

Analisis y Curacion clase 3

Mariano Dominguez@Diplodatos

18 de Mayo, 2019, Cohorte Alpha

Análisis exploratorio y curación de datos

- ▶ Mariano Dominguez @ IATE-OAC-UNC & CONICET
- ▶ Edgardo Hames @ Bitlogic
- ▶ Gabriel Miretti @ Bitlogic



Figure 1: Diplodatos @ FaMAF

Paradigmas científicos clasicos:

Siglos atras la ciencia era **empirica**, describiendo los procesos naturales.



Figure 2: Galileo observando con un telescopio

luego se desarrollaron modelos **teoricos** matematicos, generalizaciones

$$\begin{aligned}\nabla \cdot \mathbf{E} &= \frac{\rho_v}{\epsilon} && \text{(Gauss' Law)} \\ \nabla \cdot \mathbf{H} &= 0 && \text{(Gauss' Law for Magnetism)} \\ \nabla \times \mathbf{E} &= -\mu \frac{\partial \mathbf{H}}{\partial t} && \text{(Faraday's Law)} \\ \nabla \times \mathbf{H} &= \mathbf{J} + \epsilon \frac{\partial \mathbf{E}}{\partial t} && \text{(Ampere's Law)}\end{aligned}$$

Figure 3: Newton, Maxwell, Einstein, Dirac eqs

Astronomía (u otra ciencia) Computacional

En las últimas décadas se han simulado fenómenos complejos, ver: http://www.tng-project.org/movies/tng/tng50_single_galaxy_formation_g3_1080p.mp4

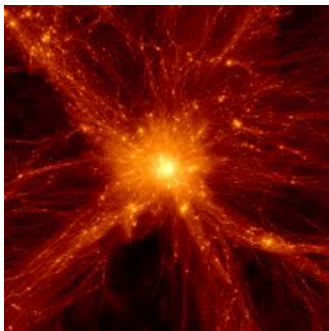


Figure 4: Simulación cosmológica LCDM

El cuarto paradigma en ciencia:

Hoy en día la exploración de datos (eScience) unifica la teoría, los experimentos y las simulaciones.

- Datos capturados por instrumentos o generados por una simulación.
- Procesados por complejos pipelines de software.
- Científico analiza bases de datos utilizando técnicas estadísticas.



Figure 5: LSST telescopio y camara

http://www.astro.caltech.edu/~george/aybi199/4th_paradigm_book_complete_lr.pdf

Herramientas para Investigacion Reproducible:

Docker combinado con Jupyter lab o RStudio son herramientas fundamentales (literate programming) y es muy probable que se unifiquen bajo Apache ARROW: <https://arrow.apache.org/> en el futuro.

La verdadera razon es la dificultad con mover los datos, i.e. las bases de datos crecen con mayor velocidad que la transferencia de los mismos: <https://skyserver.sdss.org/casjobs/>, <https://datalab.noao.edu/> , y es la razon por la que programamos en la nube: <https://aws.amazon.com/what-is-cloud-computing/> , <https://cloud.google.com> , donde resultan imperativos los contenedores.

Tanto Python, como R y Julia <https://julialang.org/> resultan muy adecuados para trabajar en ciencia de datos, visualizacion, machine learning y este ultimo esta pensado para la utilizacion optima de paralelismo con primitivas para computacion paralela a nivel de instrucciones, multicore y sistemas distribuidos. Mientras que los dos primeros permiten utilizar el stack de Apache Spark:

<https://spark.apache.org/> incluyendo asi el manejo de grandes

Por que (no) R? es FLOSS!

Introduccion al lenguaje estadistico R <http://r-project.org> y CRAN <http://cran.r-project.org/> : R consiste en una colleccion de software con una importante variedad de paquetes para analisis de datos, matematica aplicada, estadistica , graficos y diferentes utilidades. Los paquetes extras en CRAN son suministrados por individuos o comunidades de expertos en biologia, economia, geologia y otros campos (ver <https://www.jstatsoft.org/index>).

Existe una linda IDE: RStudio <https://www.rstudio.com/> y una muy buena biblioteca para graficos ggplot2 (now ggviz). Tambien existen diversas galerias de graficos en R y recientemente se ha establecido el consorcio R: <https://www.r-consortium.org/> (Microsoft compro Revolution, ver tambien h2o).

Se ha realizado un considerable esfuerzo para conectar R con otros programas, lenguajes y sistemas estadisticos. Scripts en R pueden correr facilmente desde la consola, pero mas esfuerzo es necesario para correr programas en otros lenguajes. R se conecta con C, C++, FORTRAN, JaVa, JavaScript, Matlab, Python, Perl, XLisp y Ruby. En algunos casos, las interfaces con bidimensionales

R Markdown

Esta es una presentacion en R Markdown. Markdown es un formato de sintaxis simple para crear documentos HTML, PDF y MS Word. Por mas detalles de como utilizar R Markdown ver <http://rmarkdown.rstudio.com>.

Cuando se clickea el boton **Knit**, un documento es generado que incluye el contenido asi como los outputs de de los chunks de codigos R embebidos en el documento ver <https://legacy.gitbook.com/book/bids/the-practice-of-reproducible-research/details> y <https://arxiv.org/abs/1605.04339>.

Shiny is un paquete de R que permite constriuir apps interactivos directamente en R Markdown, webpages o cosntruir dashboards. Ademas se pueden extender los Shiny apss con temas CSS, htmlwidgets y herramientas de JavaScript <https://shiny.rstudio.com/gallery> .

Estadística con R

Una noción básica es la de una muestra aleatoria.

