Aprendizaje Automático: Práctica 1

Carlos Manuel SequÍ

27 de Marzo de 2017

FUNCIONES UTILIZADAS PARA LA PRÁCTICA:

Para generar matrices de datos. Por defecto 2 puntos entre [0,1] de 2 dimensiones

```
simula_unif = function (N=2,dims=2, rango = c(0,1)){
 m = matrix(runif(N*dims, min=rango[1], max=rango[2]),
 nrow = N, ncol=dims, byrow=T)
 m
}
```

Función simula_gaus(N, dim, sigma) que genera un conjunto de longitud N de vectores de dimensión dim, conteniendo números aleatorios gaussianos de media 0 y varianzas dadas por el vector sigma. Por defecto genera 2 puntos de 2 dimensiones.

```
simula_gaus = function(N=2,dim=2,sigma)
{

  if (missing(sigma)) stop("Debe dar un vector de varianzas")
  sigma = sqrt(sigma)  # para la generaciÃ³n se usa sd, y no la varianza
  if(dim != length(sigma)) stop ("El numero de varianzas es distinto de la dimensiÃ³n")

  # genera 1 muestra, con las desviaciones especificadas
  simula_gauss1 = function() rnorm(dim, sd = sigma)
  # repite N veces, simula_gauss1 y se hace la traspuesta
  m = t(replicate(N,simula_gauss1()))
  m
}
```

Simula_recta(intervalo) una funcion que calcula los parámetros de una recta aleatoria, y = ax + b, que corte al cuadrado [-50,50]x[-50,50] (Para calcular la recta se simulan las coordenadas de 2 ptos dentro del cuadrado y se calcula la recta que pasa por ellos), se pinta o no segun el valor de parametro visible.

```
simula_recta = function (intervalo = c(-1,1), visible=F){

   ptos = simula_unif(2,2,intervalo) # se generan 2 puntos
   a = (ptos[1,2] - ptos[2,2]) / (ptos[1,1]-ptos[2,1]) # calculo de la pendiente
   b = ptos[1,2]-a*ptos[1,1]  # calculo del punto de corte

   if (visible) {  # pinta la recta y los 2 puntos
       if (dev.cur()==1) # no esta abierto el dispositivo lo abre con plot
           plot(1, type="n", xlim=intervalo, ylim=intervalo)
       points(ptos,col=3)  #pinta en verde los puntos
```

```
        abline(b,a,col=3)   # y la recta
    }
    c(a,b) # devuelve el par pendiente y punto de corte
}
```

Generar_etiquetas(vector de puntos, pendiente, punto de corte) clasifica los puntos que hay dentro del vector recibido como parámetro en función de si están por encima de la recta generada por los valores a y b. Función hecha en clase.

```
generar_etiquetas = function(puntos = simula_unif(50,2,rango=c(-50,50)), a, b)
{
    #evaluamos los puntos con la función, esto nos da la distancia desde los puntos a las
  #rectas.
    distanciaPuntoRecta = puntos[,2]-a*puntos[,1]-b
    #creamos un vector de etiquetas con el mismo tamaño que distanciaPuntoRecta
    vectorEtiquetas = distanciaPuntoRecta
    #creamos un vector de valores booleanos que separe los positivos de los negativos
    positivos = distanciaPuntoRecta >=0
    negativos = !positivos

    #ponemos los valores 1 o -1 en funcion de los vectores de booleanos obtenidos
    #anteriormente
    vectorEtiquetas[positivos] = 1
    vectorEtiquetas[negativos] = -1

    vectorEtiquetas
}
```

Asignar ruido(conjunto etiquetas, porcentaje) cambia el valor de un porcentaje dado de las etiquetas que hay dentro de conjunto_etiquetas. Función hecha en clase.

```
asignarRuido = function(conjunto_etiquetas, porcentaje=10){

    #Separamos en dos conjuntos distintos los valores que
    #hay en conjunto_etiquetas, negativos y positivos
    positivos = conjunto_etiquetas[conjunto_etiquetas>0]
    negativos = conjunto_etiquetas[conjunto_etiquetas<0]

    #calculamos el número de positivos y negativos a meter ruido
    pos_modificar = round((length(positivos)/100)*porcentaje)
    neg_modificar = round((length(negativos)/100)*porcentaje)

    #metemos el porcentaje de ruido calculado previamente
    positivos[1:pos_modificar]=-1
    negativos[1:neg_modificar]=1

    #los desordenamos para meter el factor de aleatoriedad a la hora de
    #haber metido ruido
```

```
    positivos = sample(positivos)
    negativos = sample(negativos)

    #cambiamos en el conjunto original por
    #los vectores calculados "positivos" y "negativos"
    conjuntoModificados = conjunto_etiquetas
    conjuntoModificados[conjunto_etiquetas > 0] = positivos
    conjuntoModificados[conjunto_etiquetas < 0] = negativos

    conjuntoModificados

}
```

Funcion para pintar la frontera de la función a la que se pueden añadir puntos, y etiquetas.

```
pintar_frontera = function(f,rango=c(-50,50)) {
   x=y=seq(rango[1],rango[2],length.out = 100)
   z = outer(x,y,FUN=f)
  if (dev.cur()==1) # no esta abierto el dispositivo lo abre con plot
          plot(1, type="n", xlim=rango, ylim=rango)
   contour(x,y,z, levels = 0, drawlabels = FALSE,xlim =rango, ylim=rango, xlab = "x",
           ylab = "y")
}
```

Definimos las funciones necesarias para el ejercicio 1.3:

```
f1 = function(x,y)
{
    f = (x-10)^2+(y-20)^2-400
    f
}


f2 = function(x,y)
{
    f = 0.5*(x+10)^2+(y-20)^2-400
    f
}

f3 = function(x,y)
{
    f = 0.5*(x-10)^2-(y+20)^2-400
    f
}

f4 = function(x,y)
{
    f = y-20*x^2-5*x+3
```

```
    f
}
```

Función que genera etiquetas para una función dada en un conjunto de datos.

```
generar_etiquetasParaFunciones = function(puntos = simula_unif(50,2,rango=c(-50,50)), f)
{
    #evaluamos los puntos con la función, esto nos da la distancia desde los puntos
  #a las rectas.
    distanciaPuntoRecta = f(puntos[,1],puntos[,2])
    #creamos un vector de etiquetas con el mismo tamaño que distanciaPuntoRecta
    vectorEtiquetas = distanciaPuntoRecta
    #creamos un vector de valores booleanos que separe los positivos de los negativos
    positivos = distanciaPuntoRecta >=0
    negativos = !positivos

    #ponemos los valores 1 o -1 en funcion de los vectores de booleanos obtenidos
    #anteriormente
    vectorEtiquetas[positivos] = 1
    vectorEtiquetas[negativos] = -1

    vectorEtiquetas
}
```

EJERCICIO 1:

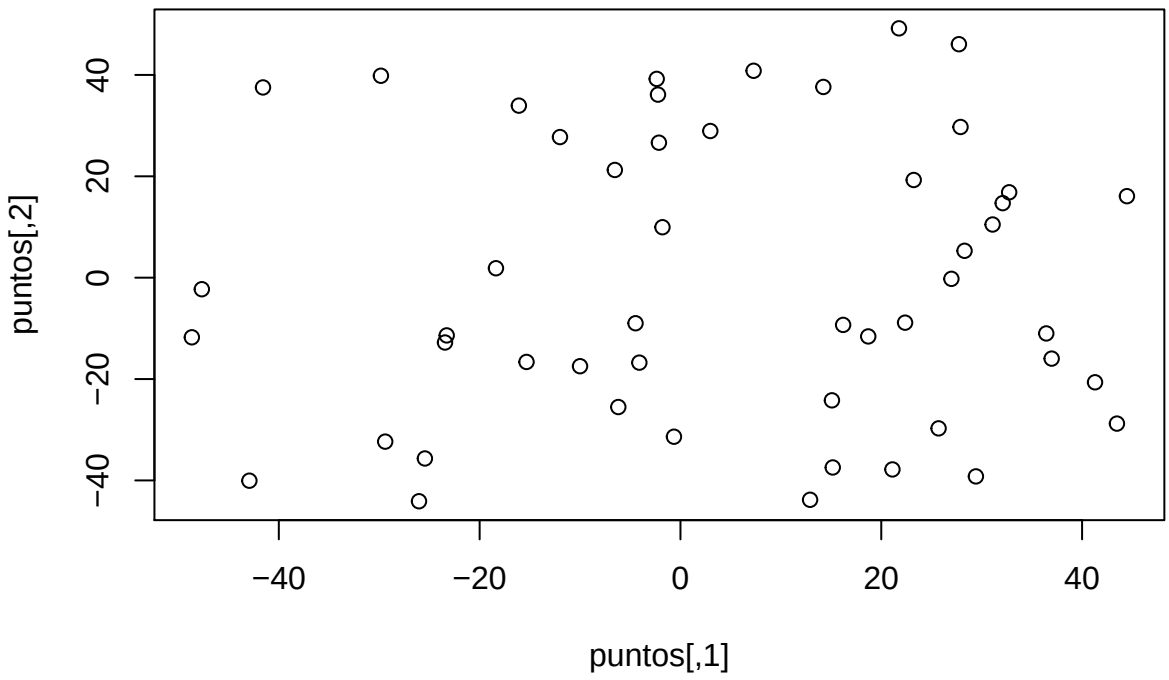
En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir clases de funciones simples.Haremos uso de tres funciones R ya programadas: -simula_unif(N,dim,rango), que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango. -simula_gaus(N,dim,sigma), que calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraido de una distribucción Gaussiana de media 0 y varianza dada, para cada dimension, por la posición del vector sigma. simula_recta(intervalo) , que simula de forma aleatoria los parámetros, v = (a,b) de una recta, y = ax + b, que corta al cuadrado [-50,50]×[-50,50].

