Introduction to R

Felipe José Bravo Márquez

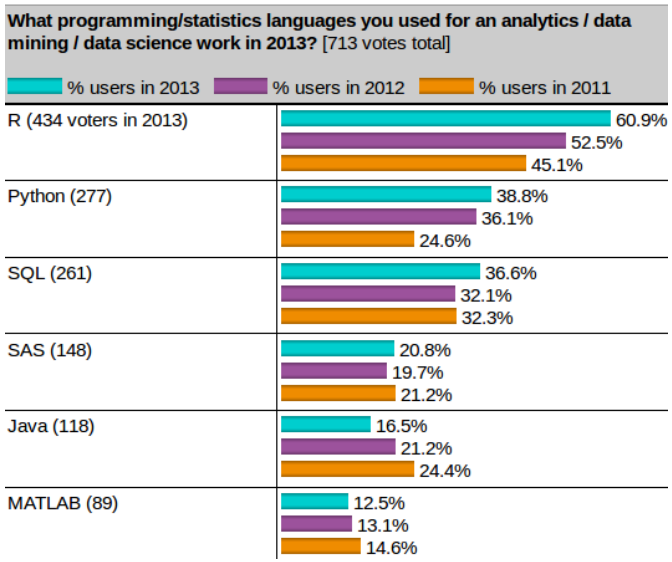
March 22, 2021

The R project for computational statistics

- R is a free statistical programming environment: `http://www.r-project.org/`
- It allows to manipulate and store data in an effective way.
- R is a complete programming language: variables, loops, conditions, functions.
- It provides many libraries that implement pretty much any statistical method.
- These libraries together with their dependencies are organized in a repository called **CRAN**: `http://cran.r-project.org/`

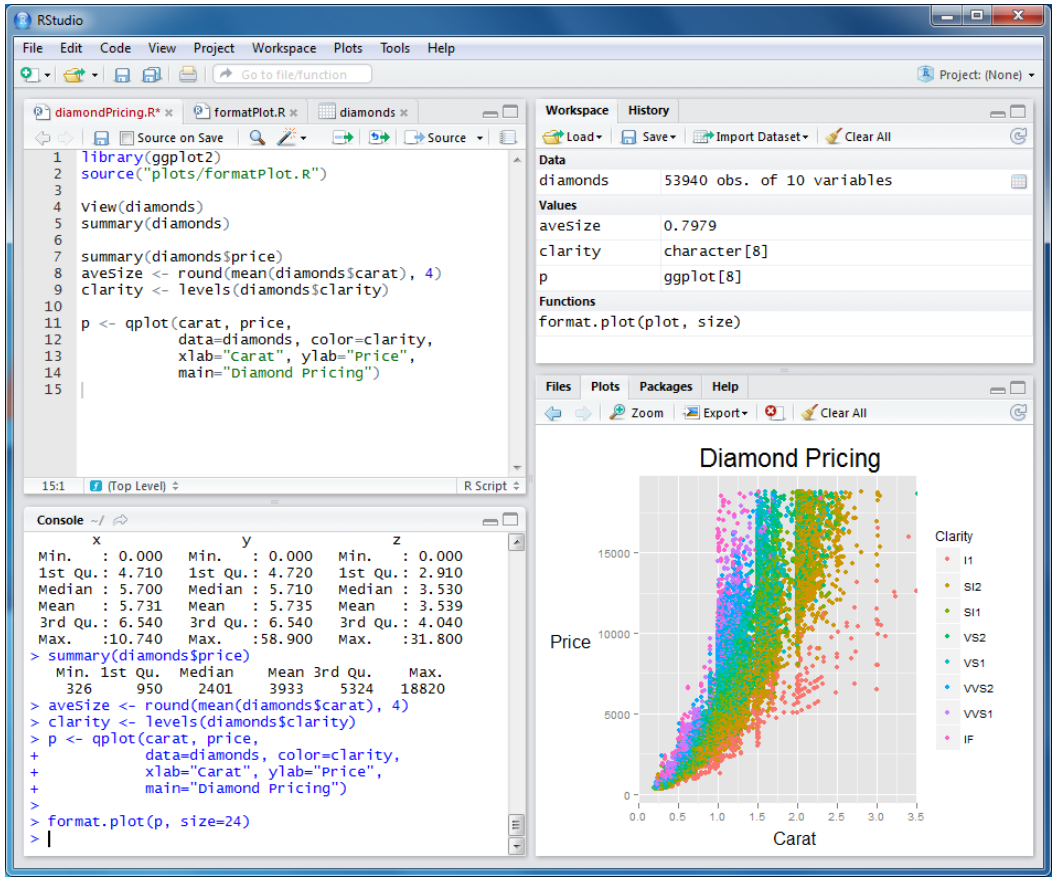
Why use R?

