

# Fundamentos de la programación estadística y Data Mining en R

## Unidad 1. Programación Estadística en R

Dr. Germán Rosati (Digital House - UNSAM - UNTREF)

19 abril, 2018

# ¿Qué es R?

- Básicamente, R es un “dialecto” de un lenguaje de los años '70: S-Language
- S fue creado en los *Bell Labs*
- R creado en 1991 por Ross Ihaka y Robert Gentleman
- 1993: R se anuncia por primera vez
- 2000: se lanza la primera versión R 1.0
- 2017: la versión más actual es la R 3.4.0

# Características de R

- Sintaxis similar a S (en su momento permitía hacer fácil el pasaje)
- Corre en casi cualquier SO/plataforma (incluso en PS3)
- Hay actualizaciones muy frecuentes (dos o tres por año)
- Funcionalidad modular: hay un conjunto de funciones básicas al cual se le van agregando diferentes paquetes con funcionalidades específicas
- Muy buenas capacidades gráficas: mucho mejores y más sofisticadas que cualquier software de procesamiento estadístico (extremo `ggplot2`)
- Lo mejor de todo: la comunidad. Cada estadístico que se le ocurre un algoritmo nuevo lo programa en R
- Lo segundo mejor: *GRATIS*

# Filosofía *Free software*

- Libertad de correr el soft con cualquier propósito (grado 0)
- Libertad de estudiar cómo funciona el programa y adaptarlo a las necesidades (grado 1). Requisito: disponer del código fuente
- Libertad de redistribuir copias (grado 2)
- Libertad de mejorar el software y lanzar las mejoras al público (grado 3). Mismo requisito que grado 1.

# Desventajas de R

- En general, los datos se cargan en memoria RAM (es más o menos lo mismo que casi cualquier otro soft). Algunos paquetes comenzaron a mejorar esta cuestión.
- La implementación es... espontánea. Si a nadie le interesa generar tu algoritmo... te toca a vos.

# Instalación de R y RStudio por primera vez

- Ir al sitio de R
- Elegir el repositorio del cual se desea descargar
- Descargar la última versión estable (actual R 3.4.0)
- Luego, es útil instalar alguna GUI para que el trabajo con el código sea más amigable.
- RStudio es una de las más usadas.
- Descargar la última versión del sitio.

# Ayuda en R

\*En general, para obtener ayuda hay dos formas

```
> help(mean)
> ?mean
```

# Dos formas de trabajar

- ① Línea de comando
  - ② Script (similar a una *syntax* de SPSS o un *.do file* de STATA)
- Son básicamente lo mismo. Solamente que en la primera se va ingresando cada comando de una vez.



# Objetos en R

- Básicamente... donde R guarda los datos.
- Hay muchos tipos de objetos en R.
- Muchos paquetes “generan” sus propios objetos
- No obstante, la mayoría de las tareas pueden resolverse con algunos pocos tipos de objetos
- En R hay cinco tipos atómicos de objetos
  - ▶ `character`
  - ▶ `numeric`
  - ▶ `integer`
  - ▶ `complex`
  - ▶ `logical`

# Objetos en R

- Cada objeto en R tiene atributos
  - ▶ nombres, nombres de dimensiones
  - ▶ dimensiones ( $R \times C$ )
  - ▶ clase
  - ▶ largo
  - ▶ etc.

En general se accede a los atributos de un objeto con la función `attributes()`

# Objetos en R: Vectores

- El objeto más básico es un vector
- Un vector solo puede constener objetos de la misma clase
- Excepción: listas, tipo especial de vector.
- Pueden crearse vectores vacíos con la función `vector()`

# Generado vectores

- Para ingresar datos se usa el signo <-
- Para comentar se usa #. Todo lo que está a la derecha de # es ignorado.

```
> x<-1  
> # PRUEBA  
> x  
## [1] 1
```

# Generado vectores

- Cuando una expresión completa se ingresa puede imprimirse o no el resultado dependiendo de la expresión

```
> x<-1      # NO SE IMPRIME NADA
> x         # SE IMPRIME AUTOMATICAMENTE EL CONTENIDO DE x
## [1] 1
> print(x)  # SE PIDE LA IMPRESION EXPLICITAMENTE
## [1] 1
```