Apartado 1 del ejercicio 1:

Dibujar una gráfica con la nube de puntos de salida correspondiente.

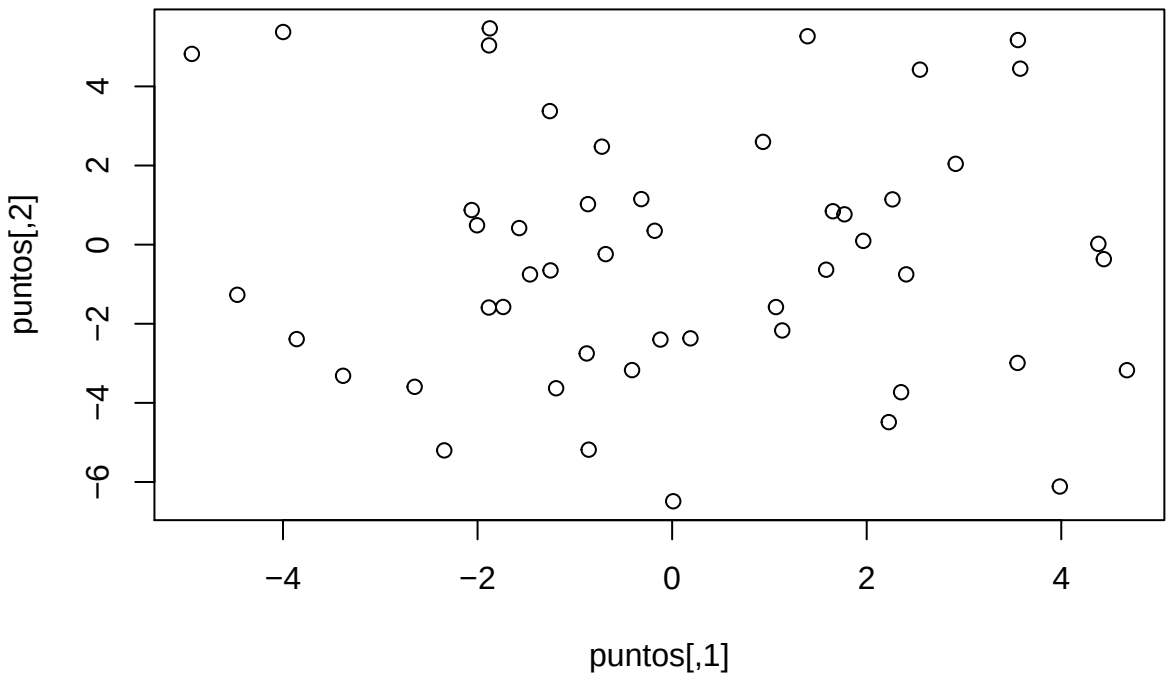
Subapartado A

```
#para la generación de números aleatorios estáticos
set.seed(1)
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = simula_unif(50,2,rango)
#dibujamos la gráfica con la nube de puntos
plot(puntos)
```



Subapartado B

```
#para la generación de números aleatorios estáticos
set.seed(2)
#definimos la varianza con la variable sigma.
sigma = c(5,7)
#generamos los puntos aleatorios con la función simula_gaus.
puntos = simula_gaus(50,2,sigma)
#dibujamos la gráfica con la nube de puntos
plot(puntos)
```



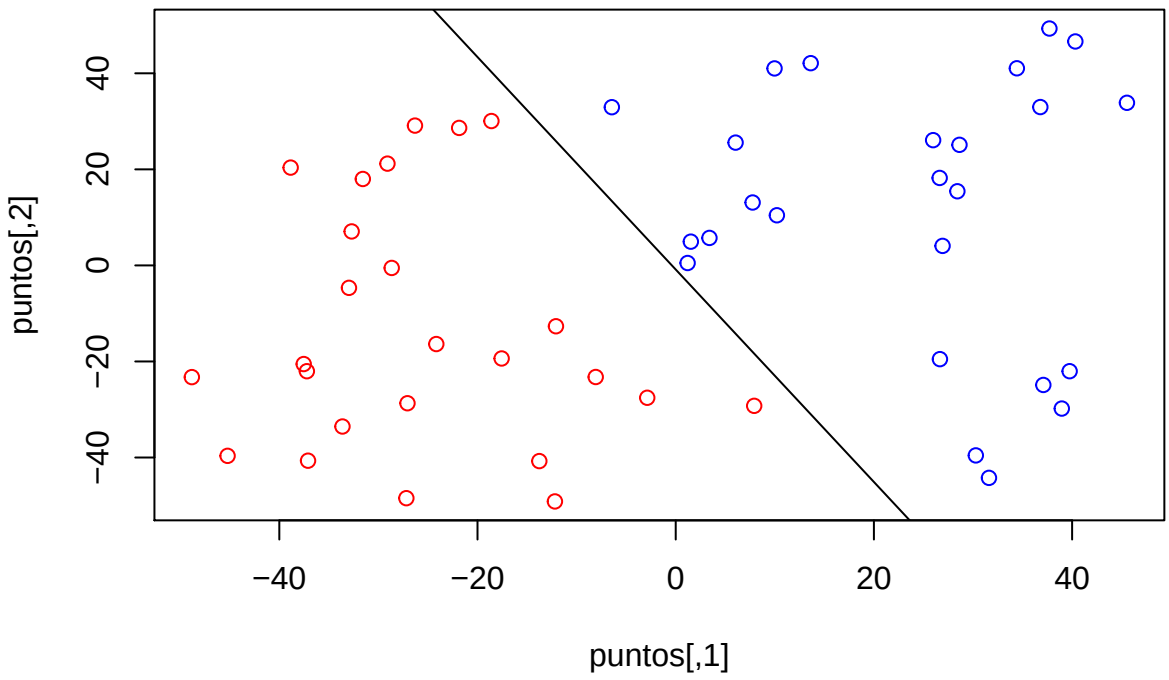
Apartado 2 del ejercicio 1:

Con ayuda de la función simula_unif() generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función f(x,y) = y-ax-b, es decir el signo de la distancia de cada punto a la recta simulada con simula_recta().

Subapartado A

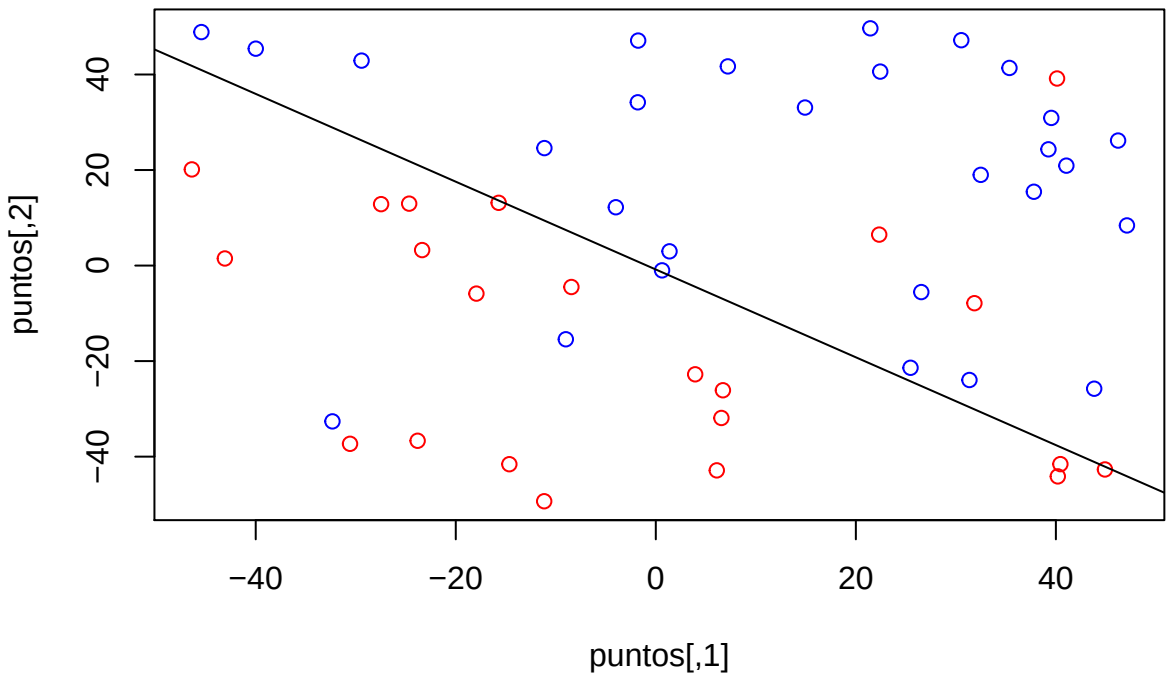
```
#para la generación de números aleatorios estáticos
set.seed(3)
#generamos los coeficientes a y b para evaluar los puntos con dicha función.
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = simula_unif(50,2,rango)
#le pasamos esos puntos a generar_etiquetas para darle etiquetas a cada uno de los
#puntos usando f(x,y) = y???ax???b, en función de si
# están por encima o por debajo.
vectorEtiquetas = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#en función de si las etiquetas recibidas son positivas o negativas los representamos
#en la gráfica de uno u otro color.
plot(puntos, col = vectorEtiquetas+3)
```

```r
#le pasamos la pendiente(a) y el punto de corte(b) a abline para que pinte la recta
abline(coeficientes[2],coeficientes[1])
```



