Introduction to R

Felipe José Bravo Márquez

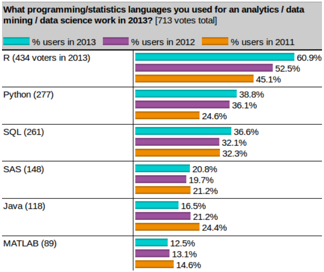
August 16, 2022

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/`

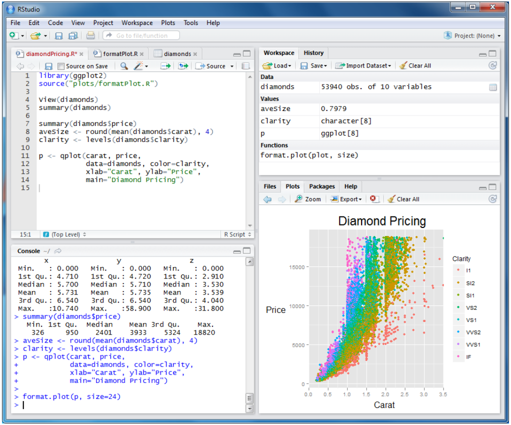
Why use R?

- 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: `https://www.rstudio.com/products/rstudio/download/`



```r
library(ggplot2)
source("plots/formatPlot.R")

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

p <- qplot(carat, price,
           data=diamonds, color=clarity,
           xlab="Carat", ylab="Price",
           main="Diamond Pricing")

```

```
       x                y                z         
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000  
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910  
 Median : 5.700   Median : 5.710   Median : 3.530  
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539  
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040  
 Max.   :10.740   Max.   :58.900   Max.   :31.800  
> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    326     950    2401    3933    5324   18820 
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
> 
> format.plot(p, size=24)
> 
```

Data	
diamonds	53940 obs. of 10 variables
Values	
aveSize	0.7979
clarity	character[8]
p	ggplot[8]
Functions	
format.plot(plot, size)	

R as a calculator

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

[1] The following slides are based on [**?**]

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**:

```
my.sum<-function(a=2,b=1){
  return(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:

```
> my.sum(3,4)
[1] 7
> my.sum()
[1] 3
```

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

```
> class(my.sum)
[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)
#to delete them all
rm(list=ls())
```

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

```
save.image("myworkspace.RData")
# we can load it in a new session
load("myworkspace.RData")
```

Vectors

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

```
ages<-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**:

```
> a.sum<-sum(ages)
> a.length<-length(ages)
> a.sum
[1] 734
> a.length
[1] 21
```

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

```
> numbers<-c(1,2,3)
> numbers+3
[1] 4 5 6
> numbers*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:

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

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

Vectors (3)

- Answer:

```
> a.mean<-sum(ages)/length(ages)
> a.mean
[1] 34.95238
> a.var<-sum((ages-a.mean)^2)/(length(ages)-1)
> a.var
[1] 747.9476
```

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

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

Vectors (4)

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

```
> c("hello",2,T)
[1] "hello" "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:

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

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

```
> names(sort(x=grades,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]`:

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

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

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

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

```
> grades[-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:

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

Operating Vectors

- We saw earlier that if we operate a scalar by a vector, the scalar applies to all the elements of the vector.
- If we 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:

```
> younger<-ages<18
> younger
 [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:

```
> ages[younger]
[1] 12  7 14  2 11 13 11
```

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

```
mean(ages[ages>=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
```

Sequences

- To create a vector consisting of a sequence of numbers we use the command **seq**:

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

- It can also be created using the operator (`:`):

```
> 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
```

Repetitions

- To create vectors that repeat a value or another vector multiple times we use the command **rep**. The first value is the object to repeat and the second is the number of repetitions:

```
> rep(10,3)
[1] 10 10 10
> rep(c("hello", "bye"),4)
[1] "hello" "bye" "hello" "bye" "hello" "bye"
"hello" "bye" "hello" "bye"
```

- Problem: Create a sequence that repeats 3 times the first 4 multiples of 7.

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

Random vector generation

- To perform experiments or simulate phenomena of known behavior it is very useful to generate random vectors.
- If we want uniformly distributed numbers between a maximum and a minimum we use **runif**:

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

- If we want numbers centered on a mean μ with a standard deviation σ, we use a normal distribution with the command **rnorm** where we know that 68% of the observations will be within the range $\mu \pm \sigma$, 95% in $\mu \pm 2\sigma$ and 99.7% in $\mu \pm 3\sigma$:

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

Random Vector Generation (2)

- When we want to model the number of arrivals per unit of time to simulate queuing models, we use the **Poisson** distribution with **rpos**. The parameter λ tells us the average number of arrivals per time period

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

- The binomial distribution can be used to represent experiments in which *k* independent trials of a coin tossing Bernoulli experiment are performed. The probability of success (e.g., getting a head) in each trial is *p*. We can simulate the number of hits obtained for *n* different Binomial experiments with the command **rbinom** (*k* is named as “size”).

