Universidad Carlos III de Madrid

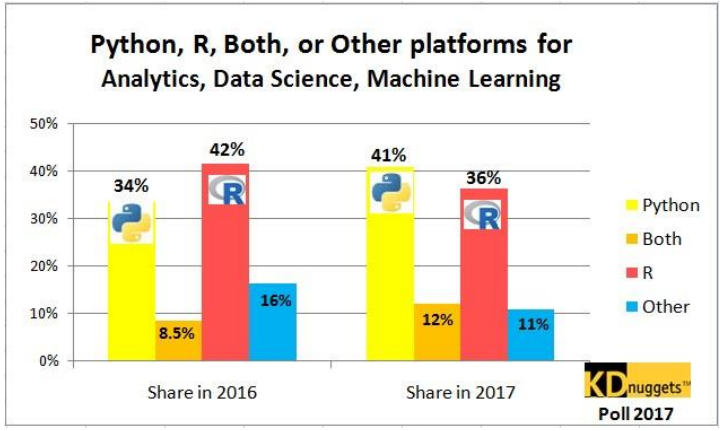
REPASO DE R

uc3m | Universidad Carlos III de Madrid

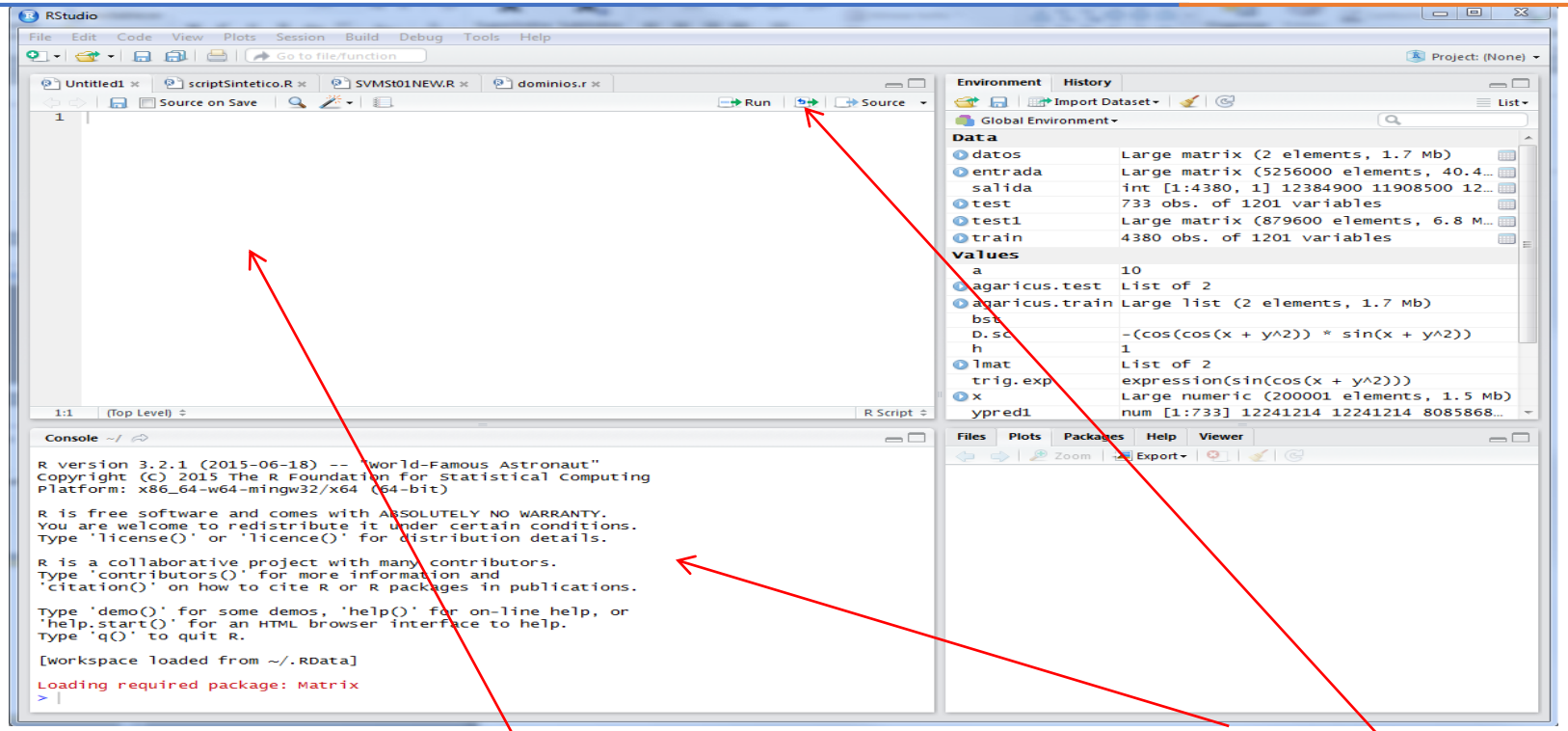
OPENCOURSEWARE
APRENDIZAJE AUTOMÁTICO PARA EL ANÁLISIS DE DATOS
GRADO EN ESTADÍSTICA Y EMPRESA
Ricardo Aler

