Contenido

Curso 2 2
Week 3 2
lapply 2
apply 3
mapply 4
split 5
Debbuging 5
Week 4 7
str() 7
Numeros Aleatorios 7
Normal 7
Generando numeros aleatorios para una regresion lineal 7
Random Sampling 10

Curso 2

Week 3

Tomás

8 de noviembre de 2019

lapply

lapply() aplica una funcion a una lista. recibe 3 argumentos:

- Una lista (x)
- Una funcion (FUN)
- Otros argumentos (...).
- Estos (...) son generalmente los argumentos que se necesitan para que la funcion (FUN) sea usada correctamente.

```
lapply(x,FUN,...)
```

Ejemplo 1:

```
x <- list(a = 1:5, b=rnorm(10))
lapply(x,mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.3596216
```

---

sapply

Es una version mejorada de lapply que busca simplificar el retorno de la funcion lapply

```
sapply(x,FUN,...)
```

- Si el retorno de la funcion de una lista de elementos de longitud 1, entrega un vector
- Si el retorno es una lista de la misma longitud cada una entrega una matriz
- Si no encuentra una forma mas eficiente, retornara una lista tal cual lapply

Ejemplo 1:

```
x <- list(a = 1:5, b=rnorm(10), c = rnorm(20,1) ,d = rnorm(100,5))
sapply(x,mean)
```

```
##           a           b           c           d
##  3.00000000 -0.01667401  0.97840777  4.84994049

class(sapply(x,mean))

## [1] "numeric"
```

---

apply

apply() aplica una funcion a una dimension especifica de un array Recibe 3 argumentos:

- Un array (x) [Vector,matriz]
- MARGIN indica que dimension queremos "mantener" (1 filas, 2 columnas)
- La funcion (FUN) que se quiere aplicar
- otros argumentos (...)

```
apply(x,MARGIN,FUN,...)
```

Ejemplo 1:

```
x <- matrix(rnorm(200),20,10)
apply(x,2,mean)

##  [1]  0.04540928  0.19112188  0.32103835 -0.47033683  0.12197411 -
0.32330068
##  [7]  0.01131228 -0.12643500 -0.20119799  0.02653277

apply(x,1,sum)

##  [1]  2.9244333 -0.7773968 -7.3743049 -2.6428985  1.2212753 -5.3691640
##  [7]  0.3064090  1.3149993  5.7987006 -6.4998583  0.1603175  3.1854963
## [13]  5.5899627  1.9983783  1.7082135 -5.2042425 -4.1623413 -2.2312351
## [19]  1.7560530  0.2195657
```

Ojo:

Las funciones de sumar/promediar columnas puede ser utilizadas directamente (son mas eficientes):

```
rowSums()
rowMeans()
colSums()
colMeans()
```

---

tapply

tapply() aplica una funcion a una parte especifica de un vector. Recibe 4 argumentos:

- Un vector (x)

- INDEX un vector de la misma longitud que x que identifica las posiciones a aplicar
- Una funcion (FUN) que se quiere aplicar
- Finalmente otros argumentos (...)

```
tapply(x,INDEX,FUN,...,simplify)
```

Ejemplo 1:

```
x <- c(rnorm(10),runif(10),rnorm(10))
f <- gl(3,10)
f

##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3

tapply(x,f,mean)

##           1           2           3
## -0.17147200  0.37322475 -0.03749873
```

---

mapply

mapply() aplica una funcion a un grupo de listas (no solo a 1 como lapply). Recibe 4 argumentos:

- Una funcion (FUN) que se quiere aplicar
- (...) argumentos que se transformaran en listas
- MoreArgs si le quiero dar mas argumentos a la fucion FUN
- simplify si quiero simplificar el resultado

```
mapply(FUN,(...),MoreArgs,simplify)
```

Ejemplo 1:

```
x <- list(rep(1,4), rep(2,3), rep(3,2),rep(4,1))
x

## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4

mapply(rep,1:4,4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

---

split

split() divide un vector/list/dataframe segun un factor Recibe 4 argumentos:

- Un vector (x)
- Factor f que indica las posiciones a cortar o grupos
- Drop indica si hay un factor vacio este debe ser botado
- (...) argumentos de la funcion

```
split(x,f,INDEX,drop,...)
```

Ejemplo 1:

```
data_1 <- read.csv('hw1_data.csv', header = TRUE, nrows = 6)
data_1
```

```
##   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 <- read.csv('hw1_data.csv', header = TRUE)
s <- split(data, data$Month)
sapply(s, function(x) colMeans(x[,c("Ozone","Solar.R","Wind")], na.rm =
TRUE))
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Debbuging

Los posibles errores de un programa se dividen en:

- Message

- Warning
- Error
- Condition (Son condiciones para saltar los 3 anteriores)

Para rastrear los errores existen 5 funciones que podrian ayudar:

- Traceback: Imprime las llamadas de las funciones que se llaman dentro de una funcion y donde ha ocurrido el error
- Debug: Se le pasa como argumento una funcion, cada vez que esta funcion es llamada para en la primera linea y se puede ir navegando a traves de la funcion mas lentamente.
- Browser: Es igual que Debug pero uno elige en que linea de una funcion ir mas lento y no desde el principio
- Trace: Permite insertar un codigo de Debug o Browser pero sin modificar el codigo original (Generalmente usado para hacer Debug a un codigo que no es tuyo)
- Recover: Puedes pausar la ejecucion de una funcion al momento que ocurre el error y rastrear este

Week 4

Tomás

13 de noviembre de 2019

str()

La funcion str() es una alternativa a la tipica funcion summary nos entrega caracterisiticas del objeto que le entregamos, str se puede leer como estructura del objeto.

Numeros Aleatorios

Se pueden recrear numeros aleatorios especialmente para simulacion y estadistica:

- rnorm : Numeros normales random
- rpois : Numeros poisson random

Estos numeros aleatorios pueden ser antecedidos por:

- d : density
- r : random number generation
- p : cumulative distribution
- q : quantile function

Normal

```
dnorm(x, mean = 0, sd = 1, log = FALSE) # Densidad de probabilidad
qnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) #
Distribucion acumulativa
pnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) # Inversa de
qnorm
rnorm(n, mean = 0, sd = 1) # Numeros random
```

Cada vez que se generan numeros aleatorios es importante plantar una semilla para poder generar los mismo numeros siempre

```
set.seed(1)
rnorm(10,0,1)
```

```
##  [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
##  [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

Generando numeros aleatorios para una regresion lineal

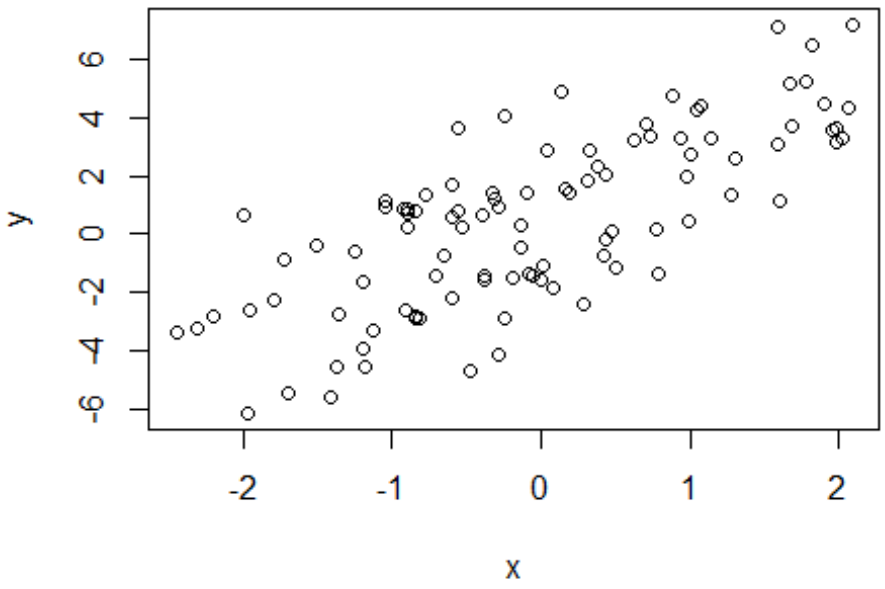
Supongamos que queremos general un modelo lineal (Considerando x variable aleatoria normal):

