Práctica 1 - Aprendizaje Automático

Néstor Rodriguez Vico

27 de marzo de 2017

Funciones de utilidad

```
set.seed(3) # semilla para los aleatorios

# por defecto genera 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 generación se usa sd, y no la varianza
  if(dim != length(sigma)) stop ("El numero de varianzas es distinto de la dimensión")
  simula_gauss1 = function() rnorm(dim, sd = sigma) # genera 1 muestra, con las
                                                    # desviaciones especificadas
  m = t(replicate(N,simula_gauss1())) # repite N veces, simula_gauss1 y se hace la
                                      # traspuesta
  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
}
```

Ejercicio sobre la complejidad de H y el ruido

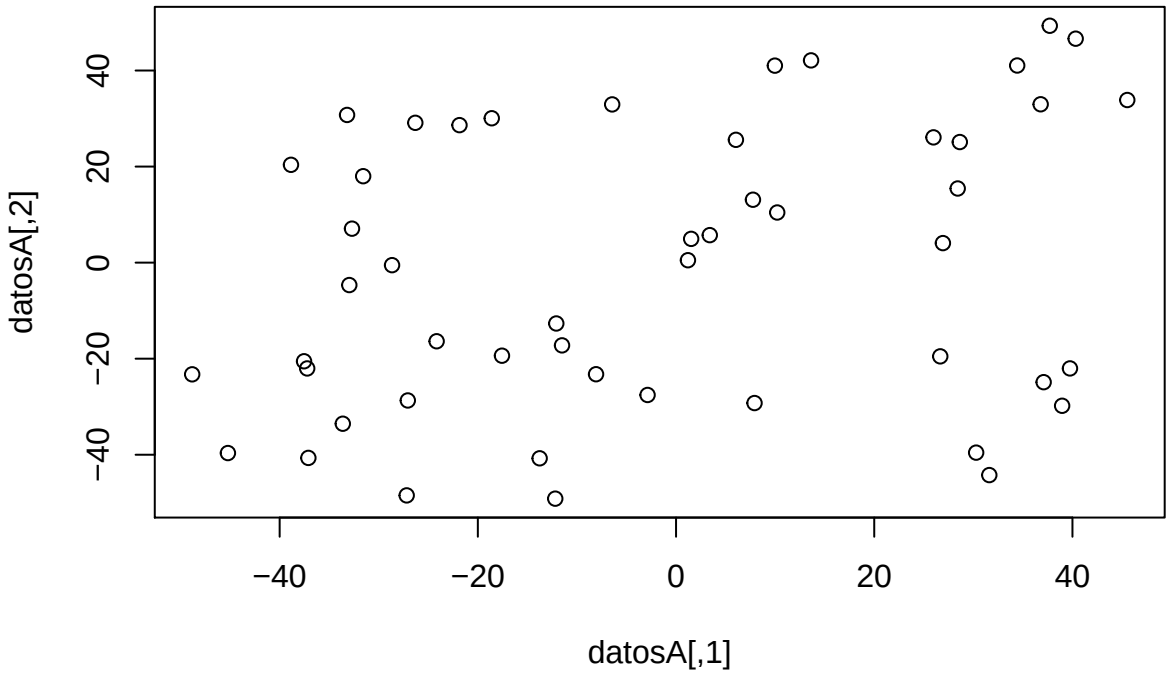
Apartado 1

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

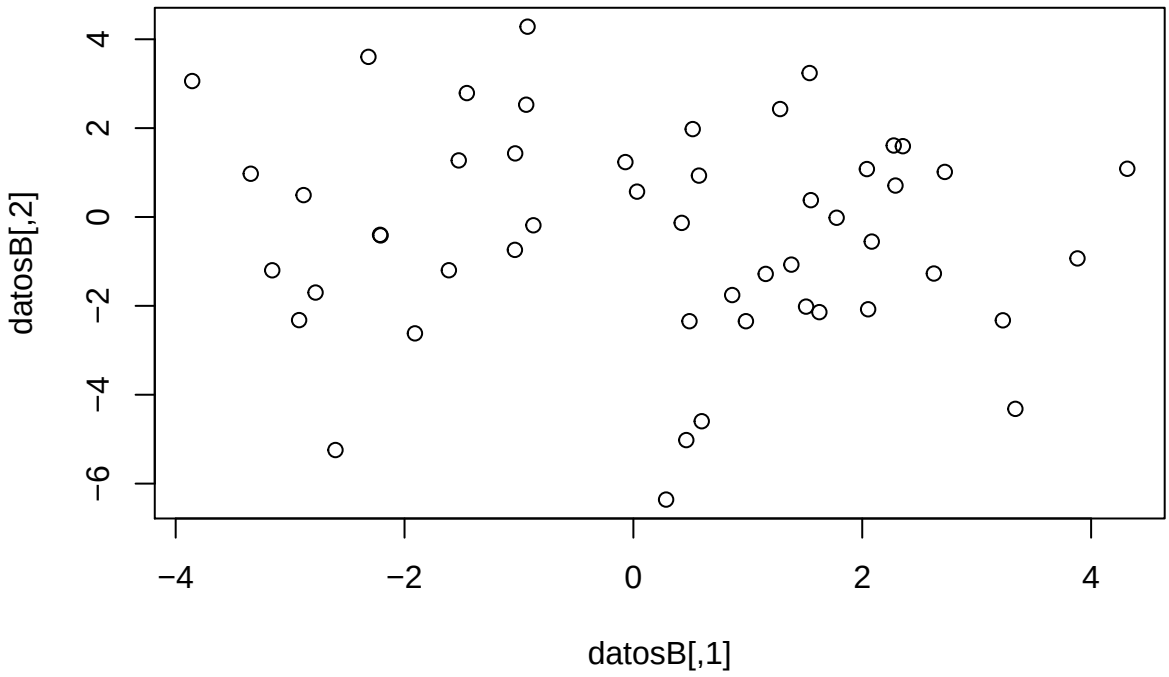
a) Considere N = 50, dim = 2, rango = [-50, +50] con simula_unif (N, dim, rango).

b) Considere N = 50, dim = 2 y sigma = [5, 7] con simula_gaus(N, dim, sigma).

```
datosA = simula_unif(50, 2, c(-50,50))
datosB = simula_gaus(50, 2, c(5,7))
plot(datosA)
```




```
plot(datosB)
```