Podemos ver como se generan las etiquetas de los puntos de manera correcta en función de la recta creada.
###Subapartado B

```r
#para la generación de números aleatorios estáticos
set.seed(4)
#generamos los coeficientes a y b para evaluar los puntos con dicha función.
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = simula_unif(50,2,rango)
#le pasamos esos puntos a generar_etiquetas para darle etiquetas a cada uno
#de los puntos usando f(x,y) = y???ax???b, en función de si
# están por encima o por debajo.
vectorEtiquetas = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#llamamos directamente a la función que se encarga de generar ruido en un conjunto de etiquetas.
#Metemos el 10% de ruido
vectorEtiquetasRuido = asignarRuido(vectorEtiquetas,10)
#en función de si las etiquetas recibidas son positivas o
#negativas los representamos en la gráfica de uno u otro color, esta vez con ruido
plot(puntos, col = vectorEtiquetasRuido+3)
#le pasamos la pendiente(a) y el punto de corte(b) a abline para que pinte la recta
abline(coeficientes[2],coeficientes[1])
```



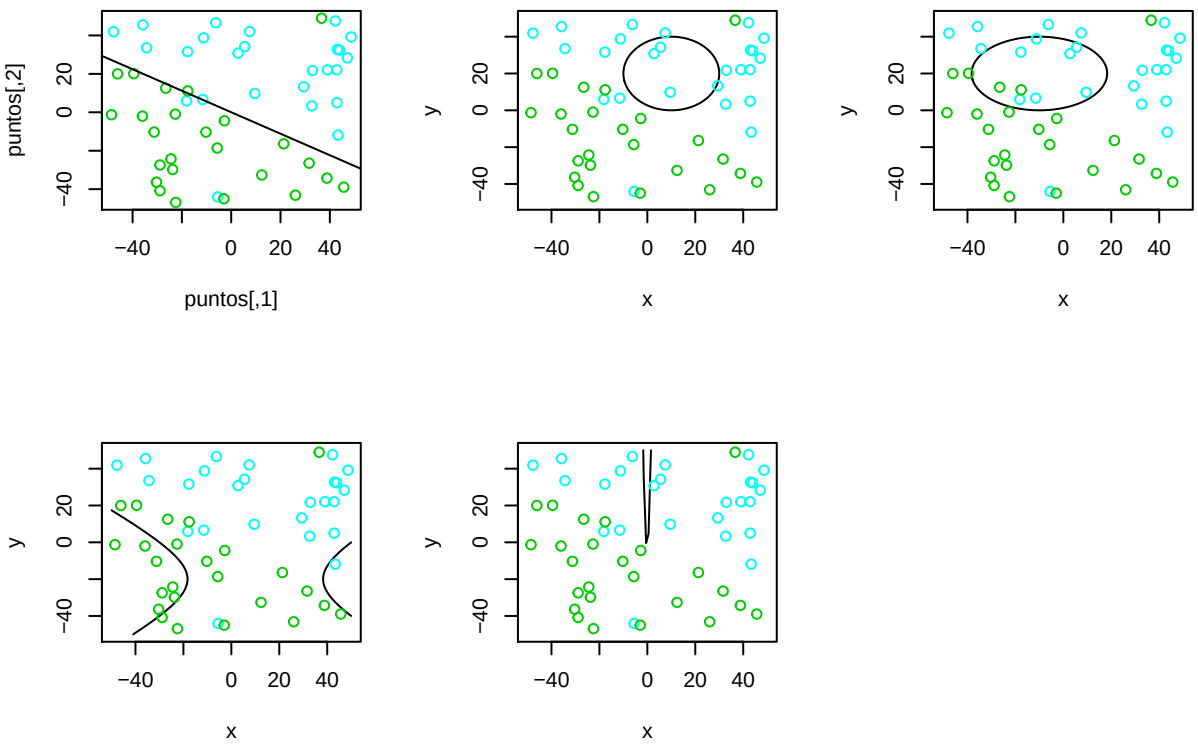
Podemos ver que se asigna ruido de forma correcta a las etiquetas de los puntos generados en la gráfica del apartado anterior.

Apartado 3 del ejercicio 1:

Supongamos ahora que las siguientes funciones de???nen la frontera de clasficación de los puntos de la muestra en lugar de una recta.Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Hemos ganado algo en mejora de clasificación al usar funciones más complejas que la dada por una función lineal ? Explicar el razonamiento.

```
#para la generación de números aleatorios estáticos
set.seed(5)
#generamos los coeficientes a y b para evaluar los puntos con dicha función.
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = simula_unif(50,2,rango)
#para pintar varias gráficas a la vez
par(mfrow=c(2,3))


#le pasamos esos puntos a generar_etiquetas para darle etiquetas a cada uno de los puntos
#usando f(x,y) = y???ax???b, en función de si
# están por encima o por debajo.
```

```
vectorEtiquetas = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#llamamos directamente a la función que se encarga de generar ruido en un conjunto de etiquetas
vectorEtiquetasRuido = asignarRuido(vectorEtiquetas,10)
#en función de si las etiquetas recibidas son positivas o negativas los representamos en
#la gráfica de uno u otro color, esta vez con ruido
plot(puntos, col = vectorEtiquetasRuido+4)
#le pasamos la pendiente(a) y el punto de corte(b) a abline para que pinte la recta
abline(coeficientes[2],coeficientes[1])

#F1
#pintamos la funcion f
pintar_frontera(f1)
#pintamos los puntos
points(x=puntos,col = vectorEtiquetasRuido+4)

#F2
#pintamos la funcion f
pintar_frontera(f2)
#pintamos los puntos
points(x=puntos,col = vectorEtiquetasRuido+4)


#F3
#pintamos la funcion f
pintar_frontera(f3)
#pintamos los puntos
points(x=puntos,col = vectorEtiquetasRuido+4)

#F4
#pintamos la funcion f
pintar_frontera(f4)
#pintamos los puntos
points(x=puntos,col = vectorEtiquetasRuido+4)
```



RESPUESTA. Como podemos observar en la comparativa de las gráficas, para nada hemos obtenido algún beneficio a la hora de realizar la separación de las muestras positivas y las negativas, en este caso, lo óptimo es comenzar con funciones lineales y, en caso de que este no sea un buen clasificador óptimo pues ya seguimos intentándolo con funciones de mayor complejidad. En este caso, como ya he dicho, el clasificador óptimo es el lineal básicamente debido a que hemos usado una función lineal para dar etiquetas a los puntos antes que nada.

EJERCICIO 2

Apartado 1 del ejercicio 2:

Implementar la función ajusta_PLA(datos,label,max_iter,vini) que calcula el hiperplano solución a un problema de clasficación binaria usando el algoritmo PLA. La entradadatos es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, label el vector de etiquetas (cada etiqueta es un valor +1 o ???1), max_iter es el número máximo de iteraciones permitidas y vini el valor inicial del vector. La función devuelve los coe???cientes del hiperplano.

