Ana Arias Botey

GRADIENTE DESCENDIENTE

1. Carga de datos y representación

Se tiene una muestra formada por 3 variables diferentes con un total de 100 observaciones. Las características de las variables son:

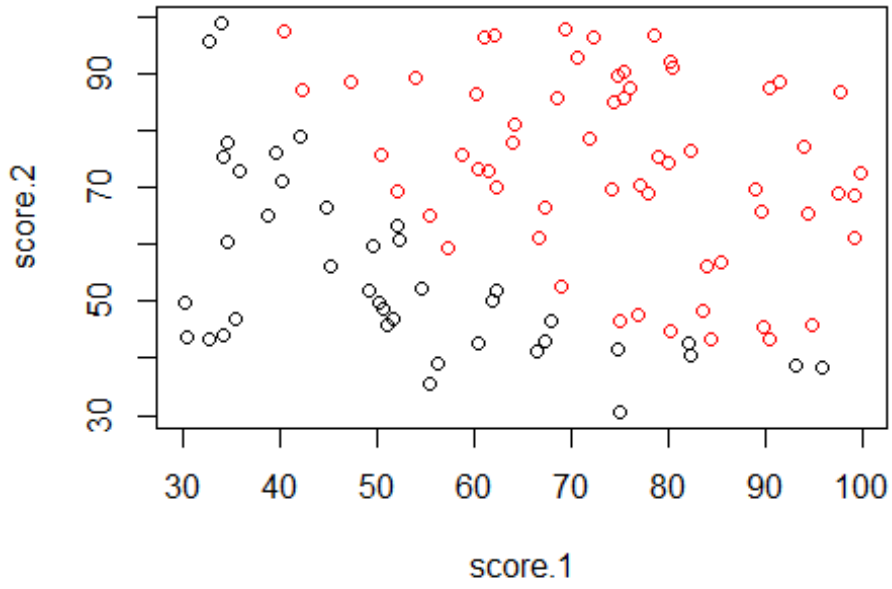
- label: toma el valor 0 ó 1. Si el alumno ha pasado la prueba y por consiguiente es admitido, tomará el valor 1. En caso contrario, toma el valor 0.

- score.1: nota de la primera parte de la prueba

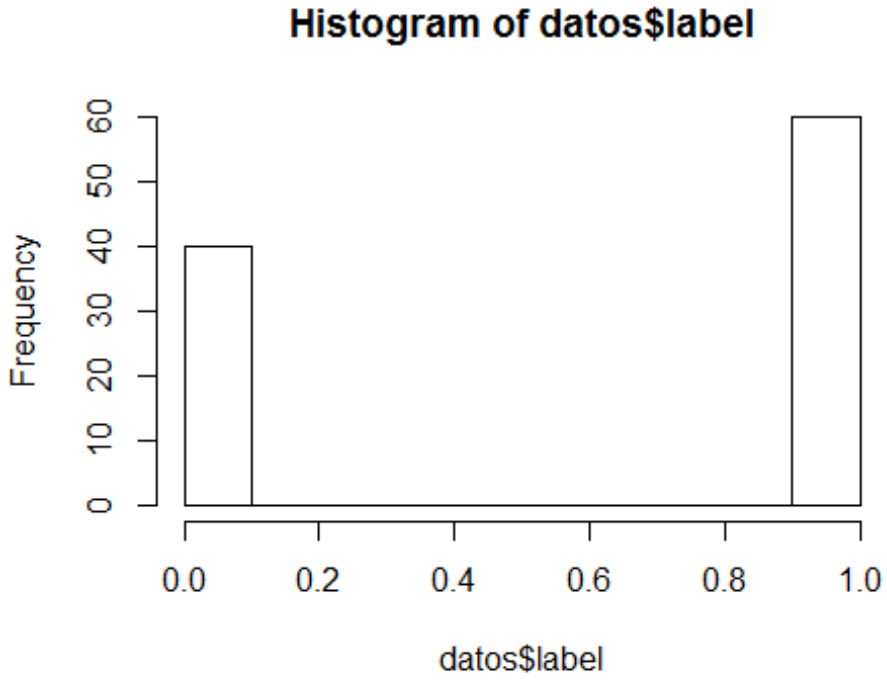
- score.2: nota de la segunda parte de la prueba