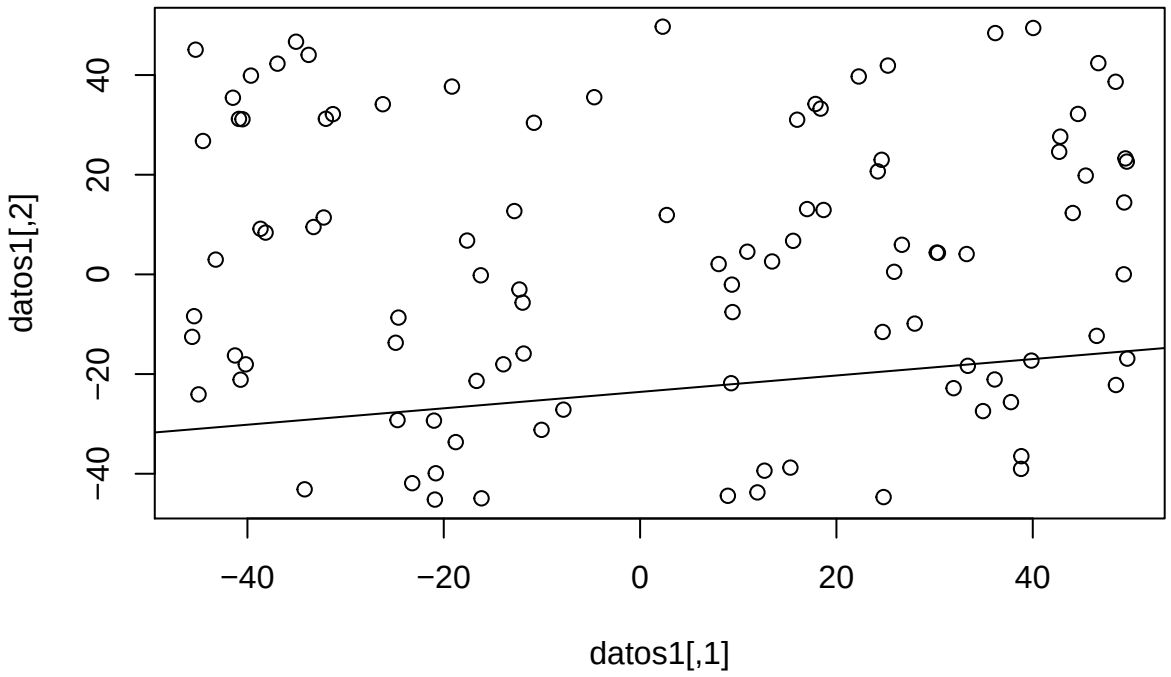
Apartado 2

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().

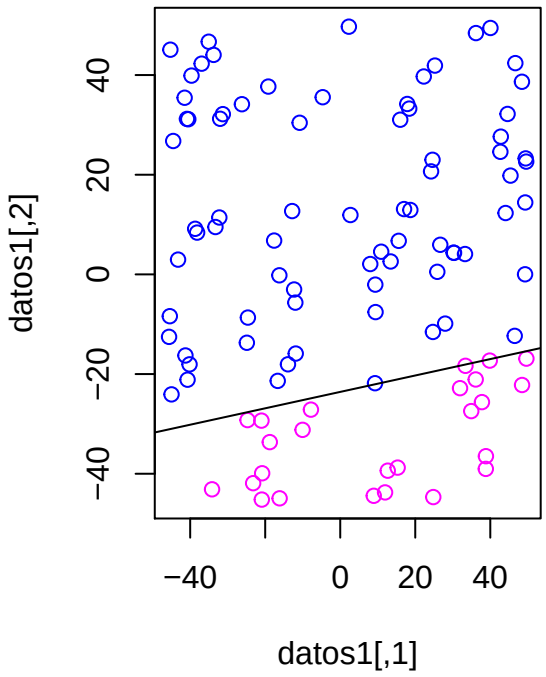
a) Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto de la recta)

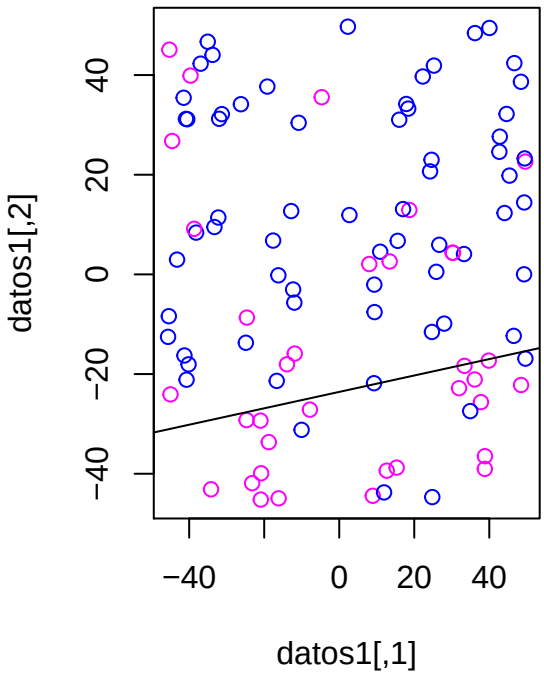
b) Modifique de forma aleatoria un 10 % etiquetas positivas y otro 10 % de negativas. Dibuje de nuevo la gráfica anterior. (Ahora hay puntos mal clasificados respecto de la recta)

```
# Función auxiliar para generar ruido en etiquetas
asignarRuido=function(etiquetas, porcentaje=10){
    positivas = which(etiquetas==1)
    negativas = which(etiquetas==-1)
    positivas = sample(positivas, length(positivas)*0.02*porcentaje, replace=FALSE)
    negativas = sample(negativas, length(negativas)*0.02*porcentaje, replace=FALSE)
    etiquetas[positivas]=-1
    etiquetas[negativas]=1
    etiquetas
}

# Generamos 100 datos de dimensión 2 en el rango -50,50
datos1 = simula_unif(100, 2, c(-50,50))
# Generamos la recta que va a determinar como clasificamos los datos
recta = simula_recta(c(-50,50))
# Vamos a ver los datos que hemos generado junto con la recta:
plot(datos1)
abline(recta[2], recta[1])
```




```
# Para realizar la clasificación voy a usar la siguiente idea:
# Para cada punto calculo el valor y de la recta en la coordenada x del punto y realizo
# la diferencia entre el valor y del punto y el valor y de la recta. Si la diferencia
# es positiva (valor del ver con la función sign()) el punto está por encima de la recta
# y si es negativa, el punto está por debajo de la recta.
# Función que calcula el valor de y para un punto x seguún la recta y = pendiente*x+valor
valor_y = function(x, pendiente, valor){
  y = pendiente*x+valor
  y
}
# Aplicamos la idea comentada anteriormente para generar las etiquetas
etiquetas = sign(valor_y(datos1[,1], recta[1], recta[2])-datos1[,2])
# Pintamos los datos junto con las etiquetas (Ojo, las etiquetas son -1 y 1 pero los
# colores no pueden ser negativos, por eso le sumamos 5)
plot(datos1, col=etiquetas+5)
# Pintamos la recta para comprobar que están bien clasificados
abline(recta[2], recta[1])
# Generamos unas nuevas etiquetas con ruido
etiquetas.R = asignarRuido(etiquetas, 10)
# Pintar dos gráficos
par(mfrow=c(1,2))
# Primero pintamos las que no tienen ruido
plot(datos1, col=etiquetas+5)
abline(recta[2], recta[1])
# Y a continuación las que tienen ruido, para comparar correctamente
plot(datos1, col=etiquetas.R+5)
abline(recta[2], recta[1])
```






```
# Pintar sólo un gráfico
par(mfrow=c(1,1))
```