- R is free software unlike Matlab, SPSS, STATA.
- It is available for many operating systems: Windows, MAC OS X, Linux.
- As of 2013 **KDnuggets** survey showed R as the preferred programming language for performing data analysis, data mining and data science.
- `http://www.kdnuggets.com/2013/08/languages-for-analytics-data-mining-data-science.html`
- Today Python has become a very strong competitor.



RStudio

- R works through the command line.
- To work in a more friendly environment we will use RStudio.
- It is also free and can be downloaded for different operating systems at this link:
  `http://www.rstudio.com/ide/download/desktop`



R as a calculator

```
> 4*5
[1] 20
> 2^3
[1] 8
> exp(-5)
[1] 0.006737947
> log(4)
[1] 1.386294
```

Declaring Variables

- Variables can be assigned using `<-` , `=` or the function `assign`.

```
a<-1
b=3
assign("three",3)
d<-a+b
ver<-T # equivalent to TRUE
word<-"hello"
```

- By convention we use the first form (`<-`).
- Variables can be of class **numeric**, **factor**, **character**, **logical**, among others.
- To see the type of a variable we use the command `class`.

```
  > class(a)
[1] "numeric"
> class(ver)
[1] "logical"
> class(word)
[1] "character"
```

Functions

- Functions are declared as variables and are created with the expression **function**:

```
suma<-function(a=2,b=1){
  a+b;
}

fac<-function(n){
  ifelse(n==1,return(1),return(n*fac(n-1)))
}
```

- Function parameters can be declared with a specific value to be used as default values when we do not provide values for those parameters:

```
> suma(3,4)
[1] 7
> suma()
[1] 3
```

- The functions are of the type **function**:

```
> class(suma)
[1] "function"
```

Help and Workspace

- To read documentation about a function we use either **help** or **?**:

```
help(ls)
?ls
#for a particular command
help("for")
```

- All variables are stored in my **workspace** environment. To list them we use the command **objects** or **ls**. To delete a variable we use **rm**:

```
objects()
ls()
rm(a)
#Para borrarlos todos
rm(list=ls())
```

- I can save all my workspace variables in a file and then retrieve my work in a future session:

```
save.image("myworkspace.RData")
#Luego lo cargamos
load("myworkspace.RData")
```

Vectors

- To work with collections of elements we declare **vectors** which are constructed with the command **c**:

```
edades<-c(21,33,12,34,23,70,90,80,7,29,14,2,
          88,11,55,24,13,11,56,28,33)
```

- To get the length of a vector we use the command **length**, then to get the sum of all elements we use **sum**:

```
> suma<-sum(edades)
> largo<-length(edades)
> suma
[1] 734
> largo
[1] 21
```

- If we operate a vector by a scalar this value is recycled for all elements of the vector:

```
> numeros<-c(1,2,3)
> numeros+3
[1] 4 5 6
> numeros*5
[1]  5 10 15
```

Vectors (2)

- Calculate the mean and variance of the vector ages using the commands **sum** and **length** based on the following equations:

$$\text{mean(edades)} = \frac{\sum_{i=1}^{n} \text{edades}_i}{n} \tag{1}$$

$$\text{variance(edades)} = \frac{\sum_{i=1}^{n} (\text{edades}_i - \text{media(edades)})^2}{n-1} \tag{2}$$

Vectors (3)

- Answer:

```
> media<-sum(edades)/length(edades)
> media
[1] 34.95238
> varianza<-sum((edades-media)^2)/(length(edades)-1)
> varianza
[1] 747.9476
```

- R has **mean** and **var** functions:

```
> mean(edades)
[1] 34.95238
> var(edades)
[1] 747.9476
```

Vectors (4)

- When we construct vectors with elements of different types, R converts them all to a single type:

```
> c("hola",2,T)
[1] "hola" "2"    "TRUE"
> c(TRUE,FALSE,500)
[1]   1   0 500
```

- The elements of a vector can be declared with names and then retrieved with the **names** command:

```
> notas<-c(Juan=4.5,Luis=6.2,Romina=3.9,Felipe=2.8,Mariana=6.7)
> names(notas)
[1] "Juan"    "Luis"    "Romina"  "Felipe"  "Mariana"
```