```
> 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
```

Categorical Variables or Factors

- In addition to numerical or logical variables, it is possible to work with categorical variables. E.g.: color, gender, social class.
- They are created with the command **factor** and the possible values of the variable are stored in the attribute **levels**.

```
> people<-factor(c("Hombre","Mujer","Mujer","Mujer","Hombre"))
> people
[1] Hombre Mujer  Mujer  Mujer  Hombre
Levels: Hombre Mujer
> class(people)
[1] "factor"
> levels(people)
[1] "Hombre" "Mujer"
#We can rename the levels
> levels(people)<-c("Man","Woman")
> people
[1] Man   Woman Woman Woman Man
Levels: Man Woman
```

Aggregating variables by category with **tapply**

- If we have a numeric vector and a categorical vector of the same length we can apply the aggregation function **tapply**.
- Example: let's create a category for the vector "ages" with levels *child*, *adolescent*, *adult*:

```
categ_ages<-ifelse(ages<12,"child",
                      ifelse(ages<18,"adolescent","adult"))
class(categ_ages)
[1] "character"
#We convert variables to factor with as.factor
categ_ages<-as.factor(categ_ages)
```

- Now, we can count the number of people per category, and calculate the mean and standard deviation for each group:

```
> tapply(ages,categ_ages,length)
adolescent      adult      child
         3         14          4
> tapply(ages,categ_ages,mean)
adolescent      adult      child
  13.00000   47.42857    7.75000
> tapply(ages,categ_ages,sd)
adolescent      adult      child
  1.000000  25.294312  4.272002
```

Strings

- We can print a string using the command **cat**:

```
> greeting<-"Hello World"
> cat(greeting)
Hello World
```

- To concatenate two strings we can use the command **paste**:

```
> paste("Hello","Bye",sep="-")
[1] "Hola-Bye"
> paste("person",1:4, sep="")
[1] "person1" "person2" "person3" "person4"
> paste(greeting,1:3, sep=" ")
[1] "Hello World 1" "Hello World 2" "Hello World 3"
```

- To extract substrings we use the command **substr**:

```
> substr(greeting,1,4)
[1] "Hell"
```

- There is a vector called **letters** that has all the letters of the alphabet. It is useful for naming variables:

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

Matrices

- Matrices are two-dimensional vectors. By default they are filled by column:

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

- To fill them per row, we can use the parameter **byrow**:

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

- We access the dimension of the matrix with the command **dim**.

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

Matrices (2)

- To access the elements of a matrix we have to specify the rows *i* and columns *j* `[i,j]`. If we leave any of the two values empty, all rows or columns are retrieved:

```
 > matrix_per_row[2,] #Second row, all columns
[1] 4 5 6
> matrix_per_row[2,1] # Second row, first column
[1] 4
> matrix_per_row[-1,-2] # All but row 1 and column 2
     [,1] [,2]
[1,]    4    6
[2,]    7    9
[3,]   10   12
```

- To access row or column names, we use **rownames** and **colnames** analogously to how we use **names** for vectors.

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

Matrices (3)

- We can add new rows or new columns to a matrix using **rbind** and **cbind** respectively:

```
  > rbind(matrix_per_row,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(matrix_per_row,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)

- Matrix multiplication is done with command %*%:

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

- If we use only the operator *, the multiplication is done element by element (only for matrices of equal dimension). This also applies to addition, subtraction, division and other operators.

Matrices (5)

- We can transpose a matrix with the command **t**:

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

- Eigenvalues and eigenvectors are calculated with **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
```

Arrays or Tensors

- Arrays (or tensors) are like matrices but with more dimensions:

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

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

, , 2

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

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

Lists

- Matrices and arrays are restricted to have all vectors of the same length and of the same type.
- Lists allow us to group objects of any type and any length:

```
my_list<-list(man="Pepe",woman="Juana",
              children=3,ages=c(4,8,12))
```

- When we access its elements using `[i]` we get a sublist:

```
> my_list[c(3,4)] # sublist
$children
[1] 3
$ages
[1]  4  8 12
```

- We have three options to access a particular item in a list:

```
my_list[[1]]
my_list[["man"]]
my_list$man

[1] "Pepe"
```

Exercise List

- Create a list with three vectors of length 100 generated by one of the mechanisms seen for generating random vectors. You can vary the distributions or parameters. Assign names to each of the vectors.

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

- Calculate the mean and standard deviation of each of the vectors in the list.

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

Aggregated calculations to Lists with **sapply** and **lapply**

- The previous exercise can be solved in a much simpler way in R with some special functions to perform aggregation over lists.
- The command **sapply** allows to apply a function to each element of a list and returns the results in a vector. Then **lapply** does the same but returns a list:

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

- Exercise: program your own version of **sapply**. Hint: In R, a function can receive another function as parameter.