En R es posible simular fácilmente esta situación con la función **sample**. Si por ejemplo quiero elegir cinco números aleatorios entre 1 y 40, escribo:

```
sample(1:40,5)
```

```
## [1] 27  9 34 21 25
```

Muestrear con reemplazo es adecuado para modelar monedas o dados. Por ejemplo para simular arrojar diez monedas podemos escribir:

```
sample(c("H","T"), 10, replace=T)
```

```
## [1] "H" "T" "H" "T" "T" "T" "H" "H" "H" "T"
```

Tambien se puede simular datos con diferentes probabilidades de cada resultado, (por ejemplo tener una tasa de exitos del 90%) utilizando el argumento prob in sample:

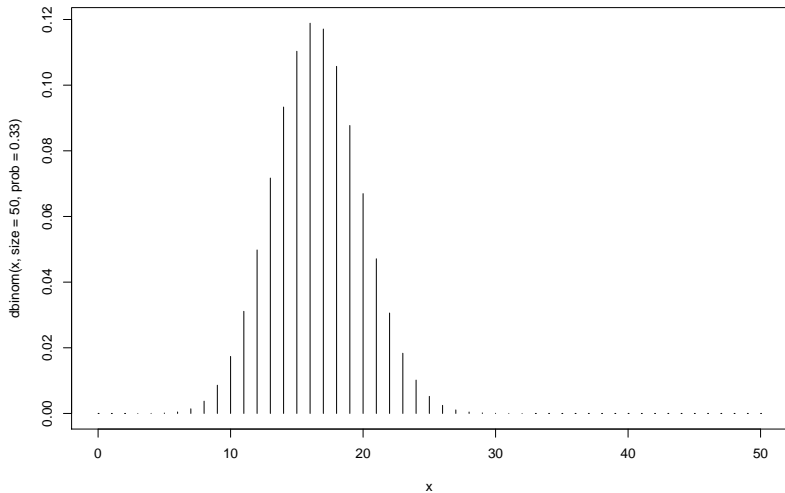
```
sample(c("succ", "fail"), 10, replace=T, prob=c(0.9, 0.1))
```

```
## [1] "fail" "succ" "succ" "succ" "succ" "succ" "succ" "succ" "s
```

Distribuciones discretas en R:

cuando las variables solo pueden tomar solamente valores finitos, es preferible dibujar un digrama de alfileres (pin), aqui podemos observar la distribucion binomial con $n=50$ y $p=0.33$.

```
x <- 0:50  
plot(x,dbinom(x,size=50,prob=.33),type="h")
```



Numeros Aleatorios:

En general suena contradictorio generar numeros aleatorios en una computadora dado que se supone que sus resultados son predecibles y reproducibles. Lo que en realidad es posible es generar secuencias de numeros pseudo aleatorios, que para todos los efectos practicos se comportan como si fueran aleatorios. Ver sobre LCGs,

<https://aaronshlegel.me/linear-congruential-generator-r.html>

https://en.wikipedia.org/wiki/Mersenne_Twister

En estadística se utilizan para crear conjuntos de datos simulados para estudiar los efectos de los algoritmos. El uso de funciones que generan numeros aleatorios es simple, por ejemplo numeros que siguen una distribucion normal:

```
n=100  
x <- rnorm (n, mean=0, sd=1)  
mean(x)
```

```
## [1] -0.05356696
```

```
sd(x)
```

```
## [1] 1.052216
```

```
quantile(x)
```

```
##           0%           25%           50%           75%          100%  
## -2.1174120 -0.8205311 -0.1323123  0.7266011  2.0909551
```

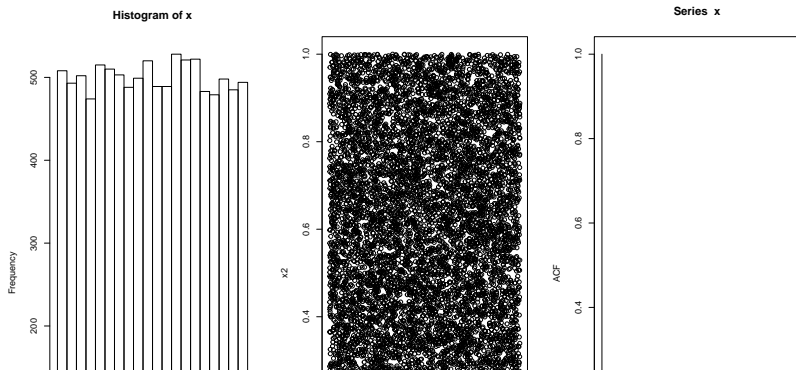
Distribution uniforme

El generador basico en R es runif, cuya entrada es el numero de valores a ser generados.

```
runif(10, min=2, max=5)
```

```
## [1] 3.551754 2.226756 2.814739 3.553073 2.487959 2.0821
## [8] 2.713716 4.599998 3.678537
```

Veamos como funciona:



Guardando las semillas.

runif no implica aleatoridad per se. runif(Nsim) es calcular una secuencia determinística basada en un número aleatorio inicial (semilla).

```
set.seed(1)  
runif(5)
```

```
## [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
```

```
set.seed(1)  
runif(5)
```

```
## [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
```

```
set.seed(2)  
runif(5)
```

```
## [1] 0.1848823 0.7023740 0.5733263 0.1680519 0.9438393
```



```
MC1seed <- .Random.seed # save the generator state  
.Random.seed <- MC1seed # restore seed
```

La transformacion Inversa del CDF:

Existe una transformacion simple, que nos permite transformar cualquier variable aleatoria en una uniforme y mas importante viceversa.

Por ejemplo si x esta dada por una densidad de probabilidad f y una CDF F , entonces vale la relacion: $F(x) = \int_{-\infty}^x f(t)dt$ y si elegimos $U = F(X)$, con U una variable aleatoria distribuida uniformemente.

