

Ciencia de Datos

Máximo Sangiácomo

Índice

1	Introducción a R	7
1.1	Primeros pasos	7
1.2	Buscar ayuda	8
1.3	Tipos de datos	8
1.4	Limpieza de memoria	8
1.5	Asignación de valores	9
1.6	Operadores aritméticos	9
1.7	Operadores relacionales	10
1.8	Operadores lógicos	11
1.9	Vectores	12
1.10	Secuencias	15
1.11	Factores	17
1.12	Matrices	17
1.13	Listas	21
1.14	Data frames	22
1.15	R base	25
1.16	Apply y tapply	25
1.17	Map	26
1.18	Loops	27
1.19	Condicionales	29
1.20	Funciones	29
1.20.1	Output más de un resultado	30
1.20.2	Argumentos con valores default	30

2 Base de datos	32
2.1 Directorio de trabajo	32
2.2 Cargar datos	33
2.2.1 Ingresar datos con <code>tidyverse</code>	33
2.2.2 Bases de Stata	34
2.3 Problemas de imputación	34
2.4 Exportar datos	36
2.5 Pipe	36
2.6 Variables	36
2.7 Merge	37
2.8 Variables: <code>group_by</code> , <code>mutate</code>	37
2.9 Guardar datos	38
2.10 Valores missing	39
2.10.1 Eliminar valores missing	40
2.11 Loop	41
2.12 Pivot (Reshape)	42
2.13 Row bind (Append)	43
2.14 Strings	44
2.15 Fechas	45
2.15.1 Manipulación de fechas	45
2.16 Análisis de datos	46
2.16.1 Tablas	47
2.17 <code>group_by</code> , <code>summarise</code>	47
2.18 Vector de resultados	49
2.19 Gráficos	49
2.20 GGPlot	50
2.21 Guardar un gráfico	52
3 Conceptos generales	53
3.1 Estimacion	53
3.2 Prediccion	53
3.3 Inferencia	54
3.4 Metodos parametricos	55

3.5	Metodos no parametricos	55
3.6	Evaluacion de la precision del modelo	55
3.6.1	Calidad del ajuste	56
3.6.2	Trade-off Sesgo-Varianza	58
3.6.3	Clasificacion	60
3.6.4	Matriz de confusion	60
3.6.5	Curva ROC	61
3.7	Cross Validation	63
3.8	Bootstrap	64
3.9	Resumen	65
4	Regresion lineal	67
4.1	Relacion entre estimacion optima y prediccion optima	68
4.2	Aplicacion practica	70
5	LASSO	76
5.1	Aplicacion practica	78
6	Logit	83
6.1	Modelo <i>logit</i>	83
6.1.1	Interpretacion de coeficientes en el modelo <i>logit</i>	84
6.2	Aplicacion practica	85
7	Arboles de decision	94
7.1	<i>Classification and Regression Tree (CART)</i>	94
7.2	Bagging	97
7.3	Random Forest	97
7.4	Boosting	97
7.4.1	Ada Boost	98
7.5	Aplicacion practica	99
7.5.1	Arboles de clasificacion	99
7.5.2	Comparacion de modelos para clasificacion	108
7.5.3	Arboles de regresion	112
7.5.4	Bagging y Random Forests	115
7.5.5	Boosting	118

8 Neural Networks	122
8.1 Single Layer Neural Networks	122
9 Análisis de clusters	125
9.1 K-Means Clustering	126
9.2 Aplicación práctica	129

Descripción del curso

El objetivo del curso es abordar distintas metodologías de análisis de datos desde el punto de vista teórico y práctico utilizando el programa R. Si bien, dada la extensión de las clases, la cobertura de cada tema no busca ser exhaustiva intenta capturar las principales intuiciones de cada método.¹

En los Capítulos 1 y 2 se hace una breve introducción al manejo de bases de datos en R. El Capítulo 3 se ocupa de los conceptos teóricos² a ser utilizados en la práctica. Luego, los Capítulos 4 a 9 revisan distintas metodologías donde primero se presentan cuestiones conceptuales y después se realizan aplicaciones prácticas utilizando mayormente los ejemplos expuestos en (James et al., 2021).³ Para seguir profundizando se recomienda ver más ejemplos en (Boehmke and Greenwell, 2020).

¹Puede suceder que algún tema no sea cubierto completamente durante las clases pero la idea es dejar el material disponible para que pueda ser revisado de manera individual.

²Se muestran las principales formulaciones matemáticas aunque la derivación formal de los resultados excede los objetivos de este curso.

³Es importante señalar que la subsección 7.5.2 presenta ejercicios que pueden servir de guía para la elaboración del **Trabajo Práctico Final**.

Capítulo 1

Introducción a R

- La programación de rutinas en *software* específico para la manipulación y análisis de bases de datos permite asegurar procesos homogéneos, documentados, fácilmente auditables, modificables y que pueden ser compartidos entre diferentes usuarios. Además, resulta sumamente útil para realizar tareas repetitivas generando ganancias de eficiencia.
- Es muy importante realizar anotaciones, tanto para compartir código con otro usuario como para uno mismo en el futuro (por ejemplo, cuáles son los insumos/*output*, los ¿por qué?). La combinación de teclas **Ctrl/Cmd + Shift + R** permite crear secciones en *scripts* que luego sirven para navegar y ser ordenados.

Importante. *A la gloria no se llega por un camino de rosas.* **Trabajar, trabajar y trabajar.** [Osvaldo Zubeldia](#).

1.1 Primeros pasos

R es un lenguaje orientado a **objetos** (vectores, listas, matrices). Si bien al principio puede parecer demasiado complejo, no es así. De hecho, una característica destacada de R es su flexibilidad.

Mientras que un software clásico muestra inmediatamente los resultados de un comando, R los almacena en un objeto, por lo que se puede realizar un análisis sin el resultado desplegado.

R (el motor) puede complementarse con **RStudio** (tablero de instrumentos) que es una IDE (*integrated development environment*) para operar de manera mas amigable (editor con sintaxis y distintos espacios de trabajo). Cada uno se encuentra disponible en:¹

1. R

¹Buscar las versiones adecuadas para el sistema operativo utilizado.

2. RStudio

Una vez instalados los programas se deben descargar “paquetes” que agregan funcionalidades al paquete que viene incorporado (base).

```
#Descarga de programa
install.packages('tidyverse')

#Carga de programa antes de utilizarlo en un script
library(tidyverse)
```

Eventualmente se puede utilizar una función específica de un paquete previamente instalado sin cargar el paquete completo. Por ejemplo, si se quiere utilizar la función `read_excel()` del paquete `readxl`.

```
readxl::read_excel()
```

1.2 Busacar ayuda

```
# Por comando
?rm # Para poder ver la ayuda el paquete debe estar instalado
help(lm)
```

- Buscar el tab Help en la ventana de abajo a la derecha de RStudio
- [Google](#)

1.3 Tipos de datos

- character/string
- numeric (integer, double)
- factor (variables categóricas, importante para clasificación)
- logical
- date

1.4 Limpieza de memoria

```
rm(list = ls()) # Elimina todos los objetos en memoria
gc() # Garbage Collection
```

1.5 Asignación de valores

```
# nombre_objeto <- valor
x <- 1
x = 5
x
```

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

```
y = x
y = 4
```

1.6 Operadores aritméticos

```
y + x
```

```
## [1] 9
```

```
y - x
```

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

```
y * x
```

```
## [1] 20
```

```
4 / 8
```

```
## [1] 0.5
```

```
8 %% 4
```

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

```
2**5
```

```
## [1] 32
```

```
2^5
```

```
## [1] 32
```

```
sqrt(9)
```

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

```
log(1)
```

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

1.7 Operadores relacionales

```
# < <= > >= == !=  
x == 1
```

```
## [1] FALSE
```

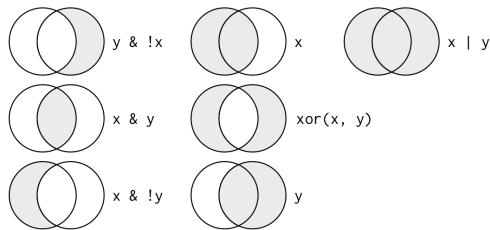
```
x == y
```

```
## [1] FALSE
```

```
y < x
```

```
## [1] TRUE
```

1.8 Operadores lógicos



Nota: Se puede usar `||` (o) y `&&` (y) para combinar múltiples expresiones lógicas. Estos operadores se llaman “cortocircuito”: tan pronto como `||` ve el primer VERDADERO devuelve VERDADERO sin calcular nada más. Tan pronto como `&&` ve el primer FALSO, devuelve FALSO.

```
# ! & / && ||
TRUE
```

```
## [1] TRUE
```

```
!FALSE
```

```
## [1] TRUE
```

```
!F
```

```
## [1] TRUE
```

```
F & T
```

```
## [1] FALSE
```

```
F | T
```

```
## [1] TRUE
```

```
x == 1 & y==4
```

```
## [1] FALSE
```

```
x == 1 | y==4
```

```
## [1] TRUE
```

```
!(x > y); x <= y
```

```
## [1] FALSE
```

```
## [1] FALSE
```

1.9 Vectores

Una característica distintiva de los vectores es que son atómicos / homogéneos.

```
a = c(0,2,5,3,8)
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
b = c("a","b","c","d","e")
typeof(b)
```

```
## [1] "character"
```

```
class(b)
```

```
## [1] "character"
```

```
f = c(F,T,F,T,T)
typeof(f)
```

```
## [1] "logical"
```

```
c(a,b)  # coerción (transforma todo en character)
```

```
## [1] "0" "2" "5" "3" "8" "a" "b" "c" "d" "e"
```

```
typeof(c(a,b))
```

```
## [1] "character"
```

```
c(NA,5,2,3,4)
```

```
## [1] NA 5 2 3 4
```

```
c(NULL,5,2,3,4)
```

```
## [1] 5 2 3 4
```

```
length(a)
```

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

```
names(a)
```

```
## NULL
```

```
names(a) = b
a
```

```
## a b c d e
## 0 2 5 3 8
```

```
## Subsetting / extraer
```

```
# por posición
a[3]
```

```
## c
## 5
```

```
a[3:5]
```

```
## c d e  
## 5 3 8
```

```
a[c(1,5)]
```

```
## a e  
## 0 8
```

```
a[-1]
```

```
## b c d e  
## 2 5 3 8
```

```
# por nombre  
a[c('a','e')]
```

```
## a e  
## 0 8
```

```
# con booleanos  
a[a >= 4]
```

```
## c e  
## 5 8
```

```
a[!(a >= 4)]
```

```
## a b d  
## 0 2 3
```

```
a[a > 2 & a < 7]
```

```
## c d  
## 5 3
```

```

a[a==0 | a==5]

## a c
## 0 5

a[a %in% c(0, 5)]

## a c
## 0 5

# muestra las posiciones que cumplen
#con la condición (no los valores)
which(a < 4)

```

```

## a b d
## 1 2 4

```

1.10 Secuencias

```

# Utilizar una funcion
# nombre_funcion(arg1 = val1, arg2 = val2, ...)
# Los argumentos se pueden nombrar,
# utilizar forma implícita o dejarlo definido por default
d = c(1,1,2,3,4,4,5,6,8,9,7,7,0)
unique(d)

```

```

##  [1] 1 2 3 4 5 6 8 9 7 0

```

```

rep(5, 10)

```

```

##  [1] 5 5 5 5 5 5 5 5 5 5

```

```

seq(1, 20)

```

```

##  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

```

```

seq(1, 20, by = 0.7)

##  [1] 1.0 1.7 2.4 3.1 3.8 4.5 5.2 5.9 6.6 7.3 8.0 8.7 9.4 10.1 10.8
## [16] 11.5 12.2 12.9 13.6 14.3 15.0 15.7 16.4 17.1 17.8 18.5 19.2 19.9

1:length(a)

## [1] 1 2 3 4 5

seq_along(a) # equivalente, pero en un caso mas seguro (ver abajo)

## [1] 1 2 3 4 5

1:20

##  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

set.seed(8)
muestra = sample(1:10, 15, replace = T)
muestra

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

sample(letters, 10, rep = T)

## [1] "n" "i" "i" "l" "g" "h" "s" "f" "g" "b"

# operaciones vectorizadas
a*c(1, 0, 10, 1, 5)

##  a  b  c  d  e
## 0  0 50  3 40

a==c(0, 1, 5, 0, 0)

##      a      b      c      d      e
##  TRUE  FALSE  TRUE  FALSE FALSE

```

```
1:6 + c(10, 2) # reciclaje (extiende el vector mas corto)
```

```
## [1] 11 4 13 6 15 8
```

1.11 Factores

```
#Variables categóricas con valores limitados
```

```
#{importante para modelos de clasificación}
```

```
data = c(1,2,2,3,1,2,3,3,1,2,3,3,1)
```

```
fdata = as.factor(data)
```

```
fdata
```

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

```
## Levels: 1 2 3
```

```
# Niveles definidos por el usuario
```

```
fdata2 = factor(data, labels = c('a', 'b', 'c'))
```

```
fdata2
```

```
## [1] a b b c a b c c a b c c a
```

```
## Levels: a b c
```

```
factor_order = c('c', 'b', 'a')
```

```
fdata3 = factor(fdata2, levels = factor_order)
```

```
fdata3
```

```
## [1] a b b c a b c c a b c c a
```

```
## Levels: c b a
```

1.12 Matrices

```
matrix(rep(3, 8))
```

```
##      [,1]
```

```
## [1,]    3
```

```
## [2,]    3
```

```
## [3,]    3
```

```

## [4,]    3
## [5,]    3
## [6,]    3
## [7,]    3
## [8,]    3

matrix(rep(3, 8), nrow = 4)

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

matrix(seq(1:12), nrow = 4)

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

matrix(seq(1:12), nrow = 4, byrow = T)

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

matrix(a, nr = 2, nc = 4) # completa pero avisa sobre las dimensiones

##      [,1] [,2] [,3] [,4]
## [1,]    0    5    8    2
## [2,]    2    3    0    5

cbind(1:5, 5:1)

##      [,1] [,2]
## [1,]    1    5
## [2,]    2    4
## [3,]    3    3
## [4,]    4    2
## [5,]    5    1

```

```

rbind(1:10, letters[10:1])

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
## [2,] "j"  "i"  "h"  "g"  "f"  "e"  "d"  "c"  "b"  "a"

mat1 = matrix(muestra, nc = 5)
mat1

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

dim(mat1)

## [1] 3 5

# subsetting/extraer
mat1[5]

## [1] 10

mat1[2, 3]

## [1] 2

mat1[3,]

## [1] 2

mat1[, 4]

## [1] 2 7 3 6 4

mat1[, 4]

## [1] 3 6 6

mat1[, 4, drop = F] # para no perder la dimensión de matriz

##      [,1]
## [1,]    3
## [2,]    6
## [3,]    6

```

```
mat1[c(1,3), c(1:3,5)]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     4     7     1     4
## [2,]     2     7     3     4
```

```
mat1[-2, -4]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     4     7     1     4
## [2,]     2     7     3     4
```

```
mat2 = matrix(sample(1:10, 4, replace = T), nc = 2)
mat3 = matrix(sample(1:10, 4, replace = T), nc = 2)
mat4 = mat2 * mat3  # elemento a elemento
mat5 = mat2 %*% mat3 # multiplicación de matrices
```

```
mat2
```

```
##      [,1] [,2]
## [1,]     5    10
## [2,]    10     6
```

```
mat3
```

```
##      [,1] [,2]
## [1,]     9     8
## [2,]    10     5
```

```
mat4
```

```
##      [,1] [,2]
## [1,]    45    80
## [2,]   100    30
```

```
mat5
```

```
##      [,1] [,2]
## [1,]   145    90
## [2,]   150   110
```

1.13 Listas

Una característica distintiva de las listas es que son recursivas y heterogéneas.

```
# Las listas pueden contener elementos de distinto tipo
ms <- list('a', 1L, 1.5, TRUE)
str(ms)
```

```
## List of 4
## $ : chr "a"
## $ : int 1
## $ : num 1.5
## $ : logi TRUE
```

```
lista1 <- list(mat1, a, b)
str(lista1)
```

```
## List of 3
## $ : int [1:3, 1:5] 4 7 2 7 10 7 1 2 3 3 ...
## $ : Named num [1:5] 0 2 5 3 8
## ... - attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
## $ : chr [1:5] "a" "b" "c" "d" ...
```

```
names(lista1) = c('a', 'b', 'c')

# subsetting/extracción
lista1$a
```

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

```
lista1[[1]]
```

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

```

lista1[[1]][1, 1]

## [1] 4

lista1['a']

## $a
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    7    1    3    4
## [2,]    7   10    2    6    8
## [3,]    2    7    3    6    4

lista1$c[5]

## [1] "e"

```

1.14 Data frames

```

df1 = data.frame(mat1)
names(df1) = c('var1', 'var2', 'var3', 'var4', 'var5')
df2 = data.frame(
  v1 = 1:10,
  v2 = 10:1,
  v3 = sample(1:10, rep = T),
  v4 = 0,
  v5 = sample(letters, 10, rep = T),
  stringsAsFactors = F)
str(df2)

## 'data.frame': 10 obs. of 5 variables:
## $ v1: int 1 2 3 4 5 6 7 8 9 10
## $ v2: int 10 9 8 7 6 5 4 3 2 1
## $ v3: int 5 3 6 5 9 10 9 10 3 7
## $ v4: num 0 0 0 0 0 0 0 0 0 0
## $ v5: chr "h" "n" "m" "e" ...

dim(df2)

## [1] 10 5

```

```
length(df2)

## [1] 5

# subsetting/extraer
df2[, 4]

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

df2[, 'v4']

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

df2$v4

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

df2['v4']

##      v4
## 1    0
## 2    0
## 3    0
## 4    0
## 5    0
## 6    0
## 7    0
## 8    0
## 9    0
## 10   0

df2$v5[1:5]

## [1] "h" "n" "m" "e" "t"
```

```
df2[3:8, c(1,5)]
```

```

##  v1 v5
## 3  3  m
## 4  4  e
## 5  5  t
## 6  6  o
## 7  7  y
## 8  8  q

# con booleanos:
df2[df2$v5 == "m",]

##  v1 v2 v3 v4 v5
## 3  3  8  6  0  m

df2[df2$v2 >= 4, 'v5']

## [1] "h" "n" "m" "e" "t" "o" "y"

df2[df2$v2 %in% c(3,5), 'v5']

## [1] "o" "q"

head(df2,5)

##  v1 v2 v3 v4 v5
## 1  1 10  5  0  h
## 2  2  9  3  0  n
## 3  3  8  6  0  m
## 4  4  7  5  0  e
## 5  5  6  9  0  t

tail(df2)

##  v1 v2 v3 v4 v5
## 5  5  6  9  0  t
## 6  6  5 10  0  o
## 7  7  4  9  0  y
## 8  8  3 10  0  q
## 9  9  2  3  0  f
## 10 10  1  7  0  h

```

```
# View(df2[c("v4", "v1")])
```

1.15 R base

Se muestran algunas funciones básicas con R base para que puedan conocerlas aunque mas adelante utilizaremos `tidyverse` para la manipulación y transformación de datos.

```
# crear una variable
df2$v6 = sample(1:10, 10, rep = T)
# rename
names(df2)[names(df2) == "v6"] = "v7"
names(df2)
```

```
## [1] "v1" "v2" "v3" "v4" "v5" "v7"
```

```
# delete
df2$v7 = NULL
names(df2)
```

```
## [1] "v1" "v2" "v3" "v4" "v5"
```

```
# Ver todos los objetos de "Enviroment":
ls()
```

```
## [1] "a"          "b"          "d"          "data"       "df1"
## [6] "df2"        "f"          "factor_order" "fdata"      "fdata2"
## [11] "fdata3"     "hook_output"  "lista1"     "mat1"       "mat2"
## [16] "mat3"        "mat4"        "mat5"       "ms"         "muestra"
## [21] "x"          "y"
```

1.16 Apply y tapply

```
# MARGIN 1 = FILAS
# MARGIN 2 = COLUMNAS
apply(df1, MARGIN = 1, sum)
```

```
## [1] 19 33 22
```

```
lista = list(1:5, 20, 12:9)
lapply(lista,mean)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 20
##
## [[3]]
## [1] 10.5
```

```
data(iris)
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 4.9 ...
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

*# Aplica una funcion a un vector en los subvectores que
define otro vector. Notar que la variable Species es factor*

```
tapply(iris$Petal.Length, iris$Species, mean)
```

```
##      setosa versicolor  virginica
##      1.462      4.260      5.552
```

1.17 Map

Toma un vector como entrada, aplica una función a cada pieza y devuelve un nuevo vector que tiene la misma longitud (y los mismos nombres) que el de entrada.

```
purrr::map(lista, mean)
```

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

```
## [1] 20
##
## [[3]]
## [1] 10.5

# Se puede aplicar una función anónima (sin nombre)
# Ver como definir una función (y sus componentes) más abajo
# Una función corta se puede usar sin llaves {}
resultado = purrr::map(lista, function(x) x + 5)
resultado
```

```
## [[1]]
## [1] 6 7 8 9 10
##
## [[2]]
## [1] 25
##
## [[3]]
## [1] 17 16 15 14
```

1.18 Loops

```
# for
for (x in 1:5) {
  print(x*2)
  print("listo")
}
```

```
## [1] 2
## [1] "listo"
## [1] 4
## [1] "listo"
## [1] 6
## [1] "listo"
## [1] 8
## [1] "listo"
## [1] 10
## [1] "listo"
```

```

resultado = c()  # Inicializa vector de resultados
for (i in seq_along(a)) {
  resultado[i] = a[i] * 2
}
resultado

## [1] 0 4 10 6 16

a * 2

##  a  b  c  d  e
## 0  4 10  6 16

# seq_along vs. length(). En un vector vacío el primero
# hace el proceso correcto (con integer(0) se interrumpe)
# Útil para un entorno de programación
conj1 = 1:10; conj2 = 20:30
x = intersect(conj1, conj2)
seq_along(x)

## integer(0)

3:seq_along(x)

## Error in 3:seq_along(x): argument of length 0

3:length(x) # secuencia descendente de 3 a 0

## [1] 3 2 1 0

# while
resultado = c()
i = 1
while (i <= length(a)) {
  resultado[i] = a[i] * 2
  i = i + 1
}
resultado

## [1] 0 4 10 6 16

```

1.19 Condicionales

```
y = 2; f = 'auto'

if (is.numeric(y)) {
  print('es numerico!')
}

## [1] "es numerico!"

if (is.numeric(y)) print('es numerico!')

## [1] "es numerico!"

if (is.numeric(f)) {
  print('es numerico!')
} else {
  print('no es numerico!')
}

## [1] "no es numerico!"

ifelse(is.numeric(y), y+8, NA)

## [1] 10

ifelse(a>=5, a+10, 0)

## a b c d e
## 0 0 15 0 18

if (this) {
  # hace esto
} else if (that) {
  # hace esto otro
} else {
  # hace otra cosa
}
```

1.20 Funciones

```

# Componentes: nombre, argumentos, cuerpo y return
# (opcional, pero generalmente se usa).
# Si el programa es para uso personal no es necesario
# validar inputs
suma = function(x, y) {
  s = x + y
  return(s)
}
res = suma(3,5)
res
## [1] 8

```

1.20.1 Output más de un resultado

```

suma2 = function(x, y) {
  s = x + y
  m = x * y
  return(list(suma = s, mult = m))
}
res = suma2(3,5)
res['suma']

```

```

## $suma
## [1] 8

```

```
res['mult']
```

```

## $mult
## [1] 15

```

1.20.2 Argumentos con valores default

```

suma3 = function(x, y = 2) {
  s = x + y
  return(s)
}
suma3(7)

```

```
## [1] 9
```

```
suma3(7, y = 0)
```

```
## [1] 7
```

Nota: se puede escribir una función (o varias) en un *script* y luego utilizar `source(mifuncion.R)` para tenerlas disponibles en el espacio de trabajo.

Capítulo 2

Base de datos

En esta clase nos vamos a centrar en el uso de `tidyverse`. Además vamos a utilizar funciones de `lubridate` y `zoo` que tienen algunas funciones especiales para trabajar con fechas.

En R existen dos tipos de bases de datos `data.frame()` y `tibble()` que son las bases de datos de `tidyverse` el mejor paquete para manipulación y transformación de datos (ver [Wickham and Grolemund \(2017\)](#)). Un `data.frame` (objeto `df`) se convierte fácilmente a `tibble` (y viceversa).

```
# Un data.frame (objeto df) se convierte fácilmente a tibble
tib = as_tibble(df)
```

Las tibbles tienen algunas funciones especiales como poder usar nombres de variables con espacio o números (se deben utilizar *back ticks*).

```
library(tidyverse)
tb <- tibble(
  `Plazo Fijo` = "espacio",
  `2000` = "numero"
)
tb

## # A tibble: 1 x 2
##   `Plazo Fijo` `2000`
##   <chr>        <chr>
## 1 espacio      numero
```

2.1 Directorio de trabajo

```
# Para ver en que directorio estamos trabajando
getwd()
# Definir directorio. Notar barras invertidas en la ruta
setwd('C:/Documentos/CianciaDatos')
```

2.2 Cargar datos

Tip. La función `fread()` del paquete `data.table` es la más eficiente para grandes volúmenes de datos porque permite paralelizar con *multithread*.