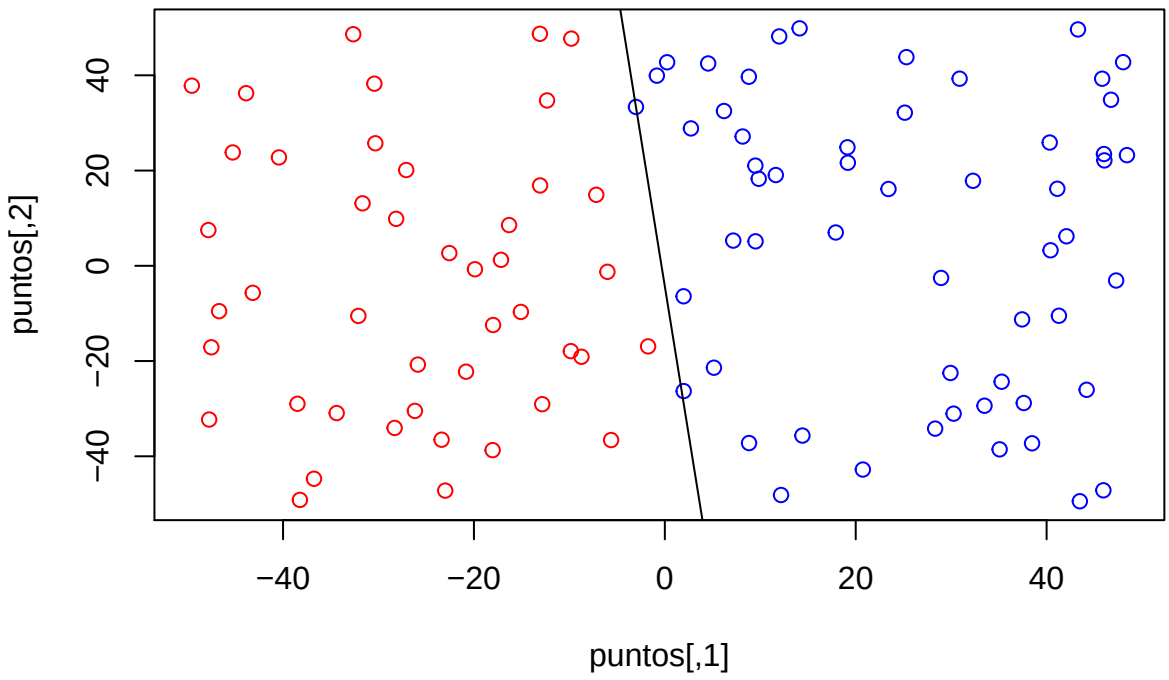
```
#generamos una matriz de puntos aleatorios con una tercera columna de unos al final
genera_puntos = function (N=2,dims=2, rango = c(0,1))
{
     m = matrix(runif(N*dims, min=rango[1], max=rango[2]),
     nrow = N, ncol=dims, byrow=T)
     m[,3] = 1
     m
}
```

```
#Función ajusta_PLA pedida en el ejercicio.
#recibe los puntos creados de forma aleatoria, el máximo de iteraciones
#permitidas antes de encontrar la mejor solución y el vector de pesos
#inicial.
ajusta_PLA = function(datos,label,max_iter,vini)
{
    #usamos un booleano para controlar el nº de iteraciones.
    #Al inicio consideramos que los datos están mal clasificados
    malClasificado = TRUE
    #variable para iterar
    i=0
    #añadimos el umbral
    vini = c(vini,1)
    #hacemos tantas iteraciones como el máximo preestablecido
    while (i < max_iter & malClasificado)
    {
        #cambiamos el
        malClasificado = FALSE
        #iteramos sobre todos los puntos de forma aleatoria
        #(por ello lo del sample)
        for (j in sample(1:length(datos[,1])))
        {
            #en caso de que no esté bien clasificado
            #(la función sign devuelve 1 o -1)
            if (sign(crossprod(datos[j,],vini)) != label[j])
            {
                #seguimos teniendo etiquetas mal clasificadas
                malClasificado = TRUE
                #actualizamos los valores del vector de pesos
                vini = vini + label[j] * datos[j,]
            }
        }

        #Iniciamos de nuevo las iteraciones sobre los puntos
        i = i+1
    }

    #devovlemos los pesos finales obtenidos y las iteraciones utilizadas
    aDevolver = list(pesos = c(vini[1],vini[2],vini[3]) , iteraciones = i)


}


#Función que me devuelve el punto de corte y la pendiente a partir
# de los pesos obtenidos con el algoritmo PLA
calculaPuntoCortePendiente = function(vini)
{

    #inicialmente tenemos la función w1*x1 + w2*x2 + b = 0, hemos de despejar
    #x2 para sacar el punto de corte y la pendiente, para así devolverlo:
    #Despejado -> x2 = -(((w1*x1) + b)/w2)
```

```
    pendiente = -vini[1]/vini[2]
    puntoCorte = -vini[3]/vini[2]
    PCP = c(pendiente,puntoCorte)
    PCP
}


#Función que me devuelve la cantidad de iteraciones utilizadas
#por el algoritmo PLA hasta encontrar la solución óptima
iteracionesPLA = function(puntos, label,max_iter, vini)
{
    #llamamos al algoritmoPLA
    vectorPesosEIterariones = ajusta_PLA(puntos, label, max_iter, vini)
    #almacenamos el valor de las iteraciones producidas
    numIteraciones = vectorPesosEIterariones$iteraciones
    #lo devolvemos
    numIteraciones
}
```

Prueba realizada para comprobar que funcionan correctamente las funciones creadas:

```
set.seed(9)
#generamos los coeficientes a y b que serán los valores iniciales del vector de pesos
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = genera_puntos(100,3,rango)
#generamos el vector de etiquetas
label = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#creamos el vini con los pesos iniciales puestos a 0 como indica el ejercicio
vini = c(0,0)
#obtenemos el vector de pesos, es decir, los coeficientes de la función aproximada y además
#las iteraciones utilizadas por el algoritmo.
vectorPesosEIterariones = ajusta_PLA(puntos, label, 1200, vini)
#obtenemos el punto de corte y la pendiente con los pesos finales obtenidos con el
#algoritmo PLA.
#Lo guardamos en una variable llamada PCP (PuntoCortePendiente), que contiene el punto
# de corte y la pendiente de la función.
#Para acceder a los pesos usamos la sintaxis correspondiente sobre la lista que devuelve mi
#función
vectorPesos = vectorPesosEIterariones$pesos
PCP = calculaPuntoCortePendiente(vectorPesos)

#pintamos los puntos obtenidos con sus etiquetas
plot(puntos, col = label+3)
#le pasamos la pendiente(a) y el punto de corte(b) a abline para que pinte la recta
abline(PCP[2],PCP[1])
```




```
print("ITERACIONES:")
```

```
## [1] "ITERACIONES:"
```

```
print(vectorPesosEIterariones$iteraciones)
```

```
## [1] 49
```

Apartado 2 del ejercicio 2

Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección.1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en [0,1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

SUBAPARTADO A):

```
#usamos los datos del apartado 1.2a
set.seed(3)
#generamos los coeficientes a y b que serán los valores iniciales del vector de pesos
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = genera_puntos(100,3,rango)
```

```r
#generamos el vector de etiquetas
label = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#creamos el vini con los pesos iniciales puestos a 0 como indica el ejercicio
vini = c(0,0)
#recogemos las iteraciones en un vector de iteraciones para luego calcular la media
cantidadIteraciones = replicate(10,iteracionesPLA(puntos, label, 1200, vini))
#calculamos la media de las iteraciones
mediaIteraciones = mean(cantidadIteraciones)
#lo sacamos por pantalla
mediaIteraciones
```

```
## [1] 3.4
```

SUBAPARTADO B):

```r
#usamos los datos del apartado 1.2a
set.seed(3)
#generamos los coeficientes a y b que serán los valores iniciales del vector de pesos
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = genera_puntos(100,3,rango)
#generamos el vector de etiquetas
label = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#recogemos las iteraciones en un vector de iteraciones para luego calcular la media
#con simula_unif(1,2,0:1) hacemos que se genere una matriz de una fila, 2 columnas y
#con valores aleatorios entre 0 y 1.
#Esto se repite (replica) 10 veces.
cantidadIteraciones = replicate(10,iteracionesPLA(puntos, label, 1200, simula_unif(1,2,0:1)))
#calculamos la media de las iteraciones
mediaIteraciones = mean(cantidadIteraciones)
#lo sacamos por pantalla
mediaIteraciones
```

```
## [1] 3.8
```

RESPUESTA.

Como podemos observar, el punto inicial no ejerce demasiada influencia sobre el número de iteraciones medio en la función PLA. Además de esto, salen números bajos de medias de iteraciones debido a que los conjuntos creados son linealmente separables, por lo que la función PLA encuentra de fácil manera la solución al problema de clasificación.

Apartado 3 del ejercicio 2:

Hacer lo mismo que antes usando ahora los datos del apartado 2b de la sección.1. ¿Observa algún comportamiento diferente? En caso a???rmativo diga cual y las razones para que ello ocurra.