Apartado 3

Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta

f(x, y) = (x-10)^2 + (y-20)^2 - 400

f(x, y) = 0,5(x+10)^2 + (y-20)^2 - 400

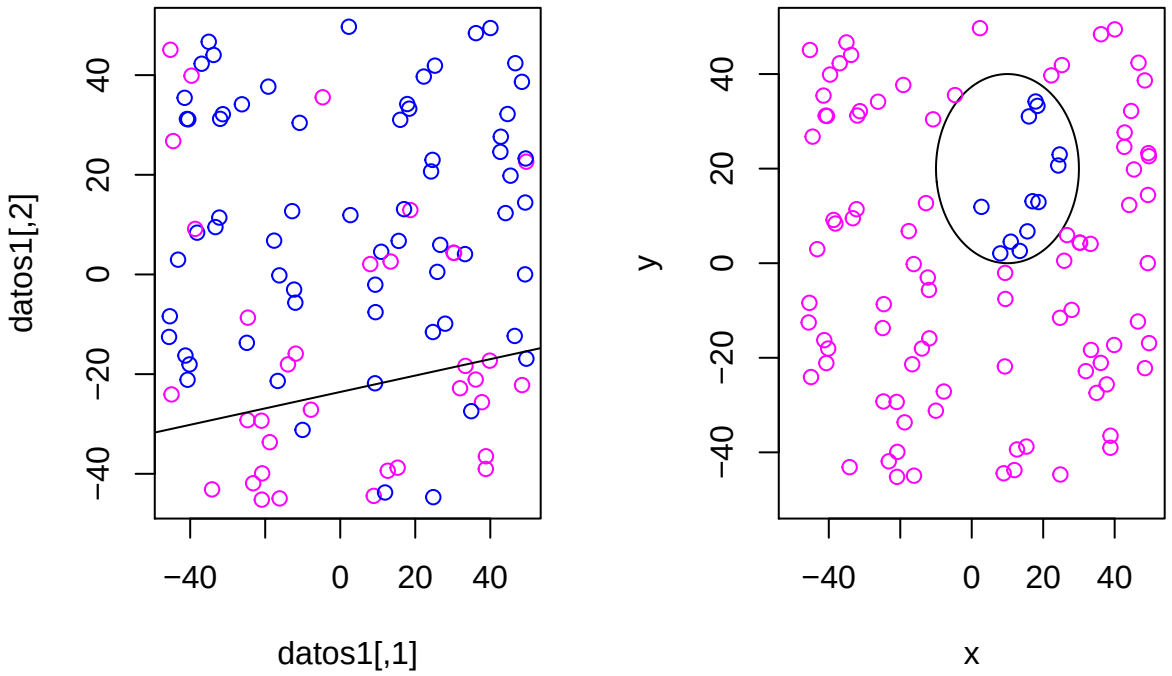
f(x, y) = 0,5(x-10)^2 - (y+20)^2 - 400

f(x, y) = y - 20x^2 - 5x + 3

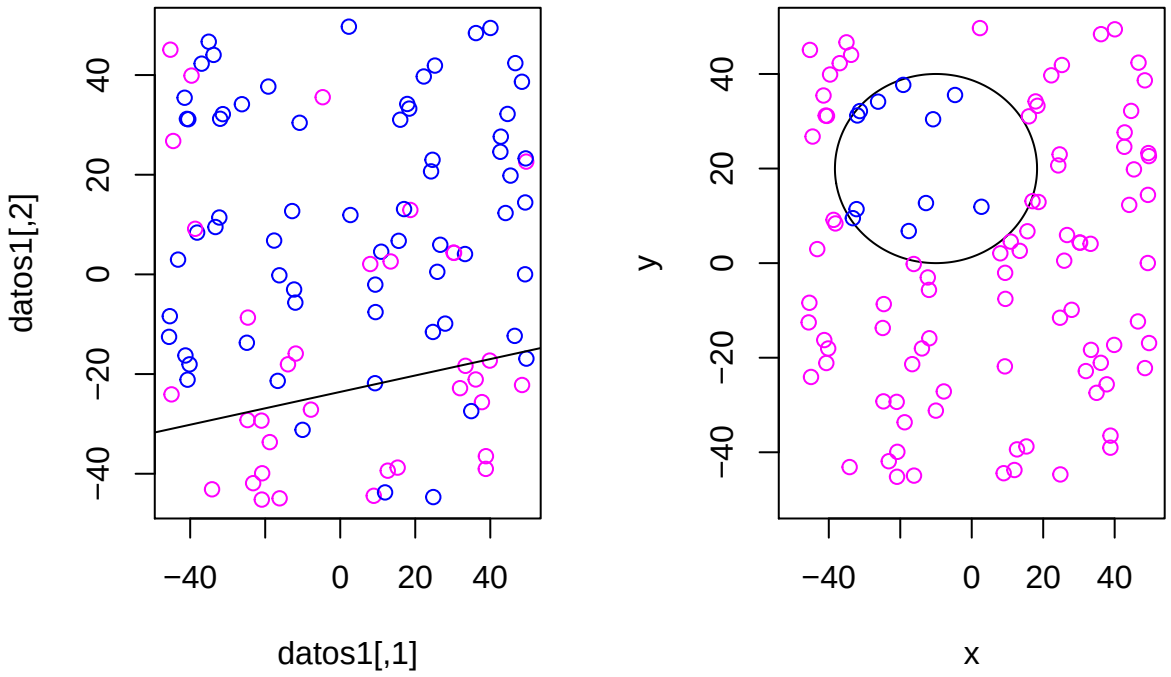
Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Hemos ganado algo en mejora de clasificación al usar funciones más complejas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

```
# 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 = 200)
   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")
}

# Ejemplo de llamada a una funcion f1_xy que hemos de definir
# pintar_frontera(f1_xy)
```

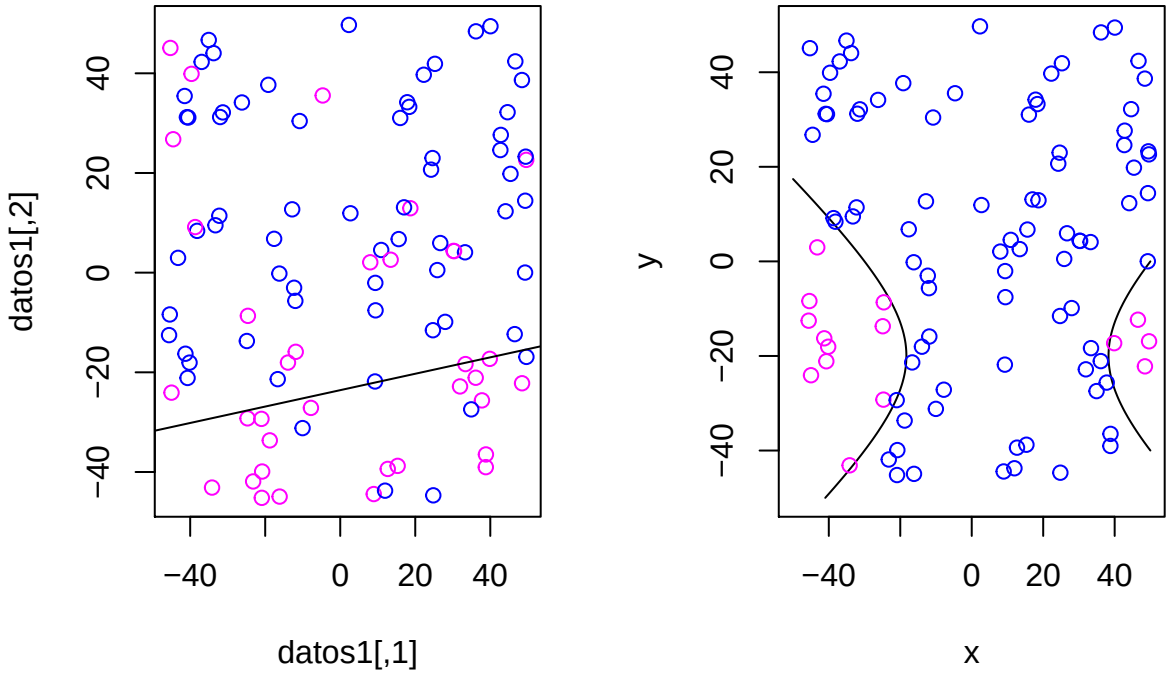
```r
# Funciones dadas por el ejercicio
funcion1 = function(x, y) (x-10)^2 + (y-20)^2 - 400
funcion2 = function(x, y) (0.5*(x+10)^2 + (y-20)^2 - 400)
funcion3 = function(x, y) (0.5*(x-10)^2 - (y+20)^2 - 400)
funcion4 = function(x, y) (y-20*x^2 - 5*x+3)

e1 = sign(funcion1(datos1[,1], datos1[,2]))
e2 = sign(funcion2(datos1[,1], datos1[,2]))
e3 = sign(funcion3(datos1[,1], datos1[,2]))
e4 = sign(funcion4(datos1[,1], datos1[,2]))

# Pintamos con ruido
par(mfrow=c(1,2))

plot(datos1, col=etiquetas.R+5)
abline(recta[2], recta[1])
pintar_frontera(funcion1); points(x=datos1[,1], y=datos1[,2], col=e1+5)
```