```
# CSV
bd = read.csv("b_datos.csv", header=TRUE, stringsAsFactors=TRUE, sep=", ")

bd = data.table::fread('b_datos.txt', header=TRUE, stringsAsFactors=F, sep='\t', nThreads=4)

bd = read.delim('./datos/b_datos.txt', header=TRUE, stringsAsFactors=F, sep='\t')

# Excel
# También puede suministrarse la ruta de acceso completa
bd = readxl::read_excel('./data/datos_wb.xlsx', sheet='1')
str(bd)

## # tibble [60 x 11] (S3: tbl_df/tbl/data.frame)
## # $ year      : num [1:60] 2011 2011 2011 2011 2011 ...
## # $ cname     : chr [1:60] "Argentina" "Brazil" "Chile" "France" ...
## # $ ccode      : chr [1:60] "ARG" "BRA" "CHL" "FRA" ...
## # $ gdp_pc2010: num [1:60] 10883 11628 13456 41369 36228 ...
## # $ gdp_pc2017: num [1:60] 24648 15323 22338 42864 42892 ...
## # $ gdp_2010   : num [1:60] 4.49e+11 2.30e+12 2.32e+11 2.70e+12 2.15e+12 ...
## # $ credit_ps : num [1:60] 14 58.1 101.3 96.8 94.1 ...
## # $ inv       : num [1:60] 17.2 20.6 23.1 22.4 19.7 ...
## # $ exports   : num [1:60] 18.4 11.6 37.8 28.4 26.9 ...
## # $ imports   : num [1:60] 16.8 12.4 34.4 30.4 28.3 ...
## # $ popu      : num [1:60] 4.13e+07 1.98e+08 1.72e+07 6.53e+07 5.94e+07 ...
```

2.2.1 Ingresar datos con tidyverse

Cuadro 2.1: Vista de la base de datos (World Bank)

year	cname	ccode	gdp_pc2010	gdp_pc2017
2,011	Argentina	ARG	10,883	24,648
2,011	Brazil	BRA	11,628	15,323
2,011	Chile	CHL	13,456	22,338
2,011	France	FRA	41,369	42,864
2,011	Italy	ITA	36,228	42,892
2,011	United Kingdom	GBR	39,729	42,294

Comando	Separador
<code>read_csv()</code>	coma
<code>read_csv2()</code>	punto y coma
<code>read_tsv()</code>	tab
<code>read_delim()</code>	otros

2.2.2 Bases de Stata

```
library(heaven)
read_dta(...)
write_dta(...)
```

2.3 Problemas de imputación

En esta sección se presentan algunas alternativas para solucionar problemas de imputación de datos, es decir, al cargar el archivo original el formato de alguna variable es distinto del esperado. Las funciones `parse_*`() toman un vector `character` y devuelven un vector del tipo indicado como lógico, entero o fecha.¹

```
# Parsear vectores
library(readr)
str(parse_logical(c("TRUE", "FALSE", "NA")))
```

```
##  logi [1:3] TRUE FALSE NA
```

```
str(parse_integer(c("1", "2", "3")))
```

```
##  int [1:3] 1 2 3
```

¹Ver más especificaciones [aquí](#).

```
str(parse_date(c("2010-01-01", "1979-10-14")))

##  Date[1:2] , format: "2010-01-01" "1979-10-14"

parse_integer(c("1", "231", ". ", "456"), na = ". ")

## [1] 1 231 NA 456

parse_double("1,23", locale = locale(decimal_mark = ","))

## [1] 1.23

challenge <- read_csv(readr_example('challenge.csv'))
tail(challenge)

## # A tibble: 6 x 2
##       x     y
##   <dbl> <date>
## 1 0.805 2019-11-21
## 2 0.164 2018-03-29
## 3 0.472 2014-08-04
## 4 0.718 2015-08-16
## 5 0.270 2020-02-04
## 6 0.608 2019-01-06

challenge <- read_csv(
  readr_example('challenge.csv'),
  col_types = cols(x = col_double(),
                    y = col_date()))
tail(challenge)

## # A tibble: 6 x 2
##       x     y
##   <dbl> <date>
## 1 0.805 2019-11-21
## 2 0.164 2018-03-29
## 3 0.472 2014-08-04
## 4 0.718 2015-08-16
## 5 0.270 2020-02-04
## 6 0.608 2019-01-06
```

2.4 Exportar datos

```
# CSV
write.csv(bd, "b_datos.csv")
write_csv()
write_excel_csv()
# TXT
write_delim()
write_tsv()

# Excel
library("xlsx")
out <- list('bd' = BD1, 't1' = TAB1, 't2' = TAB2)
write.xlsx(out, file = 'resutados.xlsx')
```

2.5 Pipe

Se llama **pipe** al símbolo `%>%` (*shortcut* con: Ctrl/Cmd + Shift + M) que cumple la función de una función compuesta. Es decir, una secuencia de operaciones del tipo: $h(g(f(x)))$.

Dicho de otra forma: $x \%>\% f \%>\% g \%>\% h$.

La sintaxis general de una función es `FN(OBJETO, ...)` y lo que hace la pipa es enviar el objeto a la posición correspondiente sin necesidad de expresarlo explícitamente. Por lo tanto, la sintaxis anterior puede expresarse equivalentemente con una pipa de la siguiente manera: `OBJETO %>% FN(, ...)`.

2.6 Variables

```
library(tidyverse)
bd1 = bd %>%
  mutate(gdp_pc2010bis = gdp_2010 / popu, # crear variable
        logGDP_pc2010 = log(gdp_pc2010),
        open = exports + imports,
        inv_demean = inv - mean(inv)) %>%
  rename(poblacion = popu) %>% # rename (newname = oldname)
  mutate(gdp_2010 = NULL)      # drop (también con select(-gdp_2010))
```

year	cname	gdp_pc2010	gdp_pc2010bis	open	inv_demean
2,011	Argentina	10,883	10,883	35.2	-2
2,011	Brazil	11,628	11,628	23.9	2
2,011	Chile	13,456	13,456	72.2	4
2,011	France	41,369	41,369	58.8	4
2,011	Italy	36,228	36,228	55.1	1
2,011	United Kingdom	39,729	39,729	62.4	-3

2.7 Merge



```
meta = readxl::read_excel('./data/datos_wb.xlsx', sheet='2')
bd = left_join(bd, meta, by=c('ccode')) # clave de union character no suele ser la mejor
```

year	cname	region
2011	Argentina	Latin America & Caribbean
2011	Brazil	Latin America & Caribbean
2011	Chile	Latin America & Caribbean
2011	France	Europe & Central Asia
2011	Italy	Europe & Central Asia
2011	United Kingdom	Europe & Central Asia

2.8 Variables: group_by, mutate

```
# Si quiero usar una función propia
demean = function(x) {x - mean(x, na.rm = TRUE)}
bd = bd %>%
  mutate(open = exports + imports) %>%
  dplyr::select(ccode, year, region, gdp_pc2017, credit_ps, inv, open) %>%
  arrange(ccode, year) %>%
  group_by(ccode) %>%
  mutate(obs = seq(1:length(ccode)),      # igual con row_number()
         gdp_gr = 100 * (gdp_pc2017 / dplyr::lag(gdp_pc2017, 1) - 1),
         credit_ps_mean = mean(credit_ps, na.rm = TRUE),
         dev = ifelse(region=='Latin America & Caribbean', 0, 1),
         gdp_dem = demean(gdp_pc2017)) %>%
  ungroup()
head(bd[c('ccode', 'dev', 'year', 'gdp_pc2017', 'gdp_gr', 'gdp_dem')],10)
```

```
## # A tibble: 10 x 6
##   ccode   dev year gdp_pc2017 gdp_gr  gdp_dem
##   <chr> <dbl> <dbl>      <dbl>  <dbl>    <dbl>
## 1 ARG      0  2011     24648.  NA     1451.
## 2 ARG      0  2012     24119. -2.15    922.
## 3 ARG      0  2013     24424.  1.27   1227.
## 4 ARG      0  2014     23550. -3.58    353.
## 5 ARG      0  2015     23934.  1.63    737.
## 6 ARG      0  2016     23190. -3.11   -7.58
## 7 ARG      0  2017     23597.  1.76    400.
## 8 ARG      0  2018     22759. -3.55   -438.
## 9 ARG      0  2019     22064. -3.06  -1133.
## 10 ARG     0  2020     19687. -10.8   -3511.
```

```
# case_when() permite evaluar más de 2 alternativas
# En el caso de tener solo 2 se puede usar la función ifelse()
```

```
df <- tibble(
  a = seq(1,5)
)
df = df %>% mutate(b = case_when(a <= 2 ~ 1,
                                    a > 2 & a <= 4 ~ 2,
                                    TRUE ~ as.double(a)))
df
```

```
## # A tibble: 5 x 2
##   a     b
##   <int> <dbl>
## 1 1     1
## 2 2     1
## 3 3     2
## 4 4     2
## 5 5     5
```

2.9 Guardar datos

```
bd = bd %>% select(ccode, year, region, gdp_gr, credit_ps, inv, open)
save(bd, file="datos_wb.rda")
```

ccode	year	region	gdp_gr	credit_ps	inv	open
ARG	2018	Latin America & Caribbean	-3.6	NA	14.7	31.2
ARG	2019	Latin America & Caribbean	-3.1	NA	13.5	32.6
ARG	2020	Latin America & Caribbean	-10.8	NA	13.4	30.5
BRA	2018	Latin America & Caribbean	1.0	60.2	15.1	28.9
BRA	2019	Latin America & Caribbean	0.7	62.6	15.3	28.5
BRA	2020	Latin America & Caribbean	-4.7	70.2	16.4	32.4
GBR	2018	Europe & Central Asia	0.6	134.6	17.8	63.0
GBR	2019	Europe & Central Asia	0.8	133.5	18.0	63.4
GBR	2020	Europe & Central Asia	-10.3	146.4	17.6	55.1
ITA	2018	Europe & Central Asia	1.1	76.7	17.8	60.3
ITA	2019	Europe & Central Asia	1.5	74.3	18.0	60.1
ITA	2020	Europe & Central Asia	-8.6	83.6	17.8	55.3

2.10 Valores missing

Son tratados como los valores más grandes de todos pero el `replace` no los considera. Notar la diferencia de comportamiento entre este último y el `arrange()`.

```
df2 <- tibble(
  a = sample(1:5, 5, replace = F),
  b = seq(5,1),
)
df2

## # A tibble: 5 x 2
##      a     b
##  <int> <int>
## 1     1     5
## 2     3     4
## 3     2     3
## 4     5     2
## 5     4     1

df2 = df2 %>% mutate(b = ifelse(b == 2, NA, b))
df2 = df2 %>% mutate(c = ifelse(b > 3, 0, b))
df2 = df2 %>% arrange(b)
df2

## # A tibble: 5 x 3
##      a     b     c
##  <int> <int> <dbl>
## 1     1     5     0
## 2     3     4     0
## 3     2     3     0
## 4     5     2     0
## 5     4     1     0
```

```
## 2   2   3   3
## 3   3   4   0
## 4   1   5   0
## 5   5   NA  NA
```

2.10.1 Eliminar valores missing

Se debe tener presente que se elimina la fila completa, por lo tanto, antes de descartarlos hay considerar si los valores *missing* son aleatorios o contienen algo de información.

```
# Volvemos a la base de WB. Recordamos la estructura
str(bd)
```

```

## tibble [60 x 7] (S3: tbl_df/tbl/data.frame)
## $ ccode      : chr [1:60] "ARG" "ARG" "ARG" "ARG" ...
## $ year       : num [1:60] 2011 2012 2013 2014 2015 ...
## $ region     : chr [1:60] "Latin America & Caribbean" "Latin America & Ca
## $ gdp_gr     : num [1:60] NA -2.15 1.27 -3.58 1.63 ...
## $ credit_ps  : num [1:60] 14 15.2 15.7 13.8 14.4 ...
## $ inv        : num [1:60] 17.2 15.9 16.3 16 15.6 ...
## $ open        : num [1:60] 35.2 30.5 29.3 28.4 22.5 ...

summary(bd[,1:4])

##      ccode           year        region        gdp_gr
## Length:60   Min.   :2011   Length:60   Min.   :-10.7750
## Class :character 1st Qu.:2013   Class :character 1st Qu.: -2.7656
## Mode  :character Median :2016   Mode  :character Median : 0.7100
##                   Mean   :2016
##                   3rd Qu.:2018
##                   Max.   :2020
##                   NA's   :6

# cuenta valores missing de
# Crecimiento del PIB y Credito al SPriu.
sum(is.na(bd$gdp_gr))

## [1] 6

sum(is.na(bd$credit_ps))

## [1] 4

```

```
sum(ifelse(is.na(bd$gdp_gr&bd$credit_ps),1,0))

## [1] 10

bd1 = na.omit(bd)
nrow(bd1)

## [1] 50

rm('bd1')
```

2.11 Loop

```
set.seed(1234)
df <- tibble(
  a = runif(100, min=0, max=100),
  b = rnorm(100, 0, 1),
  c = rnorm(100, mean=5, sd=3),
)
head(df,3)
```

```
## # A tibble: 3 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1 11.4 -1.81  3.87
## 2 62.2 -0.582  5.29
## 3 60.9 -1.11  9.92
```

```
# Reemplazar variables existentes
vars = names(df)
for (v in vars) {
  df[v] = df[v] * 100
}

head(df,3)
```

```
## # A tibble: 3 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1 1137. -181.  387.
## 2 6223. -58.2  529.
## 3 6093. -111.  992.
```

```
# Generar variables nuevas (estilo Stata, no se usa en R)
# Dividir las dos ultimas (b y c) por la primera (a)
set.seed(1234)
df1 <- tibble(
  a = rep(2, 100),
  b = rnorm(100, 0, 1),
  c = rnorm(100, 5, 3),
)
vars = names(df1[2:length(df1)])
for (v in vars) {
  df1[paste0(v, '_a')] = df1[v] / df1[['a']]
}
head(df1,3)
```

```
## # A tibble: 3 x 5
##       a     b     c   b_a   c_a
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     2 -1.21   6.24 -0.604  3.12
## 2     2  0.277  3.58  0.139  1.79
## 3     2  1.08   5.20  0.542  2.60
```

2.12 Pivot (Reshape)

country	year	key	value	country	year	cases	population
Afghanistan	1999	cases	745	Afghanistan	1999	745	19987071
Afghanistan	1999	population	19987071	Afghanistan	2000	2666	20595360
Afghanistan	2000	cases	2666	Brazil	1999	37737	172006362
Afghanistan	2000	population	20595360	Brazil	2000	80488	174504898
Brazil	1999	cases	37737	China	1999	212258	1272915272
Brazil	1999	population	172006362	China	2000	213766	1280428583
Brazil	2000	cases	80488				
Brazil	2000	population	174504898				
China	1999	cases	212258				
China	1999	population	1272915272				
China	2000	cases	213766				
China	2000	population	1280428583				

```
# Long
bdAR = bd %>%
  filter(ccode=='ARG') %>%
  select(year, credit_ps, inv) %>%
  pivot_longer(cols=-year, names_to="Var", values_to="Val") %>%
  arrange(year, desc(Var))    # Sort
```

year	Var	Val
2011	inv	17.2
2011	credit_ps	14.0
2012	inv	15.9
2012	credit_ps	15.2
2013	inv	16.3
2013	credit_ps	15.7

```
# Wide
bdAR2 = bdAR %>%
  filter(Var=='credit_ps', year <=2015) %>%
  mutate(ccode = 'ARG') %>%
  pivot_wider(id_cols=ccode, names_from=year, values_from=Val) %>%
  rename_with(~paste0("CREDIT", 2011:2015), where(is.numeric))
```

ccode	CREDIT2011	CREDIT2012	CREDIT2013	CREDIT2014	CREDIT2015
ARG	14	15.2	15.7	13.8	14.4

```
# Otra forma de llevar a long starts_with()
bdAR3 = bdAR2 %>%
  pivot_longer(cols = starts_with("CREDIT"), names_to="Var", values_to="Val") %>%
  separate(Var, c("V", "year"), sep = 6)
# unite() para concatenar variables
```

ccode	V	year	Val
ARG	CREDIT	2011	14.0
ARG	CREDIT	2012	15.2
ARG	CREDIT	2013	15.7
ARG	CREDIT	2014	13.8
ARG	CREDIT	2015	14.4

2.13 Row bind (Append)

```
bdAR = bd %>%
  filter(ccode=="ARG",
         year>=2018) %>%
  select(year, ccode, gdp_gr)
bdBR = bd %>%
  filter(ccode=="BRA",
         year>=2018) %>%
  select(year, ccode, gdp_gr)
bdARBR = rbind(bdAR, bdBR)
# Ver bind_rows() de tidyverse
```

year	ccode	gdp_gr
2018	ARG	-3.6
2019	ARG	-3.1
2020	ARG	-10.8
2018	BRA	1.0
2019	BRA	0.7
2020	BRA	-4.7

2.14 Strings

Se presentan algunas funciones básicas. Para más alternativas ver el paquete [stringr](#).

```
words = c('uno', 'dos', 'tres')
length(words) # cuenta los elementos del vector
```

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

```
str_length(words)
```

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

```
words = gsub('tres', 'cinco', words)
words
```

```
## [1] "uno"    "dos"    "cinco"
```

```
words = str_replace(words, 'uno', 'diez')
words
```

```
## [1] "diez"   "dos"    "cinco"
```

```
hi = 'hola'
nchar(hi)
```

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

2.15 Fechas

Formato de fechas.

Year	Month	Day
%Y (4 dígitos)	%m (2 dígitos)	%d (2 dígitos)
%y (2 dígitos)	%b (nombre “Jan”)	
	%B (nombre “January”)	

Nota: %y 00-69 hace referencia a 2000-2069 y, 70-99 indica 1970-1999.

```
library(zoo)
library(lubridate)
datos = readxl::read_excel('./data/datos_ts.xlsx', sheet='datos')
datos$fecha = as.Date(datos$fecha, format = '%Y-%m-%d')
datos$fecha2 = datos$fecha + days(15)
datos$fecha3 = floor_date(datos$fecha2, 'month')
datos$mes = month(datos$fecha)
datos$year = year(datos$fecha)
datos = datos[,c(1,5,6,7,8,2,3,4)] # reordena la base de datos
```

fecha	fecha2	fecha3	mes	year	ipc	tcn	emae_sa
2015-01-01	2015-01-16	2015-01-01	1	2015	57.6	8.6	144.2
2015-02-01	2015-02-16	2015-02-01	2	2015	58.5	8.7	148.0
2015-03-01	2015-03-16	2015-03-01	3	2015	59.5	8.8	147.7
2015-04-01	2015-04-16	2015-04-01	4	2015	60.9	8.9	149.8
2015-05-01	2015-05-16	2015-05-01	5	2015	62.2	9.0	149.7
2015-06-01	2015-06-16	2015-06-01	6	2015	63.0	9.1	150.6

2.15.1 Manipulación de fechas

Utilizamos el paquete `zoo`.

```
datosq <- readxl::read_excel('./data/datos_ts.xlsx', sheet='trim') %>%
  mutate(fecha=as.Date(as.yearqtr(paste(year, trim)), format="%Y %q"))
```

year	trim	ipc	tcn	emae_sa	fecha
2015	1	57.6	8.6	144.2	2015-01-01
2015	2	60.9	8.9	149.8	2015-04-01
2015	3	64.3	9.2	150.8	2015-07-01
2015	4	67.7	9.5	149.4	2015-10-01
2016	1	74.7	13.9	147.8	2016-01-01
2016	2	85.5	14.3	145.0	2016-04-01

2.16 Análisis de datos

2.16.1 Tablas

Los valores NA afectan a todas las estadísticas. Opción na.rm = F / T.

```
bd %>% summarise(
  credit_ps_media = mean(credit_ps),
  inv_max = max(inv),
  open_min = min(open)
)
```

```
## # A tibble: 1 x 3
##   credit_ps_media inv_max open_min
##             <dbl>    <dbl>     <dbl>
## 1             NA     24.9     22.5
```

2.17 group_by, summarise

```
tab = bd %>%
  dplyr::select_if(is.numeric) %>%
  pivot_longer(everything(), names_to = 'Variable', values_to = 'Value') %>%
  group_by(Variable) %>%
  summarise(
    Obs = n(),
    Media = mean(Value, na.rm = T),
    Mediana = median(Value, na.rm = T),
    SD = sd(Value, na.rm = T),
    Min = min(Value, na.rm = T),
    Max = max(Value, na.rm = T)) %>%
  ungroup()
tab
```

```
## # A tibble: 5 x 7
##   Variable   Obs     Media   Mediana     SD     Min     Max
##   <chr>     <int>     <dbl>     <dbl> <dbl>     <dbl>    <dbl>
## 1 credit_ps     60     89.7     94.6   38.9    13.7    171.
## 2 gdp_gr       60    -0.678     0.710   3.34   -10.8     4.31
## 3 inv          60     18.8     17.8    3.19    13.4    24.9
## 4 open         60     49.1     56.2    15.7    22.5    72.2
## 5 year        60  2016.    2016.    2.90  2011.    2020
```

```

tab = bd %>%
  filter(year>=2015) %>%
  group_by(year, region) %>%
  summarise(
    inv_sum = sum(inv, na.rm = T),
    inv_sd = sd(inv, na.rm = T),
    credit_ps_p50 = median(credit_ps, na.rm = T),
    obs = n()
  ) %>%
  arrange(year) %>%
  ungroup()
tab

## # A tibble: 12 x 6
##   year   region      inv_sum   inv_sd credit_ps_p50   obs
##   <dbl> <chr>        <dbl>     <dbl>        <dbl>     <int>
## 1 2015 Europe & Central Asia     55.7     2.55        95.1      3
## 2 2015 Latin America & Caribbean 57.2     4.24        66.8      3
## 3 2016 Europe & Central Asia     56.7     2.54        97.4      3
## 4 2016 Latin America & Caribbean 52.5     4.57        62.2      3
## 5 2017 Europe & Central Asia     58.0     2.76       101.       3
## 6 2017 Latin America & Caribbean 50.7     3.57        59.5      3
## 7 2018 Europe & Central Asia     58.6     2.93       104.       3
## 8 2018 Latin America & Caribbean 51.2     3.81        88.6      3
## 9 2019 Europe & Central Asia     59.6     3.26       108.       3
## 10 2019 Latin America & Caribbean 51.8     4.98        93.2      3
## 11 2020 Europe & Central Asia     58.2     3.02       124.       3
## 12 2020 Latin America & Caribbean 50.7     3.78        70.2      3

tab = tab %>% mutate(region = str_replace(region, c('Europe & Central Asia', 'Latin Am'), 'EU')) %>%
  pivot_longer(cols=-c(year, region), names_to='Var', values_to='Val') %>%
  unite(id,region, Var) %>%
  pivot_wider(id_cols=year, names_from=id, values_from=Val)
tab

## # A tibble: 6 x 9
##   year EU_inv_sum EU_inv_sd EU_credit_ps_p50 EU_obs LA_inv_sum LA_inv_sd
##   <dbl>     <dbl>     <dbl>           <dbl>     <dbl>     <dbl>     <dbl>
## 1 2015      55.7     2.55        95.1       3       57.2     4.24
## 2 2016      56.7     2.54        97.4       3       52.5     4.57
## 3 2017      58.0     2.76       101.       3       50.7     3.57
## 4 2018      58.6     2.93       104.       3       51.2     3.81
## 5 2019      59.6     3.26       108.       3       51.8     4.98
## 6 2020      58.2     3.02       124.       3       50.7     3.78
## # ... with 2 more variables: EU_credit_ps_p50 <dbl>, EU_obs <dbl>

```

2.18 Vector de resultados

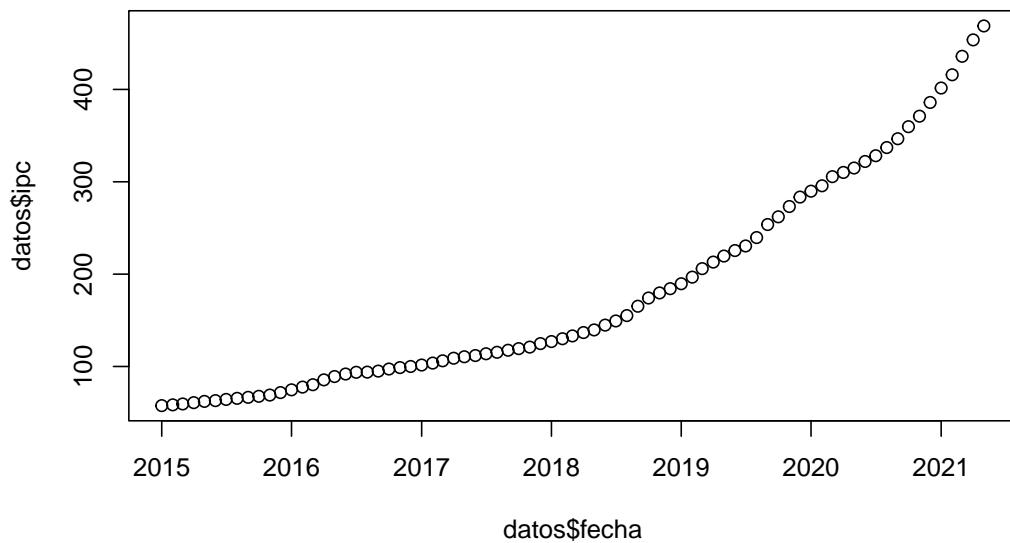
```
set.seed(1234)
df <- tibble(
  a = runif(100, min=0, max=100),
  b = rnorm(100,0,1),
  c = rnorm(100, mean=5, sd=3),
)

output <- vector("double", ncol(df)) # 1. vector de resultados (vacío)
for (i in seq_along(df)) {           # 2. secuencia
  output[[i]] <- mean(df[[i]])       # 3. cuerpo
}
output

## [1] 43.74972619  0.07975381  5.40972645
```

2.19 Gráficos

```
plot(datos$fecha, datos$ipc)
```



```
plot(datos$fecha, log(datos$ipc), type= 'l', col = 'red', xlab ='Mes', ylab ='Log(IPC)
```



2.20 GGPlot

Grammar of Graphics. Ver más detalles en ([Wickham et al., 2016](#)).