Ejemplo: Si $X \propto \exp$, entonces $F(x) = 1 - e^{-x}$. Resolviendo para x en $u = 1 - e^{-x}$ nos da $x = -\log(1 - u)$. Por lo tanto si u es uniforme, entonces $X \propto \exp$

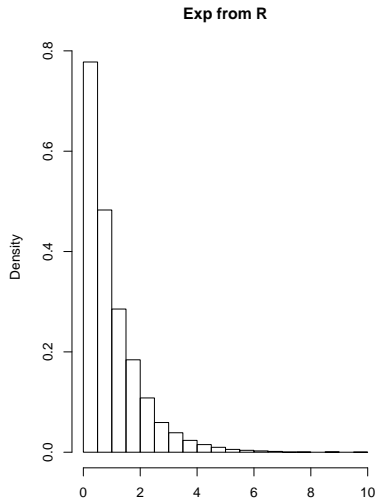
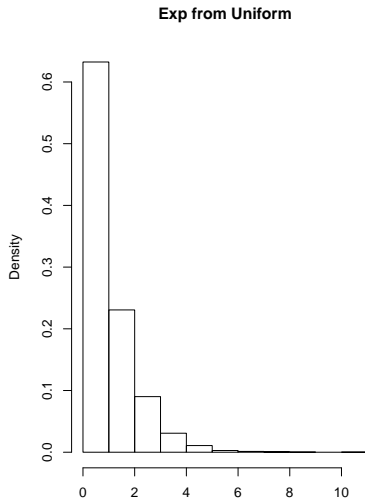
```
Nsim=10^4  
#number of random variables  
U=runif(Nsim)  
X=-log(U)  
#transforms of uniforms  
Y=rexp(Nsim)  
#exponentials from R
```

```
#plots
```

```
par(mfrow=c(1,2))
```

```
hist(X,freq=F,main="Exp from Uniform")
```

```
hist(Y,freq=F,main="Exp from R")
```



Submuestras

Los comandos `subset()`, `which()` and `ifelse()` son probablemente los mas utilizados en R. Una manera de filtrar elementos de un vector es utilizar la funcion `subset()`.

```
# create a vector  
x <- c(5,4:8,12)  
x
```

```
## [1] 5 4 5 6 7 8 12
```

```
y <- subset(x, x < 6)  
y
```

```
## [1] 5 4 5
```

Utilizando which()

identifica la posición del vector donde se cumple (is TRUE) la condición: Vea el siguiente ejemplo de cómo utilizarla:

```
# create a vector
```

```
z <- c(6:10, 12, -3)
```

```
z
```

```
## [1] 6 7 8 9 10 12 -3
```

```
which(z > 8)
```

```
## [1] 4 5 6
```

Utilizando ifelse

el comando ifelse tiene dos opciones para ejecutar. Si la condicion es TRUE se ejecuta la primera, si la condicion es FALSE se ejecuta la segunda. La sintaxis es ifelse(condition, opcion1, opcion2). Un ejemplo a continuacion.

```
# create a vector  
x <- c(-2, 5:10, 15)  
x
```

```
## [1] -2  5  6  7  8  9 10 15
```

```
# if values are < 7 will code those 1, else will become 0  
ifelse(x < 7, 1, 0)
```

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

```
# also you can do this  
ifelse(x < 7, 1, x)
```

```
## [1] 1 1 1 7 8 9 10 15
```

Code the Matrix 1:

Creamos una matriz x con numeros provenientes de una funcion normal. y llamamos a sus elementos con x[fila,columna].

```
# matrix with 12 random numbers in 4 rows  
x <- matrix(rnorm(12), nrow=4)  
x
```

```
##           [,1]      [,2]      [,3]  
## [1,]  1.2891946 -1.9527802 -0.0692199  
## [2,]  0.5955489 -0.3774369  0.1197222  
## [3,]  1.1870120  0.4915441  0.9573790  
## [4,] -0.1418306  0.1169809 -0.3900849
```

```
# find the number in 3rd row and 2nd column  
x[3,2]
```

```
## [1] 0.4915441
```

Code the Matrix 2:

tambien es posible referirse a una columna o fila u obtener las dimensiones.

```
# show second columns
```

```
x[,2]
```

```
## [1] -1.9527802 -0.3774369  0.4915441  0.1169809
```

```
# show forth row
```

```
x[4,]
```

```
## [1] -0.1418306  0.1169809 -0.3900849
```

```
# find number or columns and rows in matrix
```

```
dim(x)
```

```
## [1] 4 3
```


Utilizando lazos en R:

Cada vez que alguna operacion debe ser repetida un lazo resulta util. De acuerdo el manual de R, entre los comandos basicos de control de flujo, las construcciones para lazos son: `for`, `while` y `repeat`, con las clausulas adicionales `break` y `next`.

?Control

Un ejemplo de un lazo simple:

```
u1 <- rnorm(30) # create a vector filled with random normal
print("This loop calculates the square of the first 10 elements of
vector u1")
```

```
## [1] "This loop calculates the square of the first 10 elements of u1"
```

```
usq<-0
for(i in 1:10)
{
  usq[i]<-u1[i]*u1[i] # i-th element of u1 squared into i-th element of usq
}
```

Lazos anidados

Supongamos que queremos manipular una matriz poniendo sus elementos con valores especificos:

```
# nested for: multiplication table
mymat = matrix(nrow=30, ncol=30) # create a 30 x 30 matrix
for(i in 1:dim(mymat)[1]){
  # for each row
  for(j in 1:dim(mymat)[2]){
    # for each column
    # assign values based on position: product of two in
    mymat[i,j] = i*j
  }
}
mymat[1:5,1:5] # show just the upper left 5x5 chunk
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
```

Vectorizacion 1:

es la operacion de convertir repetidas operaciones en numeros (escalares) en operaciones en vectores o matrices. Muchos lazos pueden hacerse implicitos con vectorizacion.

el ejemplo mas elemental es la adicicion de dos vectores v1 y v2 en un vector v3, lo que puede hacerse elemento por elemento con un lazo:

```
n=100
v1 <- rnorm(n)
v2 <- rnorm(n)
v3 <- 0
#
for (i in 1:n)
{
v3[i] <-v1[i] + v2[i]
}
```

Vectorizacion 2:

o utilizando la forma vectorizada:

```
v3 = v1 + v2
```

lo que permite utilizar eficientemente rutinas muy eficientes de algebra lineal (BLAS) Comparemos el tiempo de ejecucion entre ambas soluciones:

```
m=10; n=10;
mymat<-replicate(m, rnorm(n)) # create matrix of normal random variables
mydf=data.frame(mymat) # transform into data frame
# measure loop execution
system.time(
  for (i in 1:m) {
    for (j in 1:n) {
      mydf[i,j]<-mydf[i,j] + 10*sin(0.75*pi)
    }
  }
)
```

Ejercicio: Mida el tiempo en una version vectorizada.