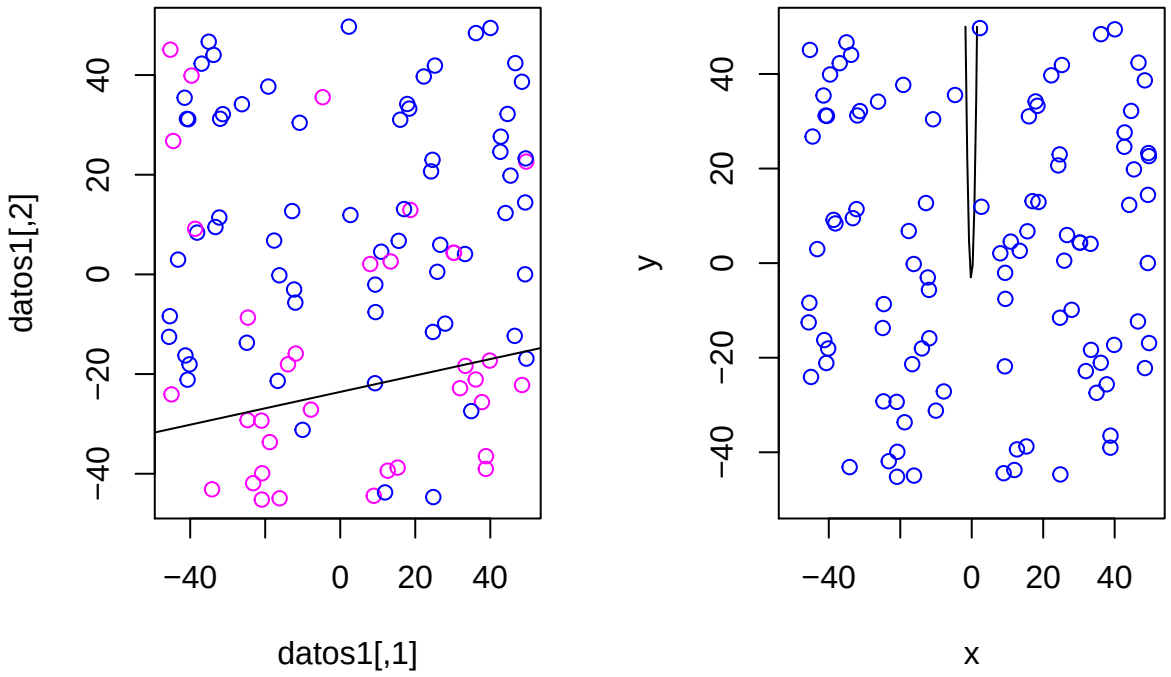

```r
plot(datos1, col=etiquetas.R+5)
abline(recta[2], recta[1])
pintar_frontera(funcion2); points(x=datos1[,1], y=datos1[,2], col=e2+5)
```




```
plot(datos1, col=etiquetas.R+5)
abline(recta[2], recta[1])
pintar_frontera(funcion3); points(x=datos1[,1], y=datos1[,2], col=e3+5)
```




```
plot(datos1, col=etiquetas.R+5)
abline(recta[2], recta[1])
pintar_frontera(funcion4); points(x=datos1[,1], y=datos1[,2], col=e4+5)
```



Podemos ver que en el caso de las gráficas de la izquierda no podemos encontrar ninguna recta que separe de manera lineal las dos clases. Lo mismo pasa en las funciones cuadráticas a excepción de la última, en la que hay infinitas rectas que pueden separar en dos clases. Este conjunto de rectas son las que dejan a los puntos por encima de la misma. En este caso se obtendría un cien por cien de acierto en la clasificación.

Ejercicio sobre el Algoritmo Perceptron

Apartado 1

Implementar la función ajusta_PLA(datos, label, max_iter, vini) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada datos 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 coeficientes del hiperplano.

```
ajusta_PLA = function(datos, label, max_iter, vini){
  seguirIterando = TRUE
  w = vini
  iter = 0
  # Debemos añadir una columna de unos para que no de problemas al realizar el producto
  # vectorial (con el debuger (browser) se puede ver el error, ya que datos[i,]
  # y w no tienen el mismo tamaño)
  datos = cbind(1,datos)
  # Recorremos todos los datos mientras se produzca algun cambio
  # en algun peso y no hayamos dado el número máximo de iteraciones
  while (seguirIterando && iter < max_iter){
    seguirIterando = FALSE
    for (i in sample(1:nrow(datos))){
      if (sign(crossprod(w, datos[i,])) != label[i]){
        # Si entramos aquí, se ha producido un cambio en algún peso, así que
        # debemos dar una iteración más
```

```
        seguirIterando = TRUE
        w = w + label[i] * datos[i,]
      }
    }
    iter = iter + 1
  }
  # Devolvemos el vector de pesos, el número de iteraciones y los coeficientes
  # del hiperplano
  resultado = list(w, iter, c(-w[1]/w[3], -w[2]/w[3]))
  resultado
}
```

Apartado 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.

```
ejecutarPLA = function(i) {solucion = ajusta_PLA(datos1, etiquetas, 1000, c(0,0,0))
                           solucion[2]}
ejecutarPLA_aleatorio = function(i) {pesos = c(simula_unif(3, 1, c(0,1)))
                           solucion = ajusta_PLA(datos1, etiquetas, 1000, pesos)
                           solucion[2]}
resultados = mapply(ejecutarPLA, 1:10)
mean(unlist(resultados))
```

```
## [1] 225.1
```

```
resultados = mapply(ejecutarPLA_aleatorio, 1:10)
mean(unlist(resultados))
```

```
## [1] 110.7
```

Podemos ver que cuando el vector de pesos se inicializa con valores aleatorios tarda menos iteraciones en converger.

Apartado 3

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

```
ajusta_PLA(datos1, etiquetas.R, 500, c(0,0,0))
```

```
## [[1]]
## [1] -350.00000   18.57857  -54.59565
##
## [[2]]
## [1] 500
##
## [[3]]
## [1] -6.4107670  0.3402939
```

```
ejecutarPLA = function(i) {pesos = c(simula_unif(3, 1, c(0,1)))
                           solucion = ajusta_PLA(datos1, etiquetas.R, 500, pesos)
```

```
                          solucion[2]}
resultados = mapply(ejecutarPLA, 1:10)
mean(unlist(resultados))
```

```
## [1] 500
```

```
ajusta_PLA(datos1, etiquetas.R, 1500, c(0,0,0))
```

```
## [[1]]
## [1] -394.00000   29.09752  -19.74795
##
## [[2]]
## [1] 1500
##
## [[3]]
## [1] -19.951435   1.473445
```

```
ejecutarPLA = function(i) {pesos = c(simula_unif(3, 1, c(0,1)))
                          solucion = ajusta_PLA(datos1, etiquetas.R, 1500, pesos)
                          solucion[2]}
resultados = mapply(ejecutarPLA, 1:10)
mean(unlist(resultados))
```

```
## [1] 1500
```

Podemos ver que en este caso se completan todas las iteraciones, sean cuales sean, ya que no esxiste una separación lineal que pueda separar los datos. He mostrado lo que sucede para 500 iteraciones y para 1500, pero si siguiesemos subiendo el valor pasaría lo mismo.