```
my_apply<-function(a_list,fun,...){
  result<-vector(length=length(a_list))
  for(i in 1:length(a_list)){
    result[i]<-fun(a_list[[i]],...)
  }
  result
}
```

Data Frames

- The data.frame is the most commonly used object type to handle datasets in R.
- A data.frame is composed of several vectors, where each vector can be of different types, but of the same length. It is equivalent to a database table:

```
ages.frame<-data.frame(edad=ages,categoria=categ_ages)

> ages.frame
   edad   categoria
1    21       adult
2    33       adult
3    12 adolescent
```

- The cells of a data.frame can be accessed in the same way as in a matrix:

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

Data Frames (2)

- We can access the elements of a data.frame as if it were an array or a list:

```
> ages.frame[3,1] # The age of the third element
[1] 12
> ages.frame$age[1:6] # The age of the first 6 elements
[1] 21 33 12 34 23 70
```

- We can also pass each variable from the data.frame to the workspace with the command **attach** and thus access them directly:

```
> attach(ages.frame)
> categ[1:3]
[1] adult      adult      adolescent
Levels: adolescent adult child
```

- We can save a data.frame in a csv file (comma separated or other character) using **write.table**:

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

- We set `row.names=F` so that it doesn't put the column names in the file.

Loading Data Frames

- We can read a data.frame from **csv** files using **read.table** and from other sources (Excel, database, etc.) using special libraries:

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

- The parameter `header` specifies whether we want to use the first row to assign names to the columns.
- In addition R comes with several datasets to experiment with. They can be viewed with the command `data()`.
- To view all available datasets from all libraries:

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

- Now we can load a dataset which will be included as a data.frame in my workspace:

  ```
  data(USArrests) # US arrests by state
  ```

Sampling

- When we have very large datasets some statistical or visualization techniques can be very computationally expensive.
- In those cases we can work with a random sample of the data.
- The idea is that if the sample is representative, the observed properties will be equivalent to those of the population.
- In R the sampling is performed with the command **sample**.
- If the sample is without replacement, we draw data randomly without replacing elements. Then the sample must be smaller than the dataset:

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

Sampling (2)

- If the sample is done with replacement we may observe duplicate data. In this way, the sample can be even larger than the original collection::

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

- When our data examples are labeled by some category and we take a sample where each category has a proportional participation to that of the original dataset, we have a stratified sample.
- Exercise: extract a random sample without replacement that has 10 rows from the data.frame **USArrests**.

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

Installing additional libraries

- R has a very active community that develops many libraries for data analysis and visualization.
- Additional libraries can be downloaded from the CRAN repository directly from the command line.
- The libraries can be installed from Rstudio interface or with the following command:

  ```
  install.packages("rpart",dependencies=T)
  ```
- Then to be able to use them they are loaded as follows: `library(rpart)`.
- A very useful set of libraries to manipulate data is *tydyverse*: `https://www.tidyverse.org/`.

  ```
  install.packages("tidyverse")
  ```
- Tidyverse includes a set of packages that provide various tools for working with data, as well as a few special ways of using those functions.

Tibble

- The tidyverse provides its own version of a data frame, which is known as a tibble.
- It comes with some smart tweaks that make it easier to work with.
- A data.frame can be converted to a tibble using the command **as_tibble**:

```
> library(tidyverse)
> ages.tibble<-as_tibble(ages.frame)
> print(ages.tibble)
# A tibble: 21 x 2
     age categ
   <dbl> <fct>
 1    21 adult
 2    33 adult
 3    12 adolescent
 4    34 adult
 5    23 adult
 6    70 adult
 7    90 adult
 8    80 adult
 9     7 child
10    29 adult
# ... with 11 more rows
```

Tibble

- We can select specific rows with the command **filter**:

```
ages.tibble %>% filter(age < 20)
```

- The command `%>%` is a pipe: it takes the output from one command and feeds it as input to the next command.
- Let's add a the column using the command **bind_cols**:

```
weights<-c(60,80,31,70,71,101,59,67,11,78,55,11,
           90,31,65,78,39,35,69,115,63)

ages.tibble <-ages.tibble %>% bind_cols("weights"=weights)
```

- Now, let's get all the people who are over 20 years old and weigh over 80

```
 ages.tibble %>% filter(age > 20 & weights > 80)
```

Tibble

- You can create a new column using the command **mutate**:

```
ages.tibble <- ages.tibble %>%
#create a new variable called future.age with the age in 10 years
mutate(future.age = age + 10)
```

- The command **mutate** can also be used to modify a column:

```
ages.tibble %>% mutate(age=1:21)
```

- The command **select** selects out only those columns you specify using their names:

```
ages.tibble %>% select(c(weights,categ))
```

Conclusions

- We have introduced R, a very popular programming language for statistical analysis.
- R provides several functionalities for manipulating datasets.
- There are several libraries that extend the basic R functionality.

References I