El objetivo de éste documento es realizar un análisis de regresión logística donde la variable dependiente es label y las variables explicativas son socore.1 y score.2.

```
setwd("C:/Users/usuario/Desktop/machine learning")
datos <- read.csv("4_1_data.csv")

plot(datos$score.1, datos$score.2, col = as.factor(datos$label), xlab =
"score.1", ylab = "score.2")
```




```r
hist(datos$label)
```



2. Muestra de entrenamiento y otra de validación

Defino muestra de entrenamiento y de test

```
# tomamos una muestra de training y otra de test
set.seed(1234)
n = nrow(datos)
id_train <- sample(1:n, 0.80*n)
datos.train <- datos[id_train,]
datos.test <- datos[-id_train,]
```

```
# train

x.train <- data.frame(rep(1,80), datos.train$score.1,
datos.train$score.2)
x <- as.matrix(x.train)
y<- datos.train$label


# test

x.test <- data.frame(rep(1,20), datos.test$score.1, datos.test$score.2)
x.test <- as.matrix(x.test)

y.test <- datos.test$label
```

Defino la función sigmoide

La ecuación de una regresión logística es:

$$\ln\left(\frac{P_i}{1-P_i}\right) = \beta_0 + \beta_1 * x_{1i} + \beta_1 * x_{2i} + \ldots + \beta_n * x_{ni} + u_i$$

Por tanto, si queremos saber la probabilidad asociada a la observación i es necesario despejar la función haciendo: pi=1/(1+exp(-z))

Se define la función sigmoide como:

```r
sigmoide <- function(z) {
  return( 1/(1+exp(-z)))
}
```

3. función de costes

La función de costes es nuestra función objetivo, es decir la función que queremos minimizar reduciendo su valor. En éste caso, representa los errores que se cometen al estimar la variable dependiente con los coeficientes beta estimados respecto del valor real.

Por tanto, el objetivo es encontrar los valores de los betas óptimos que minimicen el valor de la función objetivo de costes que representa los errores.

```r
funcionCostes <- function(parametros, x, y) {

  n <- nrow(x)
  g <- sigmoide(x %*% parametros)  # %*% este simbolo se define como multiplicaci?n de matrices
  j <-  ((-1)/n)*sum((y*log(g))+(1-y)*log(1-g))

}
```

4. Coste inicial

Tomando como valor inicial de los parámetros (beta) cero, se calcula el coste inicial. El objetivo es reducir éste coste.

```r
# inicialmente obtenemos nuestros coste m?ximo con nuestro vector de par?metros definido como ceros; a partir de ah? deberemos minimizar el coste.

parametros <- rep(0, ncol(x))

# coste m?ximo

coste_inicial = funcionCostes(parametros, x, y)
coste_inicial
```

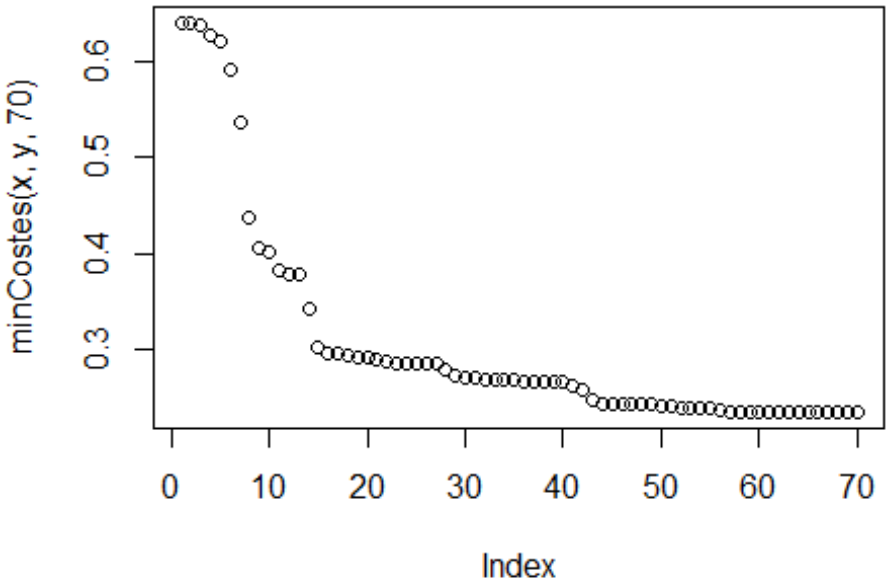
```
## [1] 0.6931472
```

```r
print(paste("coste inicial de la funcion: ",
              convergence <- c(funcionCostes(parametros, x, y)), sep =
""))
```

```
## [1] "coste inicial de la funcion: 0.693147180559945"
```