Repaso de R

- R es un lenguaje de programación (como C, C++, Java, Python, …)
- Pero especializado en análisis de datos (estadística, aprendizaje automático, ciencia de datos, minería de datos, …)
- Es uno de los lenguajes más utilizados en análisis de datos. Python es su más cercano competidor (Python destaca sobre todo en Deep Learning).



Introducción. R Studio



Nota: en Rstudio podemos trabajar en la **consola**, o escribiendo en el **editor** y mandando los comandos a la consola, marcando el código a ejecutar y **pinchando run** (o pulsando ctrl-enter). Intentaremos trabajar en el editor, para que nos quede un registro de lo que vamos haciendo.

Tipos “simples” (escalares) manejados por R

- **character**:

  b <- "cadena de caracteres"

  b <- ‘cadena de caracteres’

  class(b)

  [1] "character"

- **logical**:

  q <- TRUE

  q <- FALSE

  class(q)

  [1] "logical"

- **numeric**:

  x <- 3

  class(x)

  [1] "numeric"

- **integer**

  n <- 8L

  class(n)

  [1] "integer"

- **complex**:

  z <- -2 + 0i

  class(z)

  [1] "complex"

- Aparte, están los valores *NA* (missing values), que son de cualquier tipo, e *inf* / *NaN* (infinito / Not a Number) que son de tipo numérico.

Operadores aritméticos y lógicos

Aritméticos

Se aplican a valores numéricos, dan como resultado un valor numérico

+	suma
-	resta
*	multiplicación
/	división
^	potenciación
%/%	división entera
%%	módulo

De comparación

Comparan valores numéricos dando como resultado un valor lógico

>	mayor que
<	menor que
>=	mayor o igual
<=	menor o igual
==	igual que
!=	distinto que

Lógicos

Se aplican a valores lógicos dando como resultado un valor lógico

&	Y
|	O
!	NO

Tipos de datos “compuestos” manejados por R

Los objetos que normalmente se utilizan en R son estructuras o agrupaciones de datos

- **Vectores**: colección de datos del mismo tipo. Los escalares son realmente vectores con un solo elemento
  - **Factores**: vectores que representan variables categóricas
- **Matrices**: todos los elementos del mismo tipo, se accede a los elementos por fila y columna
- **Listas**: son listas de elementos de tipos y longitudes distintas.
- **Data frames**: son **listas** de vectores con la misma longitud. Cada columna puede tener tipos distintos. Importante: se parecen a las matrices, pero realmente son **listas**.