```
#SINTAXIS
# ggplot(data = <DATA>) +
#   <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
# se agregan layers (point, line, etc.)
# aes() "aesthetic" define la estética del gráfico
library(ggplot2)
datos1 = datos %>%
  select(fecha, ipc)

ggplot(datos1, aes(x=fecha, y=ipc)) +
  geom_line(color = 'steelblue2', size = 1.2) +
  theme_minimal() +
  labs(title = 'Indice de precios al consumidor', x = 'Mes', y ='IPC') +
  theme(legend.position = 'none') +
  NULL
```



```
# Selecciona el mejor de cada clase de acuerdo al consumo en highway
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)

g = ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(se = FALSE) +
  geom_label(aes(label = model), data = best_in_class, nudge_y = 2, alpha = 0.5) +
  labs(title = "Fuel efficiency generally decreases with engine size",
       subtitle = "Two seaters (sports cars) are an exception because of their light weight",
       x = "Engine displacement (L)",
       y = "Highway fuel economy (mpg)",
       colour = "Car type",
       caption = "Data from fueleconomy.gov") +
  NULL
g
```

Fuel efficiency generally decreases with engine size
 Two seaters (sports cars) are an exception because of their light weight



2.21 Guardar un gráfico

```
work = 'C:/Users/msang/'
filesave = paste0(work,'ts.png')
ggsave(filesave, g, width=10, height=8)
```

Capítulo 3

Conceptos generales

El **objetivo** de la Ciencia de Datos es **preparar, analizar y aprender** “algo” de los **datos**. Si se dispone de una variable *output*¹ y otras *input*² el aprendizaje se denomina **supervisado**, si sólo hay *inputs* el aprendizaje es **no supervisado**.

Dentro aprendizaje supervisado podemos distinguir la **predicción**, cuando la variable *output* es cuantitativa, de la **clasificación** donde la la variable *output* es discreta/categórica (ej. 0/1).³ Por su parte, el análisis no supervisado busca relaciones y estructura dentro de los datos (ej. distinguir *clusters*/grupos de clientes para promociones publicitarias).

3.1 Estimacion

Supongamos que se quiere estudiar la relación entre el gasto en publicidad a través de diversos canales como televisión, radio, diarios (*inputs*) y las ventas en distintos mercados (*output*).

$$Y = f(X) + \epsilon \tag{3.1}$$

donde f es una función desconocida de (X_1, X_2, X_3) y ϵ es un término de error aleatorio independiente de X con media igual a 0.

En esencia, el aprendizaje estadístico se refiere a un conjunto de enfoques para estimar f .

3.2 Prediccion

Supongamos que se dispone de datos de variables independientes por no de la dependiente, en ese caso, dado que el error en promedio es 0 podríamos predecir Y utilizando:

¹También variable dependiente o variable explicada.

²También variables independientes o variables explicativas.

³En general el interés no esta puesto en realizar inferencia/análisis condicional.

$$\hat{Y} = \hat{f}(X) \quad (3.2)$$

donde \hat{f} representa nuestra estimación de f y \hat{Y} representa la predicción de Y . En este contexto, \hat{f} a menudo se trata como una **caja negra**, en el sentido que no importa la forma exacta de \hat{f} , siempre que produzca predicciones precisas de Y .

La precisión con la que \hat{Y} se acerca a Y depende de dos cantidades, el error *reducible* y el *irreducible*. En general, \hat{f} no será una estimación perfecta de f , y esta inexactitud introducirá un error que es reducible porque potencialmente podemos mejorar la precisión de \hat{f} usando la técnica de aprendizaje estadístico más apropiada para estimar f . Sin embargo, si fuera posible estimar f exactamente de manera que la respuesta estimada $\hat{Y} = f(X)$, nuestra predicción todavía tendría algún error dado que Y también es función de ϵ , que por definición, no se puede predecir usando X . Por lo tanto, la variabilidad asociada con ϵ también afecta la precisión de nuestras predicciones. Esto se conoce como el error irreducible, porque no importa qué tan bien estimemos f , no puede reducir el error introducido por ϵ .

El término de error ϵ puede contener variables no observables que son útiles para predecir Y , por lo tanto, f no puede usarlos para su predicción. A partir de las ecuaciones (3.1) y (3.2) puede expresarse:

$$E(Y - \hat{Y})^2 = E[f(X) + \epsilon - \hat{f}(X)]^2 \quad (3.3)$$

$$= \underbrace{[f(X) - \hat{f}(X)]^2}_{\text{Reducible}} + \underbrace{Var(\epsilon)}_{\text{Irreducible}} \quad (3.4)$$

donde de $E(Y - \hat{Y})^2$ representa el promedio, o valor esperado, de la diferencia entre el valor predicho y el valor real de Y elevado al cuadrado (diferencia por exceso y defecto ponderan igual), y $Var(\epsilon)$ representa la varianza asociada al término de error ϵ .

El **foco de este curso** está en las técnicas para estimar f con el objetivo de minimizar el error reducible. Es importante tener en cuenta que el error irreducible siempre proporcionará un límite superior en la precisión de nuestra predicción de Y que en la práctica casi siempre es desconocido.⁴

3.3 Inferencia

En este caso el interés está en comprender la asociación entre Y y X_1, \dots, X_p . Se busca estimar f , pero el objetivo no es necesariamente hacer predicciones sobre Y . Ahora \hat{f} **no** puede ser tratada como una **caja negra**, porque se necesita conocer su forma exacta. Así se busca determinar (entre otras cosas):

⁴Volveremos sobre este tema en el Capítulo 4.

- Qué variables se deben incluir en el modelo
- Cómo es la relación entre la variable explicada y cada predictor
- Si la relación se puede aproximar con un modelo lineal o uno más complejo

3.4 Metodos parametricos

Se realiza en dos etapas:

- Asumir una forma funcional (**modelo**, por ejemplo lineal)

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p \quad (3.5)$$

- Estimar los parámetros (**método**, por ejemplo Mínimos Cuadrados Ordinarios - MCO-)

Si bien el problema se reduce a estimar $p + 1$ parámetros, la desventaja es que la forma funcional elegida puede diferir de la verdadera f .

3.5 Metodos no parametricos

No realizan supuestos sobre la forma funcional de f sino que tratan de buscar una estimación que se acerque lo más posible a los datos sin ser ni demasiado tosco ni demasiado ondulado.

Este enfoque puede tener una gran ventaja sobre los métodos paramétricos: al evitar el supuesto de una forma funcional particular para f , tiene el potencial para adaptarse con precisión a una gama más amplia de posibles formas para f . Cualquier enfoque paramétrico tiene la posibilidad de que la forma funcional utilizada para estimar f sea muy diferente de la verdadera f , en cuyo caso el resultado modelo no se ajustará bien a los datos. El costo es que se necesitan más datos para estimar.

3.6 Evaluacion de la precision del modelo

Ningún método domina al resto sobre todas las bases de datos posibles. En un *set* de datos en particular, un método específico puede funcionar mejor, pero algún otro método lo puede superar con otra base de datos. Por lo tanto, en cada caso se debe decidir qué método produce los mejores resultados.

3.6.1 Calidad del ajuste

Para evaluar el desempeño de un método de aprendizaje estadístico en una base de datos dada, se necesita alguna forma de medir qué tan bien sus predicciones coinciden con los datos observados. Es decir, se necesita cuantificar el grado en el cual el valor pronosticado para una observación dada está cerca de el verdadero valor de respuesta para esa observación. En el escenario de regresión, la medida más utilizada es el error medio cuadrático (*EMC*):

$$EMC = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{f}(x_i))^2 \quad (3.6)$$

donde $\hat{f}(x_i)$ es la predicción que hace \hat{f} sobre la observación i . El *EMC* será pequeño si las respuestas predichas están muy cerca de las respuestas verdaderas y será grande si para algunas observaciones difieren demasiado.

El *EMC* en (3.6) se calcula usando los **datos de entrenamiento** (*training*) que se usaron para estimar el modelo, por lo que debería denominarse con mayor precisión *EMC* de entrenamiento. Pero en general, no nos interesa realmente qué tan bien funciona el método sobre los datos de entrenamiento. Más bien, estamos interesados en la precisión de las predicciones que obtenemos cuando aplicamos nuestro método a los **datos de *test*** que no fueron visto antes (datos no utilizados para entrenar el modelo). Es decir, se busca elegir el método que produzca el menor *EMC* en la muestra de *test*.

$$Prom(y_0 - \hat{f}(x_0))^2 \quad (3.7)$$

el error de predicción cuadrático promedio para estas observaciones de *test* (y_0, x_0).

¿Qué sucede si se elige en base al *EMC* de *training* (3.6)? No hay garantía de que el método con el *EMC* de entrenamiento más bajo también tenga el *EMC* de *test* más bajo.

El panel de la izquierda de la Figura 3.1 muestra la verdadera f dada por la curva negra. Las curvas naranja, azul y verde ilustran tres posibles estimaciones de f obtenidas utilizando métodos con distintos niveles de flexibilidad. La línea naranja es el ajuste de regresión lineal, que es relativamente inflexible. Las curvas azul y verde se produjeron usando *splines* con diferentes niveles de suavidad. Es claro que a medida que aumenta el nivel de flexibilidad, las curvas se ajustan mejor a los datos observados. La curva verde es la más flexible y coincide muy bien con los datos; sin embargo, se observa que se ajusta mal a la verdadera f (en negro) porque es demasiado ondulada. Cambiando el nivel de flexibilidad del *spline* se pueden producir ajustes diferentes para estos datos.

En el panel de la derecha de la Figura 3.1 la curva gris muestra el *EMC* de **entrenamiento** en función de la flexibilidad, o más formalmente los grados de libertad (resume la flexibilidad de una curva), para una serie de *splines*. Los cuadrados naranja, azul y verde indican los *EMC* asociados con las curvas correspondientes en el panel izquierdo.

El *EMC* de entrenamiento disminuye monótonamente a medida que aumenta la flexibilidad. Dado que la verdadera f es no lineal, el ajuste lineal naranja no es lo suficientemente flexible para estimar bien f . La curva verde tiene el *EMC* de entrenamiento más bajo de los tres métodos, ya que corresponde a la más flexible de las tres curvas.

En este ejemplo, se conoce la verdadera función f , por lo que también se puede calcular el *EMC* de *test* (en general f es desconocida, por lo que esto no es posible). El *EMC* de *test* se muestra usando la curva **roja** en el panel derecho de la Figura 3.1. Como con el *EMC* de entrenamiento, el *EMC* de *test* disminuye inicialmente a medida que el nivel de flexibilidad aumenta. Sin embargo, en algún momento el *EMC* de *test* se nivela y luego empieza a aumentar. En consecuencia, las curvas naranja y verde tienen un *EMC* de *test* alto. La curva azul minimiza el *EMC* de *test*, dado que visualmente parece estimar mejor f en el panel izquierdo. La línea discontinua horizontal indica $Var(\epsilon)$, el error irreducible en la ecuación de $E(Y - \hat{Y})^2$, que corresponde al menor alcanzable por el *EMC* de *test* entre todos los métodos posibles. Por lo tanto, el suavizado de *spline* representado por la curva azul está cerca del óptimo.

Importante. En el panel de la derecha de la Figura 3.1, a medida que la flexibilidad del método de aprendizaje aumenta, se observa una disminución monótona en el *EMC* de entrenamiento y una forma de U en el *EMC* de *test*. Esta es una propiedad fundamental de aprendizaje estadístico que se mantiene independientemente de la base de datos particular en cuestión e independientemente del método estadístico que se utilice.

Cuando un método dado produce un *EMC* de entrenamiento pequeño pero un *EMC* de *test* grande, se dice que está haciendo *overfitting*/sobreajustando los datos. Esto sucede porque nuestro aprendizaje estadístico está trabajando demasiado para encontrar patrones en los datos de entrenamiento, y puede estar detectando algunos patrones que son causados por casualidad en lugar de por las verdaderas propiedades de la función desconocida f .

Overfitting se refiere específicamente al caso en el que un modelo menos flexible podría haber producido un menor error de predicción en *test*.

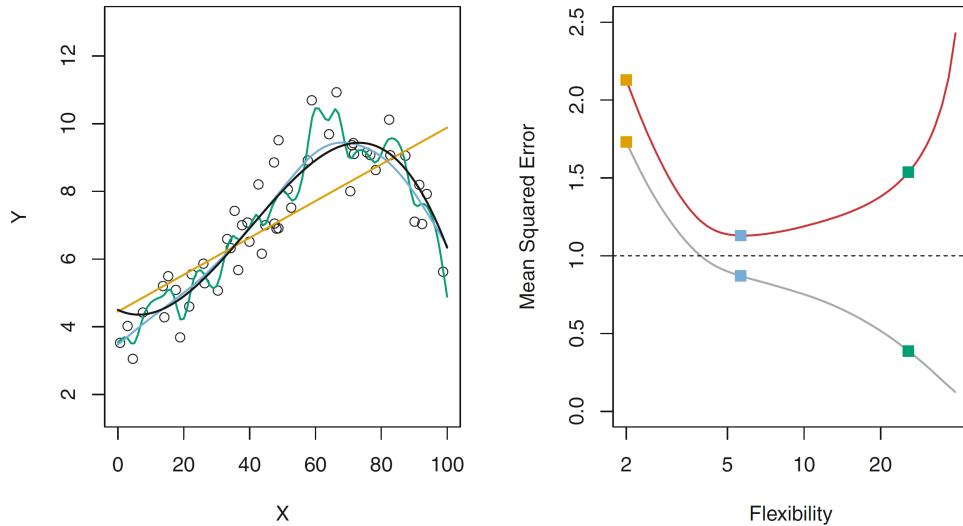


Figura 3.1: Datos en curva y ECM

La Figura 3.2 proporciona otro ejemplo en el que la verdadera f es aproximadamente lineal por lo que este tipo de modelos obtienen el menor EMC en *test* (curva roja en el panel derecho de la Figura 3.2).

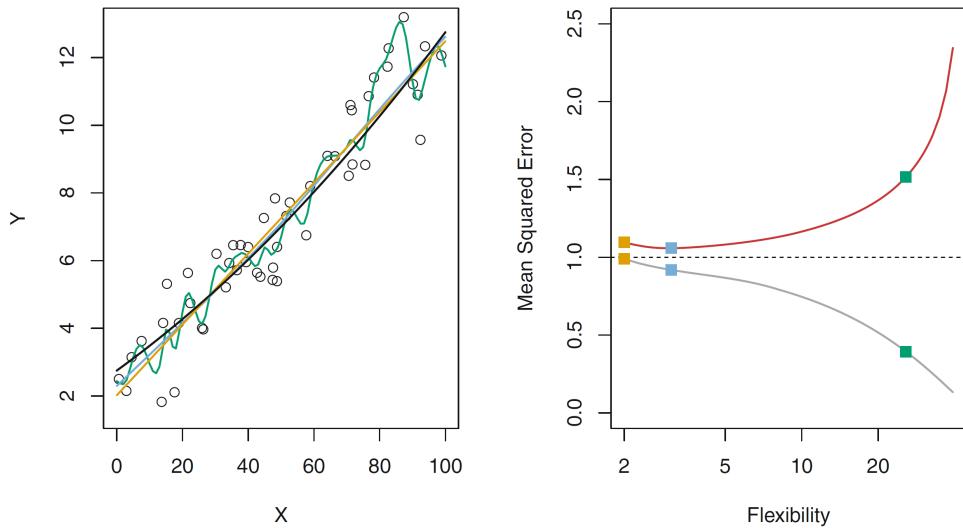


Figura 3.2: Datos lineales y EMC

3.6.2 Trade-off Sesgo-Varianza

La Figura 3.3 muestra el *trade-off* Sesgo - Varianza intuitivamente.

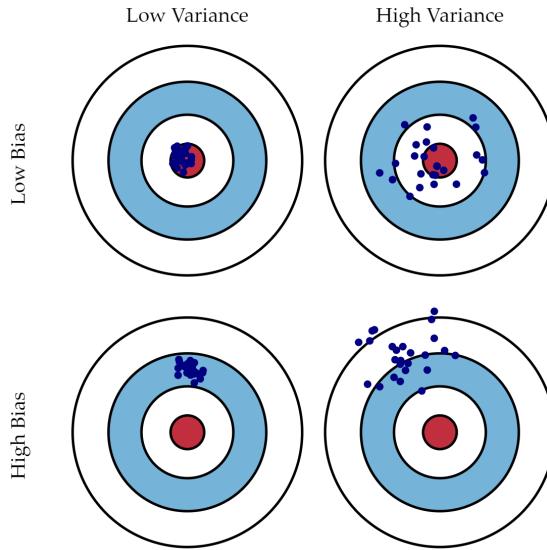


Figura 3.3: Estimación y EMC

Fuente: [Scott Fortmann-Roe](#)

La forma de U observada en las curvas EMC de *test* es el resultado de dos propiedades que compiten en los métodos de aprendizaje estadístico. El EMC de *test* esperado, para

un valor dado x_0 , puede descomponerse en la suma de tres cantidades fundamentales: la varianza de $\hat{f}(x_0)$, el sesgo al cuadrado de $\hat{f}(x_0)$ y la varianza del error ϵ .

$$E(y_0 - \hat{f}(x_0))^2 = Var(\hat{f}(x_0)) + [Sesgo(\hat{f}(x_0))]^2 + Var(\epsilon) \quad (3.8)$$

donde $E(y_0 - \hat{f}(x_0))^2$ el valor esperado de *EMC* de *test* en x_0 . Para minimizar el error de *test* esperado, se necesita seleccionar un método de aprendizaje estadístico que logre simultáneamente **baja varianza y bajo sesgo**.

La **varianza** se refiere al valor en que f cambiaría si se estimara utilizando una base de datos de entrenamiento diferente. **Sesgo** se refiere al error que se introduce al aproximar un problema de la vida real, que puede ser extremadamente complicado, por un modelo mucho más simple. Como regla general, a medida que se utilizan métodos más flexibles, la varianza aumenta y el sesgo disminuye. La tasa relativa de cambio de estas dos cantidades determina si el *EMC* de *test* aumenta o disminuye.

Los dos paneles de la Figura 3.4 ilustran la Ecuación (3.8) para los ejemplos en Figuras 3.1 y 3.2. En cada caso, la curva sólida azul representa el cuadrado del sesgo, para diferentes niveles de flexibilidad, mientras que la curva naranja corresponde a la varianza. La línea discontinua horizontal representa $Var(\epsilon)$, el error irreducible. Finalmente, la curva roja, corresponde al *EMC* de *test*, es la suma de estas tres cantidades.

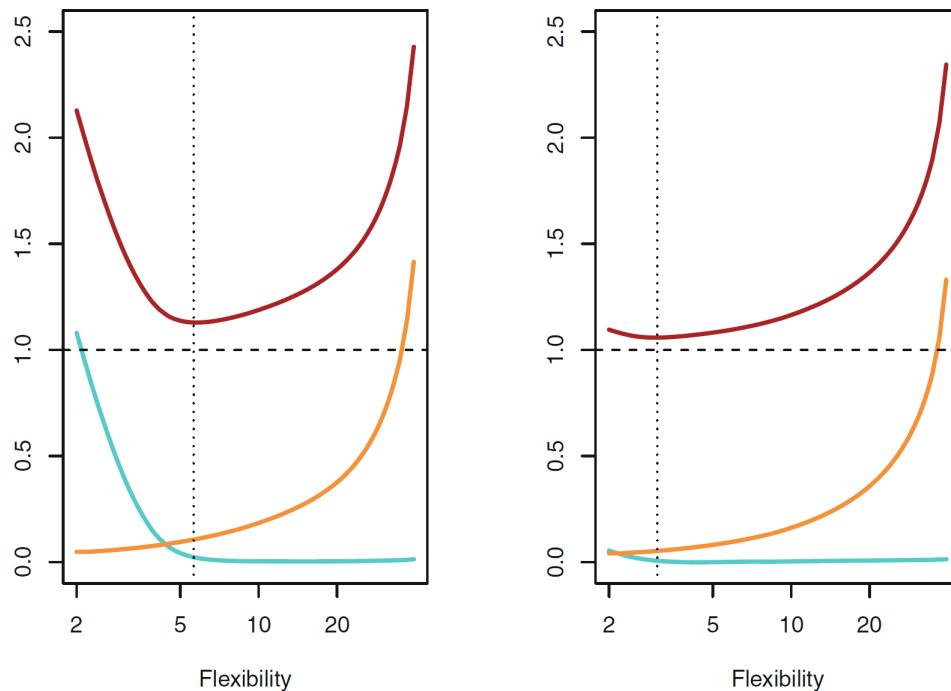


Figura 3.4: Estimacion y EMC

3.6.3 Clasificación

Muchos de los conceptos del contexto de regresión, como el *trade-off* sesgo-varianza, se transfieren al entorno de clasificación donde ahora y_i es cualitativa. El enfoque más común para cuantificar la precisión de la estimación \hat{f} es la **tasa de error** de entrenamiento, es decir, la proporción de errores que se cometen si aplicamos nuestra estimación \hat{f} a las observaciones de entrenamiento.

$$\frac{1}{n} \sum_{i=1}^n I(y_i \neq \hat{y}_i) \quad (3.9)$$

Aquí \hat{y}_i es la etiqueta de clase predicha para la i -ésima observación usando \hat{f} . Por lo tanto, $I(y_i \neq \hat{y}_i)$ es una variable indicadora que es igual a 0 si $y_i = \hat{y}_i$ ó 1 si $y_i \neq \hat{y}_i$, es decir, si la i -ésima observación fue clasificada correctamente o no por el método de clasificación.

La tasa de error de *test* asociada con un conjunto de observaciones de *test* de la forma (x_0, y_0) está dada por:

$$Prom(I(y_0 \neq \hat{y}_0)) \quad (3.10)$$

donde \hat{y}_0 es la etiqueta de clase predicha que resulta de aplicar el clasificador a la observación de *test* con predictor x_0 . Un buen clasificador es aquel para el cual el error de *test* (3.10) es el más pequeño.

3.6.3.1 Clasificador de Bayes

Es posible mostrar que bajo penalidad simétrica⁵ la tasa de error de *test* postulada en (3.10) se minimiza, en promedio, por un clasificador muy simple que asigna cada observación a la clase más probable, dados sus valores predictores. En otras palabras, se debería asignar una observación de *test* con vector predictor x_0 a la clase j para la cual (3.11) es mayor.

$$Pr(Y = j \mid X = x_0) \quad (3.11)$$

Es decir, en un problema donde sólo hay dos categorías el clasificador de Bayes predice la clase 1 si $Pr(Y = 1 \mid X = x_0) > 0.5$ y la clase 0 en caso contrario.

3.6.4 Matriz de confusión

⁵ ¿Útil para probabilidad de *default*?

		Observado	
		0	1
Predicción (decisión)	0	<i>VN</i>	<i>FN</i>
	1	<i>FP</i>	<i>VP</i>

VN: Verdadero Negativo; *FN*: Falso Negativo; *FP*: Falso Positivo; *VP*: Verdadero Positivo

Métricas para comparar modelos de clasificación. La **precisión** (*accuracy*) es la cantidad de predicciones correctas, la **sensibilidad** (*sensitivity*) es la proporción de verdaderos positivos y la **especificidad** (*specificity*) es la cantidad de *VN* identificados sobre el total de negativos.

$$\begin{aligned} \text{Precisión} &= \frac{VP + VN}{VP + VN + FP + FN} \\ \text{Sensibilidad} &= \frac{VP}{VP + FN} \\ \text{Especificidad} &= \frac{VN}{VN + FP} \end{aligned}$$

3.6.5 Curva ROC

El nombre viene de *receiver operating characteristics* (comunicación). Si se modifica el umbral $p_i > c$, cambian los resultados de la matriz de confusión. Por ejemplo, al estimar la probabilidad de *default*, para un banco podría resultar relativamente más costoso clasificar a un mal deudor como no *default* que a uno bueno como *default*. Entonces podría bajar el umbral $p_i > 0,3$ (asimétrico) para clasificar casos positivos afectando la tasa de error.

Si se define:

$$\begin{aligned} TPR &= VP/P \\ FPR &= FP/N \end{aligned}$$

La curva *ROC* representa la relación entre *true positive rate* (*TPR*) o Sensibilidad y *false positive rate* (*FPR*) o, expresado de otra manera, $(1 - \text{Especificidad})$ para todos los valores posibles de $c \in [0, 1]$. La curva *ROC* compara la proporción de verdaderos positivos con (el complemento de) la proporción de verdaderos negativos, es decir, mide la pureza alcanzada en cada categoría. De esta forma, permite medir capacidad predictiva y comparar modelos.

Casos extremos

- $c = 1$ todos clasificados como negativos ; $tpr = 0, fpr = 0$
- $c = 0$ todos clasificados como positivos ; $tpr = 1, fpr = 1$



Figura 3.5: Curva ROC

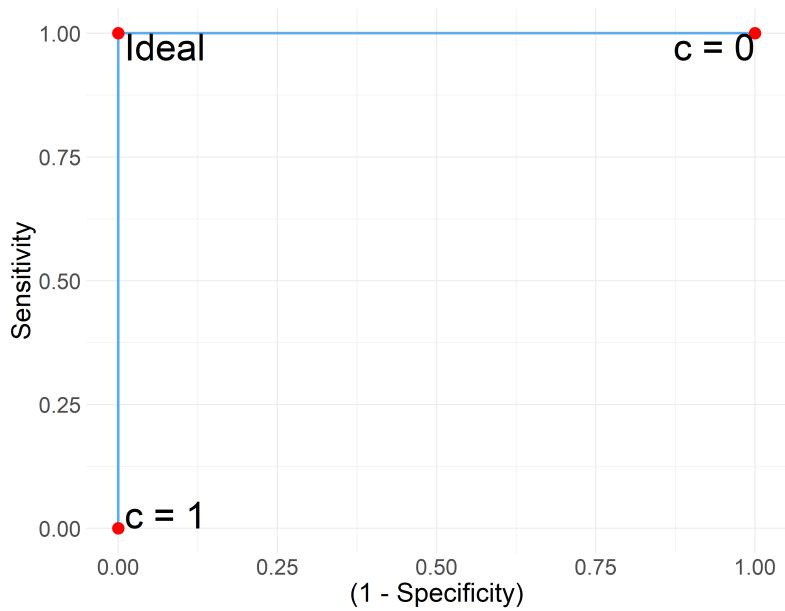


Figura 3.6: Curva ROC puntos importantes

AUC (o *AUROC*): área bajo la curva *ROC*. Cuán parecida es la curva *ROC* a la ideal, es decir cuánto *AUC* está más cerca de 1 mejor es el clasificador. Por su parte, un clasificador aleatorio debe tener un *AUC* = 0,5 (línea de 45°).

3.7 Cross Validation

Recordar la diferencia entre la tasa de error de *test* y la tasa de error de entrenamiento. El error de *test* es el error promedio que resulta de usar un método de aprendizaje estadístico para predecir la respuesta en una nueva observación, es decir, que no fue utilizada en el entrenamiento del método.

Modelos complejos predicen bien dentro de la muestra pero mal fuera de la misma (*overfit*) y nuestro interés está puesto en esta última.

Objetivo: buscar el nivel de complejidad óptimo para predecir fuera de la muestra.

Definición de pérdida:

- Regresión = $(Y - \hat{Y})^2$
- Clasificación = $1(Y \neq \hat{Y})$

k-Fold cross-validation

1. Dividir la muestra en K partes al azar.
2. Tomar $K - 1$ partes y estimar en modelo.
3. Calcular el error de predicción para los datos no utilizados.
4. Repetir para $k = 2, \dots, K$

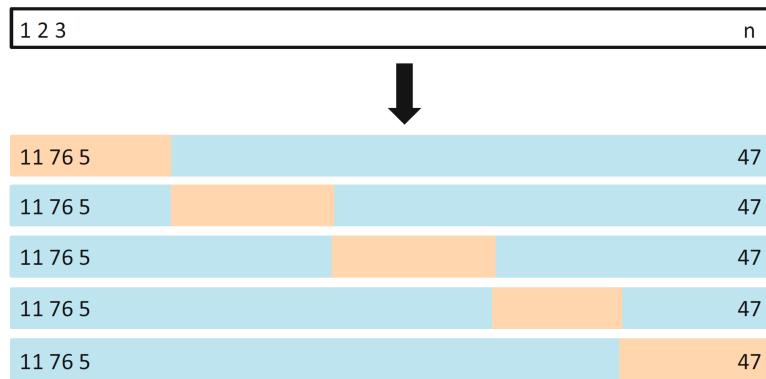


Figura 3.7: K-Fold cross-validation

La estimación por *cross-validation* del error de predicción es:

$$CV(\hat{f}) = \frac{1}{N} L(Y_i - \hat{Y}_{-k}(x_i)) \quad (3.12)$$

donde $\hat{Y}_{-k}(x_i)$ es la predicción hecha cuando la observación no fue usada para estimar. Cada observación se utiliza en dos roles: entrenamiento y *test*. De esta forma se estima el modelo K veces para construir el error de pronóstico.

Cross-validation para elección de modelos: Si α representa la complejidad de un modelo (por ejemplo el grado de un polinomio).

$$CV(\hat{f}, \alpha) = \frac{1}{N} L(Y_i - \hat{Y}_{-k}(x_i, \alpha)) \quad (3.13)$$

Computar $CV(\hat{f}, \alpha)$ para distintos valores de α y elegir el modelo que minimiza el error.⁶

3.8 Bootstrap

Bootstrap es una herramienta estadística que se puede utilizar para cuantificar la incertidumbre asociada con un estimador o un método de aprendizaje estadístico.

Dado Y_1, Y_2, \dots, Y_n iid $Y \sim (\mu, \sigma^2)$

Se quiere estimar la varianza de la media muestral $V(\bar{Y}) = \frac{\sigma^2}{n}$

Formula: $\frac{\hat{\sigma}^2}{n}$

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (Y_i - \bar{Y})^2 \quad (3.14)$$

Método sin fórmula

De los N datos originales y_1, y_2, \dots, y_N :

1. Tomar una muestra **con remplazo** de tamaño n (una observación puede entrar más de una vez y otra puede no entrar nunca).
2. Computar la media muestral de esta muestra.
3. Repetir B veces. Al terminar tendremos B estimaciones de la media.
4. Calcular la varianza de las B medias.

En términos generales

Dado Y_i con $i = 1, \dots, n$ y θ es una magnitud de interés

1. Tomar una muestra **con remplazo** de tamaño n (muestra *bootstrap*).
2. Computar $\hat{\theta}_j$, con $j = 1, \dots, B$.
3. Repetir B veces.
4. Calcular:

⁶Luego de seleccionar el modelo se estima con la muestra completa.

$$\hat{V}(\hat{\theta})_B = \frac{1}{B} \sum_{j=1}^B (\hat{\theta}_j - \bar{\hat{\theta}})^2 \quad (3.15)$$

Ejemplo:

```
set.seed(1234)
poblacion = rnorm(1000)

# Bootstrap con muestras de 300 y 10000 repeticiones:
muestra_boot = c()
for (i in 1:10000) {
  muestra = sample(poblacion, 300, replace=TRUE)
  muestra_boot = c(muestra_boot, mean(muestra))
}

# Media calculada con MUESTRA BOOTSTRAP
simulated_mean = mean(muestra_boot)

# Varianza de la MUESTRA BOOTSTRAP
simulated_var = sd(muestra_boot)^2

# Comparemos medias:
mean(poblacion); simulated_mean

## [1] -0.0265972

## [1] -0.02612666

# Comparemos varianza:
sd(poblacion)^2; simulated_var*300

## [1] 0.9946825

## [1] 0.9938018
```

3.9 Resumen

Como señala (Boehmke and Greenwell, 2020) abordar correctamente el análisis de *machine learning* significa utilizar estratégicamente los datos en procesos de aprendizaje y validación, preprocessar correctamente las variables explicativas y la variable de respuesta, ajustar los hiperparámetros y evaluar la *performance* del modelo.

La Figura 3.8 muestra gráficamente este proceso.



Figura 3.8: Proceso general de ML

Capítulo 4

Regresión lineal

En *machine learning* el objetivo principal no es estimar y hacer inferencia como en la econometría clásica sino hacer predicciones/clasificar. El modelo de regresión lineal es una herramienta útil para predecir cuando la variable de respuesta es cuantitativa.

Modelo lineal simple

$$y = \beta_0 + \beta_1 x_1 + u \quad (4.1)$$

donde u es un término de error aleatorio que captura todo lo que no puede representarse con este modelo simple (factores no observables, errores de medición, etc.).

Modelo lineal múltiple

En notación matricial:¹

$$Y = X\beta + u \quad (4.2)$$

donde la primera columna de X es la constante.

Método de Mínimos Cuadrados Ordinarios (*MCO*)

$$\hat{\beta} = \min \sum_{i=1}^n e_i^2 \quad (4.3)$$

$$\hat{\beta} = (X'X)^{-1}X'Y \quad (4.4)$$

Teorema de Gauss/Markov (TGM): Bajo los supuestos clásicos el estimador de *MCO* es el mejor dentro de los lineales e insesgados (MELI).

Definición: el R^2 mide la proporción de la variabilidad de Y explicada por el modelo. $R^2 \in [0, 1]$.

¹Para más detalles véase ([Wooldridge, 2012](#)) Apéndice E (página 807).

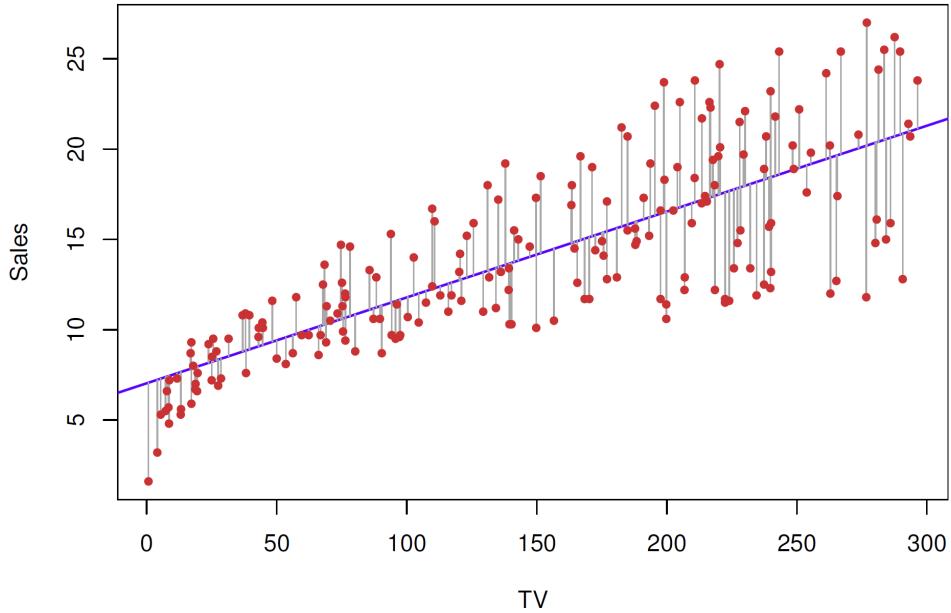


Figura 4.1: Modelo lineal y MCO

$$R^2 = 1 - \frac{SCR}{SCT} \quad (4.5)$$

Dado que la suma de cuadrados totales es igual a la suma de cuadrados explicada y la suma de cuadrados residual ($SCT = SCE + SCR$) y que SCT es una magnitud fija, por lo tanto, **MCO maximiza el R^2** .

4.1 Relacion entre estimacion optima y prediccion optima

Dado:

$$Y_i = X_i' \beta + u_i \quad (4.6)$$

con $i = 1, \dots, n$

La predicción de Y se define como:

$$\hat{Y}_i \equiv X_i' \hat{\beta} \quad (4.7)$$

Donde \hat{Y}_i es el **predictor** (variable aleatoria) y $\hat{\beta}$ es el **estimador** (parámetro). Por el TGM:

- $E(\hat{Y}_i) = X'_i \beta$ (predictor insesgado)
- $V(\hat{Y}_i) = X'_i V(\hat{\beta}) X_i = \sigma^2 X'_i (X' X)^{-1} X_i$

donde $V(\hat{\beta}) = \sigma^2 (X' X)^{-1}$

Entonces, si el *estimador* $\hat{\beta}$ es insesgado y de varianza mínima, \hat{Y}_i es un *predictor* insesgado y de varianza mínima, ambos en la clase de estimadores/predictores lineales e insesgados.

El resultado anterior surge del hecho que predecir requiere estimar (en este caso β). Retomamos el *EMC* en el caso de los **estimadores**:

$$EMC(\hat{\beta}) = E(\hat{\beta} - \beta)^2 \quad (4.8)$$

El *EMC* mide en promedio cuan lejos esta $\hat{\beta}$ (estimador) de β el parámetro que se quiere estimar.

Recordar que por definición:

- $V(\hat{\beta}) \equiv E(\hat{\beta} - E(\hat{\beta}))^2$ (dispersión)
- $Sesgo(\hat{\beta}) \equiv E(\hat{\beta}) - \beta$ (centro)

A partir de (4.8) y las definiciones anteriores reescribimos el *EMC* en términos de la descomposición sesgo-varianza:

$$EMC(\hat{\beta}) = Sesgo^2(\hat{\beta}) + V(\hat{\beta}) \quad (4.9)$$

Es decir, cuán mal estima $\hat{\beta}$ depende de cuán descentrado está en relación a la verdad (sesgo) más cuán disperso es en relación a su propio centro (varianza).

Para ver la relación entre el error de estimación y el error de predicción se define el **error de pronóstico** como:

$$Err(\hat{Y}) \equiv E(Y - \hat{Y})^2 \quad (4.10)$$

Como vimos antes, dado el modelo genérico² $Y = f(X) + u$ donde $E(u) = 0$ y $V(u) = \sigma^2$:

- $f(X)$ es la parte sistemática
- u la parte no sistemática

²No necesariamente lineal.

Nota. Resultado importante en **teoría de la predicción**: si se quiere predecir una variable aleatoria Y con una constante m el mejor predictor es su esperanza, es decir, $m = E(Y)$.³

Entonces, si $E(Y) = f(X)$, u es no observable y $f(X)$ es conocida, $f(X)$ es el mejor predictor porque, como se dijo arriba, el mejor predictor de una variable aleatoria es su esperanza.

En la práctica $f(X)$ es desconocida y, por lo tanto, se debe estimar $\hat{f}(X)$.

$$Err(\hat{Y}) = E(Y - \hat{f})^2 \quad (4.11)$$

$$Err(Y - \hat{f}) = EMC(\hat{f}) + \sigma^2 \quad (4.12)$$

En términos de la ecuación (3.3) es la suma de un error reducible (EMC) y otro irreducible (σ^2).

Nuevamente, puede verse la relación entre predicción y estimación. *Predecir* correctamente ($Err(Y - \hat{f})$) requiere *estimar* ($EMC(\hat{f})$) correctamente (porque σ^2 no se puede controlar). Es decir, tener bajo sesgo y baja varianza. Utilizando la descomposición:

$$Err(Y - \hat{f}) = \underbrace{Sesgo^2(\hat{f}) + V(\hat{f})}_{EMC} + \sigma^2 \quad (4.13)$$

Machine learning hace uso de que estrategias sesgadas pueden implicar una reducción drástica en la varianza, por lo tanto, puede ser que el mínimo EMC ocurra para predictores sesgados.

4.2 Aplicación práctica

Tip. Para hacer tablas con los resultados de las regresiones se puede utilizar el paquete **stargazer**. Algunas referencias en este [documento](#) y en este [blog](#).

La biblioteca **ISLR2** contiene la base de datos de **Boston**, que registra **medv** (mediana del valor de las casas en miles de USD) para 506 distritos censales en Boston. Buscaremos predecir **medv** usando 12 predictores como **rm** (número promedio de habitaciones por casa), **age** (edad promedio de las casas) y **lstat** (porcentaje de hogares con bajo *status* socioeconómico).

```
library(MASS)
library(ISLR2)
head(Boston)
```

³En el Capítulo 7 veremos que este resultado es importante para la metodología de árboles de decisión.

```
##      crim zn indus chas   nox     rm   age     dis rad tax ptratio lstat medv
## 1 0.00632 18 2.31     0 0.538 6.575 65.2 4.0900    1 296 15.3 4.98 24.0
## 2 0.02731 0 7.07     0 0.469 6.421 78.9 4.9671    2 242 17.8 9.14 21.6
## 3 0.02729 0 7.07     0 0.469 7.185 61.1 4.9671    2 242 17.8 4.03 34.7
## 4 0.03237 0 2.18     0 0.458 6.998 45.8 6.0622    3 222 18.7 2.94 33.4
## 5 0.06905 0 2.18     0 0.458 7.147 54.2 6.0622    3 222 18.7 5.33 36.2
## 6 0.02985 0 2.18     0 0.458 6.430 58.7 6.0622    3 222 18.7 5.21 28.7
```

Comenzaremos usando la función `lm()` para ajustar un modelo de regresión lineal simple, con `medv` como respuesta y `lstat` como predictor.⁴

```
lm.fit = lm(medv ~ lstat, data = Boston)
summary(lm.fit)

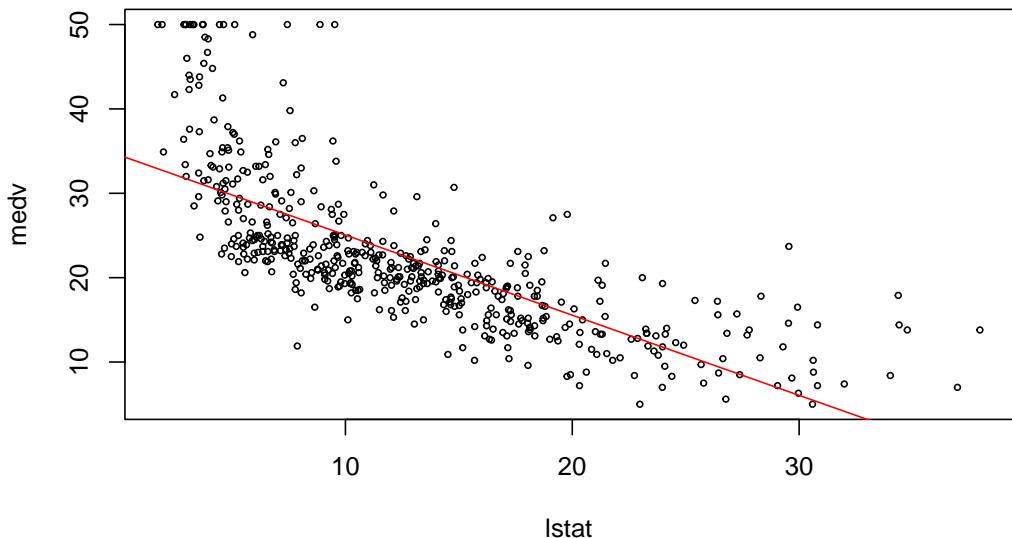
##
## Call:
## lm(formula = medv ~ lstat, data = Boston)
##
## Residuals:
##      Min      1Q  Median      3Q      Max
## -15.168 -3.990 -1.318  2.034  24.500
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 34.55384   0.56263   61.41  <2e-16 ***
## lstat       -0.95005   0.03873  -24.53  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.216 on 504 degrees of freedom
## Multiple R-squared:  0.5441, Adjusted R-squared:  0.5432 
## F-statistic: 601.6 on 1 and 504 DF,  p-value: < 2.2e-16

names(lm.fit)

## [1] "coefficients"   "residuals"      "effects"        "rank"        
## [5] "fitted.values"  "assign"         "qr"             "df.residual"  
## [9] "xlevels"         "call"           "terms"          "model"        
```

```
attach(Boston)
plot(lstat, medv, cex=.5)
abline(lm.fit, col = 'red')
```

⁴Para más detalles véase `?stats::lm`.



A continuación se examinan algunos gráficos de diagnóstico con la función `plot()`. En general, este comando producirá un gráfico a la vez (presionando *Enter* se generará el siguiente gráfico). Sin embargo, se pueden ver los cuatro gráficos juntos con las funciones `par()` y `mfrow()`, que le dicen a R que divide la pantalla de visualización en paneles separados. Por ejemplo, `par(mfrow = c(2, 2))` divide la región del gráfico en una cuadrícula de paneles de 2×2 .

- **Residuals vs. fitted values:** residuos vs. valores ajustados se utiliza para detectar patrones de variables omitidas, heterocedasticidad, etc.
- **Scale Location:** residuos estandarizados vs. valores ajustados. Similar al anterior y se utiliza para detectar patrones en los residuos.
- **Normal Q-Q:** cuantiles teóricos de la distribución normal estándar vs. cuantiles reales de residuos estandarizados. Se utiliza para evaluar la normalidad de los errores.
- **Residuals vs. Leverage:** el *leverage* es una medida de cuán influyente es una observación en el valor de los coeficientes. Este gráfico se utiliza para detectar las (posibles) observaciones influyentes y los valores atípicos al mismo tiempo.

```
par(mfrow = c(2, 2))
plot(lm.fit)
```



Para obtener la predicción y el EMC:

```
medv_hat = predict(lm.fit, data = Boston)
emc = mean((medv - medv_hat)^2)
emc
```

```
## [1] 38.48297
```

Regresión lineal múltiple:

```
lm.fit = lm(medv ~ lstat + age, data = Boston)
summary(lm.fit)
```

```
##
## Call:
## lm(formula = medv ~ lstat + age, data = Boston)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.981  -3.978  -1.283   1.968  23.158
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 33.22276    0.73085 45.458 < 2e-16 ***
## lstat        -3.77520    1.42943 -2.634  0.01092 *
## age          -0.37253   0.73333 -0.503  0.61713
```

```

## lstat      -1.03207   0.04819 -21.416 < 2e-16 ***
## age        0.03454   0.01223   2.826  0.00491 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.173 on 503 degrees of freedom
## Multiple R-squared:  0.5513, Adjusted R-squared:  0.5495
## F-statistic: 309 on 2 and 503 DF,  p-value: < 2.2e-16

```

Utilizando todas las variables de la base de datos:

```

lm.fit = lm(medv ~ ., data = Boston)
summary(lm.fit)

##
## Call:
## lm(formula = medv ~ ., data = Boston)
##
## Residuals:
##    Min     1Q   Median     3Q    Max
## -15.1304 -2.7673 -0.5814  1.9414 26.2526
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 41.617270  4.936039  8.431 3.79e-16 ***
## crim        -0.121389  0.033000 -3.678 0.000261 ***
## zn          0.046963  0.013879  3.384 0.000772 ***
## indus       0.013468  0.062145  0.217 0.828520
## chas        2.839993  0.870007  3.264 0.001173 ** 
## nox        -18.758022  3.851355 -4.870 1.50e-06 ***
## rm          3.658119  0.420246  8.705 < 2e-16 ***
## age         0.003611  0.013329  0.271 0.786595
## dis        -1.490754  0.201623 -7.394 6.17e-13 ***
## rad         0.289405  0.066908  4.325 1.84e-05 ***
## tax        -0.012682  0.003801 -3.337 0.000912 ***
## ptratio     -0.937533  0.132206 -7.091 4.63e-12 ***
## lstat      -0.552019  0.050659 -10.897 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.798 on 493 degrees of freedom
## Multiple R-squared:  0.7343, Adjusted R-squared:  0.7278
## F-statistic: 113.5 on 12 and 493 DF,  p-value: < 2.2e-16

```

Utilizando todas las variables de la base de datos menos age:

```
lm.fit1 = lm(medv ~ . - age, data = Boston)
summary(lm.fit1)

##
## Call:
## lm(formula = medv ~ . - age, data = Boston)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.1851  -2.7330  -0.6116   1.8555  26.3838
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 41.525128  4.919684  8.441 3.52e-16 ***
## crim        -0.121426  0.032969 -3.683 0.000256 ***
## zn          0.046512  0.013766  3.379 0.000785 ***
## indus       0.013451  0.062086  0.217 0.828577    
## chas        2.852773  0.867912  3.287 0.001085 **  
## nox        -18.485070 3.713714 -4.978 8.91e-07 ***
## rm          3.681070  0.411230  8.951 < 2e-16 ***
## dis        -1.506777  0.192570 -7.825 3.12e-14 ***
## rad         0.287940  0.066627  4.322 1.87e-05 ***
## tax        -0.012653  0.003796 -3.333 0.000923 ***
## ptratio     -0.934649  0.131653 -7.099 4.39e-12 ***
## lstat       -0.547409  0.047669 -11.483 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.794 on 494 degrees of freedom
## Multiple R-squared:  0.7343, Adjusted R-squared:  0.7284
## F-statistic: 124.1 on 11 and 494 DF,  p-value: < 2.2e-16
```

Capítulo 5

LASSO

Least Absolute Shrinkage and Selection Operator. Cuando el objetivo es encontrar el mejor modelo, es decir, aquel con menor error de predicción en la muestra de *test* existen distintas alternativas de selección de modelos. La idea de *shrinkage* es ajustar un modelo que contenga todos los p predictores usando una técnica que restringe o regulariza las estimaciones de coeficientes hacia cero.

Dado:

$$Y = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p + u$$

Suponer que M_k con $k = 1, \dots, K$ es una serie de modelos.

Elección de modelos: buscar en M_k el mejor modelo para predecir fuera de la muestra. La primera alternativa podría ser realizar una búsqueda exhaustiva. En las ocasiones, puede ser impráctico dado que con p predictores, se pueden construir 2^p modelos (con $p = 15$ hay 32.768 modelos).

Lasso: una manera formal y algorítmica de realizar esa tarea. Para $\lambda \geq 0$ dado, se considera la siguiente función objetivo (a minimizar):

$$R_l(\beta) = \sum_{i=1}^n (y_i - x_i' \beta)^2 + \lambda \sum_{s=2}^p |\beta_s| \quad (5.1)$$

donde x_i contiene un intercepto.

1. $\lambda = 0$ es el estimador de *MCO*
2. $\lambda = \infty$ el segundo término $\rightarrow \infty$ entonces $\beta_2, \dots, \beta_p = 0$
3. Si $0 < \lambda < \infty$ *Lasso* hace algo intermedio entre *MCO* y 0

Intuitivamente:

- $\sum_{i=1}^n (y_i - x_i' \beta)^2$ penaliza la falta de ajuste

- $\sum_{s=2}^p |\beta_s|$ penaliza complejidad

Nota. La ventaja de utilizar *lasso* es que elige automáticamente que variables entran en el modelo ($\beta_s \neq 0$) y cuales no ($\beta_s = 0$).

La Figura 5.1 muestra el valor de los coeficientes estandarizados en función de distintos valores del parámetro λ que se obtienen de aplicar *lasso* a la base de datos de Credit.¹ Cuando $\lambda = 0$, el resultado coincide con el de mínimos cuadrados, cuando λ es lo suficientemente grande, todos los coeficientes son iguales cero. Entre los dos casos extremos, moviéndose de derecha a izquierda se observa que al principio *lasso* resulta en un modelo que contiene solo el predictor de rating. Luego, ingresan al modelo student y limit casi simultáneamente, seguidas de cerca por income. Finalmente lo hacen el resto de las variables.

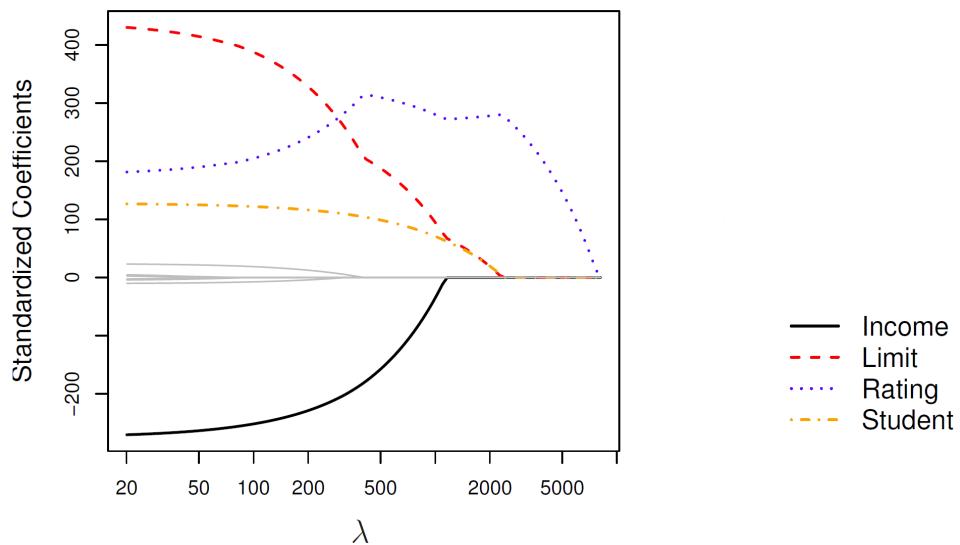


Figura 5.1: Lasso

Como señalamos arriba, si $\hat{\beta}_l = 0$, al moverse de 0 (es decir, incluir la variable) podría pensarse que la función de penalidad de *lasso* genera costos y beneficios. Por un lado, el costo por incluir la variable sube en el valor de (λ), por el otro, la penalidad de ajuste baja (más variables, mejor ajuste). Si el primero sube más rápido que lo que el segundo baja conviene quedarse en 0. En caso contrario conviene moverse afuera de 0. Por lo tanto, conviene moverse de 0 si la relación es lo suficientemente fuerte (mejora el ajuste), sino conviene evitar la penalización.

¹Datos simulados que contiene información sobre 10.000 clientes donde el objetivo es predecir quienes no pagarán su deuda de tarjeta de crédito.

5.1 Aplicacion practica

Se utiliza el paquete `glmnet` para realizar la regresión de *lasso*² con una sintaxis ligeramente diferente de otras funciones de estimación de modelos. En particular, se debe usar una matriz `x` y un vector `y` y no utiliza la sintaxis (`y ~ x`). Se busca predecir `Salary` en la base de datos `Hitters`. Antes de comenzar deben eliminarse los valores *missing*.³

```
library(ISLR2)
library(glmnet)
names(Hitters)

## [1] "AtBat"      "Hits"       "HmRun"      "Runs"       "RBI"        "Walks"
## [7] "Years"       "CAtBat"     "CHits"      "CHmRun"     "CRuns"      "CRBI"
## [13] "CWalks"     "League"     "Division"    "PutOuts"    "Assists"    "Errors"
## [19] "Salary"      "NewLeague"

dim(Hitters)

## [1] 322  20

sum(is.na(Hitters$Salary))

## [1] 59
```

La Figura de abajo muestra la base de datos completa para analizar los valores *missing*. Las variables están en el eje y y las observaciones en el eje x.