SUBAPARTADO A):

```
#generamos los datos del apartado 1.2b
set.seed(4)
#generamos los coeficientes a y b que serán los valores iniciales del vector de pesos
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = genera_puntos(100,3,rango)
#generamos el vector de etiquetas
label = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#llamamos directamente a la función que se encarga de generar ruido en un conjunto de etiquetas
vectorEtiquetasRuido = asignarRuido(label,10)
#creamos el vini con los pesos iniciales puestos a 0 como indica el ejercicio
vini = c(0,0)
#recogemos las iteraciones en un vector de iteraciones para luego calcular la media
cantidadIteraciones = replicate(10,iteracionesPLA(puntos, vectorEtiquetasRuido, 800, vini))
#calculamos la media de las iteraciones
mediaIteraciones = mean(cantidadIteraciones)
#lo sacamos por pantalla
mediaIteraciones
```

```
## [1] 800
```

SUBAPARTADO B):

```
#generamos los datos del apartado 1.2b
set.seed(4)
#generamos los coeficientes a y b que serán los valores iniciales del vector de pesos
coeficientes = simula_recta()
#definimos el rango de generación de puntos aleatorios.
rango = c(-50,50)
#generamos los puntos aleatorios con la función simula_unif.
puntos = genera_puntos(100,3,rango)
#generamos el vector de etiquetas
label = generar_etiquetas(puntos, coeficientes[1], coeficientes[2])
#llamamos directamente a la función que se encarga de generar ruido en un conjunto de etiquetas
vectorEtiquetasRuido = asignarRuido(label,10)
#recogemos las iteraciones en un vector de iteraciones para luego calcular la media
#con simula_unif(1,2,0:1) hacemos que se genere una matriz de una fila, 2 columnas y
#con valores aleatorios entre 0 y 1.
#Esto se repite (replica) 10 veces.
cantidadIteraciones = replicate(10,iteracionesPLA(puntos, vectorEtiquetasRuido, 800,
                                                  simula_unif(1,2,0:1)))
#calculamos la media de las iteraciones
mediaIteraciones = mean(cantidadIteraciones)
#lo sacamos por pantalla
mediaIteraciones
```

```
## [1] 800
```

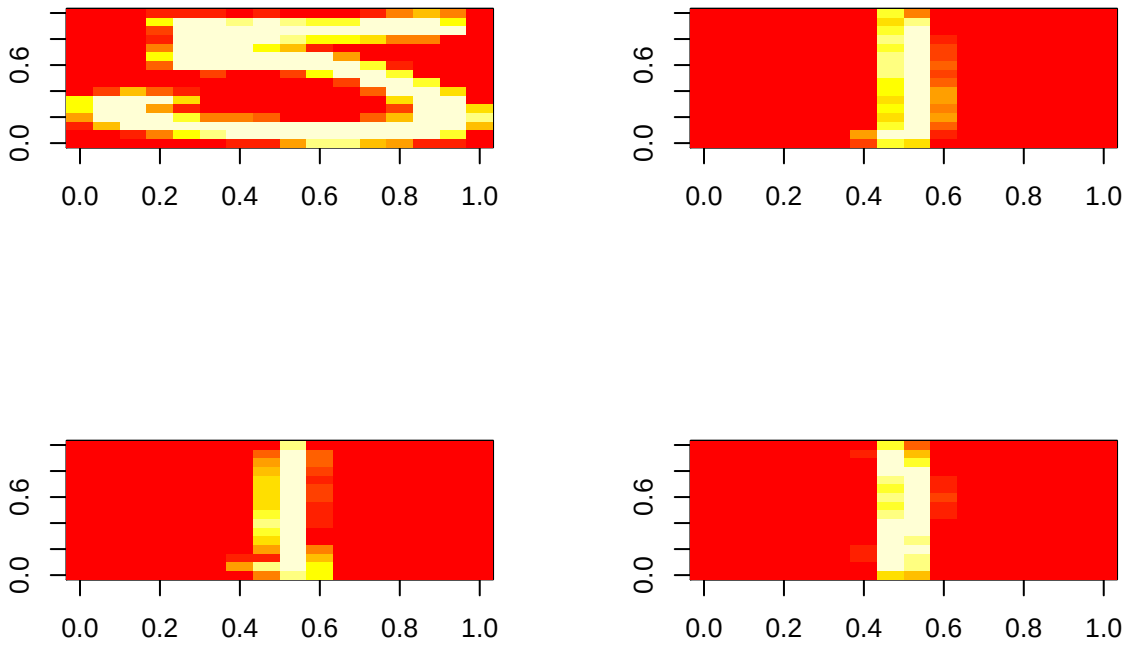
RESPUESTA. Podemos observar ahora de manera clara que los datos generados con ruido no son linealmente separable, es imposible, por lo que la función agota el máximo número de iteraciones del que ha sido dotada (800) sin hayar una solución buena al problema.

EJERCICIO 3

Apartado 1 del ejercicio 3 Leemos datos:

Abra el fichero Zip.Info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el ???chero Zip.train. Lea el ???chero Zip.train dentro de su código y visualice las imágenes (usandoparaTrabajo1.R).Seleccione solo las instancias de los números 1 y 5.Guardelas como matrices de tamaño 16x16. También está disponible el ???chero Zip.test que deberemos usar más adelante.

```
digit.train <- read.table("datos/zip.train",
                          quote="\"", comment.char="", stringsAsFactors=FALSE)

digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,] #Nos da las imagenes que son 1 o
digitos = digitos15.train[,1]  # etiquetas que tienen cada imagen. Nos dice si es un 5 o es un 1
ndigitos = nrow(digitos15.train)

# se retira la clase y se monta una matriz 3D: 599*16*16
grises = array(unlist(subset(digitos15.train,select=-V1)),c(ndigitos,16,16))
rm(digit.train)
rm(digitos15.train)

# Para visualizar los 4 primeros

par(mfrow=c(2,2))
for(i in 1:4){
  imagen = grises[i,,16:1] # se rota para verlo bien
  image(z=imagen)
}
```




```
# etiquetas correspondientes a las 4 imágenes
digitos[1:4]
```

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

Apartado 2 del ejercicio 3

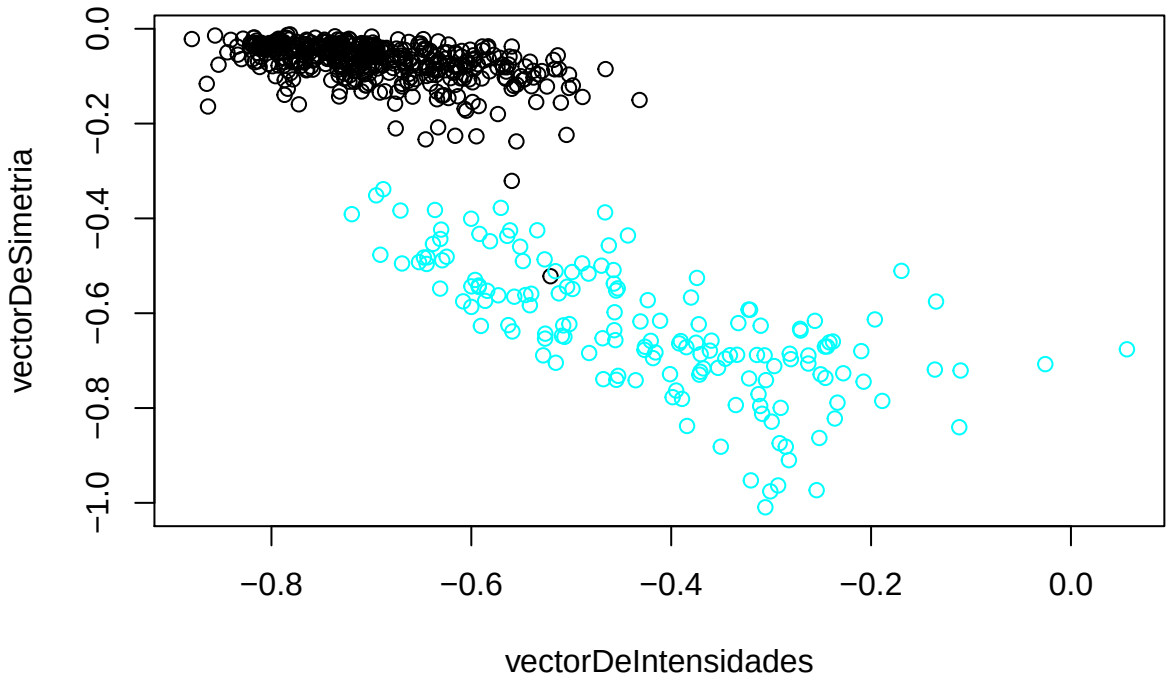
De cada matriz de números (imagen) vamos a extraer dos características: a) su valor medio; y b) su grado de simetría vertical. Para calcular el grado de simetría haremos lo siguiente: a) calculamos una nueva imagen invirtiendo el orden de las columnas; b) calculamos la diferencia entre la matriz original y la matriz invertida; c) calculamos la media global de los valores absolutos de la matriz. Conforme más alejado de cero sea el valor más asimétrica será la imagen. Representar en los ejes { X=Intensidad Promedio, Y=Simetría} las instancias seleccionadas de 1's y 5's.

SUBAPARTADO 1

Función fsimetria sacada del fichero de decsai "para trabajo 1"

```
#devuelve el grado de simetria.
fsimetria <- function(A){
  #realizamos la diferencia entre matriz original y la matriz invertida
  A = abs(A-A[,ncol(A):1])
  #calculamos la media global de los valores absolutos de la matriz
  -mean(A)
}
```

```r
#SUBAPARTADO 2
#creamos un vector de simetria
#apply: le decimos que coja todas las imagenes que contiene "grises", que lo haga con el eje x(el 1, ej
# y que le aplique a todos la función fsimetria
vectorDeSimetria = apply(grises,1,fsimetria)
#creamos un vector de medias(intensidades)
vectorDeIntensidades = apply(grises,1,mean)
#los representamos
plot(vectorDeIntensidades, vectorDeSimetria, col = digitos)
```



Apartado 3 del ejercicio 3

Ajustar un modelo de regresion lineal usando el transformación SVD sobre los datos de (Intensidad promedio, Simetria) y pintar la solución obtenida junto con los datos usados en el ajuste. Las etiquetas serán {-1,1}. Valorar la bondad del resultado usando Ein y Eout (usar Zip.test). (usar Regress_Lin(datos,label) como llamada para la función).