Vectores

- Vectores: colección de datos del **mismo tipo**.

```
> x <- c(1,3,5,7,10)
> x
[1]  1  3  5  7 10
> length(x)
[1] 5
```

- Ojo, tienen que ser valores del **mismo tipo:**

```
> y <- c(1, "dos", 3)
> y
[1] "1"   "dos" "3"
```

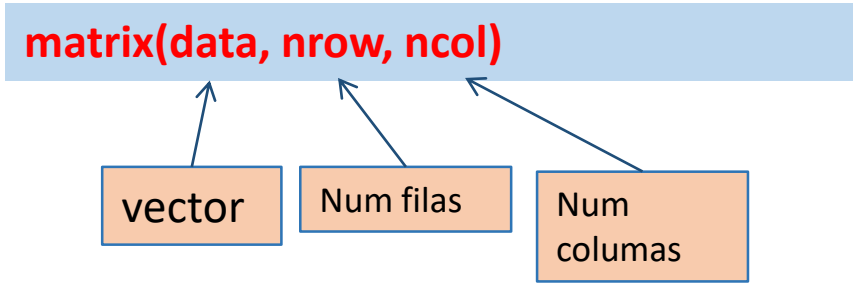
Tipos de datos “compuestos” manejados por R

Los objetos que normalmente se utilizan en R son estructuras o agrupaciones de datos

- **Vectores**: colección de datos del mismo tipo. Los escalares son realmente vectores con un solo elemento
- **Matrices**: todos los elementos del mismo tipo, se accede a los elementos por fila y columna
- **Listas**: son listas de elementos de tipos distintos.
- **Data frames**: son **listas** de vectores columna. Cada columna puede tener tipos distintos. Importante: se parecen a las matrices, pero realmente son **listas**.

Matrices

- Son como los vectores pero en dos dimensiones
- Se pueden crear con **matrix()**




```
> v<-c(4,8,2,3,89,5,4,23,4,6,1,2,3,6,1)
> m<-matrix(v,3,5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    3    4    6    3
[2,]    8   89   23    1    6
[3,]    2    5    4    2    1
```

Tipos de datos “compuestos” manejados por R

Los objetos que normalmente se utilizan en R son estructuras o agrupaciones de datos

- **Vectores**: colección de datos del mismo tipo. Los escalares son realmente vectores con un solo elemento
- **Matrices:** todos los elementos del mismo tipo, se accede a los elementos por fila y columna
- Listas: son listas de elementos de tipos distintos.
- **Data frames:** son listas de vectores columna. Cada columna puede tener tipos distintos. Importante: se parecen a las matrices, pero realmente son listas.

Listas

- Pueden contener distintos tipos de datos (números, cadenas, …)
- Los campos pueden tener nombre
- Por ejemplo, la función de R que crea modelos de regresión lineal, devuelve una lista

Ejemplo de lista

```
> linearMod <- lm(dist ~ speed, data=cars)
> str(linearMod)
List of 12
$ coefficients : Named num [1:2] -17.58 3.93
$ residuals    : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
$ effects      : Named num [1:50] -303.914 145.552 -8.115 9.885 0.194 ...
$ rank         : int 2
$ fitted.values: Named num [1:50] -1.85 -1.85 9.95 9.95 13.88 ...
$ assign       : int [1:2] 0 1
$ qr           :List of 5
$ df.residual  : int 48
$ xlevels      : Named list()
$ call         : language lm(formula = dist ~ speed, data = cars)
$ terms        :Classes 'terms', 'formula'  language dist ~ speed
$ model        :'data.frame':                  50 obs. of  2 variables:
```

Acceso a los componentes de una lista

Por nombre

```
> linearMod$coefficients
(Intercept)      speed
 -17.579095    3.932409
```

Por índice

```
> linearMod[[1]]
(Intercept)      speed
 -17.579095    3.932409
```

FUNCIONES DE ALTO NIVEL (HIGHER ORDER FUNCTIONS)

- Se llaman funciones de alto nivel porque son funciones que usan otras funciones
- En concreto, aplican una función a cada elemento de un vector o lista. Devuelven un vector o lista. Evitan bucles
- Las principales son:
  - lapply, sapply,
  - mapply
- También está apply (que es para matrices, no para listas), tapply y split.