5. Número óptimo de iteraciones

```r
# vamos a crear un bucle para obtener un mapa de puntos de las
iteraciones y poder representar como influyen el numero de iteraciones en
la funcion de costes y en el número óptimo de parametros


minCostes<-function(x, y, iteraciones=70){


  vect<-NULL
  for(i in (1: iteraciones)) {
    #parametros_optimizados <- optim(par = parametros, fn =
funcionCostes, x = x, y = y, control = list(maxit = i))
    parametros_optimizados <- optim(par = parametros, fn = funcionCostes,
x = x, y = y, control = list(maxit = i))
    vect[i] <- parametros_optimizados$value
  }
  return(vect)
}


plot(minCostes(x,y,70))
```




```
##

  iteraciones<-70
  parametros_optimizados <- optim(par = parametros, fn = funcionCostes, x
= x, y = y, control = list(maxit = iteraciones), hessian=TRUE)
  parametros_optimizados

## $par
## [1] -22.3684419   0.1791714   0.1817466
##
## $value
## [1] 0.2354468
##
## $counts
## function gradient
##      139       70
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]       [,2]      [,3]
## [1,] 0.07678373   4.805986   4.76388
```

```
## [2,] 4.80598617 326.027856 277.37700
## [3,] 4.76388011 277.377004 318.45868
```

6. Explorar la función optim

Explore other options using the optim function (see the methods section of the documentation). Explore other ways in R for estimating the Gradient Descent

```
args(optim)
```

```
## function (par, fn, gr = NULL, ..., method = c("Nelder-Mead",
##     "BFGS", "CG", "L-BFGS-B", "SANN", "Brent"), lower = -Inf,
##     upper = Inf, control = list(), hessian = FALSE)
## NULL
```

Existen diversos métodos para optimizar la función y su convergencia usando optim(). Estos métodos son: method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent")

En la salida de la función, se especifica si han convergido o no:

- Si el resultad es 0: indica que el algoritmo converge

- si el reultado es 1: indica que el algoritmo no converge. A veces basta con cambiar los límites para que converja el algoritmo.

BFGS

Este método hace uso tanto del gradiente como de una aproximación a la inversa de la matriz Hessiana de la función, esto para hacer una aproximación al cálculo de la segunda derivada. Por ser un método de aproximación a la segunda derivada se dice que es un método Quasi-Newtoniano. El problema con el método es el costo computacional para funciones de muchas variables, ya que se almacena una matriz cuadrada de datos tan grande como el cuadrado de la cantidad de variables.

Es adecuado para funciones no lineales de varias variables y la búsqueda del óptimo sin restricciones. Si bien el método hace que la convergencia al óptimo sea rápida por ser de aproximación a la segunda derivada, el costo de procesamiento es bastante alto.

```r
parametros_optimizados1 <- optim(par = parametros, fn = funcionCostes, x = x, y = y, method = "BFGS",lower=-Inf,upper=Inf,control = list(maxit = 60), hessian=TRUE)

print("---Metodo BFGS---")
```

```
## [1] "---Metodo BFGS---"
```

```r
parametros_optimizados1
```

```
## $par
## [1] -22.2587319   0.1787342   0.1804572
##
## $value
## [1] 0.2355024
##
## $counts
## function gradient
##      123       60
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]       [,2]       [,3]
## [1,] 0.07706723   4.818533   4.786323
## [2,] 4.81853301 326.499007 278.398963
## [3,] 4.78632326 278.398963 320.326608
```

Con la matriz hessiana, nos podemos cerciorar de que en efecto, el gradiente es descendente viendo si f es cóncava o convexa::

1. f es convexa sii la matriz hessiana es definida positiva
2. f es cóncava sii la matriz hessiana es definida negativa

Además, por la proposición del criterio de Sylvester, podemos ver si la función alcanza un máximo o un mínimo:

-Si todos los menores principales son mayores que 0, entonces f alcanza un m?nimo relativo en algún punto

- Si los menores principales de índice par son mayores que 0 y los de índice impar son menores que 0, entonces f alcanza un máximo relativo en algún punto.

