "iter 1.000"

```
iteraciones <- paste("iter",sep= " ",1:1000)
sample(iteraciones,10)
```

```
## [1] "iter 517" "iter 200" "iter 427" "iter 343" "iter 500" "iter 559"
## [7] "iter 952" "iter 10"  "iter 112" "iter 410"
```

Extra.

- **merge()**: Para concatenar 2 bases de datos. (lo veremos más adelante(algo similar) con dplyr)
- **aggregate()**: agrega una base de datos. su comportamiento es en fórmula **X ~ Y**, Si X es una variable numérica e Y una de categorías voy a encontrar una función de X para cada categoría de Y. Entrega un nuevo objeto que es un dataframe.
- **subset()**: realiza filtro (similar a **[]**). Le entrego un objeto y le digo que sustraer. Reduce la dimensión del objeto a los que cumplen la condición. Retorna el mismo objeto filtrado.

Vamos a RStudio

Actividad 2

Resolvamos los ejercicios propuestos con la ayuda de las expresiones regulares.

- Descargar la base de datos **Encuesta.xlsx**
- Descargar el **script2_clase1.R**