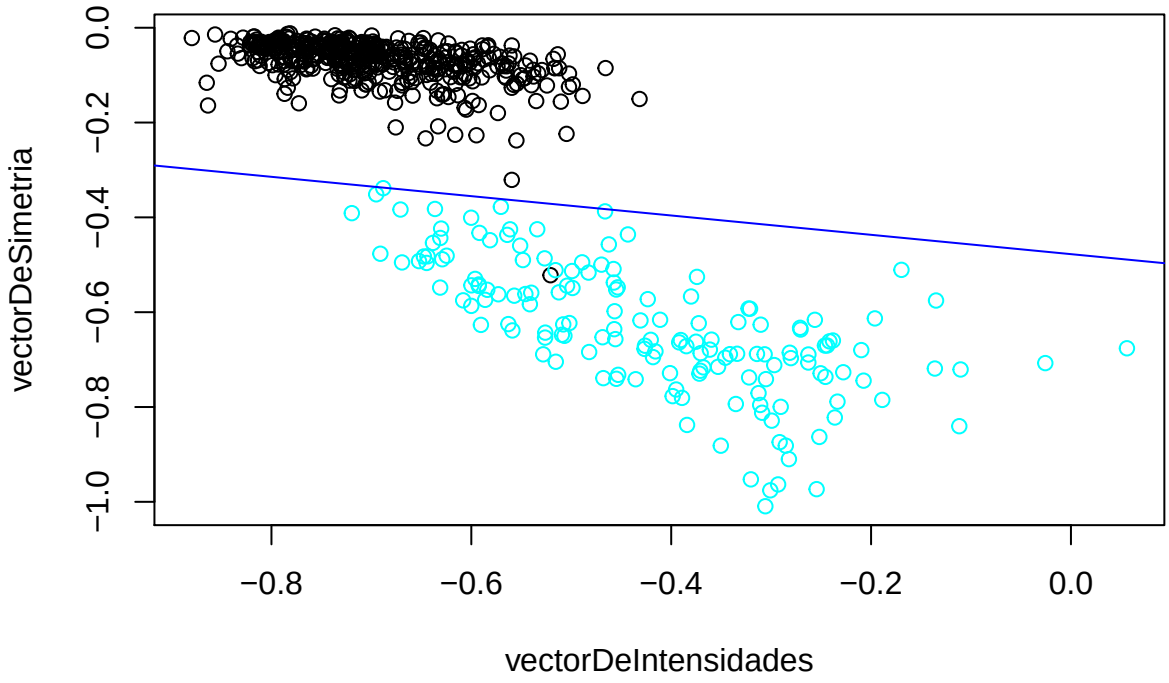
CREAMOS LA FUNCIÓN QUE SE PIDE EN EL EJERCICIO

```r
#devuelve el vector de pesos
Regress_Lin = function(datos, digitos)
{
    #creamos la matriz X con columnas intensidad, simetria, bias(normalización)
```

```
    X = cbind(datos,1)
    #cambiamos el valor de las etiquetas en la función, es decir, cambiamos el 5 por -1
    #para ello usamos "digitos" y simplemente cambiamos los 5 por -1
    Y = digitos
    Y[Y==5] = -1
    #calculamos las 3 matrices con SVD (2 matrices ortogonales y una diagonal):
    listaDeMatricesSVD = svd(t(X)%*%X)
    #realizamos la inversa de (t(X)*X)^(-1) == V*(D^t)*(t(V)), por tanto pasamos a calcular la pseudoIn
    #la pseudo inversa de D se calcula como diag(1/D) (la diagonal de 1/D)
    pseudoInversaD = diag(1/listaDeMatricesSVD$d)
    #para obtener al fin (t(X)*X)^(-1), realizamos lo dicho antes ->  V*(D^t)*(t(V))
    inversaXporXTrasp = listaDeMatricesSVD$v %*% pseudoInversaD %*% t(listaDeMatricesSVD$v)
    #multiplicamos ese resultado por la traspuesta de X para obtener al fin la pseudoTraspuesta de X
    pseudoTraspuestaX = inversaXporXTrasp %*% t(X)
    #obtenemos el vector de pesos w multiplicando la pseudoTraspuesta de X por el Vector de etiquetas Y
    vectorPesos = pseudoTraspuestaX %*% Y
    vectorPesos
}
```

CALCULAMOS LOS VECTORES DE SIMETRIA E INTENSIDAD COMO EN EL APARTADO ANTERIOR

```
#para que se pinte una sola gráfica
par(mfrow=c(1,1))
#creamos un vector de simetria
#apply: le decimos que coja todas las imagenes que contiene "grises", que lo haga con el eje x(el 1, ej
# y que le aplique a todos la función fsimetria
vectorDeSimetria = apply(grises,1,fsimetria)
#creamos un vector de medias(intensidades)
vectorDeIntensidades = apply(grises,1,mean)
#los representamos
plot(vectorDeIntensidades, vectorDeSimetria, col = digitos)

#compactamos en un vector el vector de simetrias y el de intensidades
vectoresDatos = cbind(vectorDeIntensidades,vectorDeSimetria)
#llamamos a regress_lin
vectorPesos = Regress_Lin(vectoresDatos,digitos)


#obtenemos la pendiente y el punto de corte
coeficientes = calculaPuntoCortePendiente(vectorPesos)
#pintamos la función correspondiente
abline(coeficientes[2],coeficientes[1],col=4)
```



PROCEDEMOS A LA VALORACIÓN DE LA BONDAD DEL RESULTADO CON Ein y Eout

función para calcular el error dentro y fuera de la muestra en forma de porcentaje

```
calcularError = function(datos, label, vini)
{
    cantidadMalClasificados = 0
    j=0

    #iteramos sobre todos los puntos de forma aleatoria
    #(por ello lo del sample)
    for (j in 1:nrow(datos))
    {
        #en caso de que no esté bien clasificado
        #(la función sign devuelve 1 o -1)
        producto = datos[j,]%*%vini#crossprod(datos[j,],vini)
        if (sign(producto) != sign(label[j]))
        {
            cantidadMalClasificados = cantidadMalClasificados + 1
        }
    }

    porcentajeError = (100*cantidadMalClasificados) / nrow(datos)
```

```
    porcentajeError

}
```

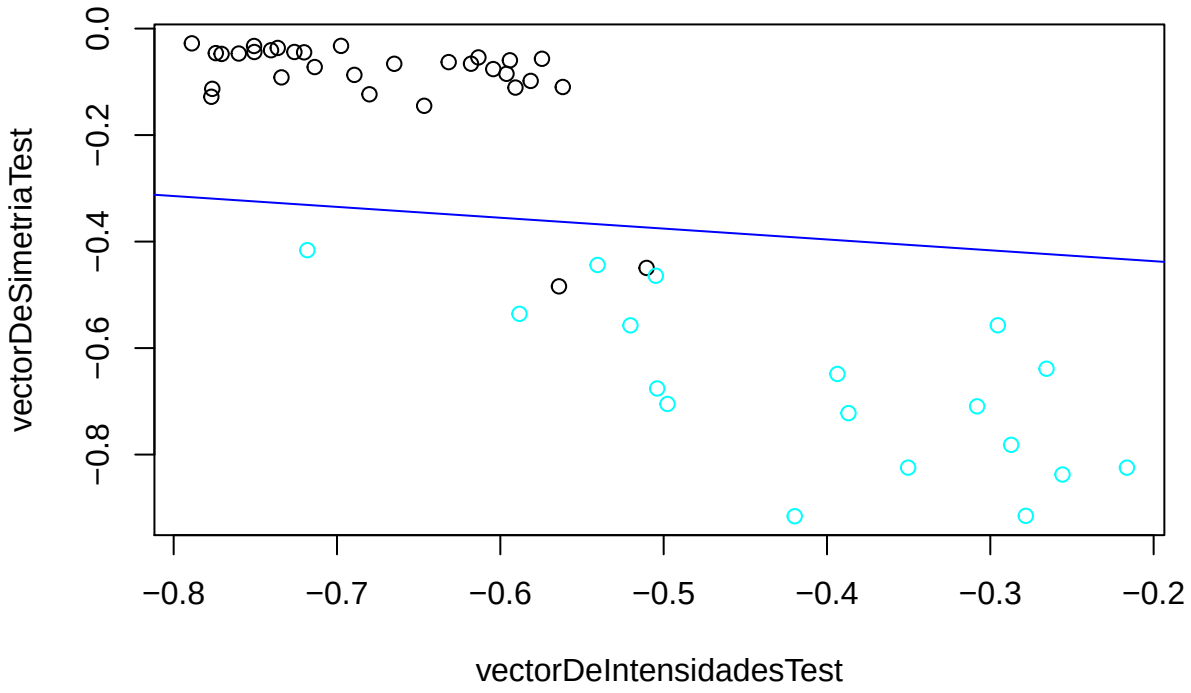
Primero obtenemos el valor de Ein a partir de los datos de entrenamiento (train):

```
#para ello primero calculamos el vector de características X
X = cbind(vectorDeIntensidades, vectorDeSimetria, 1)
#calculamos el vector Y de etiquetas sustituyendo los 5 por -1
Y = digitos
Y[Y==5] = -1
#realizamos el calculo de Ein
Ein = calcularError(X,Y,vectorPesos)
#Ya tenemos su valor:
Ein
```

```
## [1] 0.1669449
```