FUNCIONES DE ALTO NIVEL: LAPPLY

- lapply, aplica una función a una lista y devuelve una lista

```
> 6
> x = list(a=1:2, b=1:3, c=1:4, d=1:5)
> x
$a
[1] 1 2

$b
[1] 1 2 3

$c
[1] 1 2 3 4

$d
[1] 1 2 3 4 5

> lapply(x,mean)
$a
[1] 1.5

$b
[1] 2

$c
[1] 2.5

$d
[1] 3

> |
```

FUNCIONES DE ALTO NIVEL: SAPPLY

- sapply: es como lapply, pero intenta convertir el resultado a un vector (si todos los elementos de la lista a la salida tienen longitud 1 y el mismo tipo)

```
> x = list(a=1:2, b=1:3, c=1:4, d=1:5)
> x
$a
[1] 1 2

$b
[1] 1 2 3

$c
[1] 1 2 3 4

$d
[1] 1 2 3 4 5

> lapply(x,mean)
$a
[1] 1.5

$b
[1] 2

$c
[1] 2.5

$d
[1] 3

> sapply(x,mean)
  a   b   c   d
1.5 2.0 2.5 3.0
>
```

FUNCIONES DE ALTO NIVEL: SAPPLY

- El usuario puede definir sus propias funciones. Por ejemplo, esta devuelve el último elemento de cada vector en la lista

```
ultimoElemento <-  function(mivector){
  return( mivector[ length(mivector) ] )
}
```

```
x
$a
[1] 1 2
$b
[1] 1 2 3
$c
[1] 1 2 3 4
$d
[1] 1 2 3 4 5
```

```
> sapply(x, ultimoElemento)
a b c d
2 3 4 5
```

- Aunque también se puede definir la función sobre la marcha (funciones anónimas o sin nombre):

```
> sapply(x, function(mivector) mivector[ length(mivector) ])
a b c d
2 3 4 5
```

LISTAS , DATAFRAMES Y FUNCIONES DE ALTO NIVEL

- Es conveniente conocer qué son las listas y cómo usarlas, porque muchas de las variables definidas en MLR son listas.
- Pero otra importancia de las listas es que los dataframes son realmente **listas de vectores** (todos ellos con la misma longitud)
- En ocasiones queremos aplicar la misma operación (función) a cada una de las columnas del dataframe: podemos utilizar lapply (o sapply)

Tipos de datos “compuestos” manejados por R

Los objetos que normalmente se utilizan en R son estructuras o agrupaciones de datos

- **Vectores**: colección de datos del mismo tipo. Los escalares son realmente vectores con un solo elemento
  - **Factores**: vectores que representan variables categóricas
- **Matrices**: todos los elementos del mismo tipo, se accede a los elementos por fila y columna
- **Listas**: son listas de elementos de tipos y longitudes distintas.
- **Data frames**: son **listas** de vectores columna, todos ellos de la misma longitud. Cada columna puede tener tipos distintos. Importante: se parecen a las matrices, pero realmente son **listas**.

Data frames

- Creando un data frame (se hace por columnas)

```
mis.datos <-data.frame(nombre=c("luis","juan","antonio"),
            edad=c(25,30,29),  talla=c(1.75, 1.70, 1.83))
```

```
> mis.datos
   nombre edad talla
1    luis   25  1.75
2    juan   30  1.70
3 antonio   29  1.83
```

- Es la estructura adecuada para almacenar tablas de datos, porque permiten combinar en una matriz distintos tipos de datos (números, cadenas, …)
- Recordar que las matrices sólo pueden almacenar el mismo tipo de datos
- Aunque matrices y data.frames se parecen mucho, realmente son estructuras de datos muy distintas. Los data.frames no son matrices, sino **listas de vectores**.