```
set.seed(2)
x <- rnorm(100)
```

```
e <- rnorm(100,0,2)
y <- 0.5 + 2*x + e
summary(y)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -6.1180 -1.5795  0.6741  0.4970  2.9112  7.1640

plot(x,y)
```

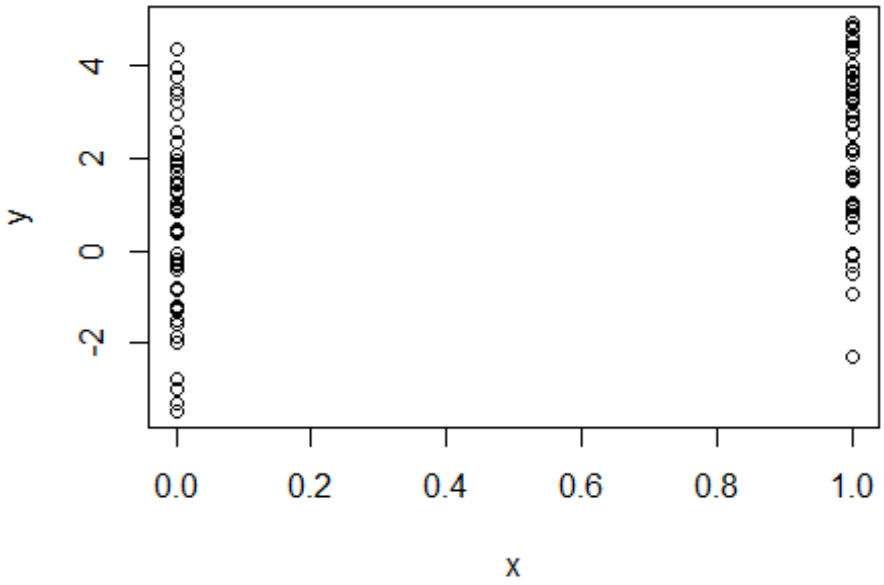


Supongamos que queremos general un modelo lineal (Considerando x variable aleatoria binomial):

```
set.seed(3)
x <- rbinom(100,1, 0.5)
e <- rnorm(100,0,2)
y <- 0.5 + 2*x + e
summary(y)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -3.4647 -0.2239  1.4970  1.4364  3.2434  4.9341

plot(x,y)
```

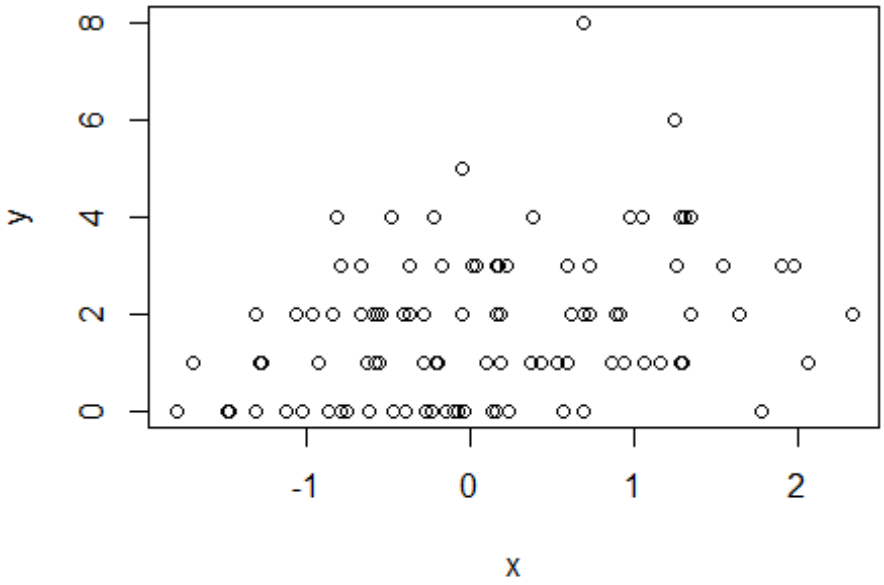


Supongamos que queremos general un modelo lineal (Considerando x una variable poisson):

```
set.seed(4)
x <- rnorm(100)
log.mu <- 0.5 + 0.3 * x
y <- rpois(100,exp(log.mu))
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.00    2.00    1.74    3.00    8.00
```

```
plot(x,y)
```



Random Sampling

Esta funcion te permite escoger un numero o elemento al azar entre un conjunto de datos que se entrega

```
set.seed(5)
sample(1:10,4)

## [1] 2 9 7 3

sample(1:10) #Obtengo una permutacion de los numeros

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

sample(letters,5)

## [1] "l" "p" "e" "i" "v"

sample(1:10,replace = TRUE)

##  [1] 10  5 10 10  3  1 10  6  8  7
```