Ejercicio sobre el Regresión Lineal

Apartado 1

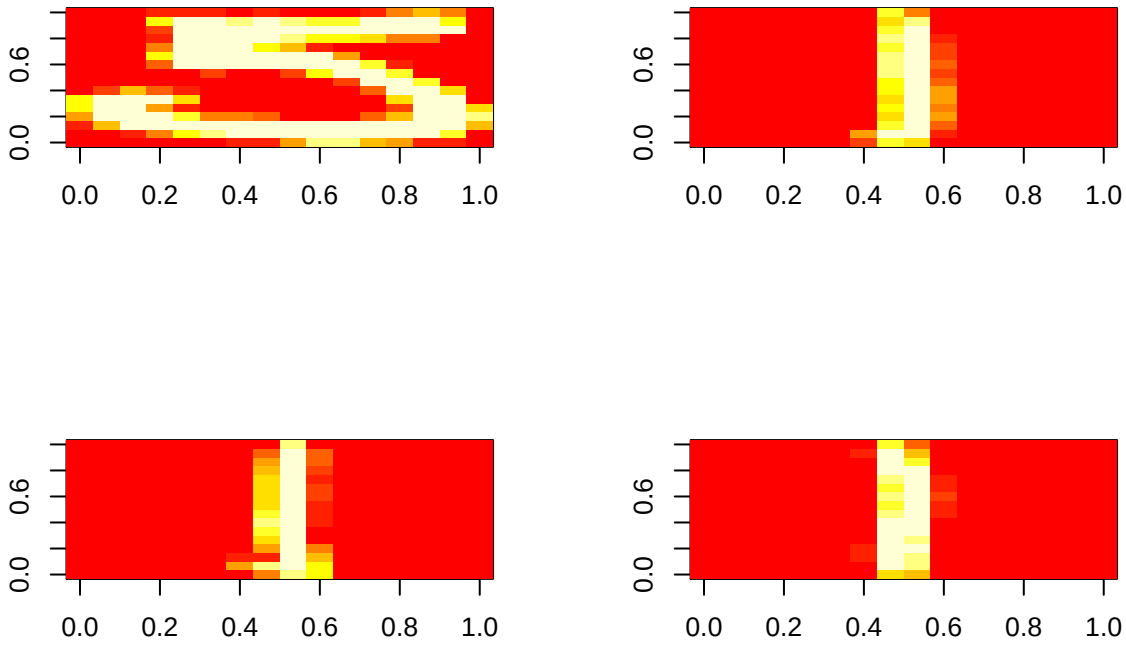
Leemos datos:
a) Abra el fichero Zip.info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el fichero Zip.train. Lea el fichero Zip.train dentro de su código y visualice las imágenes (usando paraTrabajo1.R). Seleccione solo las instancias de los números 1 y 5. Guardelas como matrices de tamaño 16x16.
b) También está disponible el fichero Zip.test que deberemos usar más adelante.

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

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : número de items leídos no es múltiplo del número de columnas
```

```
digitos1_5.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]
etiquetasTrain = digitos1_5.train[,1]
ndigitos = nrow(digitos1_5.train)

# se retira la clase y se monta una matriz 3D: 599*16*16
datos1_5Train = array(unlist(subset(digitos1_5.train,select=-V1)),c(ndigitos,16,16))
rm(digit.train)
rm(digitos1_5.train)
```

```r
# Para visualizar los 4 primeros
## ------------------------------------------------------------------------

par(mfrow=c(2,2))
for(i in 1:4){
  imagen = datos1_5Train[i,,16:1] # se rota para verlo bien
  image(z=imagen)
}
```



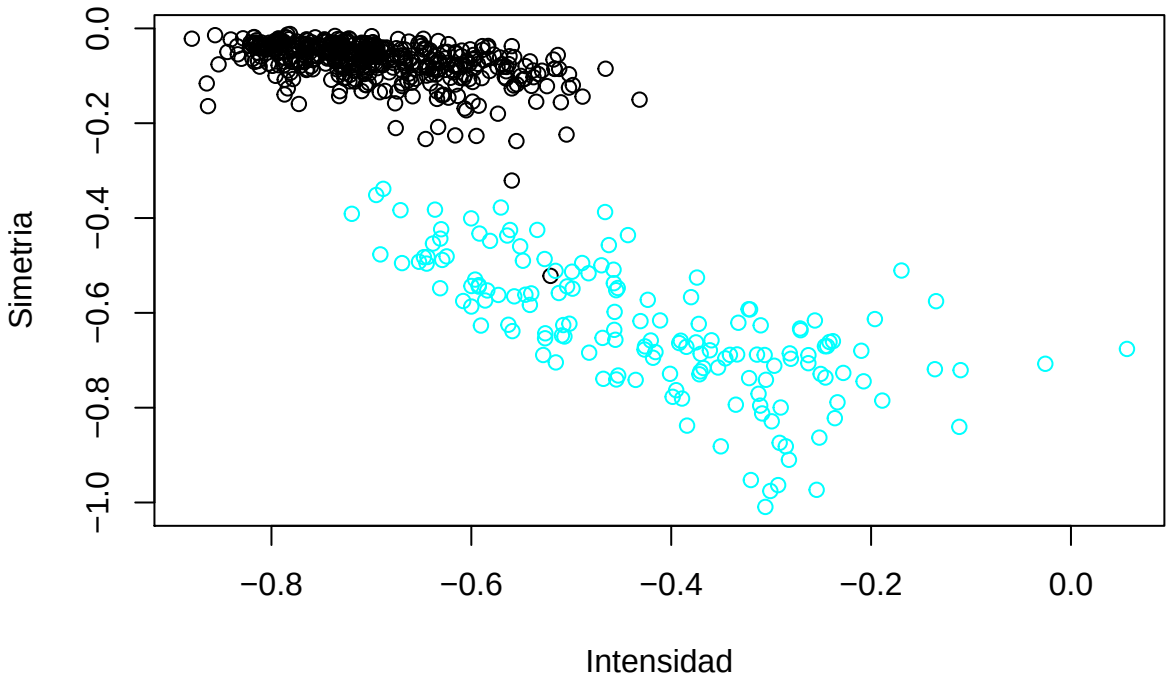
Apartado 2

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.

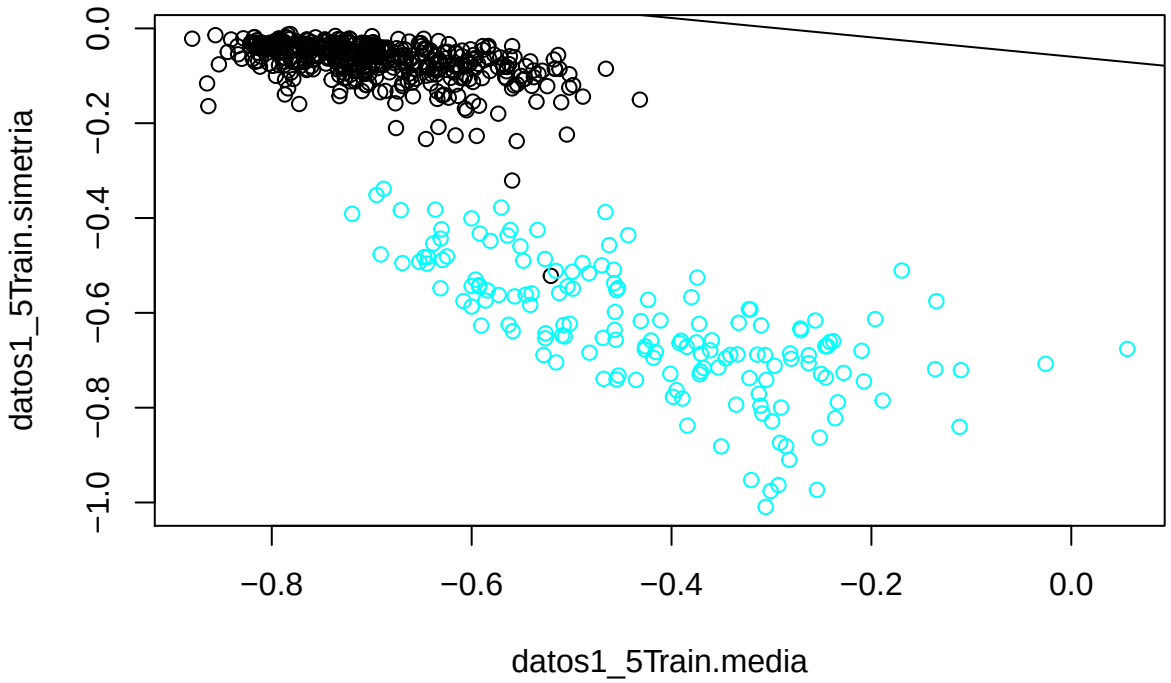
```r
simetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -mean(A)
}