```
hit = Hitters
rownames(hit) = 1:nrow(hit)
hit %>%
  is.na() %>%
  reshape2::melt() %>%
  ggplot(aes(Var2, Var1, fill=value)) +
  geom_raster() + coord_flip() +
  scale_y_continuous(NULL, expand = c(0, 0)) +
  scale_fill_discrete(name = "",
                       labels = c("Datos",
                                  "Missing")) +
  labs( x= 'Variable', title= 'Valores missing') +
  theme_minimal() +
  theme(axis.text.y = element_text(size = 7)) +
  NULL
```

²La función principal de este paquete es `glmnet()`.

³Para más detalles véase `?glmnet::glmnet` y ([Simon et al., 2011](#))



```
rm(hit)
```

```
Hitters <- na.omit(Hitters)
dim(Hitters)
```

```
## [1] 263 20
```

```
sum(is.na(Hitters))
```

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

La función `model.matrix()` es útil para crear la matriz de datos x ; no sólo produce una matriz correspondiente a los 19 predictores, sino también transforma automáticamente cualquier variable cualitativa en variables *dummies* (*one-hot encoding*). La última propiedad es importante porque `glmnet()` sólo puede tomar variables numéricas y cuantitativas.

```
x = model.matrix(Salary ~ ., Hitters) [, -1]
y = Hitters$Salary
```

Por primera vez dividimos la muestra en una base de entrenamiento y otra de *test* para estimar el error de *test* de la regresión *lasso*. Hay varias formas de dividir aleatoriamente una base de datos. Nuestro enfoque elige aleatoriamente un subconjunto de números entre 1 y n ; estos se pueden usar como índices para las observaciones de entrenamiento.

Tip. La función `createDataPartition()` de la librería `caret` es útil para crear particiones de `train` y `test`. La librería `rsample` permite realizar divisiones estratificadas.

Primero establecemos una semilla aleatoria para que los resultados obtenidos sean reproducibles.

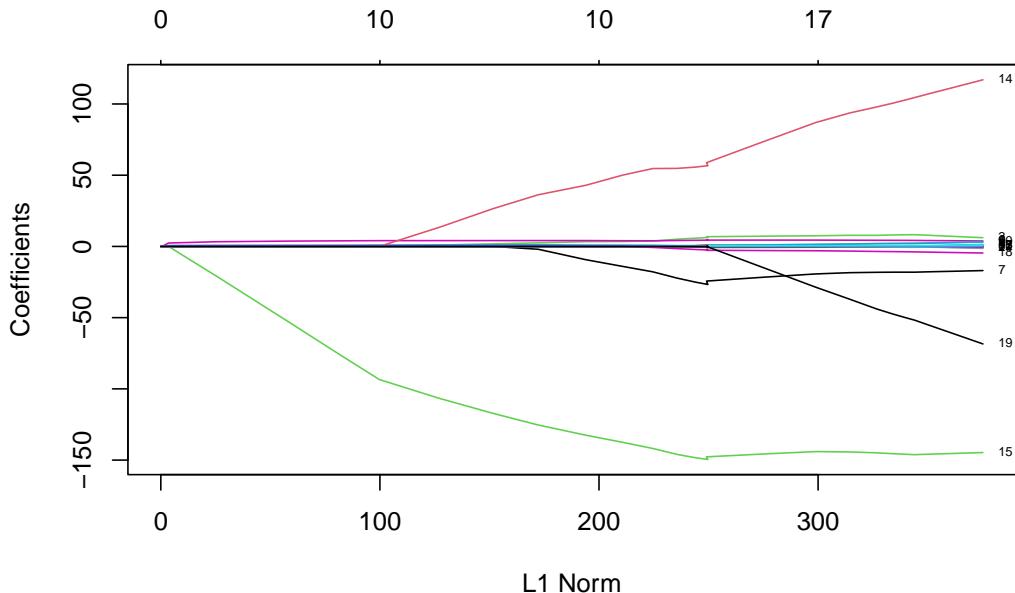
```
set.seed(1)
train <- sample(1:nrow(x), nrow(x) / 2) # selecciona la mitad de las observaciones
test <- (-train)
y.test <- y[test]
```

Para estimar un modelo *lasso* usamos la función `glmnet()` con el argumento `alpha = 1` que indica la metodología que queremos usar.⁴

También definimos una grilla⁵ de 100 valores λ que van desde $\lambda = 10^{10}$ a $\lambda = 10^{-2}$, cubriendo esencialmente la gama completa de escenarios desde el modelo nulo que contiene solo la constante, hasta el modelo de mínimos cuadrados.

Podemos ver en el gráfico de abajo que cada curva corresponde a una variable. Muestra la trayectoria de su coeficiente frente a **L1 norm** (la penalidad de LASSO) del vector de coeficientes completo a medida que varía λ . El eje de arriba indica el número de coeficientes distintos de cero para LASSO.

```
grid <- 10^seq(10, -2, length = 100)
lasso.mod <- glmnet(x[train, ], y[train], alpha = 1, lambda = grid)
plot(lasso.mod, label = TRUE)
```



⁴alpha = 0 estima con la metodología ridge.

⁵También se pueden usar valores específicos.

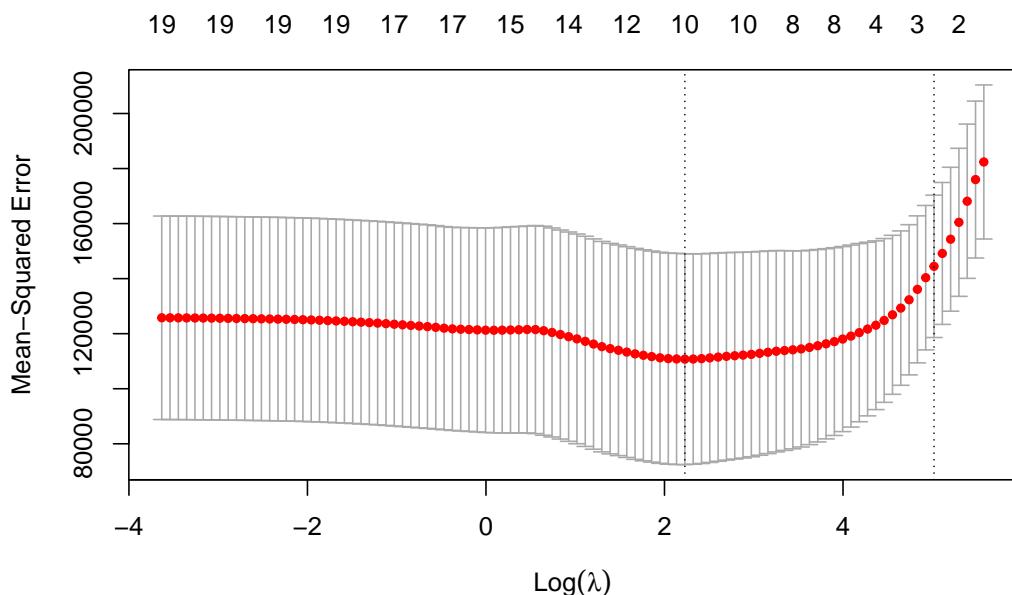
Se muestra un resumen de la ruta de `glmnet` en cada paso:

```
print(lasso.mod)

##
## Call: glmnet(x = x[train, ], y = y[train], alpha = 1, lambda = grid)
##
##      Df  %Dev  Lambda
## 1      0  0.00 1.000e+10
## 2      0  0.00 7.565e+09
## 3      0  0.00 5.722e+09
## 4      0  0.00 4.329e+09
## 5      0  0.00 3.275e+09
## 6      0  0.00 2.477e+09
....
```

Ahora realizamos un ejercicio de *cross validation* (subsección 3.7) y calculamos el error de *test* asociado. Por defecto el argumento `nfolds = 10`. Los números en la parte superior de la Figura indican el número de variables en el modelo. La primera y segunda línea discontinua vertical representan el valor de λ con el EMC mínimo y el mayor valor de λ dentro de un error estándar.

```
set.seed(1)
cv.out <- cv.glmnet(x[train, ], y[train], alpha = 1)
plot(cv.out)
```



```
names(cv.out)

## [1] "lambda"      "cvm"        "cvsd"        "cvup"        "cvlo"
## [6] "nzero"        "call"        "name"        "glmnet.fit"  "lambda.min"
## [11] "lambda.1se"   "index"

bestlam <- cv.out$lambda.min
lasso.pred <- predict(lasso.mod, s = bestlam, newx = x[test, ])
mean((lasso.pred - y.test)^2) # EMC
```

```
## [1] 143673.6
```

Finalmente, el resultado indica que 10 de los 19 coeficientes estimados son distintos de 0 (notar que hay una contradicción con las últimas dos tablas). Por lo tanto, el modelo de *lasso* con λ elegido por *cross validation* contiene solo 11 variables (más la constante).⁶

```
cv.out$nzero[cv.out$index[[1]]]
```

```
## s36
## 10
```

```
out <- glmnet(x, y, alpha = 1, lambda = grid)
lasso.coef <- predict(out, type = "coefficients", s = bestlam)[1:20, ]
lasso.coef
```

	(Intercept)	AtBat	Hits	HmRun	Runs
##	1.27479059	-0.05497143	2.18034583	0.00000000	0.00000000
##	RBI	Walks	Years	CAtBat	CHits
##	0.00000000	2.29192406	-0.33806109	0.00000000	0.00000000
##	CHmRun	CRuns	CRBI	CWalks	LeagueN
##	0.02825013	0.21628385	0.41712537	0.00000000	20.28615023
##	DivisionW	PutOuts	Assists	Errors	NewLeagueN
##	-116.16755870	0.23752385	0.00000000	-0.85629148	0.00000000

```
lasso.coef[lasso.coef != 0]
```

	(Intercept)	AtBat	Hits	Walks	Years
##	1.27479059	-0.05497143	2.18034583	2.29192406	-0.33806109
##	CHmRun	CRuns	CRBI	LeagueN	DivisionW
##	0.02825013	0.21628385	0.41712537	20.28615023	-116.16755870
##	PutOuts	Errors			
##	0.23752385	-0.85629148			

⁶Para más detalles de `cv.glmnet` ver [aquí](#).

Capítulo 6

Logit

En la base de datos `Default`,¹ la variable *default* pertenece a una de dos categorías, **Yes** o **No**. En vez de modelar esta respuesta Y directamente, la regresión logística modela la probabilidad de que Y pertenezca a un categoría. Por lo tanto, si se busca estimar la probabilidad de *default*, se puede transformar la variable original en una variable binaria 1 (Yes); 0 (No).

En general, la estrategia consiste en:

Y variable binaria 0/1.

X vector de K predictores.

Se debe construir un modelo para:

$$p = Pr(Y = 1 | X)$$

Como vimos en la subsección 3.6.3.1 el **clasificador de Bayes**:

- $\hat{Y} = 1$ si $p > 0,5$
- $\hat{Y} = 0$ si $p \leq 0,5$

Recordar que en el caso particular de la probabilidad de *default* puede resultar relativamente más costoso para un banco clasificar a un deudor malo en la categoría **No default** que a un deudor bueno como **Si default** (pérdida asimétrica). Por lo tanto, podría elegirse un umbral más bajo (ej. $p > 0,3$) para la categoría **Si default**.

6.1 Modelo *logit*

$$p = \frac{e^z}{1 + e^z} \tag{6.1}$$

¹Esta base pertenece a la librería `ISLR2`.

donde $z \equiv X\beta$ con β un vector de K coeficientes.

El panel izquierdo de la Figura 6.1 muestra la probabilidad estimada usando regresión lineal donde algunos valores de probabilidad son negativos (también podría haber valores mayores a 1 para valores elevados de la variable `balance`, ambos extremos inconsistentes con la noción de probabilidad). Los puntos naranja indican los valores 0/1 codificados por defecto (No o Si). El panel derecho exhibe las probabilidades estimadas de incumplimiento mediante regresión logística donde todos los valores $\in (0, 1)$.

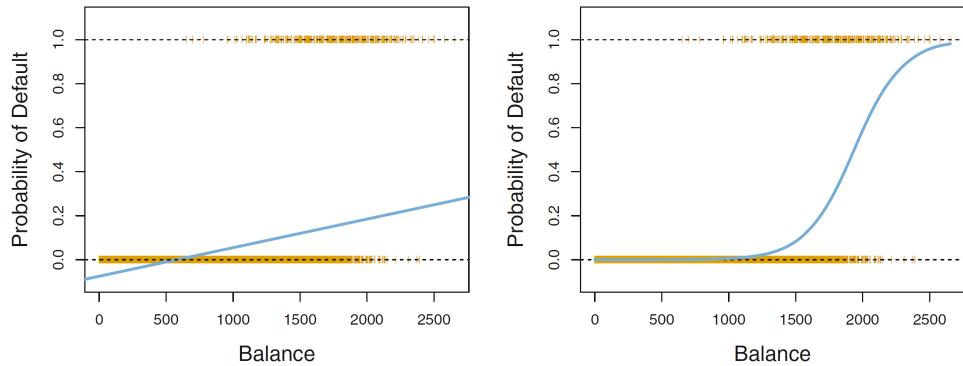


Figura 6.1: Regresión lineal vs. logística

6.1.1 Interpretacion de coeficientes en el modelo *logit*

Dada la forma funcional del modelo *logit*, primero se introduce el concepto de **odds ratio** ($\frac{p}{1-p}$) para luego derivar la interpretación de un coeficiente. El se define *odds ratio* como la probabilidad de que un evento suceda en relación a que el mismo evento no suceda. Es decir:

Por ejemplo, en promedio 1 de cada 5 personas con un *odds* de $1/4$ no pagará su deuda, ya que $p(X) = 0,2$ implica un *odds* de $\frac{0,2}{1-0,2} = 0,25 = 1/4$.

En el modelo *logit*:

$$(1-p) = \frac{1}{1+e^z} \quad (6.2)$$

Entonces el *odds ratio*:

$$\frac{p}{1-p} = e^z \quad (6.3)$$

Si se toma el logaritmo del *odds ratio* queda:

$$\ln\left(\frac{p}{1-p}\right) = z = X\beta \quad (6.4)$$

El lado izquierdo se llama *log odds* o *logit*. El modelo de regresión logística (6.1) tiene un *logit* que es lineal en X . Por lo tanto, el k -*simo* coeficiente es:

$$\beta_k = \frac{\partial \ln(\frac{p}{1-p})}{\partial X_k} \quad (6.5)$$

Aumentar X_k en una unidad cambia el *log odds* en β_k . Dado que la relación entre $p(X)$ y X en (6.1) no es lineal, β_k no corresponde al cambio en $p(X)$ asociado con el aumento de una unidad en X . La cantidad que $p(X)$ cambia debido a un cambio de una unidad en X depende del valor actual de X . Pero independientemente del valor de X , si β_k es positivo, entonces el aumento de X se asocia con el aumento de $p(X)$, y si β_k es negativo, el aumento de X se asocia con la disminución de $p(X)$.²

En este caso, $\hat{\beta}$ se estima por máxima verosimilitud (no es una forma cerrada como *MCO*).

$$\hat{\theta} = \operatorname{argmax}_{\theta} \prod_{i=1}^n L(\theta; y_i) \quad (6.6)$$

Luego se reemplazan los coeficientes estimados en (6.7) para obtener las probabilidades estimadas.

$$\hat{p}_i = \frac{e^{X_i \hat{\beta}}}{1 + e^{X_i \hat{\beta}}} \quad (6.7)$$

6.2 Aplicacion practica

Comenzaremos examinando algunas estadísticas y gráficos de los datos **Smarket**, que forman parte de la biblioteca **ISLR2**. Esta base de datos consiste en rendimientos porcentuales para el índice bursátil *S&P500* para más de 1,250 días, desde principios de 2001 hasta finales de 2005. Para cada fecha, registra los rendimientos porcentuales de los cinco días hábiles anteriores, de **Lag1** a **Lag5**. También registra el **volume** (el número de acciones negociadas el día anterior, en miles de millones), **Today** (el rendimiento porcentual en la fecha en cuestión) y **direction** (si el mercado estaba *Up* o *Down* en este día). El objetivo es predecir la “dirección” (una respuesta cualitativa) usando las otras características.

```
library(ISLR2)
names(Smarket)

## [1] "Year"      "Lag1"      "Lag2"      "Lag3"      "Lag4"      "Lag5"
## [7] "Volume"    "Today"     "Direction"
```

²Para la interpretación de coeficientes en la práctica usualmente se analizan los efectos marginales. Ver ([Wooldridge, 2012](#)) capítulo 17.

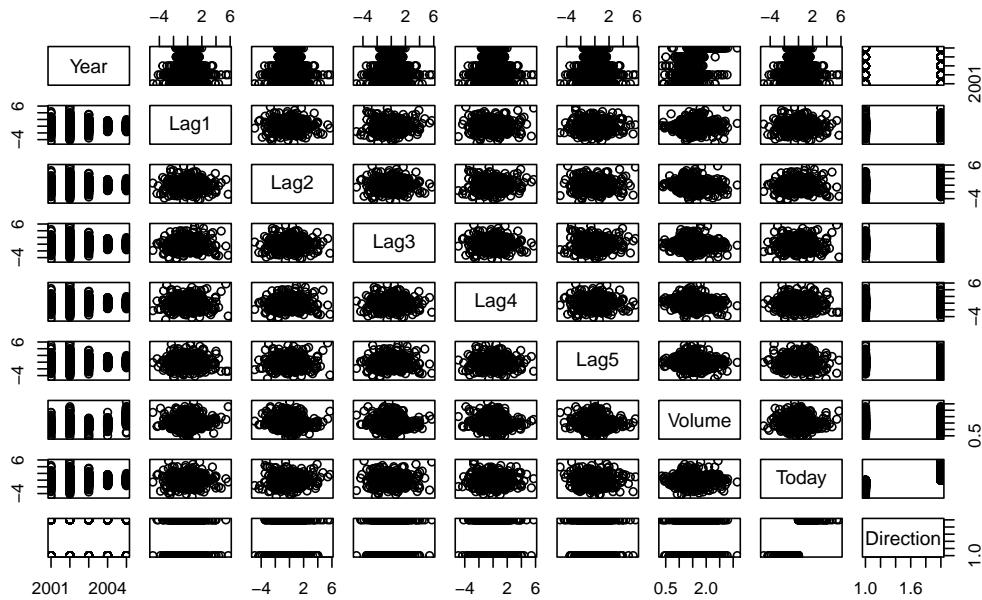
```
dim(Smarket)
```

```
## [1] 1250 9
```

```
summary(Smarket)
```

	Year	Lag1	Lag2	Lag3
##	Min. :2001	Min. :-4.922000	Min. :-4.922000	Min. :-4.922000
##	1st Qu.:2002	1st Qu.:-0.639500	1st Qu.:-0.639500	1st Qu.:-0.640000
##	Median :2003	Median : 0.039000	Median : 0.039000	Median : 0.038500
##	Mean :2003	Mean : 0.003834	Mean : 0.003919	Mean : 0.001716
##	3rd Qu.:2004	3rd Qu.: 0.596750	3rd Qu.: 0.596750	3rd Qu.: 0.596750
##	Max. :2005	Max. : 5.733000	Max. : 5.733000	Max. : 5.733000
	Lag4	Lag5	Volume	Today
##	Min. :-4.922000	Min. :-4.92200	Min. :0.3561	Min. :-4.922000
##	1st Qu.:-0.640000	1st Qu.:-0.64000	1st Qu.:1.2574	1st Qu.:-0.639500
##	Median : 0.038500	Median : 0.03850	Median :1.4229	Median : 0.038500
##	Mean : 0.001636	Mean : 0.00561	Mean :1.4783	Mean : 0.003138
##	3rd Qu.: 0.596750	3rd Qu.: 0.59700	3rd Qu.:1.6417	3rd Qu.: 0.596750
##	Max. : 5.733000	Max. : 5.73300	Max. :3.1525	Max. : 5.733000
##	Direction			
##	Down:602			
##	Up :648			
##				
##				
##				

```
pairs(Smarket)
```



La función `cor()` produce una matriz que contiene todas las correlaciones por pares entre los predictores.

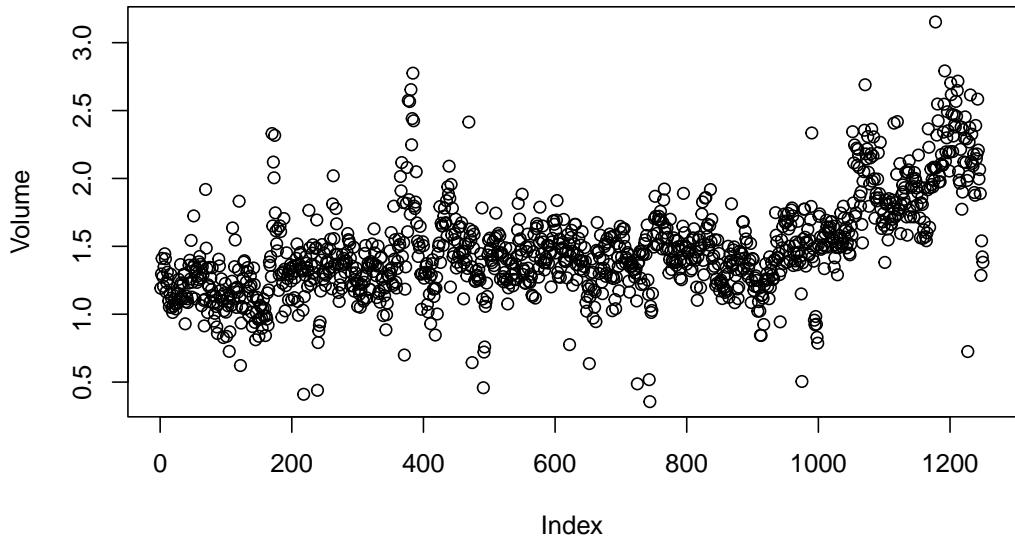
```
cor(Smarket[, -9])
```

```
##          Year      Lag1      Lag2      Lag3      Lag4
## Year  1.0000000  0.029699649  0.030596422  0.033194581  0.035688718
## Lag1  0.02969965  1.000000000 -0.026294328 -0.010803402 -0.002985911
## Lag2  0.03059642 -0.026294328  1.000000000 -0.025896670 -0.010853533
## Lag3  0.03319458 -0.010803402 -0.025896670  1.000000000 -0.024051036
## Lag4  0.03568872 -0.002985911 -0.010853533 -0.024051036  1.000000000
## Lag5  0.02978799 -0.005674606 -0.003557949 -0.018808338 -0.027083641
## Volume 0.53900647  0.040909908 -0.043383215 -0.041823686 -0.048414246
## Today  0.03009523 -0.026155045 -0.010250033 -0.002447647 -0.006899527
##          Lag5      Volume      Today
## Year   0.029787995  0.53900647  0.030095229
## Lag1  -0.005674606  0.04090991 -0.026155045
## Lag2  -0.003557949 -0.04338321 -0.010250033
## Lag3  -0.018808338 -0.04182369 -0.002447647
## Lag4  -0.027083641 -0.04841425 -0.006899527
## Lag5   1.000000000 -0.02200231 -0.034860083
## Volume -0.022002315  1.000000000  0.014591823
## Today  -0.034860083  0.01459182  1.000000000
```

Como era de esperar, las correlaciones entre las variables rezagadas y los rendimientos de hoy son cercanas a cero. En otras palabras, parece haber poca correlación entre los

rendimientos de hoy y los rendimientos de días anteriores. La única correlación sustancial es entre `Year` y `volume`. Al graficar los datos, que están ordenados cronológicamente, vemos que el `volume` aumenta con el tiempo. En otras palabras, el número promedio de acciones negociadas diariamente aumentó de 2001 a 2005.

```
attach(Smarket)
plot(Volume)
```



A continuación, ajustaremos un modelo de regresión logística para predecir la `direction` utilizando `Lag1` hasta `Lag5` y `volume`. La función `glm()` se puede utilizar para muchos tipos de modelos lineales generalizados, incluida la regresión logística. La sintaxis de la función `glm()` es similar a la de `lm()`, excepto que debemos utilizar el argumento `family = binomial` para decirle a R que ejecute una regresión logística.

```
glm.fits <- glm(Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume,
  data = Smarket, family = binomial)
summary(glm.fits)
```

```
##
## Call:
## glm(formula = Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 +
##     Volume, family = binomial, data = Smarket)
##
## Deviance Residuals:
```

```

##      Min      1Q  Median      3Q      Max
## -1.446 -1.203  1.065  1.145  1.326
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.126000  0.240736 -0.523   0.601
## Lag1        -0.073074  0.050167 -1.457   0.145
## Lag2        -0.042301  0.050086 -0.845   0.398
## Lag3         0.011085  0.049939  0.222   0.824
## Lag4         0.009359  0.049974  0.187   0.851
## Lag5         0.010313  0.049511  0.208   0.835
## Volume      0.135441  0.158360  0.855   0.392
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 1731.2 on 1249 degrees of freedom
## Residual deviance: 1727.6 on 1243 degrees of freedom
## AIC: 1741.6
##
## Number of Fisher Scoring iterations: 3

```

El valor p más pequeño está asociado con `Lag1`. El coeficiente negativo sugiere que si el mercado tuvo un rendimiento positivo ayer, es menos probable que suba hoy. Sin embargo, con un valor de 0,145, el valor de p sigue siendo relativamente grande, por lo que no hay pruebas claras de una asociación real entre `Lag1` y `direction`.

La función `predict()` se puede utilizar para predecir la probabilidad de que el mercado suba, dados los valores de los predictores. La opción `type = "response"` le dice a R que genere probabilidades de la forma $P(Y = 1|X)$. Si no se proporciona ninguna base de datos a la función `predict()`, se calculan las probabilidades para los datos de entrenamiento que se usaron para ajustar el modelo de regresión logística. Se imprimen las primeras diez probabilidades estimadas. Sabemos que estos valores corresponden a la probabilidad de que el mercado suba, en lugar de que baje, porque la función `contrasts()` indica que R ha creado una variable ficticia con un 1 para `Up`.