- We can sort a vector using the **sort** command:

```
> names(sort(x=notas,decreasing=T))
[1] "Mariana" "Luis"    "Juan"    "Romina"  "Felipe"
```

Vector Access

- R allows access to the elements of a vector by means of numerical indexes `[i]`:

```
> notas[1] # first element
Juan
 4.5
```

- The index can be another numeric vector to access more than one element:

```
> notas[c(1,5)] # first and fifth element
   Juan Mariana
    4.5     6.7
```

- If we want to omit any element we use negative indexes:

```
> notas[-2] # All but the second one
   Juan  Romina  Felipe Mariana
    4.5     3.9     2.8     6.7
```

- The elements can also be accessed by their names:

```
> notas[c("Juan","Mariana")] # Only Juan and Mariana
   Juan Mariana
    4.5     6.7
```

Operating Vectors

- We saw earlier that if I operate a scalar by a vector, the scalar applies to all the elements of the vector.
- If I now have two vectors of the same length and operate on them, the operation is done element by element (element-wise):

```
a<-c(1,2)
b<-c(3,4)
> a+b
[1] 4 6
> a*b
[1] 3 8
```

Operating Vectors (2)

- If the vectors are of different lengths, the smaller one recycles its elements:

```
> d<-c(4,5,6,9)
> a+d
[1]  5  7  7 11
> c(a,a)+d
[1]  5  7  7 11
```

- If the length of the longest is not a multiple of the length of the shortest, we get a warning:

```
> c(1,2)+c(-9,2,3)
[1] -8  4  4
Warning message:
In c(1, 2) + c(-9, 2, 3) :
  longer object length is not a multiple of shorter object length
```

Comparing Vectors

- R supports the comparison operators for numeric variables: `>`, `<`, `==`, `<=`, `>=`, `!=` in addition to `&` `|` as well as the operators **and** and **or** for logical variables:

```
> menores<-edades<18
> menores
 [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[17]  TRUE  TRUE FALSE FALSE FALSE
```

- If we give a vector an index of logical variables we retrieve the values where the index takes the true value:

```
> edades[menores]
[1] 12  7 14  2 11 13 11
```

- Exercise: calculate the average age of the elements older or equal to 18 years old.

```
mean(edades[edades>=18])
```

Null Values

- In R, missing values are written as `NA`. It is common that they appear when we read data from a database. Some functions do not accept null values so they must be taken into account.

```
> missing_vector<-c(12,15,NA)
> missing_vector
[1] 12 15 NA
```

- To check if a variable is null we use the command `is.na`:

```
> missing_vector[!is.na(missing_vector)]
[1] 12 15
```

Secuencias

- Para crear un vector formado por una secuencia de números usamos el comando **seq**:

```
> pares<-seq(from=2,to=20,by=2)
> cuatro_mult<-seq(from=4,by=4,length=100)
> pares
 [1]  2  4  6  8 10 12 14 16 18 20
```

- También se pueden crear usando el operador (`:`):

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,10,1)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Repeticiones

- Para crear vectores que repitan un valor u otro vector varias veces usamos el comando **rep**. El primer valor es el objeto a repetir y el segundo es el número de repeticiones:

```
> rep(10,3)
[1] 10 10 10
> rep(c("hola","chao"),4)
[1] "hola" "chao" "hola" "chao" "hola" "chao" "hola" "chao"
```

- Problema: Crear una secuencia que repita 3 veces los 4 primeros múltiplos de 7.

```
> rep(seq(from=7,by=7,length=4),3)
 [1]  7 14 21 28  7 14 21 28  7 14 21 28
```

Generación de vectores aleatorios

- Para realizar experimentos o simular fenómenos de comportamiento conocido es muy útil generar vectores aleatorios.
- Si queremos números uniformemente distribuidos entre un máximo y un mínimo usamos **runif**:

```
> runif(n=5, min = 1, max = 10)
[1] 5.058862 1.737830 9.450956 9.149376 2.652774
```

- Si queremos números centrados en una media μ y con una desviación estándar σ, usamos una distribución normal con **rnorm** donde sabemos que el 68% de las observaciones estarán alrededor $\mu \pm \sigma$, el 95% en $\mu \pm 2\sigma$ y el 99.7% en $\mu \pm 3\sigma$:

```
> rnorm(n=5, mean = 10, sd = 4)
[1] 12.081286  2.636001 16.001953  0.120463  6.211835
```