# Aplicamos la función para calcular la simetria y la intensidad a cada matriz
# de la lista de matrices
datos1_5Train.media = apply(datos1_5Train, 1, mean)
datos1_5Train.simetria = apply(datos1_5Train, 1, simetria)
plot(datos1_5Train.media, datos1_5Train.simetria,
     xlab="Intensidad", ylab="Simetria", col=etiquetasTrain)
```



Apartado 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 funcion).

```
# Se necesita el paquete matrixcalc. En caso de no estar instalado, se instalará
Regress_Lin = function(datos, label) {
  # Nos aseguramos de que vamos a trabar con matrices
  datos = as.matrix(datos)
  label = as.matrix(label)

  x = t(datos) %*% datos
  matrices = svd(x)
  auxiliar = matrices$v%*%diag(1/matrices$d)%*%t(matrices$v)
  pseudo = auxiliar%*%t(datos)
  w = pseudo%*%label
  w
  unlist(w)
}

datosTrain = matrix(c(rep(1, length(datos1_5Train.simetria)), datos1_5Train.media,
                      datos1_5Train.simetria), nrow = length(datos1_5Train.simetria))
modelo = Regress_Lin(datosTrain, etiquetasTrain)
plot(datos1_5Train.media, datos1_5Train.simetria, col=etiquetasTrain)
abline(-modelo[1]/modelo[3], -modelo[2]/modelo[3])
```



Como podemos ver, no se separa correctamente, lo cual nos daría que pensar que tenemos el algoritmos de a regrsión mal implementado. Tras pensar un rato y analizar l que estamos haciendo he encontrado el error. Si observamos el contenido de nuestras etiquetas es el siguiente:

```
etiquetasTrain
```

```
##   [1] 5 1 1 1 1 1 1 1 5 1 1 1 1 1 5 1 1 1 1 5 1 5 5 1 1 1 1 1 5 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 5 5 5 1 5 1 1 1 5 1 1 1 1 5 1 5 5 1 1 5 5 1 5 5 1 1 5 1 1
##  [71] 5 5 1 5 1 1 5 5 1 1 1 5 1 1 1 1 1 5 1 1 1 1 1 1 1 1 1 5 1 1 1 5 1 1
## [106] 1 1 1 1 5 1 5 1 1 1 1 1 1 1 5 1 5 1 1 5 1 1 5 1 1 1 5 1 1 1 1 1 1 1
## [141] 1 1 1 1 5 1 5 5 1 5 5 5 1 1 1 1 1 5 5 5 1 1 5 1 1 1 5 5 5 5 5 5 1 5
## [176] 5 1 1 5 5 1 5 1 1 1 1 1 1 1 1 1 1 1 1 1 5 1 5 5 5 1 1 1 5 1 1 1 1 5
## [211] 1 1 1 1 1 1 1 1 1 5 1 5 1 5 1 1 1 1 1 1 1 1 1 1 1 1 1 5 1 1 1 1 1 5
## [246] 1 1 1 1 1 1 5 5 1 5 1 5 1 1 5 1 1 1 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [281] 1 5 1 1 1 1 5 1 1 1 5 1 1 1 5 1 1 5 1 1 5 5 1 1 1 5 1 1 1 1 1 1 1 1
## [316] 5 5 1 1 1 1 1 1 1 1 5 1 1 5 1 1 1 1 1 1 1 1 5 1 1 1 1 1 1 1 5 1 5
## [351] 5 5 1 5 1 5 1 1 5 1 5 1 5 1 1 1 1 1 1 1 1 1 5 1 1 1 1 1 5 1 1 1 1 1
## [386] 1 1 1 1 1 1 1 1 1 1 5 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [421] 1 1 5 1 1 1 5 1 1 1 1 1 5 5 1 1 1 1 1 1 1 1 1 1 1 1 1 5 5 5 5 5 1
## [456] 5 1 1 5 1 5 1 1 5 1 1 1 1 1 5 1 1 1 1 1 1 1 5 1 1 1 1 5 5 1 1 5 1 1 5
## [491] 1 1 1 5 1 5 1 1 1 1 5 1 1 1 1 1 5 5 5 1 5 5 5 1 5 1 5 5 1 5 5 5 1 1 1
## [526] 1 1 1 5 1 5 5 5 1 5 1 1 5 1 1 1 1 5 1 1 1 1 1 1 1 5 1 1 1 1 1 5 1 1 1
## [561] 1 1 5 5 5 5 1 1 1 1 1 5 5 1 1 1 5 1 5 1 1 1 5 1 1 1 1 1 5 5 1 1 1 1 1 1
## [596] 5 1 1 5
```

Como bien podemos observar las etiquetas son 1 y 5. El problema reside en que deben ser -1 y 1, para poder usar el signo (positivo o negativo) así que antes de llamar a la función de regresion debemos cambiar dichas etiquetas. Por intentar conservar algo de la información original voy a cambiar los 5 por -1 y mantener los 1 como 1.

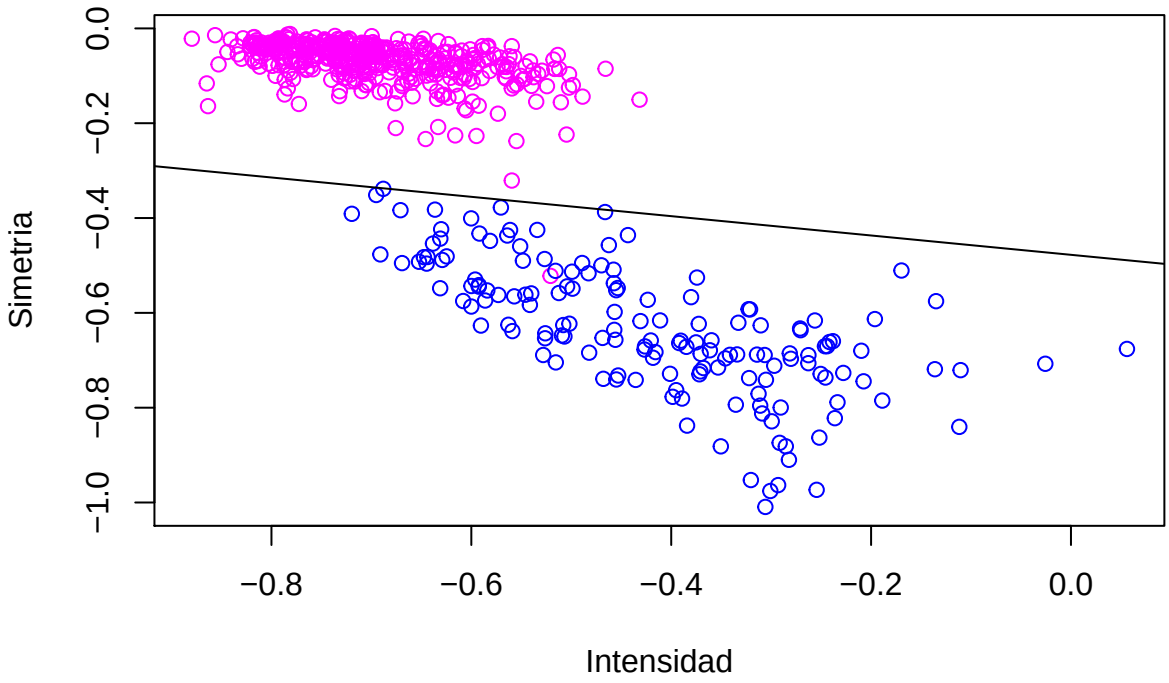
```
etiquetasPositivasNegativas = replace(etiquetasTrain, etiquetasTrain==5, -1)
etiquetasPositivasNegativas
```

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