# Generado vectores

- El operador `:` sirve para generar secuencias de enteros
- También puede usarse la función `seq()` para secuencias más complejas

```
> x<-1:10
> x
##      [1]  1  2  3  4  5  6  7  8  9 10
```

# Generado vectores

- La función `c()` sirve para crear vectores de objetos

```
> x<-c(1.2,2.3,3.8)      #numérico
> x<-c(TRUE, FALSE, TRUE) #lógico
> x<-c("a","b","c")      #character
> x<-9:20                 #integer
```

- También puede usarse la función `vector()`

```
> x<-vector("numeric",length=10)
> x
## [1] 0 0 0 0 0 0 0 0 0 0
```

# Mezclando vectores

- Cuando se mezclan diferentes objetos en un vector R hace uso de la “coerción”: se fuerza a los objetos a transformarse en una misma clase

```
> y<-c(1.2, "a")           #character
> x<-c(FALSE, 2)           #numérico
> x<-c(FALSE, "a")         #character
```



# Coerción explícita

- Puede aplicarse de forma explícita la coercion; se usan las funciones `as.[class]`
- Esta coercion puede aplicarse a otros objetos, no solo a vectores

```
> x<-0:6
> as.character(x)
## [1] "0" "1" "2" "3" "4" "5" "6"
> as.logical(x)
## [1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
## [1] "0" "1" "2" "3" "4" "5" "6"
```

# Objetos en R: Matrices

- Las matrices son vectores en con un atributo *dimensión*
- Las podemos considerar vectores de dos dimensiones (filas y columnas)
- Se construyen en el sentido de las columnas

```
> m<-matrix(1:6,ncol=2,nrow=3)
```

```
> m
```

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

# Objetos en R: Matrices

- Puedo crear matrices de muchas formas: una útil es concatenado por filas o columnas vectores ya existentes con las funciones `rbind()` y `cbind()`

```
> x<-1:3
> y<-4:6
> cbind(x,y)
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
> rbind(x,y)
##    [,1] [,2] [,3]
## x     1   2   3
## y     4   5   6
```

# Objetos en R: Listas

- Las listas son una clase especial de vector: pueden contener elementos de cualquier tipo en su interior.
- Son objetos muy importantes en R: la mayoría de los resultados de los modelos que se aplican en R devuelven como output un objeto que es una lista.
- Otro ejemplo: al importar datos de un GIS (.shp, .geojson, geodatabase, etc.) R los interpreta como una lista.

# Objetos en R: Listas

```
> x<-list(1,"a",TRUE, 4.6)
> x
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 4.6
```

# Objetos en R: Factors

- Se usan para representar datos categóricos (nominales u ordinales). Son algo así como un vector de integers con una etiqueta asociada (algo parecido a SPSS).
- Algunas funciones como `lm()` y `glm()` hacen un uso especial de los factores
- Suele ser mejor usar factors que integers porque son “autodescriptivos”: “Ocupado”, “Desocupado”, “Inactivo” es mejor que 1,2,3.

# Objetos en R: Factors

```
> x<-factor(c("Ocup","Ocup","Desoc","Ocup"  
+            ,"Inact"))  
> x  
## [1] Ocup  Ocup  Desoc Ocup  Inact  
## Levels: Desoc Inact Ocup
```

# Objetos en R: Factors

- Se puede generar factors ordenados con el argumento `levels`.

```
> x<-factor(c("A","A","M","B"  
+           , "B"),  
+           levels=c("B","M","A"))  
> x
```

```
## [1] A A M B B  
## Levels: B M A
```

```
> table(x)
```

```
## x  
## B M A  
## 2 1 2
```



# Objetos en R: Missing Values

- Son denotados NA o NaN
- `is.na()` se usa para testear si en un objeto hay NA's
- `is.nan()` se usa para testear si en un objeto hay NaN's
- NA tiene una clase: hay integer NA's, character NA's
- Todo NaN es un NA, lo inverso no es cierto