La familia de instrucciones `apply`.

Esta compuesta de funciones intrinsecamente vectorizada y esta compuesta de funciones (`[s,l,m,r, t,v]apply`) para manipular datos en forma de matrices en una forma repetitiva, evitando el uso explicito de lazos. Las primeras tres son las mas utilizadas:

1. `apply` permite aplicar una funcion a filas (primer indice) o columnas (segundo indice) de una matriz.
2. `lapply` permite aplicar una dada funcion a cada elemento de una lista.
3. `sapply` igual que la anterior, pero se obtiene un vector en lugar de una lista.

```
#### elementary example of apply function
# define matrix mymat by replicating the sequence 1:5 for 4 rows
mymat<-matrix(rep(seq(5), 4), ncol = 5)
# mymat sum on rows
apply(mymat, 1, sum)
```

```
## [1] 15 15 15 15
```

```
# mymat sum on columns
apply(mymat, 2, sum)
```

```
## [1] 10 11 12 13 14
```

```
# produce a summary column wise (for each column)
apply(mymat, 2, function(x, y) summary(mymat))
```

```
##           [,1]           [,2]           [,3]
## [1,] "Min.      :1.00" "Min.      :1.00" "Min.      :1.00"
## [2,] "1st Qu.:1.75" "1st Qu.:1.75" "1st Qu.:1.75"
## [3,] "Median  :2.50" "Median  :2.50" "Median  :2.50"
## [4,] "Mean    :2.00" "Mean    :2.00" "Mean    :2.00"
## [5,] "Var     :1.00" "Var     :1.00" "Var     :1.00"
## [6,] "Std. Dev.:1.00" "Std. Dev.:1.00" "Std. Dev.:1.00"
## [7,] "Sample   :5" "Sample   :5" "Sample   :5"
## [8,] "Range    :4" "Range    :4" "Range    :4"
## [9,] "Max.     :5" "Max.     :5" "Max.     :5"
```

Importando y exportando datos.

Los datos en R pueden guardarse como archivos .Rdata con la función save. Que luego pueden leerse en R con load.

```
a <- 1:10  
save(a, file = "Data.Rdata")  
rm(a)  
load("Data.Rdata")  
print(a)
```

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

Presentando los Data Frames:

El siguiente ejemplo crea un dataframe a y lo guarda como un archivo CSV con write.csv. Luego el dataframe es cargado desde el archivo a una variable b con read.csv.

```
var1 <- 1:5
var2 <- (1:5) / 10
var3 <- c("Diplo", "Datos", "en", "FaMAF", "@UNC")
a <- data.frame(var1, var2, var3)
names(a) <- c("VariableInt", "VariableReal", "VariableChar")
write.csv(a, "Data.csv", row.names = FALSE)
#rm(a)
b <- read.csv("Data.csv")
print(b)
```

```
##      VariableInt VariableReal VariableChar
## 1              1           0.1          Diplo
## 2              2           0.2          Datos
## 3              3           0.3             en
## 4              4           0.4          FaMAF
```


Creando un Data Frame

Un data frame es similar a una matriz, pero puede contener elementos numericos o texto. La funcion que se utiliza para crear data frames es `dataframe()` por ej:

```
# create a data frame
hospital <- c("Cordoba", "Buenos Aires")
pacientes <- c(150, 350)
df <- data.frame(hospital, pacientes)
df
```

```
##           hospital pacientes
## 1      Cordoba      150
## 2 Buenos Aires      350
```

```
# structure
str(df)
```

```
## 'data.frame':    2 obs. of  2 variables:
## $ hospital : Factor w/ 2 levels "Buenos Aires",...: 2 1
```

Read Write Table

La funcion `write.table` guarda el contenido de un objeto en un archivo. El objeto es tipicamente un marco de datos ('data.frame'), pero puede ser cualquier otro tipo de objeto (vector, matriz,. . .).

La funcion `read.table` crea un marco de datos ('data frame') y constituye la manera mas usual de leer datos en forma tabular.

```
misdatos <- read.table("data.dat")  
misdatos$hospital
```

```
## [1] Cordoba      Buenos Aires  
## Levels: Buenos Aires Cordoba
```

```
misdatos["hospital"]
```

```
##      hospital  
## 1      Cordoba  
## 2 Buenos Aires
```

crea un marco de datos denominado `misdatos`

cada variable recibira por defecto el nombre V1, V2,... y puede ser accedida individualmente escribiendo `misdatosV1`, `misdatosV2`,... , o escribiendo `misdatos["V1"]`, `misdatos["V2"]`,... , o, tambien escribiendo `misdatos[,1]`, `misdatos[,2]`, .. etc

Existen varias opciones con valores por defecto (aquellos usados por R si son omitidos por el usuario). Para solicitar ayuda utilizar ?

```
?read.table
```

Para mas ejemplos de uso de R puede consultarse <https://www.computerworld.com/article/2497464/business-intelligence/top-r-language-resources-to-improve-your-data-skills.html> en particular es muy recomendable el paquete <http://swirlstats.com/>

Para extender el manejo de R ver <http://adv-r.had.co.nz/> y como construir paquetes <http://r-pkgs.had.co.nz/> y por supuesto <https://www.r-graph-gallery.com>

coffe & mate breack?

THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG
PILE OF LINEAR ALGEBRA, THEN COLLECT
THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL
THEY START LOOKING RIGHT.