```

glm.probs <- predict(glm.fits, type = "response")
glm.probs[1:10]

```

```

##      1      2      3      4      5      6      7      8
## 0.5070841 0.4814679 0.4811388 0.5152224 0.5107812 0.5069565 0.4926509 0.5092292
##      9      10
## 0.5176135 0.4888378

```

```
contrasts(Direction)
```

```
##      Up
## Down  0
## Up    1
```

Para hacer una predicción sobre si el mercado subirá o bajará en un día en particular, debemos convertir estas probabilidades pronosticadas en etiquetas de clase, `Up` o `Down`. Los siguientes dos comandos crean un vector de predicciones de clase basado en si la probabilidad prevista de un aumento del mercado es mayor o menor que 0,5.

```
glm.pred <- rep("Down", 1250)
glm.pred[glm.probs > .5] = "Up"
```

El primer comando crea un vector de 1.250 elementos `Down`. La segunda línea transforma en `Up` todos los elementos para los que la probabilidad predicha de un aumento del mercado supera los 0,5. Dadas estas predicciones, la función `table()` se puede usar para producir una matriz de confusión (subsección 3.6.4) para determinar cuántas observaciones se clasificaron correcta o incorrectamente. Al ingresar dos vectores cualitativos, R creará una tabla de 2×2 con recuentos del número de veces que ocurrió cada combinación. Por ejemplo, pronosticó `Up` y el mercado aumentó, predijo `Up` y el mercado disminuyó, etc.

Los elementos de la diagonal de la matriz de confusión indican predicciones correctas (*accuracy*), mientras que los fuera de la diagonal representan predicciones incorrectas. Por lo tanto, el modelo predijo correctamente que el mercado subiría en 507 días y que bajaría en 145 días, para un total de $507 + 145 = 652$ predicciones correctas. La función `mean()` se puede utilizar para calcular la fracción de días en los que la predicción fue correcta. En este caso, la regresión logística predijo correctamente el movimiento del mercado 52,2% del tiempo.

```
table(glm.pred, Direction)
```

```
##          Direction
## glm.pred Down  Up
##      Down 145 141
##      Up    457 507
```

```
(507 + 145) / 1250
```

```
## [1] 0.5216
```

```
mean(glm.pred == Direction)
```

```
## [1] 0.5216
```

Tip. La función `confusionMatrix()` de la librería `caret` calcula la matriz de confusión con estadísticas completas.

```
library('caret')
confusionMatrix(factor(glm.pred), factor(Direction))
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction Down   Up
##       Down    145 141
##       Up      457 507
##
##             Accuracy : 0.5216
##                 95% CI : (0.4935, 0.5496)
##       No Information Rate : 0.5184
##       P-Value [Acc > NIR] : 0.4216
##
##             Kappa : 0.0237
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.2409
##             Specificity  : 0.7824
##       Pos Pred Value : 0.5070
##       Neg Pred Value : 0.5259
##             Prevalence : 0.4816
##             Detection Rate : 0.1160
##       Detection Prevalence : 0.2288
##             Balanced Accuracy : 0.5116
##
##             'Positive' Class : Down
##
```

A primera vista, parece que el modelo de regresión logística funciona un poco mejor que un modelo aleatorio. Sin embargo, este resultado es engañoso porque entrenamos y testeamos el modelo en el mismo conjunto de observaciones de 1.250. En otras palabras, $100\% - 52.2\% = 47.8\%$, es la tasa de error de **entrenamiento**. Como hemos visto

anteriormente, la tasa de error de entrenamiento suele ser demasiado optimista: tiende a subestimar la tasa de error de *test*.

Para evaluar mejor la precisión del modelo de regresión logística, podemos ajustar el modelo usando parte de los datos y luego examinar qué tan bien predice los datos **retenidos**. Esto producirá una tasa de error más realista, en el sentido de que en la práctica estamos interesados en el rendimiento de nuestro modelo, no en los datos que usamos para ajustar el modelo, sino en los días futuros para los que se desconocen los movimientos del mercado.

Para implementar esta estrategia, primero crearemos un vector correspondiente a las observaciones de 2001 a 2004. Luego usaremos este vector para crear una base de datos retenidos de observaciones de 2005.

```
train <- (Year < 2005)
Smarket.2005 <- Smarket[!train, ]
dim(Smarket.2005)

## [1] 252   9

Direction.2005 <- Direction[!train]
```

El objeto **train** es un vector de 1.250 elementos, correspondientes a las observaciones de la base de datos. Los elementos del vector que corresponden a observaciones que ocurrieron antes de 2005 se establecen en **VERDADERO**, mientras que los que corresponden a observaciones en 2005 se establecen en **FALSO**. El objeto **train** es un vector *booleano*, ya que sus elementos son **VERDADERO** y **FALSO**. Los vectores *booleanos* se pueden utilizar para obtener un subconjunto de filas o columnas de una matriz. Por ejemplo, el comando **Smarket[train,]** elegirá una submatriz de la base de datos del mercado de valores, correspondiente solo a las fechas anteriores a 2005, ya que son aquellos para los que los elementos de **train** son **VERDADERO**. El símbolo **!** (negación) se puede utilizar para invertir todos los elementos de un vector *booleano*. Es decir, **!train** es un vector similar a **train**, excepto que los elementos que son **VERDADERO** en **train** se cambian a **FALSO** en **!train**, y los elementos que son **FALSO** en **train** se cambian a **VERDADERO** en **!train**. Por lo tanto, **Smarket[!train,]** produce una submatriz de los datos del mercado de valores que contiene solo las observaciones para las cuales **train** es **FALSO**, es decir, las observaciones con fechas en 2005 (252 observaciones).

Ahora ajustamos un modelo de regresión logística usando solo el subconjunto de las observaciones que corresponden a fechas anteriores a 2005, usando el argumento **subset**. Luego obtenemos las probabilidades pronosticadas de que el mercado de valores suba para cada uno de los días de la base de *test*, es decir, para los días de 2005.

```
glm.fits <- glm(Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume,
  data = Smarket, family = binomial, subset = train)

glm.probs <- predict(glm.fits, Smarket.2005, type = "response")
```

Tener en cuenta que hemos entrenado y testeado el modelo en dos bases de datos completamente separadas: el entrenamiento se realizó solo con las fechas anteriores a 2005 y la prueba se realizó solo con las fechas de 2005. Finalmente, calculamos las predicciones para 2005 y las comparamos con los movimientos reales del mercado durante ese período de tiempo.

```
glm.pred <- rep("Down", 252)
glm.pred[glm.probs > .5] <- "Up"
table(glm.pred, Direction.2005)
```

```
##          Direction.2005
## glm.pred Down Up
##      Down 77 97
##      Up   34 44
```

```
mean(glm.pred == Direction.2005)
```

```
## [1] 0.4801587
```

```
mean(glm.pred != Direction.2005)
```

```
## [1] 0.5198413
```

La notación `!=` significa que *no es igual a* (o distinto), por lo que el último comando calcula la tasa de error en los datos de *test*. Los resultados son bastante decepcionantes: la tasa de error de *test* es de 52%. Por supuesto, este resultado no es tan sorprendente, dado que, en general, no se esperaría poder utilizar los rendimientos de días anteriores para predecir el rendimiento futuro del mercado.

Capítulo 7

Arboles de decisión

Los métodos basados en árboles para regresión y clasificación estratifican o segmentan el espacio predictor en varias regiones. Para hacer una predicción para una observación dada normalmente utiliza el valor de respuesta promedio de las observaciones de la base de entrenamiento en la región a la que pertenece. En el caso de clasificación se asigna a la categoría mayoritaria dentro del nodo terminal.

7.1 *Classification and Regression Tree (CART)*

En el caso de árboles de regresión, si Y es la respuesta y X_1 y X_2 los *inputs* se parte el espacio (X_1, X_2) en dos regiones, en base a una sola variable (partición horizontal o vertical). Dentro de cada región proponemos como predicción la media muestral de Y en cada región.

Se busca elegir la variable y el punto de partición de manera óptima (mejor ajuste global). Es computacionalmente inviable considerar cada posible partición del espacio de atributos en J regiones. Por lo tanto, toma un enfoque *top-down greedy* que se conoce como división binaria recursiva. El enfoque es *top-down* porque comienza en la parte superior del árbol (en cuyo punto todas las observaciones pertenecen a una sola región) y luego divide sucesivamente el espacio predictor; cada división se indica a través de dos nuevas ramas más abajo en el árbol. Es *greedy* porque en cada paso del proceso de construcción del árbol, la mejor división se hace en ese paso en particular, en lugar de mirar hacia adelante y elegir una división que conducirá a un mejor árbol en algún paso futuro.

El panel izquierdo de la Figura 7.1 muestra un árbol de regresión para predecir el logaritmo del salario (en miles de dólares) de un jugador de béisbol, basado en la cantidad de años que ha jugado en las ligas mayores y la cantidad de *hits* que hizo en el año anterior. En un nodo interno dado, la etiqueta (de la forma $X_j < t_k$) indica la rama izquierda que sale de esa división, y la rama de la derecha corresponde a $X_j \geq t_k$. Por ejemplo, la división en la parte superior del árbol da como resultado dos ramas grandes. El la rama izquierda corresponde a `Years < 4,5`, y la rama derecha corresponde a `Years >= 4,5`.¹

¹Al estar arriba, `Years` es la variable más importante para explicar el salario.

El árbol tiene dos nodos internos y tres nodos terminales u hojas. El número en cada hoja es la media de la variable de respuesta de las observaciones que caen allí. Por ejemplo, la predicción para el nodo terminal de la izquierda es $e^{5,107} \times 1.000 = \165.174 . El panel derecho la Figura 7.1 muestra las regiones en función de Years y Hits.

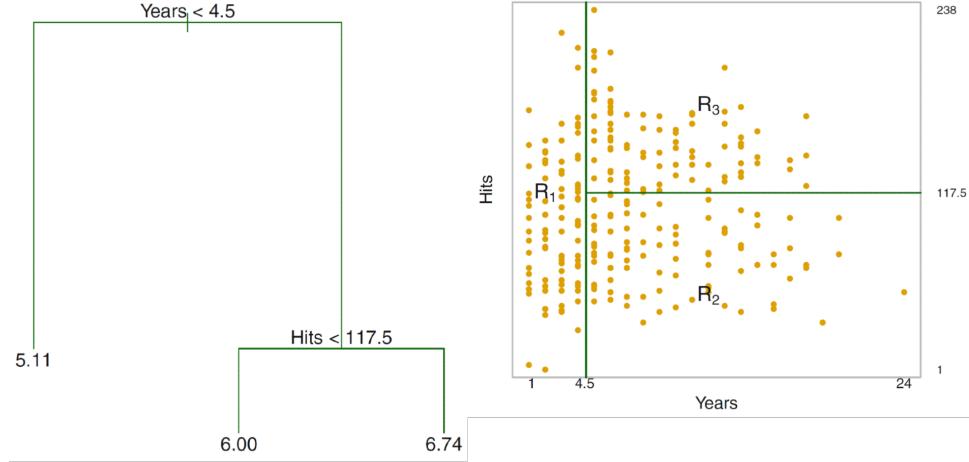


Figura 7.1: Árbol de regresión

Notar:

- Cada región tiene su propio modelo.
- Ciertas variables importan en determinadas regiones y no en otras (*Hits*).

Dado Y y X un vector de p variables con n observaciones el algoritmo busca determinar cuál variable usar para la partición y que punto de esa variable usar para la partición. Si j es la variable de partición y el punto de partición es s , se definen los siguientes semiplanos:

$$R_1(j, s) = X \mid X_j < s$$

$$R_2(j, s) = X \mid X_j \geq s$$

Se trata de buscar la variable de partición X_j y el punto de partición s que resuelvan (minimizar el *EMC* en cada región):

$$\sum_{i:x_i \in R_1(j, s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j, s)} (y_i - \hat{y}_{R_2})^2 \quad (7.1)$$

Donde \hat{y}_{R_1} y \hat{y}_{R_2} es el promedio de la respuesta en las regiones 1 y 2, respectivamente. Para cada variable y punto de partición, la minimización interna se corresponde con la **media** dentro de cada región.²

²Recordar que si se quiere predecir una variable aleatoria Y con una constante m el mejor predictor es su esperanza, es decir, $m = E(Y)$.

¿Cuándo parar de realizar divisiones?

Un árbol demasiado extenso sobreajusta (*overfit*) los datos. Pero dado que el proceso es secuencial y cada corte no mira lo que puede suceder después, si se detiene el proceso demasiado pronto se puede perder un “gran” corte más abajo. *Pruning*: ajustar un árbol grande y luego podarlo (*prune*) usando un criterio de *cost-complexity*.

Weakest link pruning

Un subárbol $T \in T_0$ es un árbol que se obtiene colapsando los nodos terminales de otro árbol (cortando ramas).

Cost-complexity del árbol T :

$$C_\alpha(T) = \sum_{m=1}^{|T|} n_m Q_m(T) + \alpha |T| \quad (7.2)$$

con $Q_m(T) = \frac{1}{n_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$ (impureza) y n_m cantidad de observaciones en cada partición.

El primer término mide el (mal) ajuste y el segundo la complejidad. Cuando $\alpha = 0$, entonces el subárbol T simplemente será igual a T_0 , porque en ese caso (7.2) solo mide el error de entrenamiento. Sin embargo, a medida que α aumenta, hay que pagar un costo por tener un árbol con muchos nodos terminales, por lo que (7.2) tenderá a minimizarse para un subárbol más pequeño.³ Entonces el **objetivo** es, para un α dado, encontrar la poda óptima que minimiza $C_\alpha(T)$.

Para cada α hay un único subárbol T_α que minimiza $C_\alpha(T)$. El mecanismo de búsqueda de T_α (poda óptima dado α) de *weakest link* consiste en eliminar sucesivamente las ramas que producen el mínimo incremento en $\sum_{m=1}^{|T|} n_m Q_m(T)$ (impureza). Recordar que un árbol grande aumenta la varianza, por lo tanto, se colapsa la partición menos necesaria. Un árbol más pequeño con menos divisiones (es decir, menos regiones R_1, \dots, R_J) tiene menor varianza y es más fácil de interpretar a costa de un pequeño sesgo.

El proceso eventualmente colapsa en el nodo inicial, pero pasa por una sucesión de árboles, desde el más grande, hasta el más chico, por el proceso de *weakest link pruning*. El árbol óptimo T_α pertenece a esta sucesión.

Classification tree

Un árbol de clasificación es muy similar a un árbol de regresión, excepto que se utiliza para predecir una respuesta cualitativa en lugar de una cuantitativa. Recordar que para un árbol de regresión, la respuesta predicha para una observación está dada por la respuesta media de las observaciones de entrenamiento que pertenecen al mismo nodo terminal. En contraste, para un árbol de clasificación, predice que cada observación pertenece a la clase que ocurre más comúnmente en las observaciones de entrenamiento en la región a la que pertenece. Se basa en el error de clasificación o índice de Gini (pureza), análogo a *EMC* en un árbol de regresión.

³Idea similar a *Lasso*.

7.2 Bagging

Ventajas y desventajas de *CART*:

- Forma inteligente de representar no linealidades.
- Arriba quedan las variables más relevantes entonces es fácil de comunicar. Reproduce proceso decisorio humano.
- Si la estructura es lineal, *CART* no anda bien.
- Poco robusto, variaciones en los datos modifican el resultado.

Un método de *ensemble* es un enfoque que combina muchos modelos simples en uno único y potencialmente muy poderoso. Los modelos simples se conocen como modelos de aprendizaje débil, ya que por sí mismos pueden generar predicciones mediocres.

Una posible solución es el *bootstrap aggregation* que consiste en tomar como predicción el promedio de las predicciones *bootstrap*.⁴

$$\hat{f}_{bag} = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}(x) \quad (7.3)$$

Esta idea se basa en que la varianza del promedio es menor que la de una predicción sola. Bajo independencia si $V(x) = \sigma^2$ entonces $V(\bar{x}) = \frac{\sigma^2}{n}$. Pero existe el **problema** que si hay un predictor fuerte (siempre va a ser seleccionado primero), los distintos árboles son muy similares entre sí por lo que habrá alta correlación.

7.3 Random Forest

Busca bajar la correlación entre los árboles del *bootstrap*. Al igual que en *bagging*, construye una serie de árboles de decisión en muestras de entrenamiento *bootstrap*. Pero al construir estos árboles de decisión, cada vez que se considera una división en un árbol, se elige como candidatos de división una muestra aleatoria de m predictores del conjunto completo de p predictores ($m < p$).

7.4 Boosting

Tanto *bagging* como *random forest* son técnicas paralelas. Es decir, cada clasificador individual se construye independientemente de los otros. La ventaja es que se pueden crear

⁴Recordar muestras con remplazo Sección 3.8.

muchos clasificadores simultáneamente, pero la desventaja de que los clasificadores no aprenden unos de otros.

En *boosting* los árboles van creciendo secuencialmente, es decir, cada árbol crece usando información de árboles elaborados previamente. *Boosting* no implica un muestreo *bootstrap*; en cambio cada árbol se ajusta a una versión modificada de la base de datos original.

Boosting se basa en el concepto de **weak classifier**: un clasificador marginalmente mejor que tirar una moneda, donde la tasa de error apenas mejor que 0,5. Por ejemplo, un árbol como vimos en *CART* con pocas ramas (*stump*: clasificador con dos ramas). El promedio ponderado de una sucesión de clasificadores débiles genera una notable mejora.

7.4.1 Ada Boost

Adaptive boosting. Dadas las siguientes definiciones:

- $y \in -1, 1$
- Clasificador: $\hat{y} = G(X)$
- Error de predicción = $\frac{1}{N} \sum_{i=1}^N I(y_i \neq G(x_i))$

Se describe el proceso:

1. Comienza con pesos $w_i = \frac{1}{N}$
2. Para $m = 1, \dots, M$:
 - Calcula una predicción
 - Calcula el error de predicción agregado
 - Calcula $\alpha_m = \ln[\frac{1-err_m}{err_m}]$
 - Actualiza los ponderadores $w_i \leftarrow w_i c_i$

con $c_i = \exp[\alpha_m \underbrace{I(y_i \neq G(x_i))}_{0/1}]$

3. *Output*: $G(x) = \text{sgn}[\sum_{m=1}^M \alpha_m G_m(x)]$ (signo del promedio).

Si i estuvo correctamente predicha, $c_i = 1$, entonces no hay ajuste. Caso contrario, $c_i = e^{\alpha_m} = \frac{1-err_m}{err_m} > 1$. Notar que si siempre se predice la clase mayoritaria la tasa de error nunca puede ser mayor al 50% y por eso la expresión anterior es mayor a 1.

En cada paso el método da más importancia relativa a las observaciones mal predichas. **Paso final**: promedio ponderado de predicciones en cada paso. La Figura 7.2 muestra el proceso gráficamente.



Figura 7.2: Ada boost

7.5 Aplicacion practica

La biblioteca `tree` se utiliza para construir árboles de clasificación y regresión.

Tip. Cada una de las metodologías que veremos a continuación posee hiperparámetros cuyos valores *default* pueden ser modificados en búsqueda de una mejor optimización. Se recomienda revisar el `help` de cada una para conocer las alternativas.

7.5.1 Arboles de clasificacion

```
library(tree)
```

Primero usamos árboles de clasificación para analizar la base de datos `Carseats`, donde `Sales` es una variable continua, por lo que comenzamos recodificándola como una variable binaria. Usamos la función `ifelse()` para crear una variable, llamada `High`,⁵ que toma un valor de `Yes` si la variable `Sales` excede 8 y toma un valor de `No` en caso contrario.

```
library(ISLR2)
attach(Carseats)
High <- factor(ifelse(Sales <= 8, 'No', 'Yes'))
```

⁵Notar que se construye como `factor` dado que así lo requieren las funciones que utilizaremos más adelante.

Finalmente, usamos la función `data.frame()` para unir `High` con el resto de los datos de `Carseats`.

```
Carseats <- data.frame(Carseats, High)
```

Ahora usamos la función `tree()` para ajustar un árbol de clasificación con el fin de predecir `High` usando todas las variables excepto `Sales`. La sintaxis de la función `tree()` es bastante similar a la de la función `lm()`.

```
tree.carseats <- tree(High ~ . - Sales, Carseats)
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"      "Price"        "Income"        "CompPrice"     "Population"
## [6] "Advertising"    "Age"          "US"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

Vemos que la tasa de error de entrenamiento es 9%. Para árboles de clasificación, la desviación reportada en la salida de `summary()` esta dada por:

$$-2 \sum_m \sum_k n_{mk} \log \hat{p}_{mk},$$

donde n_{mk} es el número de observaciones en el nodo terminal m que pertenecen a la clase k . Una desviación pequeña indica un árbol que proporciona un buen ajuste a los datos (de entrenamiento). La **desviación media residual** informada es simplemente la desviación dividida por $n - |T_0|$, que en este caso es $400 - 27 = 373$.

Una de las propiedades más atractivas de los árboles es que se pueden representar gráficamente. Usamos la función `plot()` para mostrar la estructura de árbol y la función `text()` para mostrar las etiquetas de los nodos. El argumento `pretty = 0` indica a R que incluya los nombres de categoría para cualquier predictor cualitativo, en lugar de simplemente mostrar una letra para cada categoría.

```
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



La variable más importante para **Sales** parece ser la ubicación de las estanterías, ya que la primera rama diferencia las ubicaciones **Good** de las ubicaciones **Bad** y **Medium**.

Si solo escribimos el nombre del objeto del árbol, R imprime la salida correspondiente a cada rama del árbol. R muestra el criterio de división (ej. **Price < 92,5**), el número de observaciones en esa rama, el desvío, la predicción general para la rama (**Yes** o **No**) y la fracción de observaciones en esa rama que toma valores de **Yes** y **No** (entre paréntesis). Las ramas que conducen a los nodos terminales se indican con asteriscos.

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##      * denotes terminal node
##
## 1) root 400 541.500 No ( 0.59000 0.41000 )
## 2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
## 4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##     8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##        16) CompPrice < 110.5 5  0.000 No ( 1.00000 0.00000 ) *
##        17) CompPrice > 110.5 5  6.730 Yes ( 0.40000 0.60000 ) *
##     9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##        18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##        19) Population > 207.5 20  7.941 Yes ( 0.05000 0.95000 ) *
## 5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
## 10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##    20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
```

```

##          40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##            80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##              160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##              161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##                81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##                41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##              21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##                42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##                  84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##                  85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##                    170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##                    171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                      342) Age < 49.5 13   16.050 Yes ( 0.30769 0.69231 ) *
##                      343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##                    43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##                      86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##                      87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##                        174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                          348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                          349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##                        175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##                      11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##                        22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##                          44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##                            88) Income < 100 9   12.370 No ( 0.55556 0.44444 ) *
##                            89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##                          45) CompPrice > 130.5 11  0.000 Yes ( 0.00000 1.00000 ) *
##                        23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##                          46) CompPrice < 122.5 10  0.000 No ( 1.00000 0.00000 ) *
##                          47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##                            94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##                            95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##                          3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##                            6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##                              12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##                                24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##                                25) Price > 109 9   11.460 No ( 0.66667 0.33333 ) *
##                              13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##                            7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##                              14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##                              15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *

```

Para evaluar correctamente la *performance* de un árbol de clasificación con estos datos, debemos estimar el error de *test* en lugar de calcular el error de entrenamiento. Dividimos

las observaciones en la base de entrenamiento y una base de *test*,⁶ se construye el árbol usando la base de entrenamiento, y se evalúa su desempeño en los datos en *test*. La función `predict()` se usa para este propósito. En el caso de un árbol de clasificación, el argumento `type = "class"` instruye a R para devolver una predicción de clase. Este modelo produce predicciones correctas en alrededor del 77% de los casos en la base de datos de *test*.

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)

##           High.test
## tree.pred  No Yes
##       No 104 33
##       Yes 13 50

(104 + 50) / 200

## [1] 0.77
```

A continuación, consideramos si podar el árbol podría llevar a mejores resultados. La función `cv.tree()` realiza una *cross-validation* para determinar el nivel óptimo de complejidad del árbol; se utiliza *cost complexity pruning* para seleccionar una secuencia de árboles.

Usamos el argumento `FUN = prune.misclass` para indicar que queremos que la tasa de error de clasificación guíe el proceso de *cross-validation* y poda, en lugar del valor predeterminado para la función `cv.tree()` que es el desvío. La función `cv.tree()` reporta el número de nodos terminales de cada árbol considerado (`size`), así como la tasa de error correspondiente y el valor del parámetro *cost complexity* utilizado (`k`, que corresponde a α en la ecuación (7.2)).

```
set.seed(7)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)

## [1] "size"    "dev"     "k"       "method"
```

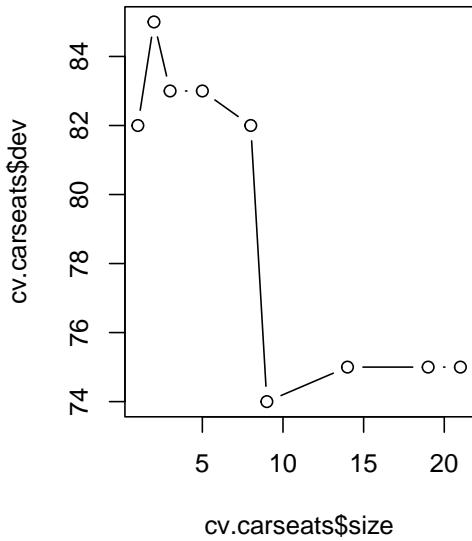
⁶Específicamente se construye un índice para las observaciones de `train` y se usa el opuesto para `test`.