# Objetos en R: Missing Values

```
> x<-c(15,20,31,NA,12)
> is.na(x)
## [1] FALSE FALSE FALSE  TRUE FALSE
> is.nan(x)
## [1] FALSE FALSE FALSE FALSE FALSE
> x<-c(15,NaN,31,NA,12)
> is.na(x)
## [1] FALSE  TRUE FALSE  TRUE FALSE
> is.nan(x)
## [1] FALSE  TRUE FALSE FALSE FALSE
```

# Objetos en R: Data Frames

- Son lo más parecido a lo que entendemos en ciencias sociales por base de datos. Se usan para almacenar datos tabulares.
- Para R son un tipo especial de list, en la cual cada elemento de la lista tiene la misma longitud (`length()`)
- Cada elemento de la list puede ser pensado por una columna y el largo de cada elemento de la list es el nro. de files (`nrow()`)
- A diferencia de las matrix cada data frame puede almacenar elementos (columnas) de diferentes tipos.
- Tienen un atributo especial: `row.names`
- Cuando veamos impo y expo de datos, veremos que cuando uno llama a la función `read.csv()` o `read.table()` o similares lo que devuelven son un data frame
- Pueden ser transformados en matrices usando `data.matrix()`

# Objetos en R: Data Frames

```
> x<-data.frame(ident=1:3, tratamiento=c("D","B","B"))
> x
##      ident tratamiento
## 1      1           D
## 2      2           B
## 3      3           B
> nrow(x)
## [1] 3
> ncol(x)
## [1] 2
> row.names(x)
## [1] "1" "2" "3"
```

# Objetos en R: names

- Los diferentes objetos que vimos tienen nombres
- Vectores:

```
> x<-1:4  
> names(x)  
## NULL  
> names(x)<-c("a","p","c","x")
```

# Objetos en R: names

- Listas:

```
> m<-list(a=1, b="A",c=TRUE, d=15.3242342)
> names(m)
## [1] "a" "b" "c" "d"
```

# Objetos en R: names

- Data Frames:

```
> x<-data.frame(ident=1:4, tratamiento=c("D","B","A","D"))  
> names(x)  
## [1] "ident"          "tratamiento"
```

# Subsetting/Slicing en R

- Ahora, ¿cómo extraemos partes de los objetos? ¿Cómo aplicamos transformaciones a un subconjunto de un determinado objeto?
- A ambas operaciones se las llama *slicing* o *subsetting*, respectivamente
- Cada objeto tiene alguna forma de subsetting o slicing. Hay varios operadores
- `[]` devuelve siempre un objeto de la misma clase que el original
- `[[` se usa para extraer datos de un dataframe o una lista. No necesariamente devuelve siempre un objeto de la misma clase que el original
- `$` se usa para extraer elementos de un dataframe o una lista por nombre.



# Subsetting/Slicing en R: Vectores

- Orden

```
> x<-1:10
> x[1:4]
## [1] 1 2 3 4
> x[4:1]
## [1] 4 3 2 1
> x[c(1,5,8)]
## [1] 1 5 8
> x[-c(1,5,8)]
## [1] 2 3 4 6 7 9 10
```

# Subsetting/Slicing en R: Matrices

```
> x<-1:4
> y<-c("Z","Z","E","F")
> mat<-cbind(x,y)
> mat[1,]
##      x      y
## "1" "Z"
> mat[,1]
## [1] "1" "2" "3" "4"
> mat[,2]
## [1] "Z" "Z" "E" "F"
> mat[3,2]
##      y
## "E"
```

# Subsetting/Slicing en R: Data Frames

```
> x<-1:4
> y<-c("Z","Z","E","F")
> z<-c(TRUE,FALSE,FALSE,FALSE)
> df<-as.data.frame(cbind(x,y,z))
> df[,2]
## [1] Z Z E F
## Levels: E F Z
> df$x
## [1] 1 2 3 4
## Levels: 1 2 3 4
> df$z[1]
## [1] TRUE
## Levels: FALSE TRUE
```

# Subsetting/Slicing en R: Listas

```
> x<-list(foo = 1:4, bar = 0.6)
> x[1]
## $foo
## [1] 1 2 3 4
> x[[1]]
## [1] 1 2 3 4
> x$bar
## [1] 0.6
> x['bar']
## $bar
## [1] 0.6
> x[['bar']]
## [1] 0.6
```