Algunos Datasets intrinsecos:

iris dataset

```
## 'data.frame':    150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0
## $ Species      : Factor w/ 3 levels "setosa","versicolor"
```

bodyfat dataset, brief excursion to `install.packages("")` details:

Exploracion de Datos 1:

► 1-Checkeando las dimensiones

```
dim(iris)
```

```
## [1] 150    5
```

► 2 nombre de las variable o columnas

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal"
## [5] "Species"
```

► 3 Structura

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9
```

Exploracion de Datos 2:

► 4 Atributos

```
attributes(iris)
```

```
## $names
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

```
## [5] "Species"
```

```
##
```

```
## $class
```

```
## [1] "data.frame"
```

```
##
```

```
## $row.names
```

```
##      [1]      1      2      3      4      5      6      7      8      9     10     11     12     13
```

```
##     [18]     18     19     20     21     22     23     24     25     26     27     28     29     30
```

```
##     [35]     35     36     37     38     39     40     41     42     43     44     45     46     47
```

```
##     [52]     52     53     54     55     56     57     58     59     60     61     62     63     64
```

```
##     [69]     69     70     71     72     73     74     75     76     77     78     79     80     81
```

```
##     [86]     86     87     88     89     90     91     92     93     94     95     96     97     98
```

```
##    [103]    103    104    105    106    107    108    109    110    111    112    113    114    115
```

Exploracion de Datos 3:

- 5 Veamos las primeras 5 filas

```
iris[1:5,]
```

##		Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1		5.1	3.5	1.4	0.2	setosa
## 2		4.9	3.0	1.4	0.2	setosa
## 3		4.7	3.2	1.3	0.2	setosa
## 4		4.6	3.1	1.5	0.2	setosa
## 5		5.0	3.6	1.4	0.2	setosa

- 6 Veamos los valores de alguna columna

```
iris[1:10, "Sepal.Length"]
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
```


Imputacion de valores NA 1:

```
# Introduce missing values  
set.seed(100)  
original<-iris  
iris[sample(1:nrow(iris), 40), "Sepal.Length"] <- NA  
library(mice)
```

```
## Loading required package: lattice
```

```
##
```

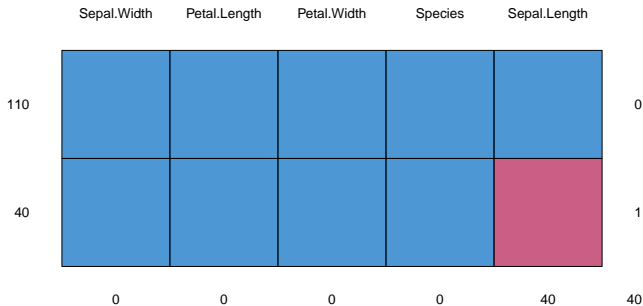
```
## Attaching package: 'mice'
```

```
## The following objects are masked from 'package:base':
```

```
##
```

```
##      cbind, rbind
```

```
md.pattern(iris)
```

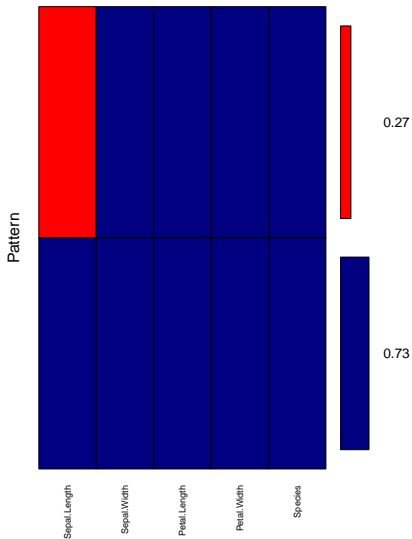
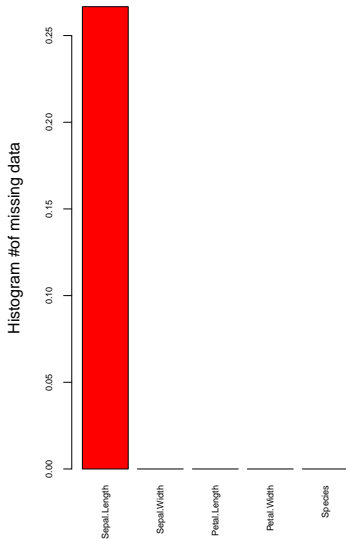


```
##
```

```
Sepal.Width Petal.Length Petal.Width Species Sepal.Length
```

Imputacion de valores NA 2:

Esto puede visualizarse como



Imputacion de valores NA 3:

Para tratar con valores perdidos, el metodo principal es imputar esos valores por ejemplo con la media, mediana, moda o valores cercanos. Otra opcion si se disponen de suficientes datos es ignorar esa medicion (na.action=na.omit)

```
impute(iris$Sepal.Length, mean) # replace with mean
```

##	1	2	3	4	5	
##	5.100000	4.900000	4.700000	4.600000	5.000000	5.400000
##	8	9	10	11	12	
##	5.000000	5.806364*	4.900000	5.400000	4.800000	4.800000
##	15	16	17	18	19	
##	5.806364*	5.700000	5.400000	5.100000	5.700000	5.100000
##	22	23	24	25	26	
##	5.806364*	4.600000	5.100000	5.806364*	5.000000	5.000000
##	29	30	31	32	33	
##	5.200000	4.700000	4.800000	5.400000	5.200000	5.806364*
##	36	37	38	39	40	
##	5.000000	5.500000	4.000000	5.806364*	5.100000	5.000000

Exploracion de variables individuales 1:

► 1 Distribucion de cada variable