Ahora leemos los datos del test para obtener Eout:

```
digit.test <- read.table("datos/zip.test",
                         quote="\"", comment.char="", stringsAsFactors=FALSE)

digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,] #Nos da las imagenes que son 1 o 5 pe
digitos = digitos15.test[,1]  # etiquetas que tienen cada imagen. Nos dice si es un 5 o es un 1
ndigitos = nrow(digitos15.test)

# se retira la clase y se monta una matriz 3D: 599*16*16
grises = array(unlist(subset(digitos15.test,select=-V1)),c(ndigitos,16,16))
rm(digit.test)
rm(digitos15.test)

# Para visualizar los 4 primeros
## ------------------------------------------------------------------------
par(mfrow=c(1,1))

#creamos un vector de simetria pero de los datos del test esta vez
#apply: le decimos que coja todas las imagenes que contiene "grises", que lo haga
#con el eje x(el 1, eje que me coge cada imagen completa)
# y que le aplique a todos la función fsimetria
vectorDeSimetriaTest = apply(grises,1,fsimetria)
#creamos un vector de medias(intensidades)
vectorDeIntensidadesTest = apply(grises,1,mean)
#los representamos
plot(vectorDeIntensidadesTest, vectorDeSimetriaTest, col = digitos)

#pintamos la función correspondiente con los pesos obtenidos en el train
abline(coeficientes[2],coeficientes[1],col=4)
```




```
##Obtenemos el valor de Eout usando el vector de pesos del train también:
#calculamos el vector X
X = cbind(vectorDeIntensidadesTest, vectorDeSimetriaTest, 1)
#calculamos el vector de etiquetas Y
Y = digitos
Y[Y==5] = -1
#calculamos el error fuera de la muestra
Eout = calcularError(X,Y,vectorPesos)
#Ya tenemos el valor de Eout, procedemos a comparar los valores
#Eout y Ein (salida en tanto por ciento):
```

```
Eout
```

```
## [1] 4.081633
```

```
Ein
```

```
## [1] 0.1669449
```

Como podemos observar Ein es bastante próximo a Eout y, además este último tiene un valor de tan solo un 4.08, lo que indica que el error es muy pequeño y además que el error Ein es representativo del Eout.

EJERCICIO 4

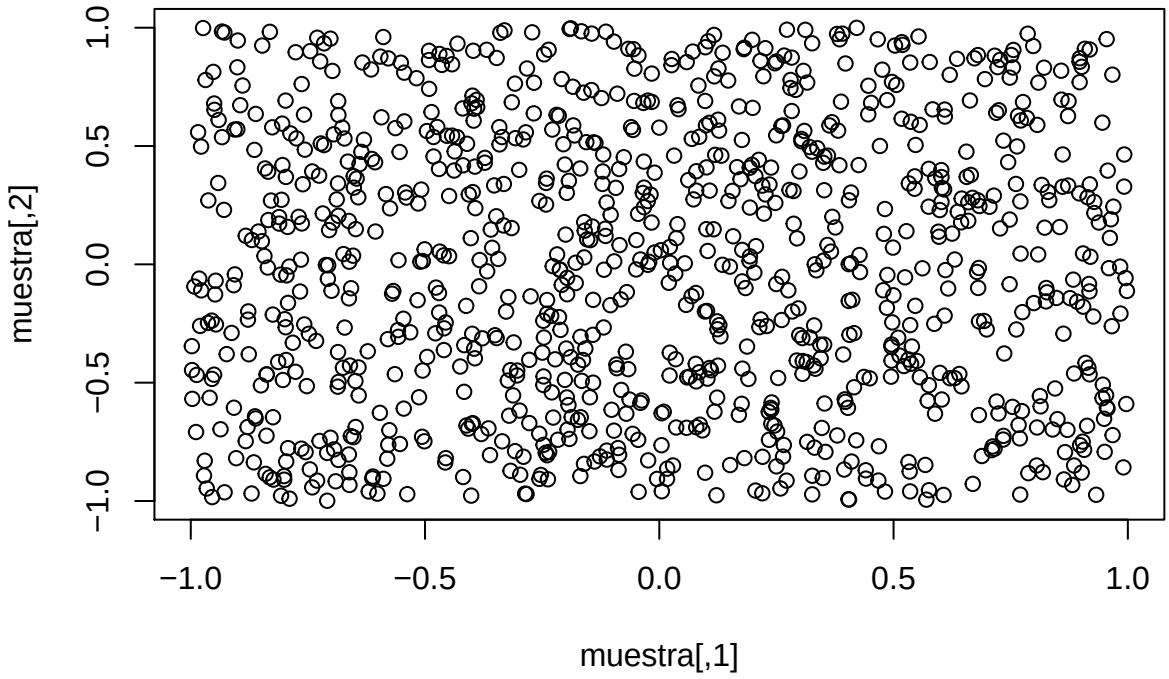
En este apartado exploramos como se transforman los errores Ein y Eout cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función simula_unif(N,2,size) que nos devuelve N

coordenadas 2D de puntos uniformemente muestreados dentro del cuadrado de???nido por [-size,size]×[-size,size]

EXPERIMENTO 1

SUBAPARTADO A

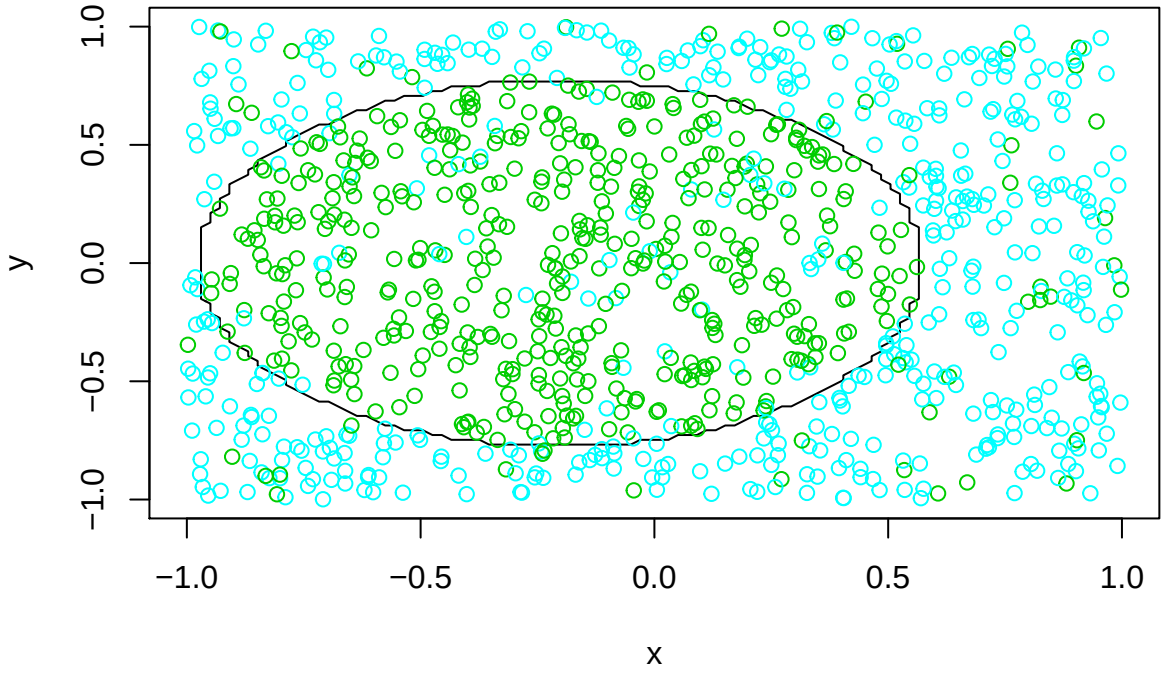
```
#generamos la muestra de entrenamiento indicada
set.seed(12)
rango = c(-1,1)
muestra = simula_unif(1000,2,rango)
plot(muestra)
```