# Subsetting/Slicing en R: Listas

```
> x<-list(foo = 1:4, bar = 0.6, baz = 'hola')
> x[c(1,3)]
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hola"
```

# Subsetting/Slicing en R: Listas

- El operador `[[ ]]` puede usarse con índices calculados. `$` solo puede usarse con nombres

```
> x<-list(foo = 1:4, bar = 0.6, baz = 'hola')
> name<-'bar' # Indice computado
> x[[name]]
## [1] 0.6
> x$name
## NULL
> x$foo
## [1] 1 2 3 4
```

# Subsetting/Slicing en R: Listas con elementos anidados

- El operador `[[ ]]` puede tomar cualquier secuencia de enteros

```
> x<-list(a = list(10,12,14), b = c(3.14, 2.81))
> x[[c(1,3)]]
## [1] 14
> x[[1]][[3]]
## [1] 14
> x[[c(2,1)]]
## [1] 3.14
```

## Subsetting/Slicing en R: Indexado Lógico

- En R es posible utilizar un vector de booleanos (TRUE-FALSE) como insumo para hacer subsetting/slicing. Tiene que tener la misma longitud que el objeto que vamos a subsetear
- La idea es que los miembros del objeto original que están en la “posición” TRUE del vector lógico son extraídos para el slice, mientras que los que están en la “posición” FALSE, no.

```
> x<-c("aa", "bb", "cc", "dd", "ee")  
> index<-c(FALSE, TRUE, FALSE, TRUE, FALSE)  
> x[index]  
## [1] "bb" "dd"
```



# Subsetting/Slicing en R: Indexado Lógico

- Esto se aplica a cualquier objeto. Es particularmente útil para trabajar con data frames, por ejemplo, para quedarnos con casos que cumplen determinada condición. Es similar a hacer un filtro en SPSS o STATA.

## Valores NA: eliminarlos

- Puede ser pensado como un caso de indexado lógico:
- Relativamente fácil: `is.na()` devuelve un vector de lógicos

```
> x<-c(1,2,NA,4,NA)
> no<-is.na(x)
> x1<-x[!no]
> x2<-x[!is.na(x)]
> x1
## [1] 1 2 4
> x2
## [1] 1 2 4
```

## Valores NA: eliminarlos

- Si hay varias columnas con NA's, entonces puede usarse `complete.cases()`

```
> x<-c(1,2,NA,4)
> y<-c(NA,"a",NA,"n")
> m<-cbind(x,y)
> m1<-m[complete.cases(m),]
> m1
##      x    y
## [1,] "2" "a"
## [2,] "4" "n"
```

# Importando Datos en R

- Varias funciones importantes:

- ▶ `read.table()` y `read.csv()` para leer datos tabulares
- ▶ `readLines()` para leer líneas en un archivo de texto
- ▶ `read.spss()` en el paquete *foreign* que unifica muchas las funciones de importación exportación

# Importando Datos en R: `read.table()`

- De las más usadas. El formato es similar al resto
  - ▶ `file`: string con el nombre y la ruta del archivo a importar
  - ▶ `header`: logical indicado si el archivo tiene cabeceras
  - ▶ `sep`: string que indica cómo se separan las columnas
  - ▶ `colClasses`: character vector que indica la clase de cada columna (“character”, “integer”, etc.)
  - ▶ `nrows`: cantidad de filas en el dataset
  - ▶ `skip`: cantidad de filas que hay que saltar contando desde el principio en el dataset
  - ▶ `stringAsFactors`: logical que indica si al importar el archivo deben codificarse los strings como factors

# Importando Datos en R: `read.table()`

- Si el archivo no es muy grande puede usarse directamente:

```
> data<-read.table("prueba.txt")
```

- R va a realizar algunas tareas automáticamente:
  - ▶ saltar las líneas que empiezan con `#`
  - ▶ calcular cuántas filas hay y cuánta memoria necesita reservar
  - ▶ identificar la clase de cada columna (si uno lo define con los argumentos resulta más eficiente)
  - ▶ `read.csv()` es idéntico a `read.table()` con la diferencia de que el separador es la coma.