```
iris<-original  
summary(iris)
```

```
##      Sepal.Length      Sepal.Width      Petal.Length      Petal.  
## Min.      :4.300      Min.      :2.000      Min.      :1.000      Min.  
## 1st Qu.:5.100      1st Qu.:2.800      1st Qu.:1.600      1st Qu.  
## Median :5.800      Median :3.000      Median :4.350      Median  
## Mean   :5.843      Mean   :3.057      Mean   :3.758      Mean  
## 3rd Qu.:6.400      3rd Qu.:3.300      3rd Qu.:5.100      3rd Qu.  
## Max.   :7.900      Max.   :4.400      Max.   :6.900      Max.  
##           Species  
## setosa      :50  
## versicolor:50  
## virginica  :50  
##  
##  
##
```

Exploracion de variables individuales 2:

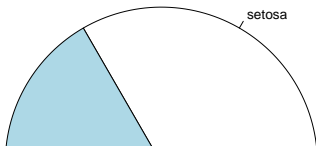
► 2 Frecuencia

```
table(iris$Species)
```

```
##  
##      setosa versicolor  virginica  
##          50          50          50
```

► 3 Pie chart

```
pie(table(iris$Species))
```



Exploracion de variables individuales 3:

- ▶ 4 media y varianza de la Sepal.Length

```
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

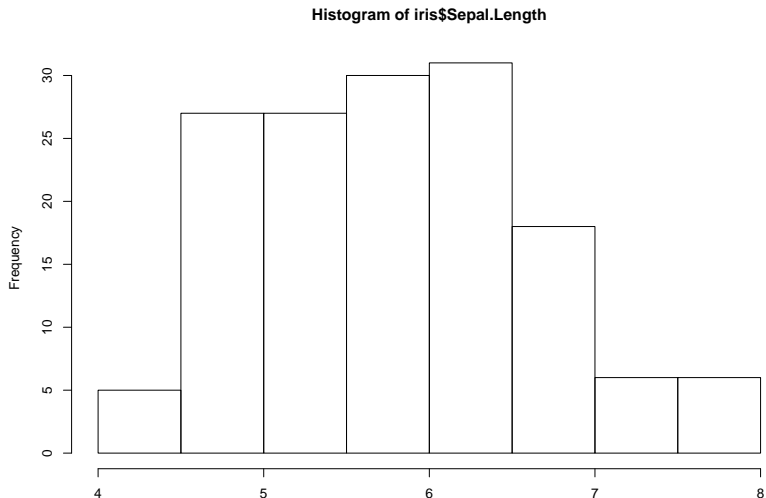
```
var(iris$Sepal.Length)
```

```
## [1] 0.6856935
```

Exploracion de variables individuales 4:

► 5 Histogramas

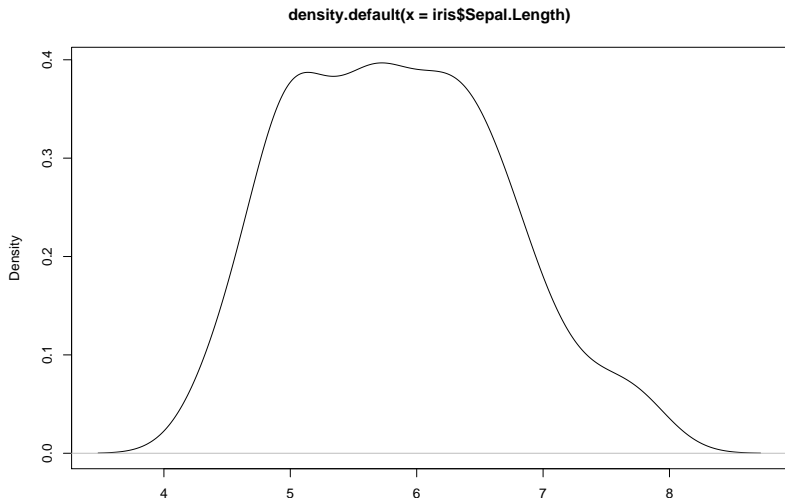
```
hist(iris$Sepal.Length)
```



Exploracion de variables individuales 5:

► 6 Densidad

```
plot(density(iris$Sepal.Length))
```



Errores como features:

El vocabulario internacional de metrologia (VIM) define una cantidad como **una propiedad de un fenomeno, cuerpo o substancia, donde la propiedad tiene una magnitud que puede ser expresada como un numero y una referencia.**

donde tipicamente el numero es el **valor de una cantidad** obtenida mediante un procedimiento de medicion, y la referencia es la **unidad de medicion.**

Adicionalmente, toda cantidad debe tener asociada alguna indicacion sobre la calidad de la medicion, esto es un atributo cuantificable conocido como **incerteza o error**, que caracteriza que dispersion de valores que pueden ser atribuibles a una dada medicion.

Las incertezas pueden ser directamente medidas o derivadas en el caso de una medicion indirecta ($\text{Potencia} = \text{Voltaje} * \text{corriente}$) y deben obtenerse por propagacion. Ver las librerias `units` y `errors` en CRAN, para un uso adecuado de las mismas como features o cuando se generan nuevos.

Técnicas de detección de outliers:

Outliers son valores extremos que se desvían de las observaciones de los datos, pueden indicar una variabilidad en la medición, errores experimentales o un descubrimiento fenomenal.

Pueden ser de en una variable (o feature) o en varias, simples o colectivos. Las causas más comunes que los generan pueden ser: errores de carga de datos (humanos o computacionales), errores instrumentales, intencionales (inyecciones), procesamiento de los datos!, errores de muestreo (fuentes disímiles o diferentes unidades), naturales o novedades.