Generación de vectores aleatorios (2)

- Cuando queremos modelar un número de arribos por unidad de tiempo para simular modelos de colas, usamos la distribución de **Poisson** con **rpos**. El parámetro λ nos dice la cantidad promedio de llegadas en un período:

```
> rpois(n=10, lambda = 3)
 [1] 1 3 8 6 1 1 6 3 4 7
```

- Un experimento de distribución binomial se basa en tener *n* experimentos, donde en cada experimento realizamos *k* intentos de un fenómeno cuya probabilidad de acierto en cada intento es *p*. Con el comando **rbinom** podemos simular la cantidad de aciertos obtenidos en cada experimento.

```
> rbinom(n=10,size=2,prob=0.5)
 [1] 0 1 2 1 1 0 2 0 0 1
> rbinom(n=10,size=2,prob=0.7)
 [1] 1 2 2 1 0 1 2 2 2 2
> rbinom(n=10,size=2,prob=0.2)
 [1] 0 0 0 0 1 0 1 0 1 0
```

Variables Categóricas o Factores

- Además de las variables numéricas o lógicas, se puede trabajar con variables categóricas. Ej: color, sexo, clase social.
- Se crean con el comando **factor** y los posibles valores de la variable se guardan en el atributo **levels**.

```
> gente<-factor(c("Hombre","Mujer","Mujer","Mujer","Hombre"))
> gente
[1] Hombre Mujer  Mujer  Mujer  Hombre
Levels: Hombre Mujer
> class(gente)
[1] "factor"
> levels(gente)
[1] "Hombre" "Mujer"
#Puedo renombrar a los niveles
> levels(gente)<-c("Man","Woman")
> gente
[1] Man   Woman Woman Woman Man
Levels: Man Woman
```

Agregando variables por categorías con **tapply**

- Si tenemos un vector numérico y otro categórico del mismo largo podemos aplicar una función de agregación.
- Ejemplo: Creo una categoría para el vector edades de niveles *niño*, *adolescente*, *adulto*:

```
categ_edades<-ifelse(edades<12,"niño",
                     ifelse(edades<18,"adolescente","adulto"))
class(categ_edades)
[1] "character"
#Convierto a factor con as.factor
categ_edades<-as.factor(categ_edades)
```

- Ahora cuento la cantidad de personas por categoría, y calculo la media y la desviación estándar para cada grupo:

```
tapply(edades,categ_edades,length)
adolescente      adulto        niño
          3          14           4
> tapply(edades,categ_edades,mean)
adolescente      adulto        niño
   13.00000    47.42857     7.75000
> tapply(edades,categ_edades,sd)
adolescente      adulto        niño
   1.000000   25.294312    4.272002
```

Manejo de Strings

- Puedo imprimir un string usando el comando **cat**:

```
> saludo<-"Hola Mundo"
> cat(saludo)
Hola Mundo
```

- Para concatenar dos strings uso el comando **paste**:

```
> paste("Hola","Chao",sep="-")
[1] "Hola-Chao"
> paste("persona",1:4, sep="")
[1] "persona1" "persona2" "persona3" "persona4"
> paste(saludo,1:3, sep=" ")
[1] "Hola Mundo 1" "Hola Mundo 2" "Hola Mundo 3"
```

- Para extraer sub-cadenas usamos el comando **substr**:

```
> substr(saludo,1,4)
[1] "Hola"
```

- Existe un vector llamado **letters** que tiene todas las letras del abecedario, útil para nombrar variables:

```
> letters[1:4]
[1] "a" "b" "c" "d"
```

Matrices

- Las matrices son vectores de dos dimensiones. Por defecto se van llenando por columna:

```
> matriz_por_col<-matrix(data=1:12,nrow=3,ncol=4)
> matriz_por_col
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

- Para llenarlas por fila uso el parámetro **byrow**:

```
> matriz_por_fil<-matrix(data=1:12,nrow=4,ncol=3,byrow=T)
> matriz_por_fil
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

- Accedemos a la dimensión de la matriz con el comando **dim**.

```
 > dim(matriz_por_fil)
[1] 4 3
```

Matrices (2)

- Para acceder a los elementos de una matriz tengo que especificar las filas *i* y las columnas *j* `[i,j]`. Si dejo alguno de los dos valores vacío se recuperan todos las filas o columnas:

```
 > matriz_por_fil[2,] #Segunda fila, todas las columnas
[1] 4 5 6
> matriz_por_fil[2,1] # Segunda fila, primera columna
[1] 4
> matriz_por_fil[-1,-2] # Descarto fila 1 y columna 2
     [,1] [,2]
[1,]    4    6
[2,]    7    9
[3,]   10   12
```