- Si no es ninguno de los casos anteriores, entonces f es un puto silla

En nuestra matriz hessiana todos los menores son positivos, veamos que ocurre con el determinante:

```
hessiano=parametros_optimizados$hessian
det(hessiano)
```

```
## [1] 11.10634
```

Vemos que también el determinante es positivo, luego por el teorema de silvester se tiene que f alcanza un mínimo. Además por ser todos los menores positivos y la matriz definida positiva, se tiene que f es una función convexa.

```
#Comparaci?n entre las estimaciones
#datos.train
modelo1<-glm(label~score.1+ score.2, family="binomial", data=datos.train)
modelo1$coefficients
```

```
## (Intercept)     score.1     score.2
## -23.8041438   0.1907735   0.1931052
```

```
parametros_optimizados1$par
```

```
## [1] -22.2587319   0.1787342   0.1804572
```

Vemos que son parecidos, luego el modelo está correctamente optimizado

7. comprobamos que los resultados son correctos

```
library(testthat)
```

```
## Warning: package 'testthat' was built under R version 3.4.2
```

```
parametros <-parametros_optimizados1$par
# probability of admission for student (1 = b, for the calculos)
new_student <- c(1,25,78)
print("Probability of admission for student:")
```

```
## [1] "Probability of admission for student:"
```

```
print(prob_new_student <- sigmoide(t(new_student) %*% parametros))
```

```
##             [,1]
## [1,] 0.02378289
```

```
test_that("Test TestGradiente",{
  #parametros <- testGradiente(x = x, y = y)

  new_student <- c(1,25,78)
  prob_new_student <- sigmoide(t(new_student) %*% parametros)
  print(prob_new_student)
  expect_equal(as.numeric(round(prob_new_student, digits = 8)),
0.02378289)

})
```

```
##             [,1]
## [1,] 0.02378289
```

8. predecimos con el testing

```
# primeros definimos nuestro Xi con la columna de unos:

x.test <- data.frame(rep(1,20), datos.test$score.1, datos.test$score.2)
x.test <- as.matrix(x.test)
# x.test

parametros <- as.matrix(parametros_optimizados$par)
probabilidades <- sigmoide((x.test %*% parametros))
probabilidades
```

```
##              [,1]
##  [1,] 0.995870859
##  [2,] 0.017031726
##  [3,] 0.997808003
##  [4,] 0.999598439
##  [5,] 0.517673646
##  [6,] 0.536316828
##  [7,] 0.103738331
##  [8,] 0.026432026
##  [9,] 0.865478338
## [10,] 0.998453228
## [11,] 0.007435511
## [12,] 0.995654579
## [13,] 0.013298261
## [14,] 0.068252093
## [15,] 0.999520468
## [16,] 0.048601239
## [17,] 0.015574169
## [18,] 0.999808930
## [19,] 0.999907306
## [20,] 0.999799396
```

```
#prediccion <- sigmoide(z)
#prediccion
```

8. Matriz de confusión

Test the TestGradientDescent function with the training set (4_1_data.csv). Obtain the confusion matrix.

```
probabilidades[abs(probabilidades-1)<abs(probabilidades-0)]=1
probabilidades[abs(probabilidades-1)>=abs(probabilidades-0)]=0

table(y.test, probabilidades)
```

```
##       probabilidades
## y.test  0  1
##      0  8  0
##      1  0 12
```

```
acierto<-100*sum(diag(table(y.test, probabilidades)))/sum(table(y.test,
probabilidades))
acierto
```

```
## [1] 100
```

9. Implementar un algoritmo equivalente a la función optim

Implement the algorithm step by step using the update rule and an iterative algorithm, do not use the optim function. Research about regularization in logistic regression and explain it

Tenemos que cambiar los valores de Os para minimizar el coste. El descenso de gradiente se usa para minimizar el costo. Cada vez que se actualizan los valores, se espera que el costo se minimice.

Dada una función definida por un conjunto de parámetros, el descenso del gradiente comienza con un conjunto inicial de valores de parámetros (que llamaré theta en el algoritmo) y se desplaza iterativamente hacia un conjunto de valores de parámetros que minimiza la función. Esta minimización iterativa se logra usando cálculo, dando pasos en la dirección negativa del gradiente de función.

La variable alpha controla la longitud de una iteración. Si damos un paso demasiado grande, podemos pasar el mínimo. Sin embargo, si tomamos pequeños pasos, se necesitarán muchas iteraciones para llegar al mínimo.