Explorando multiples variables 1:

- ▶ 1 covariance of two variables

```
cov(iris$Sepal.Length, iris$Petal.Length)
```

```
## [1] 1.274315
```

- ▶ 2 Correlation of two variables

```
cor(iris$Sepal.Length, iris$Petal.Length)
```

```
## [1] 0.8717538
```

Explorando multiples variables 2:

► 3 Distribution in subsets

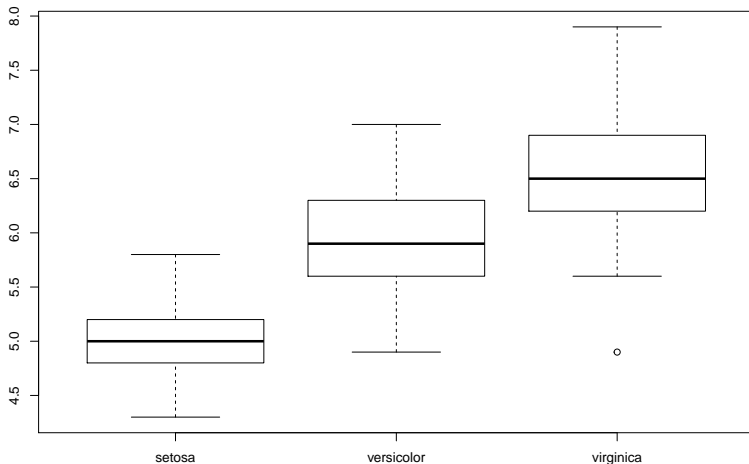
```
aggregate(Sepal.Length ~ Species, summary, data=iris)
```

```
##      Species Sepal.Length.Min. Sepal.Length.1st Qu. Sepal.Length.Max.
## 1      setosa           4.300           4.800           5.800
## 2 versicolor           4.900           5.600           6.900
## 3  virginica           4.900           6.225           7.900
##      Sepal.Length.Mean Sepal.Length.3rd Qu. Sepal.Length.Max.
## 1           5.006           5.200           5.800
## 2           5.936           6.300           6.900
## 3           6.588           6.900           7.900
```

Explorando multiples variables 3:

► 4 Box Plot

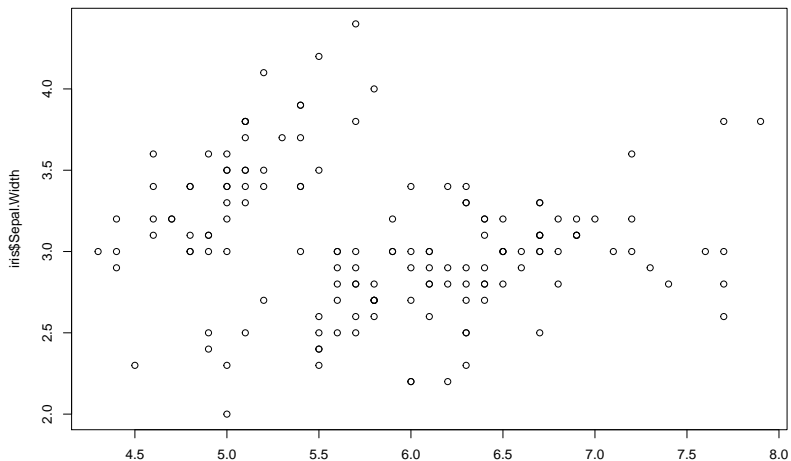
```
boxplot(Sepal.Length~Species, data=iris)
```



Explorando multiples variables 4:

► 5 Scatter plot

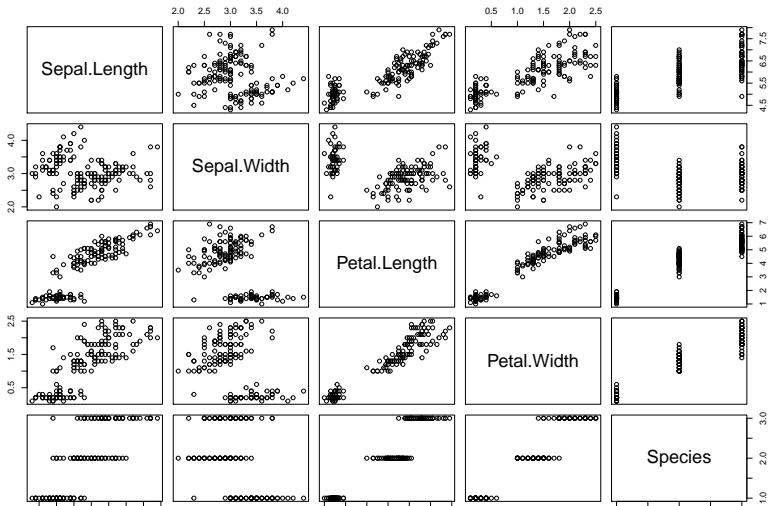
```
plot(iris$Sepal.Length, iris$Sepal.Width)
```



Explorando multiples variables 5:

► 6 Pairs plot

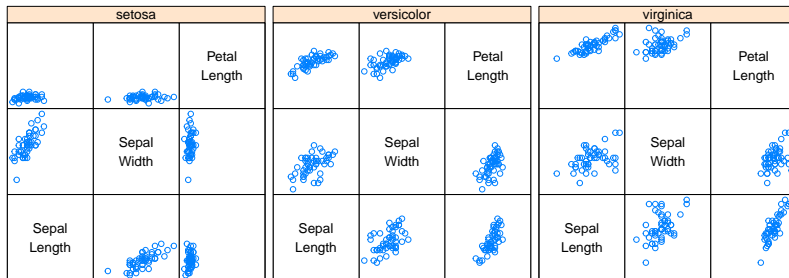
```
pairs(iris)
```



Explorando multiples variables 6:

- 7 other complicated plots

```
library(lattice)
splom(~iris[1:3] | Species, data = iris, pscales = 0, varnames = c("Sepal Length", "Petal Length", "Sepal Width"))
```