```
cv.carseats
```

```
## $size
## [1] 21 19 14 9 8 5 3 2 1
##
## $dev
## [1] 75 75 75 74 82 83 83 85 82
##
## $k
## [1] -Inf 0.0 1.0 1.4 2.0 3.0 4.0 9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"           "tree.sequence"
```

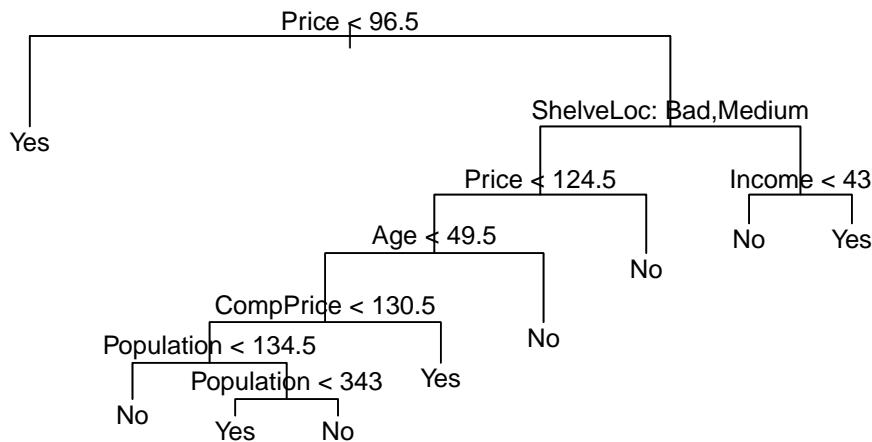
A pesar de su nombre, `dev` corresponde al número de errores de *cross-validation*. El árbol con 9 nodos terminales da como resultado solo 74 errores de *cross-validation*. Graficamos la tasa de error en función tanto del `size` y `k`.

```
par(mfrow = c(1, 2))
plot(cv.carseats$size, cv.carseats$dev, type = "b")
plot(cv.carseats$k, cv.carseats$dev, type = "b")
```



Ahora aplicamos la función `prune.misclass()` para podar el árbol y obtener el árbol de 9 nodos terminales.

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



¿Qué tan bien se desempeña este árbol podado en la base de datos de *test*? Una vez más, aplicamos la función `predict()`.

```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##           High.test
## tree.pred No Yes
##       No  97  25
##       Yes 20  58
```

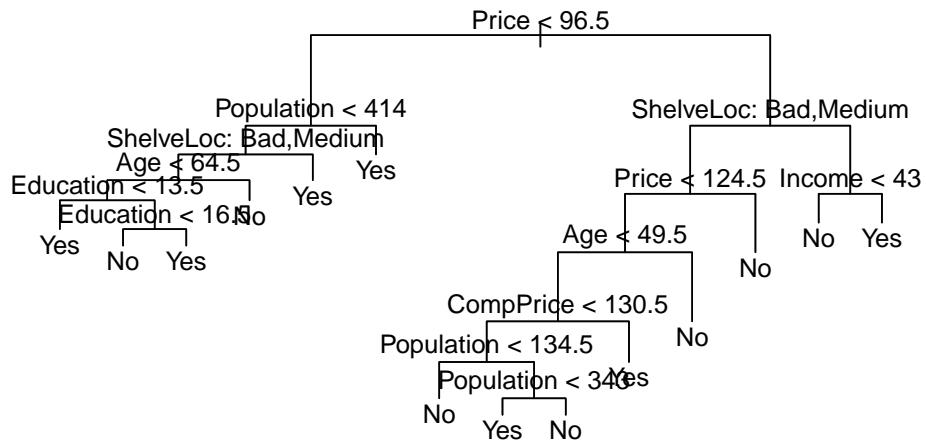
```
(97 + 58) / 200
```

```
## [1] 0.775
```

Ahora el 77,5% de las observaciones de *test* se clasifican correctamente, por lo que el proceso de poda no solo produjo un árbol más interpretable, sino que también mejoró ligeramente la precisión de la clasificación.

Si aumentamos el valor de `best`, obtenemos un árbol podado más grande con menor precisión de clasificación:

```
prune.carseats <- prune.misclass(tree.carseats, best = 14)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##        No 102 31
##        Yes 15 52
```

```
(102 + 52) / 200
```

```
## [1] 0.77
```

Para calcular el área bajo la curva (subsección 3.6.5) se deben estimar probabilidades. Por ello, volvemos a estimar el árbol óptimo con la opción `type = 'vector'` y el contraste contra la clase de interés.

```
library(yardstick)
prune.carseats <- prune.misclass(tree.carseats, best = 9)
tree.pred9 <- predict(prune.carseats, Carseats.test, type = "vector")

prune.carseats <- prune.misclass(tree.carseats, best = 14)
tree.pred14 <- predict(prune.carseats, Carseats.test, type = "vector")

bd = data.frame(obs=factor(High.test), pred=tree.pred9[, 'Yes'])
roc_auc(bd, truth=obs, pred, event_level = 'second')$.estimate

## [1] 0.7921429
```

Curva ROC:

```
library(ggplot2)
plot_roc = function(tab_obs_pred) {
  dat = tab_obs_pred %>% pivot_longer(-obs, names_to="Modelo", values_to="prob")
  dat %>% group_by(Modelo) %>%
    roc_curve(obs, prob, event_level = 'second') %>% autoplot()
}
tab = tibble(
  obs = factor(High.test),
  Tree9 = tree.pred9[, 'Yes'],
  Tree14 = tree.pred14[, 'Yes'],
  Azar = runif(nrow(bd), 0, 1)
)
plot_roc(tab)
```



7.5.2 Comparacion de modelos para clasificacion

Importante. Ideas para el **Trabajo Práctico Final**.

En esta subsección se presenta la aplicación de 3 modelos diferentes `rpart` (CART), `ranger` (*Random Forest*) y `lightgbm` (*Boosting*) y se compara la *performace* a través de la curva ROC. Por lo tanto, en todos los casos se predicen probabilidades.

Notar que, por un lado, se puede optimizar (*tuning*) hiperparámetros y, por el otro, se puede predecir la clase para luego construir la matriz de confusión. Tareas para el TPF...

Comenzamos. Primero se cargan las librerías a utilizar y se generan los datos a utilizar.

```
library(rpart)
library(ranger)
library(lightgbm)
library(Matrix)
library(mltools)
library(data.table)
library(tidyverse)

Carseats.train <- Carseats[train, ]
High.train <- High[train]

formula <- formula(High ~ . - Sales)
```

Aplicación de `rpart`.

```

rpart.mod = rpart(formula,
                  data = Carseats.train,
                  control = rpart.control(minsplit = 20,
                                         minbucket = 6,
                                         cp = 0,
                                         xval = 0,
                                         maxdepth = 16))
rpart.prob = predict(rpart.mod, Carseats.test)

rpartVarImp = as_tibble_row(rpart.mod$variable.importance) %>%
  mutate(id = 1) %>%
  pivot_longer(cols = -id, names_to = 'Variable', values_to = 'Value') %>%
  mutate(id = NULL) %>%
  arrange(desc(Value))
rpartVarImp

```

```

## # A tibble: 10 x 2
##   Variable     Value
##   <chr>       <dbl>
## 1 Price       17.1
## 2 Age         10.5
## 3 CompPrice   9.44
## 4 ShelveLoc   7.47
## 5 Population  5.52
## 6 Income      5.12
## 7 Advertising 3.69
## 8 Education   1.85
## 9 US          1.81
## 10 Urban      0.817

```

Aplicación de **ranger**.

```

set.seed(1234)
ranger.mod = ranger(formula,
                     data = Carseats.train,
                     probability = TRUE,
                     num.trees = 300,
                     min.node.size = 15,
                     mtry = 3,
                     splitrule ='gini',
                     importance ='impurity')

names(ranger.mod)

```

```

## [1] "predictions"           "num.trees"
## [3] "num.independent.variables" "mtry"
## [5] "min.node.size"          "variable.importance"
## [7] "prediction.error"       "forest"
## [9] "splitrule"              "treetype"
## [11] "call"                   "importance.mode"
## [13] "num.samples"            "replace"

ranger.prob = predict(ranger.mod, Carseats.test)

```

Aplicación de `lightgbm`. Muy poderoso pero con sintaxis más compleja.

```

set.seed(4321)

# Union de las 2 bases para el One Hot Encoding
dataset <- rbind(Carseats.train, Carseats.test)
# Borrar la clase de dataset antes del One Hot Encoding
dataset$High = NULL

# Generar One-hot encoding (matriz dispersa)
options(na.action='na.pass')
dataset_sparse <- one_hot(data.table(dataset))

# Para poder separar de nuevo en train/test
fin1 <- nrow(Carseats.train)
ini2 <- fin1 + 1
fin2 <- fin1 + nrow(Carseats.test)

# Formato requerido por LGBM
High.tr = ifelse(High.train=='Yes',1,0) # Se necesita que sea binaria (0/1)
dat_train = lgb.Dataset(data = data.matrix(dataset_sparse[1:fin1,]),
                        label = High.tr,
                        free_raw_data=FALSE)

dat_test <- as.matrix(dataset_sparse[ini2:fin2, ])

lgb.grid = list(objective = "binary",
                 num_iterations=126,
                 bagging_fraction=0.7,
                 feature_fraction=0.451065109894262,
                 learning_rate=0.063063082671059,
                 min_data_in_leaf=71,
                 num_leaves=551,

```

```

lambda_11=0.219802451012323,
min_gain_to_split=0.336965265567414,
max_bin=31)

lgbm.mod = lgb.train(data = dat_train,
                      params = lgb.grid)

## [LightGBM] [Info] Number of positive: 81, number of negative: 119
## [LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing
## You can set `force_col_wise=true` to remove the overhead.
## [LightGBM] [Info] Total Bins 221
.....

lgbm.prob = predict(lgbm.mod, dat_test)

```

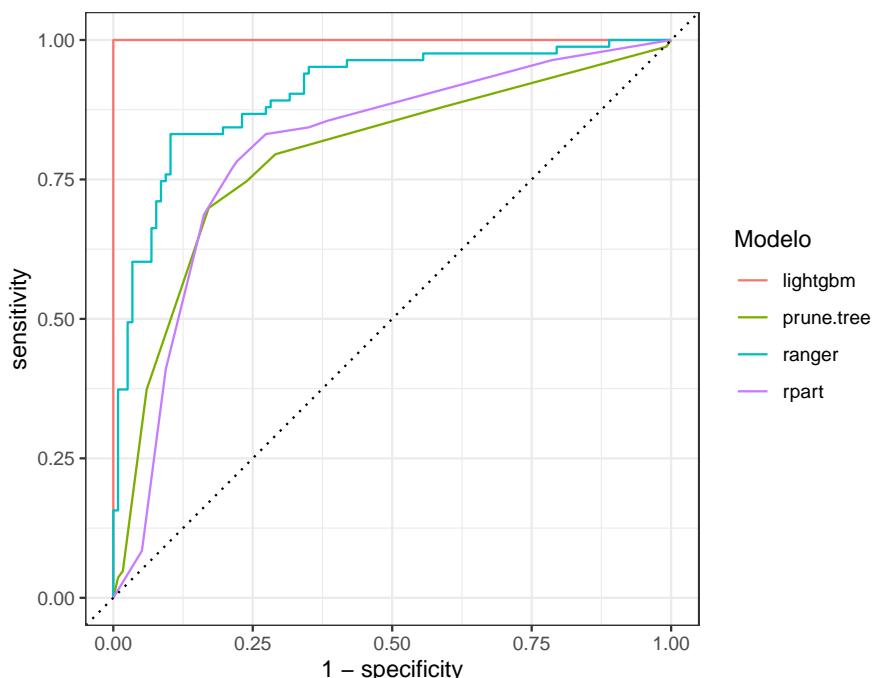
Curvas ROC:

```

tab = tibble(obs = factor(High.test),
              rpart = rpart.prob[, 'Yes'],
              prune.tree = tree.pred9[, 'Yes'],
              ranger = ranger.prob$predictions[, 'Yes'],
              lightgbm = lgbm.prob)

plot_roc(tab)

```



Finalmente se calculan las AUC:

```
tab_auc = tibble(rpart = roc_auc(tab, truth=obs, rpart, event_level = 'second')$.estim
                 prune.tree = roc_auc(tab, truth=obs, prune.tree, event_level = 'second')
                 ranger = roc_auc(tab, truth=obs, ranger, event_level = 'second')$.estima
                 lightgbm = roc_auc(tab, truth=obs, lightgbm, event_level = 'second')$.es
                 aux = 1)
tab_auc = tab_auc %>% pivot_longer(-aux, names_to="Modelo", values_to="AUC") %>%
  dplyr::select(Modelo, AUC) %>% arrange(desc(AUC))
tab_auc

## # A tibble: 4 x 2
##   Modelo      AUC
##   <chr>     <dbl>
## 1 lightgbm    1
## 2 ranger     0.905
## 3 rpart      0.804
## 4 prune.tree  0.792
```

7.5.3 Arboles de regresion

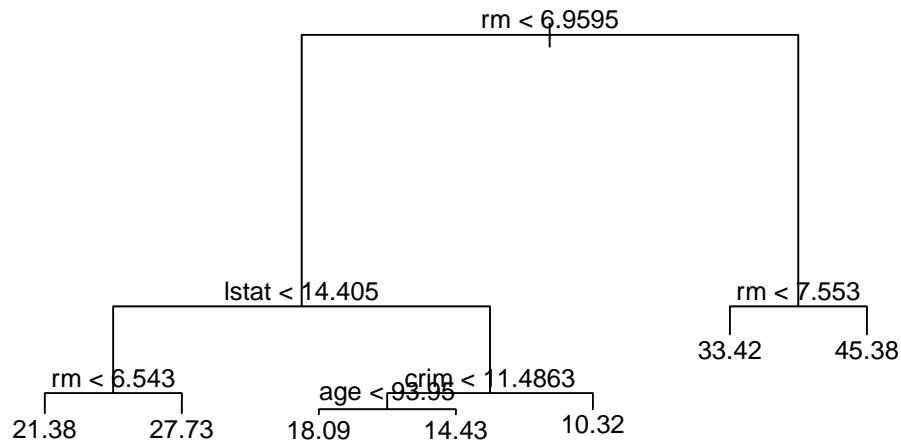
Aquí ajustamos un árbol de regresión a la base de datos Boston. Primero, creamos una base de entrenamiento y ajustamos el árbol a esos datos.

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston) / 2)
tree.boston <- tree(medv ~ ., Boston, subset = train)
summary(tree.boston)

##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "rm"      "lstat"   "crim"    "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##      Min.  1st Qu.  Median  Mean  3rd Qu.  Max.
## -10.1800 -1.7770 -0.1775  0.0000  1.9230 16.5800
```

Notar que la salida de `summary()` indica que solo cuatro de las variables han sido usadas en la construcción del árbol. En el contexto de un árbol de regresión, el desvío es simplemente la suma de los errores al cuadrado del árbol. Ahora graficamos el árbol.

```
plot(tree.boston)
text(tree.boston, pretty = 0)
```



La variable `lstat` mide el porcentaje de personas con bajo nivel socioeconómico, mientras que la variable `rm` corresponde al número promedio de habitaciones. El árbol indica que valores más altos de `rm`, o valores más bajos de `lstat`, corresponden a casas más caras. Por ejemplo, el árbol predice un precio medio de vivienda de \$45.400 para viviendas en distritos censales en los que `rm` $\geq 7,553$.

Podríamos haber estimado un árbol mucho más grande, usando el argumento `control = tree.control(nobs = length(train), mindev = 0)` en la función `tree()`.

Ahora usamos la función `cv.tree()` para ver si podar el árbol mejorará el rendimiento.

```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = "b")
```



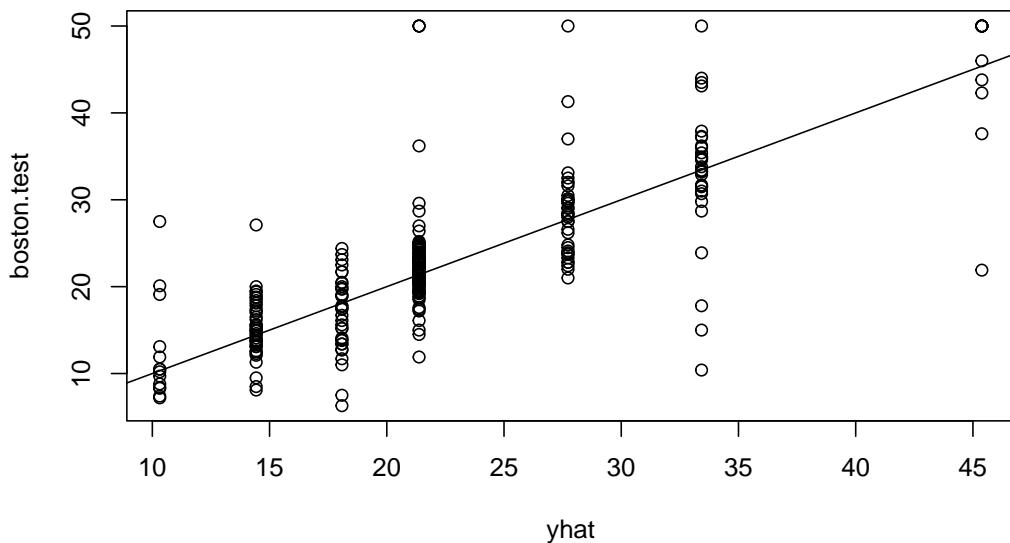
En este caso, el árbol más complejo se selecciona mediante *cross-validation*. Sin embargo, si deseamos podar el árbol, podemos hacerlo de la siguiente manera, usando la función `prune.tree()`:

```
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



De acuerdo con los resultados de la *cross-validation*, usamos el árbol no podado para hacer predicciones en la base de *test*.

```
yhat <- predict(tree.boston, newdata = Boston[-train, ])
boston.test <- Boston[-train, "medv"]
plot(yhat, boston.test)
abline(0, 1)
```



```
mean((yhat - boston.test)^2)
```

```
## [1] 35.28688
```

En otras palabras, el *EMC* de la la base de *test* asociado con el árbol de regresión es de 35,29. Por lo tanto, la raíz cuadrada del *EMC* es de alrededor de 5,941, lo que indica que este modelo produce predicciones de *test* que están (en promedio) dentro de aproximadamente \$5.941 del valor medio real de la vivienda para el distrito censal.

7.5.4 Bagging y Random Forests

Aquí aplicamos *bagging* y *random forests* a los datos de *Boston*, usando el paquete **randomForest** en R. Los resultados exactos obtenidos en esta sección pueden depender de la versión de R y la versión del paquete **randomForest** instalado. Recordar que el *bagging* es simplemente un caso especial de un *random forests* con $m = p$. Por lo tanto, la función **randomForest()** se puede utilizar para realizar *random forests* y *bagging*. Realizamos el *bagging* de la siguiente manera:

```

library(randomForest)
set.seed(1)
bag.boston = randomForest(medv ~ ., data = Boston, subset = train, mtry = 12, importance = TRUE)
bag.boston

##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,
##                Type of random forest: regression
##                Number of trees: 500
##                No. of variables tried at each split: 12
##
##                Mean of squared residuals: 11.40162
##                % Var explained: 85.17

```

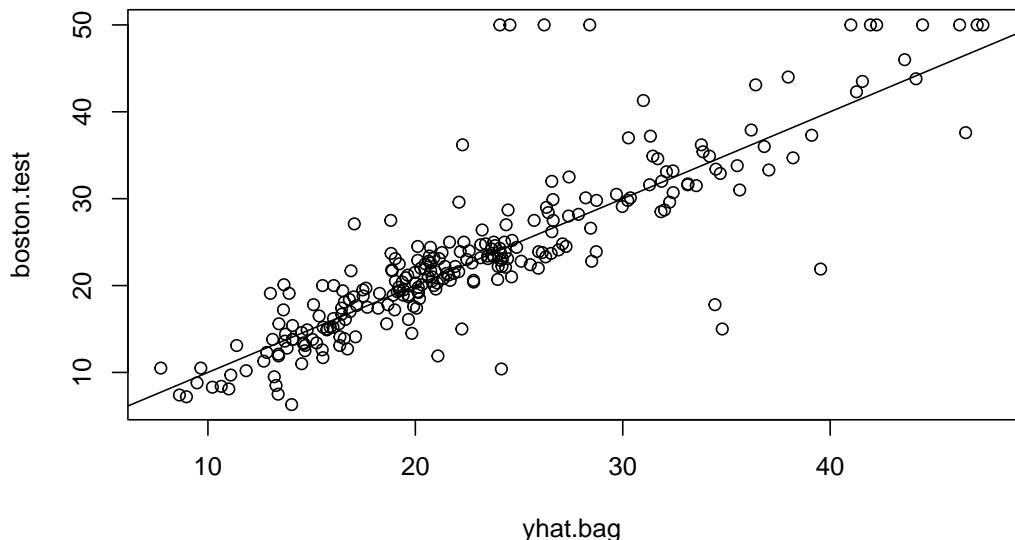
El argumento `mtry = 12` indica que se deben considerar todos los predictores (hay 12 en total) para cada división del árbol; en otras palabras, se debe realizar el *bagging*.

¿Qué tan bien se desempeña este modelo en la base de *test*?

```

yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
plot(yhat.bag, boston.test)
abline(0, 1)

```



```
mean((yhat.bag - boston.test)^2)
```

```
## [1] 23.41916
```

El *EMC* de la base de *test* asociado con el árbol de regresión *bagging* es de 23,42, aproximadamente dos tercios del obtenido utilizando un árbol único podado de manera óptima (35, 29). Podríamos cambiar el número de árboles elaborados por `randomForest()` usando el argumento `ntree`:

```
bag.boston <- randomForest(medv ~ ., data = Boston,
  subset = train, mtry = 12, ntree = 25)
yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
mean((yhat.bag - boston.test)^2)
```

```
## [1] 25.75055
```

La estimación de un *random forests* procede exactamente de la misma manera, excepto que usamos un valor más pequeño del argumento `mtry`. Por defecto, `randomForest()` usa $p/3$ variables cuando construye un *random forests* de árboles de regresión, y \sqrt{p} variables cuando construye un *random forests* de árboles de clasificación. Aquí usamos `mtry = 6`.

```
set.seed(1)
rf.boston <- randomForest(medv ~ ., data = Boston,
  subset = train, mtry = 6, importance = TRUE)
yhat.rf <- predict(rf.boston, newdata = Boston[-train, ])
mean((yhat.rf - boston.test)^2)
```

```
## [1] 20.06644
```

El *EMC* de la base de datos de *test* es de 20,07; esto indica que *random forests* produjo una mejora con respecto a *bagging* en este caso.

Usando la función `importance()`, podemos ver la importancia de cada variable.

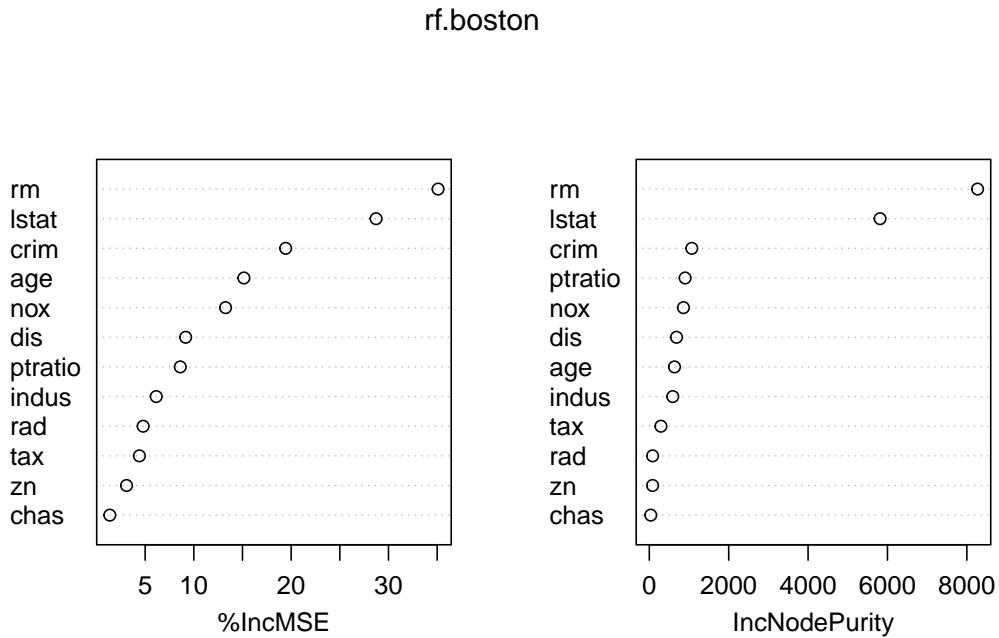
```
importance(rf.boston)
```

	%IncMSE	IncNodePurity
## crim	19.435587	1070.42307
## zn	3.091630	82.19257
## indus	6.140529	590.09536
## chas	1.370310	36.70356
## nox	13.263466	859.97091

```
## rm      35.094741  8270.33906
## age    15.144821  634.31220
## dis     9.163776  684.87953
## rad     4.793720  83.18719
## tax     4.410714  292.20949
## ptratio 8.612780  902.20190
## lstat   28.725343 5813.04833
```

Se reportan dos medidas de importancia de las variables. El primero se basa en la disminución media de la precisión en las predicciones sobre las muestras *out of bag* cuando se permuta una variable determinada. La segunda es una medida de la disminución total en la impureza de los nodos que resulta de las divisiones sobre esa variable, promediada sobre todos los árboles. En el caso de los árboles de regresión, la impureza del nodo se mide por el *RSS* de entrenamiento, y para los árboles de clasificación por el desvío. Los gráficos de estas medidas de importancia se pueden producir utilizando la función `varImpPlot()`.