- Para acceder a los nombres de las filas o columnas usamos **rownames** y **colnames** de forma análoga a como usamos **names** para los vectores.

```
> rownames(matriz_por_fil)<-paste("r",1:4,sep="")
> colnames(matriz_por_fil)<-paste("c",1:3,sep="")
> matriz_por_fil["r2","c3"]
[1] 6
```

Matrices (3)

- Puedo agregarle nuevas filas o nuevas columnas a una matriz usando **rbind** y **cbind** respectivamente:

```
  > rbind(matriz_por_fil,r5=1:3)
    c1 c2 c3
r1   1  2  3
r2   4  5  6
r3   7  8  9
r4  10 11 12
r5   1  2  3
> cbind(matriz_por_fil,c4=4:1)
    c1 c2 c3 c4
r1   1  2  3  4
r2   4  5  6  3
r3   7  8  9  2
r4  10 11 12  1
```

Matrices (4)

- Operaciones algebraicas como la multiplicación de matrices se hace con `%*%`:

```
>a<-matriz_por_col %*% matriz_por_fil
      c1  c2  c3
[1,] 166 188 210
[2,] 188 214 240
[3,] 210 240 270
```

- Si usamos solamente el operador `*`, la multiplicación se hace elemento por elemento (sólo para matrices de igual dimensión). Esto aplica también para la suma, la resta, la división y otro tipo de operadores.

Matrices (5)

- Podemos transponer una matriz con **t**:

```
> t(a)
   [,1] [,2] [,3]
c1  166  188  210
c2  188  214  240
c3  210  240  270
```

- Los valores y vectores propios se calculan con **eigen**:

```
> eigen(a)
$values
[1] 6.483342e+02 1.665808e+00 3.437970e-14

$vectors
           [,1]        [,2]       [,3]
[1,] -0.5045331 -0.76077568  0.4082483
[2,] -0.5745157 -0.05714052 -0.8164966
[3,] -0.6444983  0.64649464  0.4082483
```

Arreglos o Tensores

- Los arreglos (o tensores) son como las matrices pero de más dimensiones:

```
> arreglo<-array(1:8, dim=c(2,2,2))
> arreglo
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

> arreglo[1,2,1]
[1] 3
```

Listas

- Las matrices me restringen a que todos los vectores sean del mismo largo y del mismo tipo.
- Las listas me permiten agrupar objetos de cualquier tipo y de cualquier largo:

```
milista<-list(hombre="Pepe",mujer="Juana",
              hijos=3,edades=c(4,8,12))
```

- Cuando accedo a sus elementos usando `[i]` recupero una sub-lista:

```
> milista[c(3,4)] # Sublista
$hijos
[1] 3
$edades
[1]  4  8 12
```

- Para acceder a una elemento particular tengo tres opciones:

```
milista[[1]]
milista[["hombre"]]
milista$hombre

[1] "Pepe"
```

Ejercicio Lista

- Crear una lista que tenga tres vectores de largo 100 generado por alguno de los mecanismos vistos para generar vectores aleatorios. Pueden variar las distribuciones o los parámetros. Asígnele nombres a cada uno de los vectores.

```
vectores<-list(normal=rnorm(n=100,mean=10,sd=5),
               poisson=rpois(n=100,lambda=10),
               uniforme=runif(n=100,min=5,max=15))
```

- Calcule la media y la desviación estándar de cada uno de los vectores de la lista.

```
medias<-vector()
desv<-vector()
for(i in 1:length(vectores)){
  medias[i]<-mean(vectores[[i]])
  desv[i]<-sd(vectores[[i]])
}
> medias
[1] 10.589222 10.390000  9.579866
> desv
[1] 5.155478 2.711349 2.905810
```

Cálculos agregados a Listas con **sapply** y **lapply**

- El ejercicio anterior se puede resolver de manera mucho más sencilla en R con unas funciones especiales para realizar agregación sobre listas.
- El comando **sapply** permite aplicar una función a cada elemento de una lista y devuelve los resultados en un vector. Luego **lapply** hace lo mismo pero retorna una lista:

```
 > sapply(vectores,mean)
   normal   poisson  uniforme
10.589222 10.390000  9.579866
> sapply(vectores,sd)
  normal  poisson uniforme
5.155478 2.711349 2.905810
```