Para implementar el algoritmo es necesario avanzar hacia abajo por la superficie de nuestra función objetivo. Para ello, es necesario calcular el gradiente en función de los parámetros a optimizar de la función objetivo.

El gradiente determina la pendiente de la recta tangente y de esta forma se puede avanzar por la superficie de la función variando su valor.

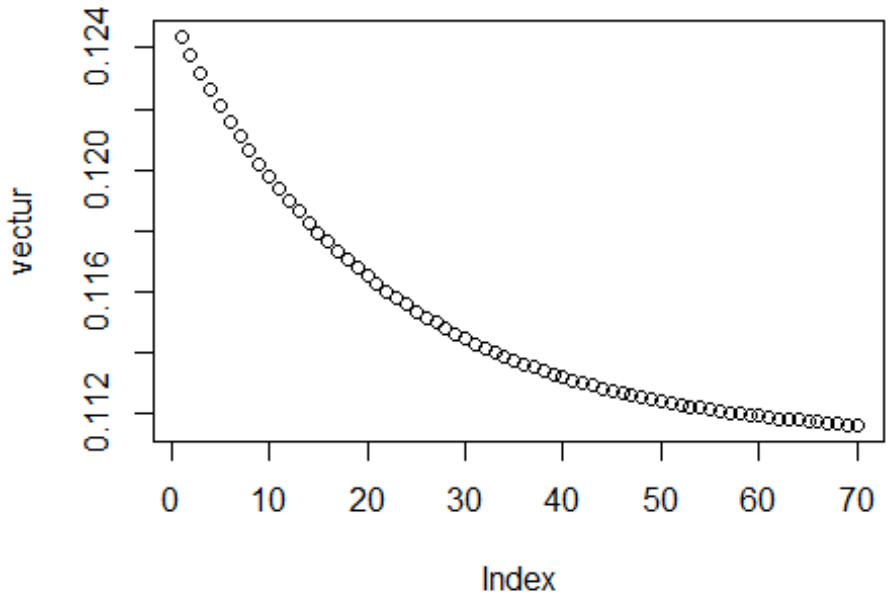
Además se tomará como función de costes:

```
coste <-  sum((sigmoide(x%*%theta)- y)^2)/(2*m)
```

Esta función es equivalente a la que se ha utilizado para hacer el resto de apartados. Sin embargo para derivar esta función de costes respecto de cada uno de los thetas me resultaba un procedimiento menos arduo utilizando ésta función.

```
# x es x.train
# y es y.train
# la sigmoide sigue estando definida igual, definimos una nueva funci?n
de costes y unos parametros nuevos



theta <- c(0,0,0)
theta1 <- 0
theta2 <- 0
theta3 <- 0
m <- nrow(x)

theta <- as.matrix(theta)
coste <-  sum((sigmoide(x%*%theta)- y)^2)/(2*m)
coste

## [1] 0.125

alpha <- 0.00001


# dibujamos el coste vs iteraciones para ver el numero
```

```r
iteraciones <- 70
vectur <- NULL


#estimamos las betas(theta)

for(i in 1:iteraciones) {
  theta1 <- theta1 - alpha * (1/m) * sum((sigmoide(x%*%theta)- y)*x[,1])
  theta2 <- theta2 - alpha * (1/m) * sum((sigmoide(x%*%theta)- y)*x[,2])
  theta3 <- theta3 - alpha * (1/m) * sum((sigmoide(x%*%theta)- y)*x[,3])

  theta <- data.frame(theta1, theta2, theta3)
  theta <- as.matrix(t(theta))
  vectur[i] <- sum((sigmoide(x%*%theta)- y)^2)/(2*m)
}

theta <- data.frame(theta1, theta2, theta3)
theta

##         theta1      theta2      theta3
## 1 1.335494e-05 0.004301632 0.003847023

plot(vectur)
```




```r
parametros_optimizados=theta
```

Como se observa, los parámetros salen números diferentes a los de la función optim. Sin embargo, pueden ser una combinación lineal de los otros.

Una vez estimados los parámetros vemos si ha disminuido el coste. El coste final es el último coeficiente del vector en el que se han acumulado los valores:

```
costefinal=vectur[iteraciones]
```

o bien también se puede determinar con la función definida de costes e introduciendo los parámetros_optimizados.