```
##  [24]  1  1  1  1  1 -1  1  1  1  1  1  1  1  1  1  1  1 -1 -1 -1  1 -1
##  [47]  1  1  1 -1  1  1  1  1 -1  1 -1 -1  1  1 -1 -1  1 -1 -1  1  1 -1  1
##  [70]  1 -1 -1  1 -1  1  1 -1 -1  1  1  1 -1  1  1  1  1  1 -1  1  1  1  1
##  [93]  1  1  1  1  1  1 -1  1  1  1 -1  1  1  1  1  1  1 -1  1 -1  1  1  1
## [116]  1  1  1  1 -1  1 -1  1  1 -1  1  1 -1  1  1  1 -1  1  1  1  1  1  1
## [139]  1  1  1  1  1  1 -1  1 -1 -1  1 -1 -1 -1  1  1  1  1  1 -1 -1 -1  1
## [162]  1 -1  1  1  1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1  1  1 -1 -1  1 -1  1  1
## [185]  1  1  1  1  1  1  1  1  1  1  1  1 -1  1 -1 -1 -1  1  1  1 -1  1  1
## [208]  1  1 -1  1  1  1  1  1  1  1  1  1 -1  1 -1  1 -1  1  1  1  1  1  1
## [231]  1  1  1  1  1  1  1  1 -1  1  1  1  1  1 -1  1  1  1  1  1  1 -1 -1
## [254]  1 -1  1 -1  1  1 -1  1  1  1 -1  1  1  1  1  1  1  1  1  1  1  1  1
## [277]  1  1  1  1  1 -1  1  1  1  1 -1  1  1  1 -1  1  1  1 -1  1  1 -1  1
## [300]  1 -1 -1  1  1  1 -1  1  1  1  1  1  1  1  1  1 -1 -1  1  1  1  1  1
## [323]  1  1  1 -1  1  1 -1  1  1  1  1  1  1  1  1  1 -1  1  1  1  1  1  1
## [346]  1  1 -1  1 -1 -1 -1  1 -1  1 -1  1  1 -1  1 -1  1 -1  1  1  1  1  1
## [369]  1  1  1  1  1 -1  1  1  1  1  1 -1  1  1  1  1  1  1  1  1  1  1  1
## [392]  1  1  1  1 -1 -1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
## [415]  1  1  1  1  1  1  1  1 -1  1  1  1 -1  1  1  1  1  1 -1 -1  1  1  1
## [438]  1  1  1  1  1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1  1 -1  1
## [461] -1  1  1 -1  1  1  1  1  1 -1  1  1  1  1  1  1  1 -1  1  1  1  1 -1
## [484] -1  1  1 -1  1  1 -1  1  1  1 -1  1 -1  1  1  1  1 -1  1  1  1  1  1
## [507] -1 -1 -1  1 -1 -1 -1  1 -1  1 -1 -1  1 -1 -1 -1  1  1  1  1  1  1 -1
## [530]  1 -1 -1 -1  1 -1  1  1 -1  1  1  1  1 -1  1  1  1  1  1  1  1 -1  1
## [553]  1  1  1  1 -1  1  1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1  1  1  1
## [576] -1  1 -1  1  1  1 -1  1  1  1  1  1 -1 -1  1  1  1  1  1  1 -1  1  1
## [599] -1
```

Como podemos ver las etiquetas ahora son -1 y 1, lo que nosotros necesitamos. Ahora si podemos aplicar nuestro algoritmo y ver que la salida obtenida es la correcta.

```
modelo = Regress_Lin(datosTrain, etiquetasPositivasNegativas)
plot(datos1_5Train.media, datos1_5Train.simetria,
     xlab="Intensidad", ylab="Simetria", col=etiquetasPositivasNegativas+5)
abline(-modelo[1]/modelo[3], -modelo[2]/modelo[3])
```



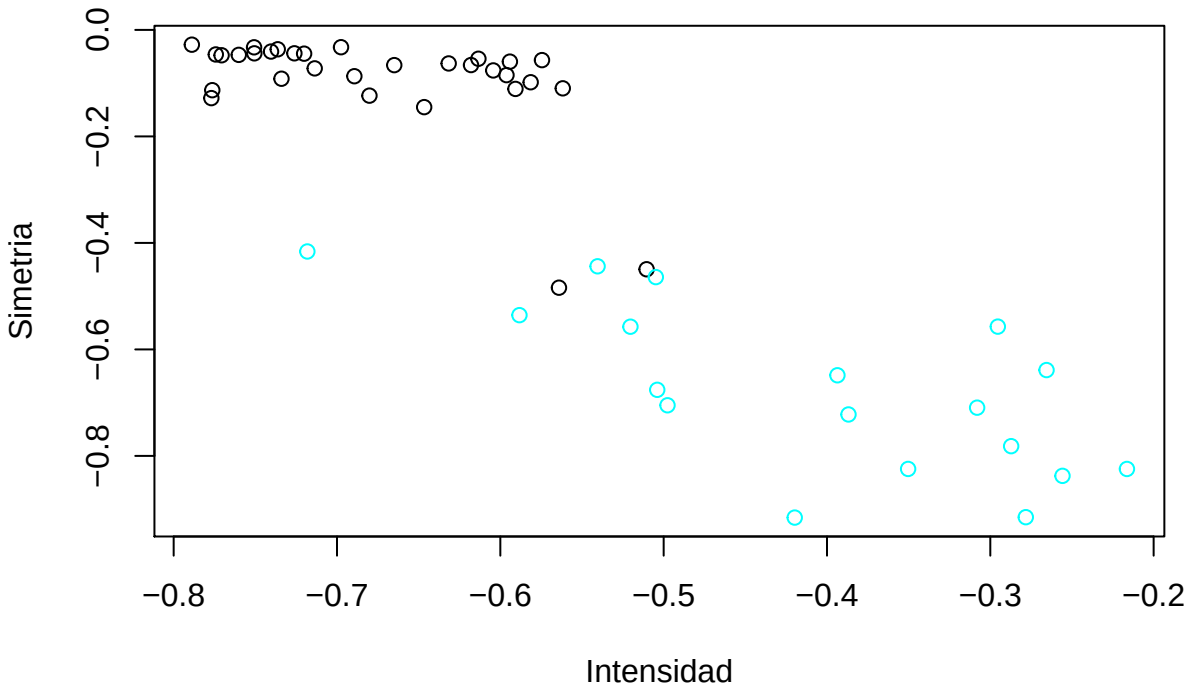
Calculamos el error en la muestra (Ein) como el porcentaje de las etiquetas mal calculadas por nuestra regresión. Viendo la gráfica anterior sabemos que no va a ser cero, porque hay un punto que no se ha clasificado correctamente.