Data frames

- Ejemplo: airquality (medidas de la calidad del aire en Nueva York)

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Data frames. Ejercicio

- Hallar la media de las temperaturas de airquality en Mayo, excluyendo los NA de la media

```
> temp.mayo <- airquality[airquality$Month==5,"Temp"]
> mean(temp.mayo)
[1] 65.54839
```

Data frames. Solución

- Hallar la media de las temperaturas de airquality en Mayo (mes 5), excluyendo los NA de la media.
- Tres posibles soluciones:

```
> mean(airquality[airquality$Month==5,"Ozone"], na.rm = TRUE)
[1] 23.61538
> mean(airquality[airquality$Month==5,]$Ozone, na.rm = TRUE)
[1] 23.61538
> mean(airquality$Ozone[airquality$Month==5], na.rm = TRUE)
[1] 23.61538
> with(airquality, mean(Ozone[Month==5], na.rm = TRUE))
[1] 23.61538
>
```

Los data.frames son listas de columnas (no matrices)

- Aunque matrices y dataframes se parecen mucho, realmente son estructuras de datos muy distintas. Los dataframes no son matrices, sino **listas de vectores**.
- Por ejemplo, el data.frame *cars* es una lista de dos columnas (vectores). Esto es importante, porque a los data.frames se les pueden aplicar **funciones de alto nivel**, como vamos a ver.

```
> str(cars)
'data.frame':  50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

Los data.frames son listas de columnas (no matrices)

- Al ser los data.frames listas de columnas, podemos usar *lapply* o *sapply* para aplicar la misma operación a cada una de las columnas. Ejemplos:
  - Calcular la media y/o la desviación típica de cada columna
  - Calcular el número de NA's en cada columna
  - …
- También podemos usar *lapply* para **transformar** cada columna del data.frame. Ejemplos:
  - Normalizar o estandarizar columnas
  - Imputar columnas (substituir NA's por la media, por ejemplo)
  - …

LISTAS , DATAFRAMES Y FUNCIONES DE ALTO NIVEL

- Supongamos que queremos calcular la media de cada una de las columnas del dataframe airquality

```
> airquality[1:10,]
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67     5   1
2     36     118  8.0   72     5   2
3     12     149 12.6   74     5   3
4     18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
7     23     299  8.6   65     5   7
8     19      99 13.8   59     5   8
9      8      19 20.1   61     5   9
10    NA     194  8.6   69     5  10
>
>
> sapply(airquality, mean)
    Ozone   Solar.R      Wind      Temp     Month       Day
       NA        NA  9.957516 77.882353  6.993464 15.803922
>
> sapply(airquality, mean, na.rm=TRUE)
     Ozone    Solar.R       Wind       Temp      Month        Day
 42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
> |
```

- Nota: en el caso de la media de las columnas de un data.frame también podemos usar *colMeans*()

```
> colMeans(airquality, na.rm=TRUE)
    Ozone   Solar.R      Wind      Temp     Month       Day 
42.129310 185.931507  9.957516 77.882353  6.993464 15.803922 
```

- Aunque curiosamente, en R base no existe *colSd*() para calcular las desviaciones típicas, con lo que habría que usar *sapply* (o bien la librería *matrixStats*)

LISTAS , DATAFRAMES Y FUNCIONES DE ALTO NIVEL

- Supongamos que queremos calcular la media **y** la desviación de cada una de las columnas del dataframe airquality
- Una posibilidad sería llamar dos veces a *sapply*, una para la media y otra para la desviación. Otra posibilidad es usar una función que devuelva *mean* y *sd*.