Por último se determina la matriz de confusión:

```
zi_estimado <- x.test %*% as.matrix(t(theta))

prediccion_zi <- zi_estimado

prediccion_zi[abs(prediccion_zi - 1) < abs(prediccion_zi - 0)] = 1
prediccion_zi[abs(prediccion_zi - 1) > abs(prediccion_zi - 0)] = 0
prediccion_zi
```

```
##       [,1]
##  [1,]    1
##  [2,]    0
##  [3,]    1
##  [4,]    1
##  [5,]    1
##  [6,]    1
##  [7,]    0
##  [8,]    0
##  [9,]    1
## [10,]    1
## [11,]    0
## [12,]    1
## [13,]    0
## [14,]    0
## [15,]    1
## [16,]    0
## [17,]    0
## [18,]    1
## [19,]    1
## [20,]    1
```

```
table(y.test, prediccion_zi)
```

```
##       prediccion_zi
## y.test  0  1
```

```
##      0  8  0
##      1  0 12
```

Al igual que por el método optim, se obtiene que el número de aciertos es el 100%

10. Inestabilidad: tomar diferentes muestras y ver que existe un porcentaje significativo de diferencia entre lo que dice un modelo y lo que dice otro

Tomamos una muestra diferente cambiando la semilla y se repiten exactamente los mismos pasos para determinar la matriz de confusión:

```
# tomamos una muestra de training y otra de test
set.seed(6789)
n = nrow(datos)
id_train <- sample(1:n, 0.80*n)
datos.train <- datos[id_train,]
datos.test <- datos[-id_train,]
```

creamos nuestras matrices de train y test

```
# train

x.train <- data.frame(rep(1,80), datos.train$score.1,
datos.train$score.2)
x <- as.matrix(x.train)
y<- datos.train$label


# test

x.test <- data.frame(rep(1,20), datos.test$score.1, datos.test$score.2)
x.test <- as.matrix(x.test)

y.test <- datos.test$label

iteraciones<-70
  parametros_optimizados <- optim(par = parametros, fn = funcionCostes, x
= x, y = y, method = "BFGS",lower=-Inf,upper=Inf,control = list(maxit =
iteraciones), hessian=TRUE)
  parametros_optimizados

## $par
##              [,1]
## [1,] -25.6965196
## [2,]   0.2219931
## [3,]   0.2047872
##
```

```
## $value
## [1] 0.1727471
##
## $counts
## function gradient
##       81       39
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##            [,1]       [,2]       [,3]
## [1,] 0.05167528   3.171769   3.098491
## [2,] 3.17176942 206.203822 181.074228
## [3,] 3.09849109 181.074228 196.308504
```

```
# primeros definimos nuestro Xi con la columna de unos:

x.test <- data.frame(rep(1,20), datos.test$score.1, datos.test$score.2)
x.test <- as.matrix(x.test)
# x.test

parametros <- as.matrix(parametros_optimizados$par)
probabilidades <- sigmoide((x.test %*% parametros))
probabilidades
```

```
##               [,1]
##  [1,] 4.613250e-05
##  [2,] 5.858813e-02
##  [3,] 1.226435e-01
##  [4,] 1.128310e-02
##  [5,] 9.488318e-01
##  [6,] 2.239961e-02
##  [7,] 1.006552e-01
##  [8,] 3.589052e-01
##  [9,] 2.275686e-01
## [10,] 9.999972e-01
## [11,] 7.526413e-01
## [12,] 2.653891e-04
## [13,] 7.023994e-05
## [14,] 9.890093e-01
## [15,] 7.867953e-01
## [16,] 4.477764e-01
## [17,] 1.637196e-01
## [18,] 9.999832e-01
## [19,] 8.209852e-01
## [20,] 9.999027e-01
```

```r
#prediccion <- sigmoide(z)
#prediccion

probabilidades[abs(probabilidades-1)<abs(probabilidades-0)]=1
probabilidades[abs(probabilidades-1)>=abs(probabilidades-0)]=0

table(y.test, probabilidades)
```

```
##       probabilidades
## y.test  0  1
##      0 12  3
##      1  0  5
```

```r
acierto<-100*sum(diag(table(y.test, probabilidades)))/sum(table(y.test,
probabilidades))
acierto
```

```
## [1] 85
```

Se obtiene un 85% de aciertos que teniendo una muestra tan pequeña de test (30 observaciones) es significativamente diferente a los resultados anteriores donde conseguíamos un 100% de aciertos, por tanto existe inestabilidad.