Scatter Plot Matrix

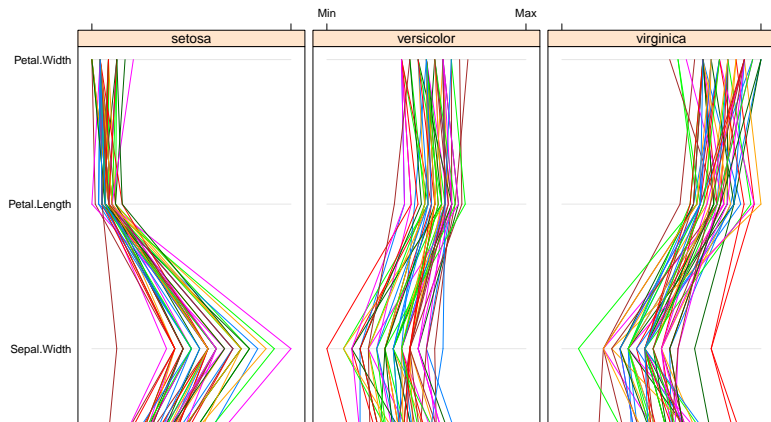
Explorando multiples variables 7:

```
parallel(~iris[, 1:4] | Species, data = iris, layout = c(3,
```

```
## Warning: 'parallel' is deprecated.
```

```
## Use 'parallelplot' instead.
```

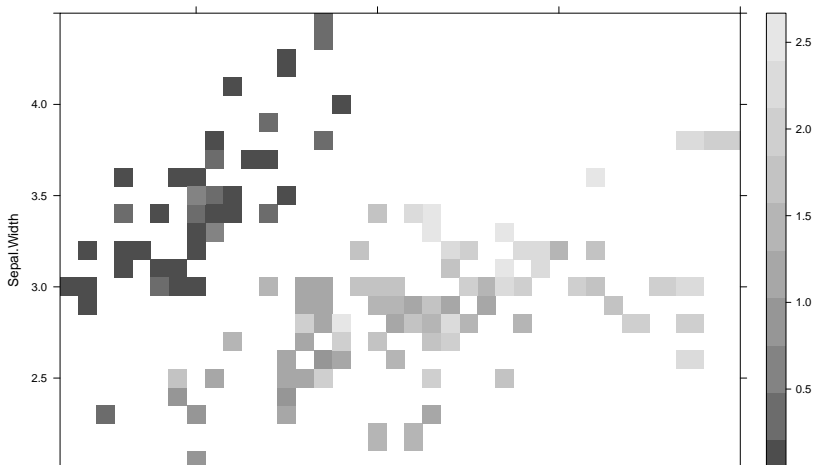
```
## See help("Deprecated")
```



More Exploration

► Level Plot

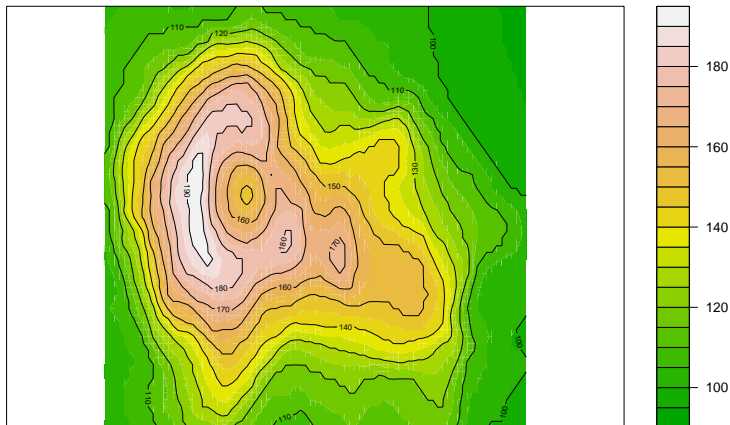
```
library(lattice)  
print(levelplot(Petal.Width~Sepal.Length*Sepal.Width, iris))
```



More Exploration

► Contour

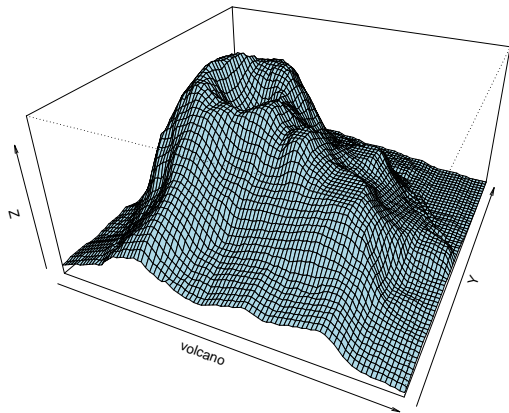
```
filled.contour(volcano, color = terrain.colors, asp = 1, p
```



More Exploration

► 3D Surface

```
persp(volcano, theta = 25, phi = 30, expand = 0.5, col = "green")
```



More Exploration

- Interactive 3D Scatter Plot

```
library(rgl)  
plot3d(iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)
```

Writing plots as pdf/ps.

- ▶ Save as a .PDF file

```
pdf("myPlot.pdf")  
x <- 1:50  
plot(x, log(x))  
graphics.off()
```

- ▶ Save as a postscript file

```
postscript("myPlot.ps")  
x <- -20:20  
plot(x, x^2)  
graphics.off()
```

Writing plots as png/jpeg

- Find temp.. or save as png or jpg

```
jpeg("plot.jpg")  
plot(x, 1/x)  
dev.off()
```

```
## pdf
```

```
## 2
```


Ejercicios:

Visualizacion es una herramienta muy importante para la generacion de intuicion, pero raramente uno tiene los datos en la forma necesaria. Frecuentemente se necesitara crear nuevas variables o simplemente reordenarlas.

Exploraremos ahora la manipulacion basica utilizando un conjunto de datos sobre los vuelos en Nueva York en 2013.

```
library(nycflights13)
flights<-nycflights13::flights
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>         <int>         <dbl>
## 1  2013     1     1     517             515           2
## 2  2013     1     1     533             529           4
## 3  2013     1     1     542             540           2
## 4  2013     1     1     544             545          -1
```

Practico 1: Entregar un Rmd donde se encuentren todos los vuelos que:

- ▶ Que arribaron con un retraso de mas de dos horas.
- ▶ Volaron hacia Houston (IAH o HOU)
- ▶ Fueron operados por United, American o Delta.
- ▶ Salieron en Verano (Julio, Agosto y Septiembre)
- ▶ Arrivaron mas de dos horas tarde, pero salieron bien.
- ▶ Salieron entre medianoche y las 6 am.