```
> airquality[1:10,]
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67     5   1
2     36     118  8.0   72     5   2
3     12     149 12.6   74     5   3
4     18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
7     23     299  8.6   65     5   7
8     19      99 13.8   59     5   8
9      8      19 20.1   61     5   9
10    NA     194  8.6   69     5  10
>
> m_sd <- function(x) {
    return(c(mean(x, na.rm=TRUE), sd(x, na.rm=TRUE)))
  }
>
> sapply(airquality, m_sd)
          Ozone   Solar.R     Wind     Temp    Month      Day
[1,] 42.12931 185.93151 9.957516 77.88235 6.993464 15.80392
[2,] 32.98788  90.05842 3.523001  9.46527 1.416522  8.86452
>
```

EJERCICIOS:

- Usar *sapply* que devuelva una matriz (como en la transparencia anterior) con el valor mínimo y máximo de cada columna.
- Usar *sapply* que devuelva el número de NA's de cada columna. Nota, si x es un vector, podemos saber el número de NA's en ese vector así: *sum(is.na(x)).*

LISTAS , DATAFRAMES Y FUNCIONES DE ALTO NIVEL

- Sea el data.frame CO2 donde hay columnas de distintos tipos: factores y números.

```
> head(CO2)
  Plant   Type  Treatment conc uptake
1   Qn1 Quebec nonchilled   95   16.0
2   Qn1 Quebec nonchilled  175   30.4
3   Qn1 Quebec nonchilled  250   34.8
4   Qn1 Quebec nonchilled  350   37.2
5   Qn1 Quebec nonchilled  500   35.3
6   Qn1 Quebec nonchilled  675   39.2
```

- Si calculamos la media de las columnas con *lapply(CO2, mean)* va a dar error (también con *colMeans(CO2)*), porque *mean* no funciona con factores o characters.
- Vamos a hacer una función que, si la columna es numérica, calcule su media, y si es no-numérica, calcule su moda

Media si número, moda si no-número

```
medializa <- function(x) {
  if(is.numeric(x)){
    mean(x)
  } else {
    pracma::Mode(x)
  }
}
```

```
> lapply(CO2, medializa)
$`Plant`
[1] "Qn1"

$Type
[1] "Quebec"

$Treatment
[1] "nonchilled"

$conc
[1] 435

$uptake
[1] 27.2131
```

TRANSFORMACIÓN DE DATAFRAMES

- En ocasiones es útil transformar cada una de las columnas de un data.frame, aplicándoles la misma operación.
- Por ejemplo, supongamos que queremos dividir cada valor de una columna por la media de esa columna.
- Nótese que lapply devuelve una **lista**, la cual hay que **convertirla a dataframe** al final

```
> aq_procesado <- as.data.frame( lapply( airquality, function(x) x / mean(x, na.rm=TRUE)))
>
> head(aq_procesado)
      Ozone   Solar.R      Wind      Temp     Month       Day
1 0.9731942 1.0218817 0.7431572 0.8602719 0.7149533 0.06327543
2 0.8545120 0.6346423 0.8034132 0.9244713 0.7149533 0.12655087
3 0.2848373 0.8013704 1.2653758 0.9501511 0.7149533 0.18982630
4 0.4272560 1.6834156 1.1549065 0.7960725 0.7149533 0.25310174
5        NA        NA 1.4361011 0.7190332 0.7149533 0.31637717
6 0.6646204        NA 1.4963571 0.8474320 0.7149533 0.37965261
>
```

TRANSFORMACIÓN DE DATAFRAMES: IMPUTACIÓN

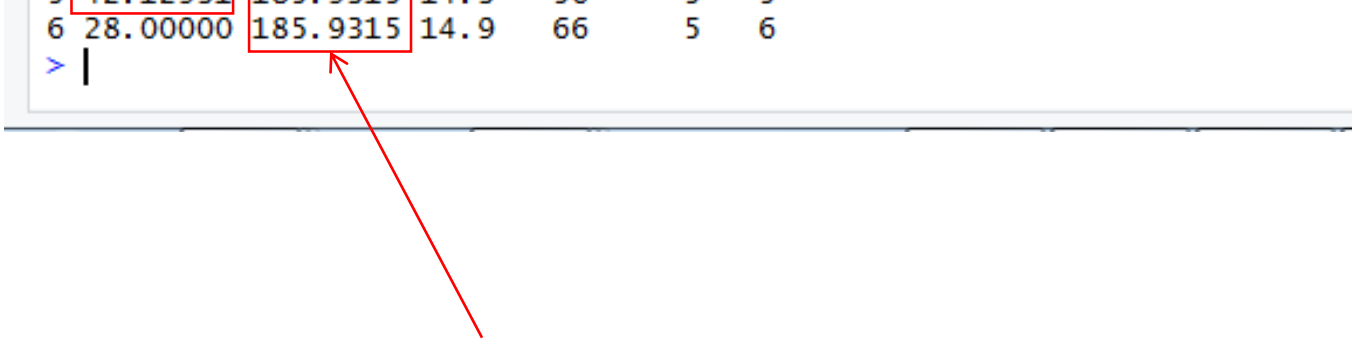
- Una transformación muy común en aprendizaje automático y estadística es la imputación
- La imputación consiste en convertir los valores faltantes (missing values) o NAs en algún valor numérico. Lo más sencillo que se puede hacer es cambiar los NA's por el valor medio de la columna
- Recordemos que una columna de un data.frame es un vector. Para cambiar algunos valores de un vector que cumplan cierta condición, se puede usar la función **ifelse**. Por ejemplo, el siguiente código transforma los NA's por cero.