# Importando Datos “grandes” en R: `read.table()`

- Si se enfrentan con un dataset grandes pueden
  - ▶ consultar la documentación de la función
  - ▶ Setear el argumento `colClasses`: suele ser mucho más rápido dado que R no tiene que realizar la tarea de detectar los tipos de cada columna.
  - ▶ Setar `nrows` esto ayuda a usar menos memoria... no a que sea necesariamente más rápido.

```
> init<-read.table("prueba.txt",nrows=100)
> classes<-sapply(init,class)
> todos<-read.table("prueba.txt"
>                   ,colClasses = classes)
```

# Importando Datos “grandes” en R: `read.table()`

- Además de todo esto es importante hacer una estimación gruesa de la memoria que va a requerir cargar el dataset.
- Si la memoria requerida es más de la disponible en el sistema...
- Cálculo grueso: supongamos un dataframe con 2.000.000 de filas y 200 columnas todos datos numéricos
  - ▶  $2.000.000 \times 200 \times 8$  bytes/numerics
  - ▶ 2.400.000.000 bytes
  - ▶  $2.400.000.000 \text{ bytes} / 2^{20} \text{ bytes/MB}$
  - ▶ 2.288,8 MB
  - ▶ 2.24 GB



# Funciones en R

- Las funciones se crean con el argumento `function()` y se almacenan como un objeto en R.

```
> fun<-function(argumentos){  
>   #[se hace algún cómputo]  
> }
```

- Las funciones pueden pasarse como argumento a otras funciones
- Pueden anidarse: puede definirse una función dentro de una función
- El valor que devuelve una función es la última expresión en el cuerpo de la función

# Funciones en R

- Las funciones tienen un input (argumentos) y devuelven un output (valor)
  - ▶ Argumentos formales: los que están incluidos en la definición de la función
  - ▶ La función `formal()` devuelve los argumentos formales de una función
  - ▶ No todas las funciones en R usan argumentos formales
  - ▶ Los argumentos pueden estar “missing” o tener valores asignados por defecto

# Funciones en R: matching de argumentos

- Los argumentos pueden ser matcheados por nombre o por posición. Las siguientes son expresiones equivalentes:

```
> data<-rnorm(100)
> sd(data)
## [1] 0.9708136
> sd(x=data)
## [1] 0.9708136
> sd(x=data,na.rm=TRUE)
## [1] 0.9708136
> sd(na.rm=TRUE,x=data)
## [1] 0.9708136
> sd(na.rm=TRUE, data)
## [1] 0.9708136
```

## Funciones en R: matching de argumentos

- Si bien no es recomendable, se pueden “mezclar” matching de argumentos por nombre y por posición. Cuando un argumento se llama por nombre se lo saca de la lista y el resto de los argumentos se llaman por la posición.

```
> args(lm)
## function (formula, data, subset, weights, na.action, method,
##      model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##      contrasts = NULL, offset, ...)
## NULL
```

- Por ende, las siguientes expresiones son equivalentes

```
> lm(data=datos, y~x, model=FALSE, 1:100)
> lm(y~x, data=datos, 1:100, model=FALSE)
```

# Funciones en R: definiendo una función

```
> squared<-function(x,b,y=NULL,...){  
+   return(x**2)  
+ }  
> squared(2)  
## [1] 4
```

- Pueden usarse argumentos que son seteados por defecto en NULL
- La evaluación de los argumentos es “lazy”, es decir que solamente se evalúan los argumentos a medida que son necesarios.
  - ▶ En la función anterior el argumento b no se evalúa, entonces llamando squared(2) no arroja error porque el 2 se matchea en función a la posición

# Funciones en R: definiendo una función

```
> prt<-function(a,b){  
>   print(a)  
>   print(b)  
> }  
> prt(2)
```

- Aquí si arroja un error: b debe ser evaluado luego de print(a)

# Funciones en R: definiendo una función

- Puede agregarse un argumento "..."
- Se usa para pasar a la función argumentos de otras funciones y no tener que copiar toda la lista de argumentos
- También es útil cuando el número de argumentos de la función no puede ser conocido de antemano.
- Un punto a tener en cuenta: todos los argumentos luego del "... " deben ser nombrados explícitamente y no pueden ser matcheados parcialmente.