- Ejercicio, programar una propia versión de **sapply**. Hint: En R una funciones puede recibir otra función como parámetro y aplicarla de manera genérica.

```
myapply<-function(lista,fun,...){
  resultado<-vector(length=length(lista))
  for(i in 1:length(lista)){
    resultado[i]<-fun(lista[[i]],...)
  }
  resultado
}
```

Data Frames

- El data.frame es el tipo de colección de datos más utilizada para trabajar con datasets en R.
- Un data.frame se compone de varios vectores, donde cada vector puede ser de distintos tipos, pero del mismo largo. Es equivalente a una tabla de una base de datos:

```
edades.frame<-data.frame(edad=edades,categoria=categ_edades)
```

```
> edades.frame
   edad   categoria
1    21       adulto
2    33       adulto
3    12 adolescente
```

- Las dimensiones de un data.frame se acceden de la misma manera que en una matriz:

```
> length(edades.frame)
[1] 2
> dim(edades.frame)
[1] 21  2
```

Data Frames (2)

- Puedo acceder a los elementos como si fuese una matriz o una lista:

```
> edades.frame[3,1] # La edad del tercer elemento
[1] 12
> edades.frame$edad[1:6] # La edad de los primeros 6 elementos
[1] 21 33 12 34 23 70
```

- También puede pasar cada variable del data.frame a mi workspace con el comando **attach** y así accederlas directamente:

```
attach(edades.frame)
> categoria[1:3]
[1] adulto      adulto      adolescente
Levels: adolescente adulto niño
```

- Puedo guardar un data.frame en un archivo csv (separado por comas u otra carácter) usando **write.table**:

```
write.table(x=edades.frame,file="edades.csv",sep=",",row.names=F)
```

- Pongo `row.names=F` para que no ponga los nombres de las columnas en el archivo.

Cargando Data Frames

- Puedo leer un data.frame desde archivos **csv** de manera nativa y desde otras fuentes (Excel, base de datos, etc.) usando librerías especiales:

```
my.frame<-read.table(file="edades.csv",header=T,sep=",")
```

- El parámetro `header` específica si quiero usar la primera fila para asignarle nombres a las columnas.
- Además R provee varias colecciones de datos para experimentar. Se pueden ver como el comando `data()`.
- Para ver todos los datasets disponibles de todas las librerías:

```
data(package = .packages(all.available = TRUE))
```

- Ahora podemos cargar un dataset, que se incluye como data.frame en mi workspace:

```
data(USArrests) # Arrestos en Estados Unidos por estado
```

Muestreo

- Cuando tenemos datasets muy grandes algunas técnicas estadísticas o de visualización pueden ser muy costosas computacionalmente.
- Se puede trabajar con una muestra aleatoria de los datos.
- La idea es que si la muestra es representativa, la propiedades observadas serán equivalentes a las de la población.
- En R se realiza el muestreo con el comando **sample**.
- Si la muestra es sin reemplazo, sacamos datos de manera aleatoria sin reponer el elemento. Entonces la muestra debe ser de menor tamaño que el dataset:

```
> sample(edades,size=4,replace=F)
[1] 80 88 12 23
```

Muestreo (2)

- Si la muestra es con reemplazo poddemos observar datos duplicados. De esta forma, la muestra puede ser incluso de mayor tamaño que la colección original:

```
sample(edades,size=100,replace=T)
```

- Cuando tenemos que los datos vienen etiquetados por alguna categoría y tomamos una muestra donde cada categoría tiene una participación proporcional a la de la colección original, tenemos un muestreo estratificado.
- Ejercicio: extraer una muestra aleatoria sin reemplazo que tenga 10 filas del data.frame **USArrests**.

```
USArrests[sample(1:(dim(USArrests)[1]),size=10,replace=F),]
```

Instalando librerías adicionales

- R tiene una comunidad muy activa que desarrolla muchas librerías para el análisis y la visualización de datos.
- Se pueden descargar librerías adicionales desde el repositorio CRAN directamente desde R.
- Las librerías se pueden instalar desde Rstudio o con el siguiente comando:

```
install.packages("rpart",dependencies=T)
```

- Luego para poder usarlas se cargan de la siguiente forma: `library(rpart)`.
- Un conjunto de liberías muy útiles para manipular datos es *tydyverse*: `https://www.tidyverse.org/`.

```
install.packages("tidyverse")
```

Bilbiografía I

Venables, William N., David M. Smith, and R Development Core Team. *An introduction to R.*, 2002.