```
> mivector <- c(1,2,3, NA, 4, 5, 6, NA)
>
> mivector
[1]  1  2  3 NA  4  5  6 NA
>
> mivector_sin_NAs <- ifelse(is.na(mivector), 0, mivector)
>
> mivector_sin_NAs
[1] 1 2 3 0 4 5 6 0
>
```

TRANSFORMACIÓN DE DATAFRAMES: IMPUTACIÓN

- Ahora, en lugar de transformar los NA's en cero, los transformaremos en la media de la columna, y lo aplicaremos ya a todo el data.frame airquality.

```
> imputacion <- function(x) {
+   ifelse(is.na(x), mean(x, na.rm=TRUE), x)
+ }
>
> aq_imputado <- as.data.frame( lapply( airquality, imputacion))
>
> head(aq_imputado)
      Ozone  Solar.R Wind Temp Month Day
1 41.00000 190.0000  7.4   67     5   1
2 36.00000 118.0000  8.0   72     5   2
3 12.00000 149.0000 12.6   74     5   3
4 18.00000 313.0000 11.5   62     5   4
5 42.12931 185.9315 14.3   56     5   5
6 28.00000 185.9315 14.9   66     5   6
>
```



EJERCICIO:

- Vamos a hacer ahora una transformación similar a la imputación, usando también **ifelse**.
- Vamos a substituir los valores “pequeños” de cada columna por cero. Los valores pequeños serán aquellos que sean sólo un 10% del valor máximo de la columna.

TRANSFORMACIÓN DE DATAFRAMES: NORMALIZACIÓN

- Una de las transformaciones más comunes de dataframes en aprendizaje automático es la **normalización**.
- La normalización consiste en hacer que todas las columnas tengan un rango de valores entre cero y uno. Es bueno para los algoritmos de AA que todos los atributos tengan el mismo rango (peso, altura, …)
- Es decir: col_normalizada = (col – min(col)) / (max(col) – min(col)))
- También se podría estandarizar: col_std = (col – mean(col)) / sd(col)

Sin normalizar

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
>
```

Normalizado

```
> head(aq_normalizado)
      Ozone   Solar.R      Wind      Temp Month        Day
1 0.23952096 0.5596330 0.3000000 0.2682927     0 0.00000000
2 0.20958084 0.3394495 0.3315789 0.3902439     0 0.03333333
3 0.06586826 0.4342508 0.5736842 0.4390244     0 0.06666667
4 0.10179641 0.9357798 0.5157895 0.1463415     0 0.10000000
5         NA        NA 0.6631579 0.0000000     0 0.13333333
6 0.16167665        NA 0.6947368 0.2439024     0 0.16666667
>
```