SUBAPARTADO B

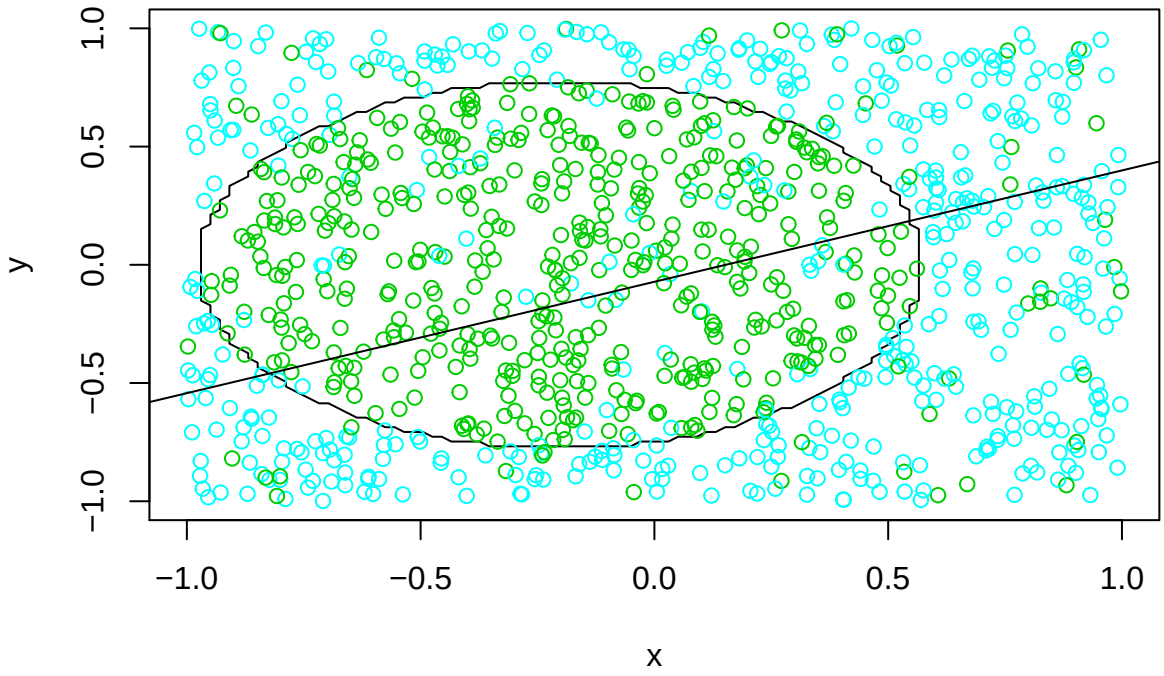
```
#función dada en el ejercicio
f5 = function(x,y)
{
    valor = sign((x+0.2)^2 + y^2-0.6)
    valor
}
#generamos las etiquetas con ruido y pintamos la función con los puntos
etiquetas = generar_etiquetasParaFunciones(muestra, f5)
```

```
etiquetasRuido = asignarRuido(etiquetas,10)
pintar_frontera(f5,rango)
points(muestra, col = etiquetasRuido+4)
```



SUBAPARTADO C

```
#pintamos lo del apartado anterior
pintar_frontera(f5,rango)
points(muestra, col = etiquetasRuido+4)

#calculamos la recta de regresión y la pintamos
vectorPesos = Regress_Lin(muestra, etiquetasRuido)
abline(vectorPesos[2],vectorPesos[1])
```




```
#realizamos el calculo de Ein
muestra = cbind(muestra,1)
Ein = calcularError(muestra,etiquetasRuido,vectorPesos)
```

Error dentro de la muestra

```
Ein
```

```
## [1] 37.1
```

SUBAPARTADO D

```
cantidadEin = 0
cantidadEout = 0
#Preestablecemos el rango según se dice en el ejercicio
rango = c(-1,1)
for(i in 1:1000)
{
    #generamos la muestra de entrenamiento indicada
    muestraIn = simula_unif(1000,2,rango)

    #generamos la muestra de TEST indicada
    muestraOut = simula_unif(1000,2,rango)

    #generamos etiqueas con ruido en la muestra train
    etiquetasIn = generar_etiquetasParaFunciones(muestraIn, f5)
```

```
    etiquetasRuidoIn = asignarRuido(etiquetasIn,10)

    #generamos etiquetas con ruido en el test
    etiquetasOut = generar_etiquetasParaFunciones(muestraOut, f5)
    etiquetasRuidoOut = asignarRuido(etiquetasOut,10)

    #aplicamos regresión lineal con la muestra train
    vectorPesosIn = Regress_Lin(muestraIn, etiquetasRuidoIn)
    muestraIn = cbind(muestraIn,1)
    #calculamos error dentro de la muestra
    Ein = calcularError(muestraIn,etiquetasRuidoIn,vectorPesosIn)
    muestraOut = cbind(muestraOut,1)
    #calculamos error fuera de la muestra
    Eout = calcularError(muestraOut, etiquetasRuidoOut, vectorPesosIn)

    #actualizamos valores
    cantidadEin = cantidadEin + Ein
    cantidadEout = cantidadEout + Eout
}

#calculamos porcentajes
cantidadEin = cantidadEin/1000
cantidadEout =  cantidadEout/1000
```

Mostramos por pantalla las medias en porcentaje de los errores interno y externo la muestra (respectivamente).

```
cantidadEin
```

```
## [1] 39.8424
```

```
cantidadEout
```

```
## [1] 40.058
```

SUBAPARTADO E

Como podemos observar el error es prácticamente el máximo (muy próximo al 50%), y esto es debido a que estamos aplicando regresión lineal en un problema que es imposible de separar mediante una función lineal, debido a ello sale un porcentaje de error tan alto. También podemos afirmar que el error dentro de la muestra es muy próximo al error fuera de esta, por lo que es representativo.

EXPERIMENTO 2

SUBAPARTADO A

APARTADO A

```
  #generamos la muestra de entrenamiento indicada
  set.seed(14)
    rango = c(-1,1)
    muestraIn = simula_unif(1000,2,rango)
    #creamos el vector de características indicado en el ejercicio
    muestraIn2 = cbind(muestraIn, muestraIn[,1]*muestraIn[,2], muestraIn[,1]^2, muestraIn[,2]^2)

    #generamos etiquetas con ruido
```

```
    etiquetasIn = generar_etiquetasParaFunciones(muestraIn2, f5)
    etiquetasRuidoIn = asignarRuido(etiquetasIn,10)

    #calculamos el vector de pesos
    vectorPesosIn = Regress_Lin(muestraIn2, etiquetasRuidoIn)
    muestraIn2 = cbind(muestraIn2,1)
    #calculamos el error interno a la muestra
    Ein = calcularError(muestraIn2,etiquetasRuidoIn,vectorPesosIn)
    Ein
```

```
## [1] 14.2
```

APARTADO B

```
cantidadEin = 0
cantidadEout = 0
for(i in 1:1000)
{
    #generamos la muestra de TRAIN
    rango = c(-1,1)
    muestraIn = simula_unif(1000,2,rango)
    muestraIn2 = cbind(muestraIn, muestraIn[,1]*muestraIn[,2], muestraIn[,1]^2, muestraIn[,2]^2)

    #generamos la muestra de TEST
    muestraOut = simula_unif(1000,2,rango)
    muestraOut2 = cbind(muestraOut, muestraOut[,1]*muestraOut[,2], muestraOut[,1]^2, muestraOut[,2]^2)

    #generamos las etiquetas para muestraIn2
    etiquetasIn = generar_etiquetasParaFunciones(muestraIn2, f5)
    etiquetasRuidoIn = asignarRuido(etiquetasIn,10)

    #generamos las etiquetas para muestraOut2
    etiquetasOut = generar_etiquetasParaFunciones(muestraOut2, f5)
    etiquetasRuidoOut = asignarRuido(etiquetasOut,10)

    #calculamos el vector de pesos para l amatriz i
    vectorPesosIn = Regress_Lin(muestraIn2, etiquetasRuidoIn)
    #incluimos la columna de 1
    muestraIn2 = cbind(muestraIn2,1)
    #calculamos el error dentro de la muestra
    Ein = calcularError(muestraIn2,etiquetasRuidoIn,vectorPesosIn)
    #incluimos la columna de 1
    muestraOut2 = cbind(muestraOut2,1)
    #calculamos el error fuera de la muestra
    Eout = calcularError(muestraOut2, etiquetasRuidoOut, vectorPesosIn)
    #actualizamos los valores de cantidades Ein y Eout
    cantidadEin = cantidadEin + Ein
    cantidadEout = cantidadEout + Eout
}

cantidadEin = cantidadEin/1000
cantidadEout =  cantidadEout/1000
```

```
cantidadEin
```

```
## [1] 14.2518
```

```
cantidadEout
```

```
## [1] 14.4891
```

APARTADO C

En este experimento podemos observar una gran mejora en la clasificación, esto es debido al uso de una función más compleja que las lineales, lo que hace que realice una clasificación con un nivel de optimidad muy superior al experimento 1, concretamente el error ha disminudo del 40% al 14%, una gran mejora.

RESPUESTA AL SEGUNDO PUNTO DEL EXPERIMENTO 2:

Es evidente, como ya he dicho, que para este tipo de problemas el ajuste de regresión lineal es mucho menos productivo que el ajuse con funciones de mayor complejidad como hemos demostrado en el experimento 2, disminuyendo el porcentaje de error dentro y fuera de la muestra en casi un 25% (porcentaje que seria mayor incluso si no hubiese ruido).