```
varImpPlot(rf.boston)
```



Los resultados indican que en todos los árboles considerados en el *random forests*, el tamaño de la casa (`rm`) y la riqueza de la comunidad (`lstat`) son las dos variables más importantes.

7.5.5 Boosting

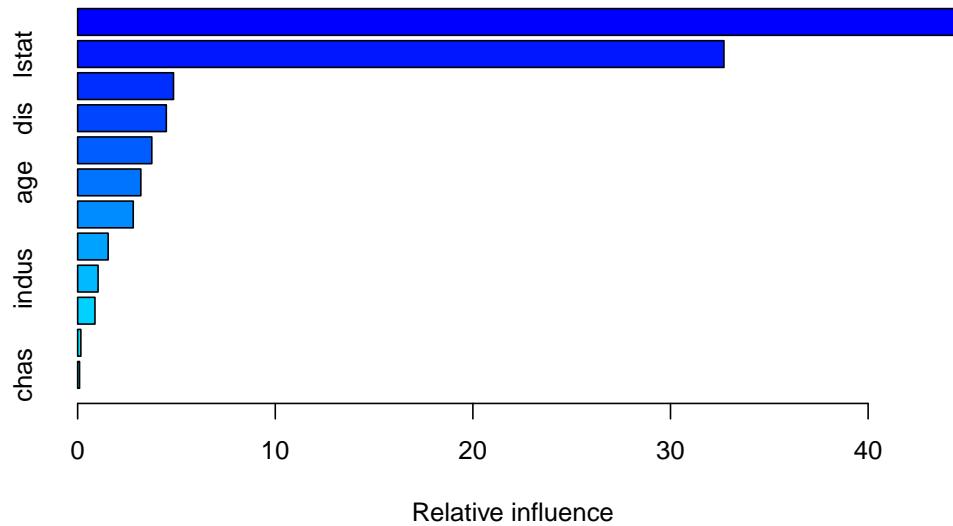
Aquí usamos el paquete `gbm` y la función `gbm()`, para ajustar árboles de regresión *boosting* a la base de datos Boston. Ejecutamos `gbm()` con la opción `distribution = "gaussian"`

ya que este es un problema de regresión; si fuera un problema de clasificación binaria, usaríamos `distribution = "bernoulli"`. El argumento `n.trees = 5000` indica que queremos 5.000 árboles, y la opción `interaction.depth = 4` limita la profundidad de cada árbol.

```
library(gbm)
set.seed(1)
boost.boston <- gbm(medv ~ ., data = Boston[train, ],
                     distribution = 'gaussian', n.trees = 5000, interaction.depth = 4)
```

La función `summary()` produce un gráfico y estadísticas de influencia relativa.

```
summary(boost.boston)
```

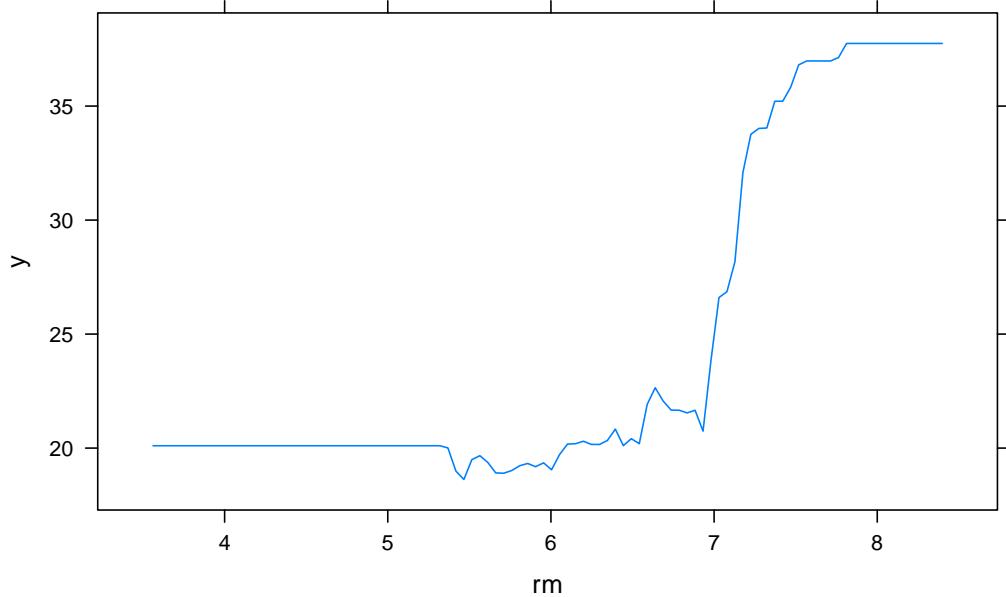


```
##           var      rel.inf
## rm      rm 44.48249588
## lstat   lstat 32.70281223
## crim    crim  4.85109954
## dis     dis  4.48693083
## nox    nox  3.75222394
## age     age  3.19769210
## ptratio ptratio 2.81354826
## tax     tax  1.54417603
## indus   indus 1.03384666
## rad     rad  0.87625748
```

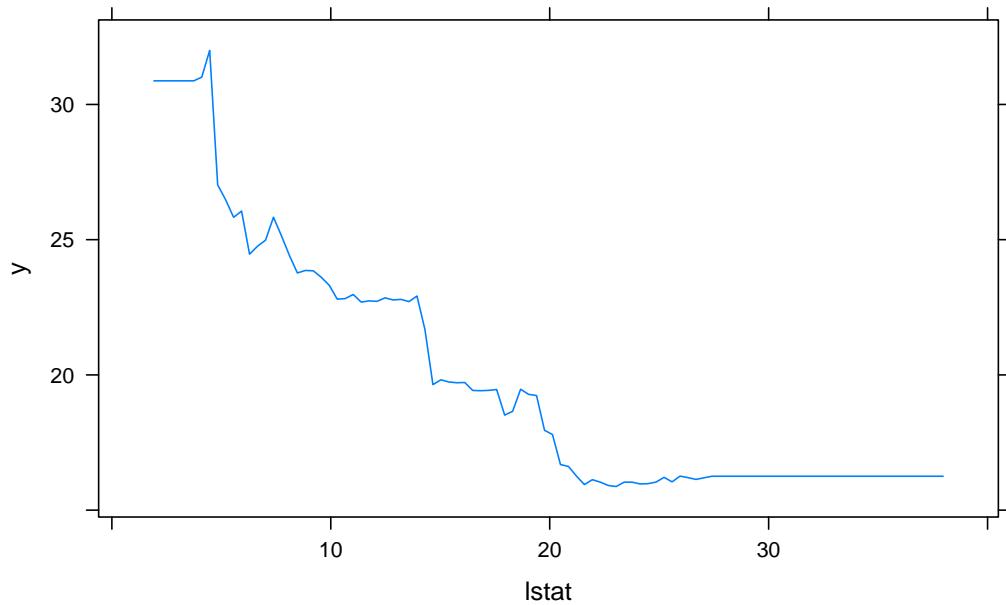
```
## zn          zn  0.16220479
## chas        chas 0.09671228
```

Nuevamente, vemos que `rm` y `lstat` son las variables más importantes. También podemos producir **gráficos de dependencia parcial** para estas dos variables. Estos gráficos ilustran el efecto marginal de las variables seleccionadas en la respuesta después de **integrar** las otras variables. En este caso, como cabría esperar, los precios medios de la vivienda aumentan con `rm` y disminuyen con `lstat`.

```
plot(boost.boston, i = "rm")
```



```
plot(boost.boston, i = "lstat")
```



Ahora usamos el modelo *boosting* para predecir `medv` en la base de *test*:

```
yhat.boost <- predict(boost.boston,
  newdata = Boston[-train, ], n.trees = 5000)
mean((yhat.boost - boston.test)^2)

## [1] 18.39057
```

El *EMC* de *test* obtenido es de 18,39, inferior al *EMC* de *test* de *random forests* y *bagging*. Podemos realizar un *boosting* con un valor diferente del parámetro de contracción λ . El valor predeterminado es 0,001, pero esto se modifica fácilmente. Aquí tomamos $\lambda = 0,2$.

```
boost.boston <- gbm(medv ~ ., data = Boston[train, ],
  distribution = "gaussian", n.trees = 5000,
  interaction.depth = 4, shrinkage = 0.2, verbose = F)
yhat.boost <- predict(boost.boston,
  newdata = Boston[-train, ], n.trees = 5000)
mean((yhat.boost - boston.test)^2)

## [1] 16.54778
```

En este caso, usar $\lambda = 0,2$ produce a un *EMC* de *test* más bajo que $\lambda = 0,001$.

Capítulo 8

Neural Networks

Una *Neural Network* es una red de neuronas artificiales inspiradas en la biología humana configuradas para realizar tareas específicas como la clasificación o el reconocimiento de patrones.

8.1 Single Layer Neural Networks

Una red neuronal toma un vector de entrada de p variables $X = (X_1, X_2, \dots, X_p)$ y construye una función no lineal $f(X)$ para predecir la respuesta Y . Antes vimos modelos de predicción no lineales (ej. árboles) pero lo que distingue a las redes neuronales de estos métodos es la estructura particular del modelo. La Figura 8.1 muestra una red neuronal simple para modelar una respuesta cuantitativa utilizando $p = 4$ predictores. En la terminología de las redes neuronales, las cuatro variables X_1, \dots, X_4 componen las unidades en la capa de entrada. Las flechas indican que cada una de las variables de la capa de entrada alimenta a cada una de las K unidades de la capa de oculta (puede haber K ; en este caso hay 5). El modelo de red neuronal tiene unidades ocultas de la forma:

$$f(X) = \beta_0 + \sum_{k=1}^K \beta_k h_k(X) \quad (8.1)$$

$$= \beta_0 + \sum_{k=1}^K \beta_k g(w_{k0} + \sum_{j=1}^p w_{kj} X_j) \quad (8.2)$$

Se construye aquí en dos pasos. Primero las K activaciones A_k , $k = 1, \dots, K$, en la capa oculta se calculan como funciones de las características de entrada X_1, \dots, X_p :

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^p w_{kj} X_j) \quad (8.3)$$

donde $g(z)$ es una función de activación no lineal que se especifica de antemano. Se puede pensar en cada A_k como una transformación diferente $h_k(X)$ de las características originales. Estas activaciones K de la capa oculta luego alimentan la capa de salida, lo que da como resultado:

$$f(X) = \beta_0 + \sum_{k=1}^K \beta_k A_k \quad (8.4)$$

un modelo de regresión lineal en las $K = 5$ activaciones. Todos los parámetros β_0, \dots, β_k y w_{10}, \dots, w_{Kp} deben estimarse a partir de los datos. La función de activación más utilizada en las redes neuronales modernas es el *ReLU* (*rectified linear unit*), que toma la forma:

$$g(z) = (z)_+ = \begin{cases} 0 & \text{si } z < 0 \\ z & \text{en otro caso} \end{cases} \quad (8.5)$$

El modelo representado en la Figura 8.1 obtiene 5 variables nuevas mediante el cálculo de 5 combinaciones lineales diferentes de X y luego transforma cada una a través de una función de activación $g(\cdot)$, el modelo final es lineal en estas últimas.

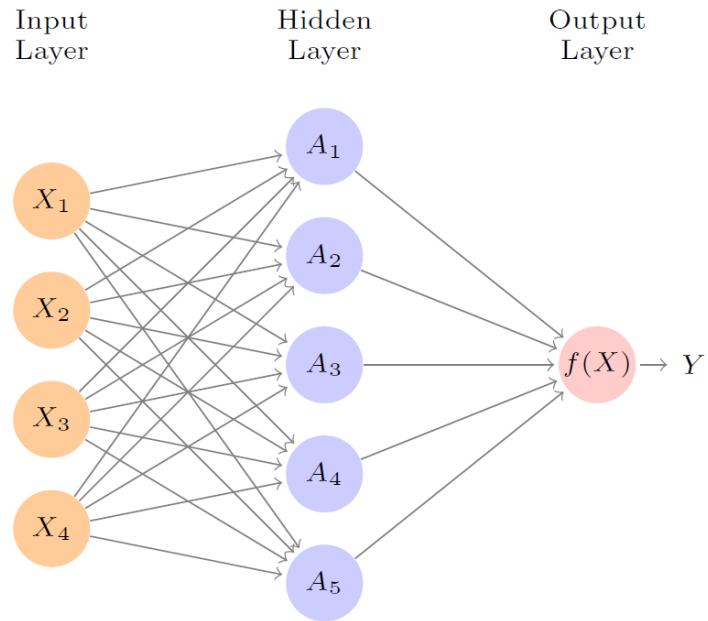


Figura 8.1: Single Layer Neural Network

La no linealidad en la función de activación $g(\cdot)$ es esencial, ya que sin ella el modelo $f(X)$ en la ecuación (8.2) colapsaría en un modelo lineal simple en X_1, \dots, X_p . Además, tener una función de activación no lineal permite que el modelo capture no linealidades complejas y efectos interacción.

Ajustar una red neuronal requiere estimar los parámetros desconocidos en la ecuación (8.2). cuando la variable de respuesta es cuantitativa normalmente se busca minimizar el EMC.

Capítulo 9

Analysis de clusters

Clustering se refiere a un conjunto amplio de técnicas para encontrar subgrupos o *clusters*, en una base de datos. Cuando agrupamos las observaciones, se busca dividirlos en grupos distintos para que las observaciones dentro cada grupo sean bastante similares entre sí, mientras que las observaciones en grupos diferentes sean muy distintas entre sí. Por ello, se debe definir qué significa que dos (o más) observaciones sean similares o diferentes.

Si X es una matriz de n filas (observaciones) y p columnas, cada fila es un “punto” de p dimensiones y cada columna se corresponde con una variable. Por ejemplo, una base con 30 alumnos y cuatro notas de distintas materias (cada alumno es un “punto”).

Clustering busca armar grupos de puntos con esos datos. Se trata de un problema **no supervisado** porque se trata de descubrir estructura, grupos distintos, en base a los datos. Esto es útil, por ejemplo, para segmentación de mercado.

Dissimilarity

Suponer dos puntos $x, z \in \mathbb{R}^p$. ¿Cuán disímiles son x y z , con p coordenadas cada uno?

En el caso de variables cuantitativas la distancia euclídea¹ es:

$$d(x, z) = \left[\sum_{j=1}^p (x_j - z_j)^2 \right]^{\frac{1}{2}} \quad (9.1)$$

Definición: *Dissimilarity Matrix*

Si D es una matriz $N \times N$ donde $D_{ij} = D(x_i, x_j)$ es el *input* del análisis de *clusters*. Idealmente las D_{ij} son verdaderas distancias, de modo que D es simétrica y con diagonal principal nula. El análisis es muy sensible a la elección de D .

Cluster analysis

Cada punto está indizado por $i \in 1, \dots, N$ y supongamos que sabemos de antemano que hay K *clusters*. A su vez, cada *cluster* está indizado por $k \in 1, \dots, K$. Entonces, un mecanismo de *clusters* asigna cada punto a un solo *cluster*. Es decir:

¹Agrega disimilitudes para cada atributo.

$k = C(i)$, donde $C(i)$ es un “enconder”

$C(i): 1, \dots, N \rightarrow 1, \dots, K$.

Por lo tanto, el análisis de *cluster* busca encontrar $C^*(i)$ óptimo, en base a la matriz de *dissimilarities*.

Si se considera la siguiente función de pérdida:

$$W(C) = \frac{1}{2} \sum_{k=1}^K \left[\sum_{i,j/C(i)=C(j)=k} d(x_i, x_j) \right] \quad (9.2)$$

Intuitivamente $W(C)$ agrega las disimilitudes dentro de cada *cluster*.

Si se define T como la disimilitud total, entre todas las observaciones (no depende de la clusterización):

$$T = \frac{1}{2} \sum_{i,j} d_{i,j} \quad (9.3)$$

$$= \frac{1}{2} \sum_{k=1}^K \left[\sum_{i,j/C(i)=C(j)=k} d(x_i, x_j) \right] + \sum_{k=1}^K \left[\sum_{i,j/C(i)=C(j) \neq k} d(x_i, x_j) \right] \quad (9.4)$$

$$T = W(C) + B(C) \quad (9.5)$$

Donde $B(C)$ es la agregación de las distancias **entre clusters**. Entonces, si se propone como objetivo minimizar la disimilitud dentro de los *clusters*, como T esta fijo, minimizar $W(C)$ es equivalente a maximizar $B(C)$.

9.1 K-Means Clustering

En *K-Means Clustering* se busca dividir las observaciones en un número preespecificado de grupos. Luego el algoritmo asignará cada observación a exactamente uno de los K *clusters*.

La Figura 9.1 muestra los resultados obtenidos al realizar *K-means clustering* en un ejemplo simulado con 150 observaciones en dos dimensiones utilizando tres valores diferentes de K .

$$W(C) = \sum_{k=1}^K N_k \sum_{C(i)=k} \|x_i - \bar{x}_k\|^2 \quad (9.6)$$

donde $\bar{x}_k = (\bar{x}_{1k}, \dots, \bar{x}_{pk})$ es el vector de medias asociado con el *cluster* k y $N_k = \sum_{i=1}^N I(C(i) = k)$.

Algoritmo de *k-means*:

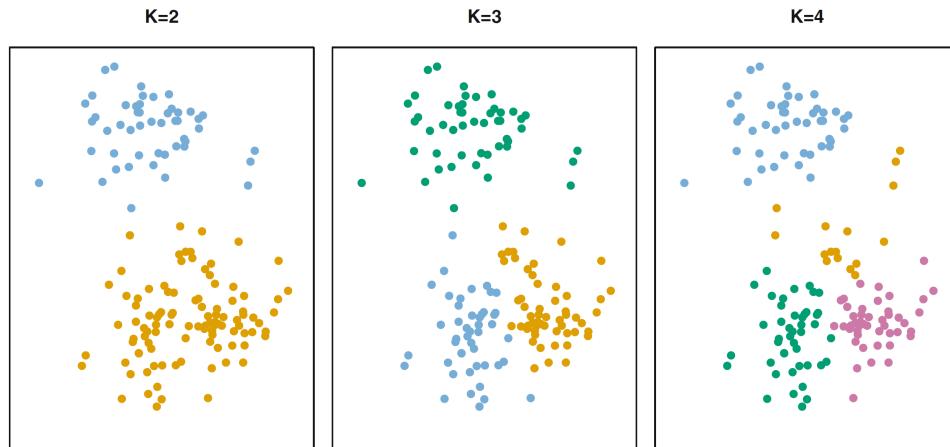


Figura 9.1: Clusters

1. Asignar aleatoriamente un número, del 1 al K , a cada una de las observaciones. Estos sirven como asignaciones de grupos iniciales para las observaciones.
 2. Iterar hasta que las asignaciones de *clusters* dejen de cambiar.
- Para cada uno de los K *clusters*, calcular el centroide del *cluster*. El k -ésimo centroide del *cluster* es el vector de p medias de atributos para las observaciones en el k -ésimo *cluster*.
 - Asignar cada observación al *cluster* cuyo centroide esté más cerca (usando la distancia euclíadiana).

El mecanismo optimiza primero dentro del *cluster* (elige las medias) y luego optimiza reasignando las observaciones, dejando quietas las medias.

Dado que el algoritmo de *K-means* encuentra un óptimo local en lugar de un óptimo global, los resultados obtenidos dependerán de la asignación inicial (aleatoria) de *clusters* de cada observación en el paso 1 del algoritmo. Por esta razón, es importante ejecutar el algoritmo varias veces desde diferentes puntos aleatorios iniciales. Luego se selecciona la mejor solución, es decir, aquella para la cual el objetivo (9.6) es el más pequeño.

La Figura 9.3 muestra los óptimos locales obtenidos ejecutando *K-means clustering* seis veces usando seis asignaciones iniciales diferentes.

Cuestiones prácticas

- Inicialización: puede ser en base a *clusters* o medias.
- Número de *clusters*: No hay un mecanismo comúnmente aceptado. En algunos casos es exógeno.
- La *within dissimilarity* $W(C)$ cae con el número de *clusters*.
- K óptimo se corresponde con un quiebre en el dibujo de $W(C)$ incrementando la cantidad de *clusters* (*trade-off* sesgo-varianza determinado por cantidad de *clusters*).

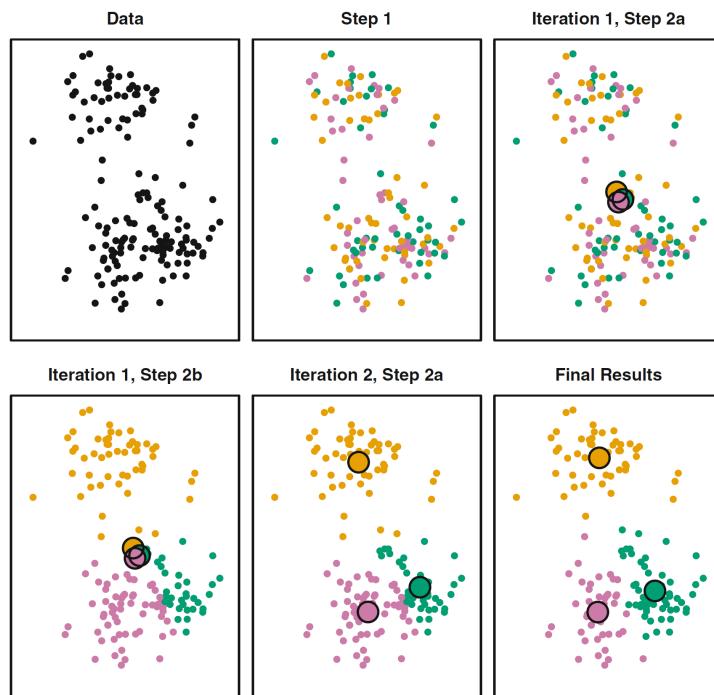


Figura 9.2: Algoritmo

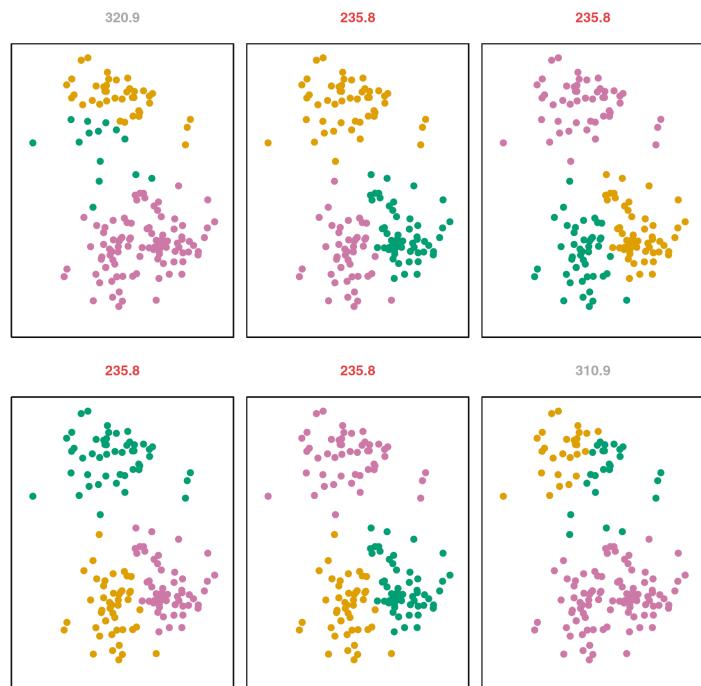


Figura 9.3: Algoritmo

9.2 Aplicacion practica

La función `kmeans()` realiza *K-means clustering* en R. Comenzamos con un ejemplo simulado simple en el que realmente hay dos grupos en los datos: las primeras 25 observaciones tienen un cambio medio relativo a las siguientes 25 observaciones.

```
set.seed(2)
x <- matrix(rnorm(50 * 2), ncol = 2)
x[1:25, 1] <- x[1:25, 1] + 3
x[1:25, 2] <- x[1:25, 2] - 4
```

Ahora estimamos *K-means clustering* con $K = 2$

```
km.out <- kmeans(x, 2, nstart = 20)
```

Las asignaciones de *clusters* de las 50 observaciones están contenidas en `km.out$cluster`.

km.out\$cluster

El *K-means clustering* separó perfectamente las observaciones en dos grupos a pesar de que no proporcionamos ninguna información de grupo a `kmeans()`. Podemos graficar los datos, con cada observación coloreada de acuerdo con su asignación de grupo.

```
plot(x, col = (km.out$cluster + 1),  
      main = "K-Means Clustering Results with K = 2",  
      xlab = "", ylab = "", pch = 20, cex = 2)
```



En este ejemplo, sabíamos que realmente había dos *clusteres* porque generamos los datos. Sin embargo, para datos reales, en general no se conoce el verdadero número de *clusters*. Podríamos haber realizado *K-means clustering* en este ejemplo con $K = 3$.

```
## [1] "cluster"      "centers"       "totss"        "withinss"      "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"

plot(x, col = (km.out$cluster + 1),
      main = "K-Means Clustering Results with K = 3",
      xlab = "", ylab = "", pch = 20, cex = 2)
```

K-Means Clustering Results with K = 3



Cuando $K = 3$, *K-means clustering* divide los dos grupos.

Para ejecutar la función `kmeans()` en R con múltiples asignaciones de *clusteres* iniciales, usamos el argumento `nstart`. Si se usa un valor de `nstart` mayor que uno, entonces *K-means clustering* se realizará usando múltiples asignaciones aleatorias en el Paso 1 del algoritmo, y la función `kmeans()` reportará solo los mejores resultados. Aquí comparamos el uso de `nstart = 1` con `nstart = 20`.

```
set.seed(4)
km.out <- kmeans(x, 3, nstart = 1)
km.out$tot.withinss
```

```
## [1] 104.3319
```

```
km.out <- kmeans(x, 3, nstart = 20)
km.out$tot.withinss
```

```
## [1] 97.97927
```

Tener en cuenta que `km.out$tot.withinss` es la suma total de cuadrados dentro del grupo, que buscamos minimizar realizando *K-means clustering*. Las sumas de cuadrados individuales dentro del grupo están contenidas en el vector `km.out$withinss`.

Se recomienda ejecutar siempre *K-means clustering* con un valor grande de `nstart`, como 20 o 50, ya que de lo contrario puede obtenerse un óptimo local indeseable.

Al realizar un *K-means clustering*, además de utilizar varias asignaciones de agrupamiento iniciales, también es importante establecer una semilla aleatoria usando la función `set.seed()`. De esta manera, las asignaciones de grupos iniciales en el Paso 1 pueden ser replicadas y la salida de *K-means* será totalmente reproducible.

Bibliografía

- Boehmke, B. and Greenwell, B. (2020). *Hands-On Machine Learning with R*.
- James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021). *An Introduction to Statistical Learning*. Springer.
- Simon, N., Friedman, J., Hastie, T., and Tibshirani, R. (2011). Regularization paths for cox's proportional hazards model via coordinate descent. *Journal of Statistical Software*, 39(5):1–13.
- Wickham, H. and Grolemund, G. (2017). *R for Data Science*.
- Wickham, H., Navarro, D., and Pedersen, T. L. (2016). *ggplot2: Elegant Graphics for Data Analysis*.
- Wooldridge, J. (2012). *Introductory Econometrics: A Modern Approach*, volume 5th edition. South-Western College Publishing.