EJERCICIO:

- Usar *lapply* y *as.data.frame*, como en la antepenúltima transparencia, para normalizar las columnas de airquality.
- ¿Qué ocurriría si una columna es constante? La diferencia entre máximo y mínimo sería cero y el cociente daría infinito. Escribir una solución mejor que la anterior, para normalizar las columnas del data.frame, pero no transformando aquellas columnas constantes. Para esto tendréis que utilizar la función **if**, como hicimos hace algunas transparencias. Podéis comprobar si vuestro programa funciona con este data.frame
  - aq_chungo <- cbind(airquality, col_chunga = 3)

```
> aq_chungo <- cbind(airquality, col_chunga = 3)
> head(aq_chungo)
  Ozone Solar.R Wind Temp Month Day col_chunga
1    41     190  7.4   67     5   1          3
2    36     118  8.0   72     5   2          3
3    12     149 12.6   74     5   3          3
4    18     313 11.5   62     5   4          3
5    NA      NA 14.3   56     5   5          3
6    28      NA 14.9   66     5   6          3
>
```

EJERCICIO:

- ¿Qué ocurriría si una columna es constante? La diferencia entre máximo y mínimo sería cero y el cociente daría infinito. Escribir una solución mejor que la anterior, para normalizar las columnas del data.frame, pero no transformando aquellas columnas constantes. Para esto tendréis que utilizar la función **if**, como hicimos hace algunas transparencias. Podéis comprobar si vuestro programa funciona con este data.frame
  - aq_chungo <- cbind(airquality, col_chunga = 3)

```
> aq_chungo <- cbind(airquality, col_chunga = 3)
> head(aq_chungo)
  Ozone Solar.R Wind Temp Month Day col_chunga
1    41     190  7.4   67     5   1          3
2    36     118  8.0   72     5   2          3
3    12     149 12.6   74     5   3          3
4    18     313 11.5   62     5   4          3
5    NA      NA 14.3   56     5   5          3
6    28      NA 14.9   66     5   6          3
>
```

```
normalizaBis <- function(x){
  if(sd(x, na.rm=TRUE) == 0){
    rep(1, length(x))
  } else {
    (x-min(x, na.rm=TRUE)) / (max(x, na.rm=TRUE)-min(x, na.rm=TRUE))
  }
}

midataframe <- cbind(airquality, constante=3)

res <- as.data.frame(lapply(midataframe, normalizaBis))
print(head(res))
```

EJERCICIO:

- ¿Qué ocurriría si una columna no es numérica? Escribid una solución mejor que la anterior de tal manera que sólo se normalicen columnas numéricas que no sean constantes. Habrá que usar **if**, **is.numeric,** y tal vez && o ||
- Podéis probar vuestra solución con este otro data.frame, el cual contiene una columna con cadenas de caracteres.

```
> aq_mezclado <- cbind(airquality, col_chunga = 3, col_string = "A")
> head(aq_mezclado)
  Ozone Solar.R Wind Temp Month Day col_chunga col_string
1    41     190  7.4   67     5   1          3          A
2    36     118  8.0   72     5   2          3          A
3    12     149 12.6   74     5   3          3          A
4    18     313 11.5   62     5   4          3          A
5    NA      NA 14.3   56     5   5          3          A
6    28      NA 14.9   66     5   6          3          A
>
```

```
normalizaTris <- function(x){
  if(is.numeric(x)){
    if(sd(x, na.rm=TRUE) == 0){
      rep(1, length(x))
    } else {
     (x-min(x, na.rm=TRUE)) / (max(x, na.rm=TRUE)-min(x, na.rm=TRUE))
    }
  } else {
    return(x)
  }
}

midataframe <- cbind(airquality, constante=4, cadena="pepe")

res <- as.data.frame(lapply(midataframe, normalizaTris))
print(head(res))
```