```
calculadas = sign(t(modelo)%*%t(datosTrain))
diferencia = calculadas == etiquetasPositivasNegativas
Ein = length(diferencia[diferencia == FALSE]) / nrow(datosTrain)
Ein
```

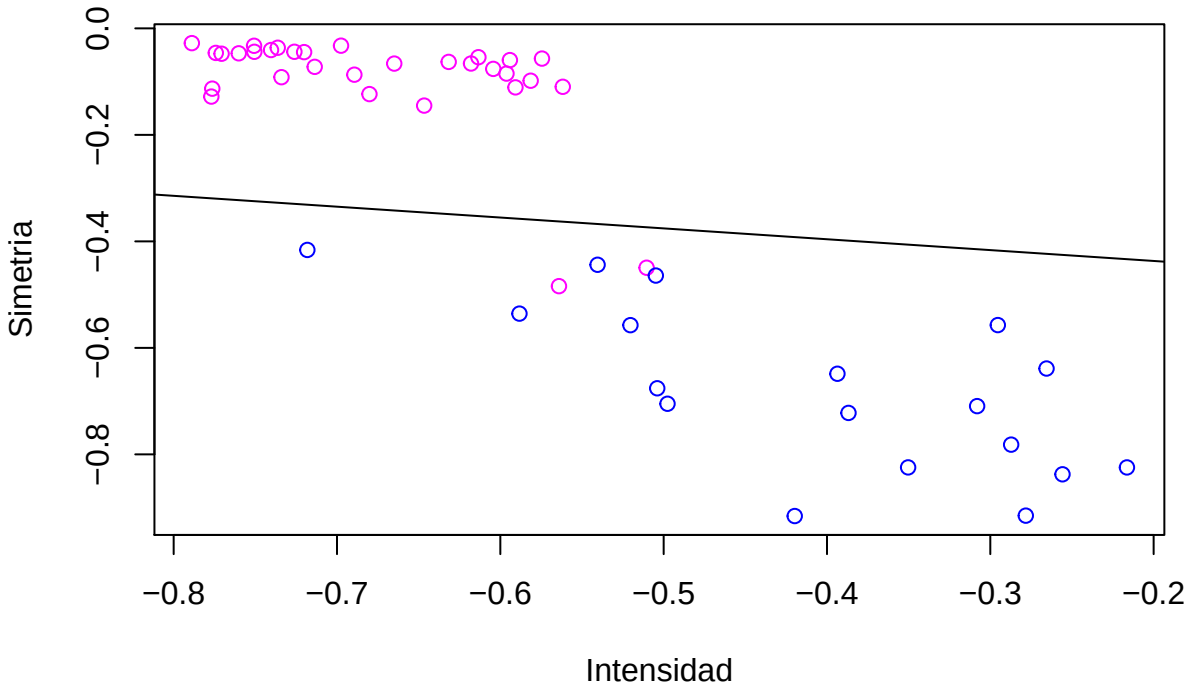
```
## [1] 0.001669449
```

Para calcular el error en fuera de la muestra debemos repetir el proceso pero esta vez ya con los pesos calculados y aplicarlos sobre los datos de test.

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

digitos1_5.test = digit.test[digit.test$V1==1 | digit.test$V1==5,]
etiquetasTest = digitos1_5.test[,1]
ndigitos = nrow(digitos1_5.test)

# se retira la clase y se monta una matriz 3D: 599*16*16
datos1_5Test = array(unlist(subset(digitos1_5.test,select=-V1)),c(ndigitos,16,16))
rm(digit.test)
rm(digitos1_5.test)

datos1_5Test.media = apply(datos1_5Test, 1, mean)
datos1_5Test.simetria = apply(datos1_5Test, 1, simetria)
plot(datos1_5Test.media, datos1_5Test.simetria,
     xlab="Intensidad", ylab="Simetria", col=etiquetasTest)
```




```
# Debemos construir la misma matriz que en el ejercicio anterior
datosTest = matrix(c(rep(1, length(datos1_5Test.simetria)), datos1_5Test.media,
                  datos1_5Test.simetria), nrow = length(datos1_5Test.simetria))
# Tener cuidado con las etiquetas
etiquetasPositivasNegativas = replace(etiquetasTest, etiquetasTest==5, -1)
# Y repetir el proceso, pero con los pesos ya calculados
plot(datos1_5Test.media, datos1_5Test.simetria,
     xlab="Intensidad", ylab="Simetria", col=etiquetasPositivasNegativas+5)
abline(-modelo[1]/modelo[3], -modelo[2]/modelo[3])
```



Como podemos ver la clasificación es bastante buena. Vamos a calcular el error fuera de la muestra (Eout)

para hacernos una idea de como de buena es.

```
calculadas = sign(t(modelo)%*%t(datosTest))
diferencia = calculadas == etiquetasPositivasNegativas
Eout = length(diferencia[diferencia == FALSE]) / nrow(datosTest)
Eout
```

```
## [1] 0.04081633
```

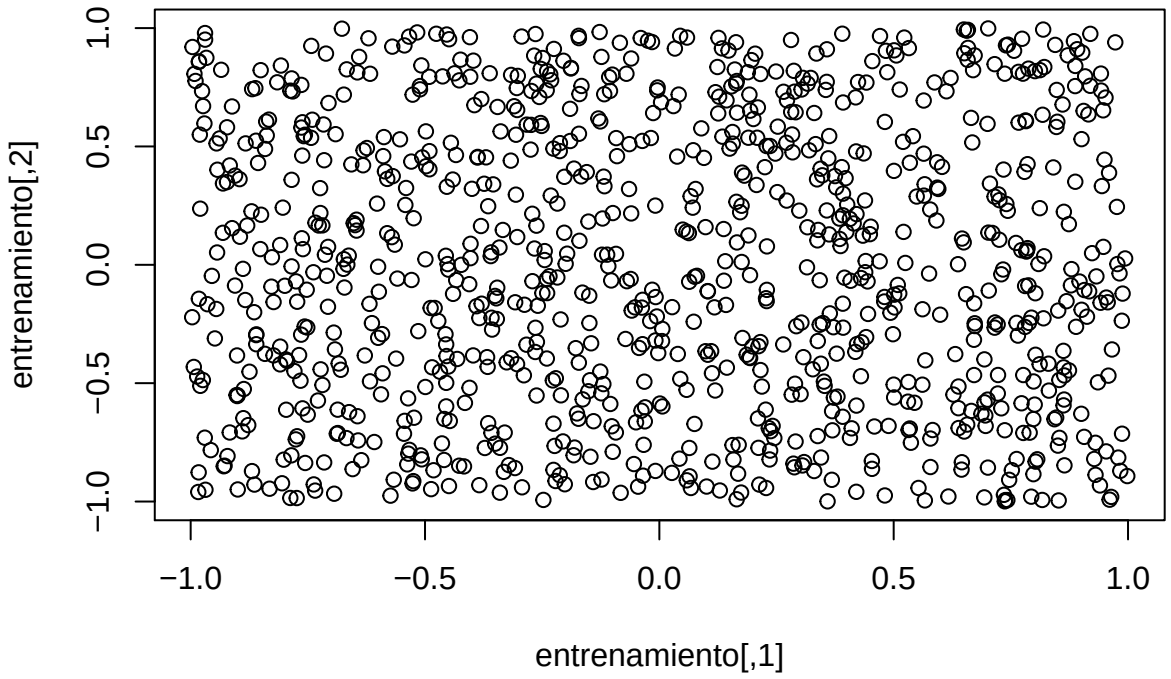
Podemos ver que ambos errores (Ein y Eout) son bastante similares y que, además, Ein es un valor cercano a 0, lo cual nos indica que nuestro modelo funciona correctamente.

Apartado 4

Experimento-1

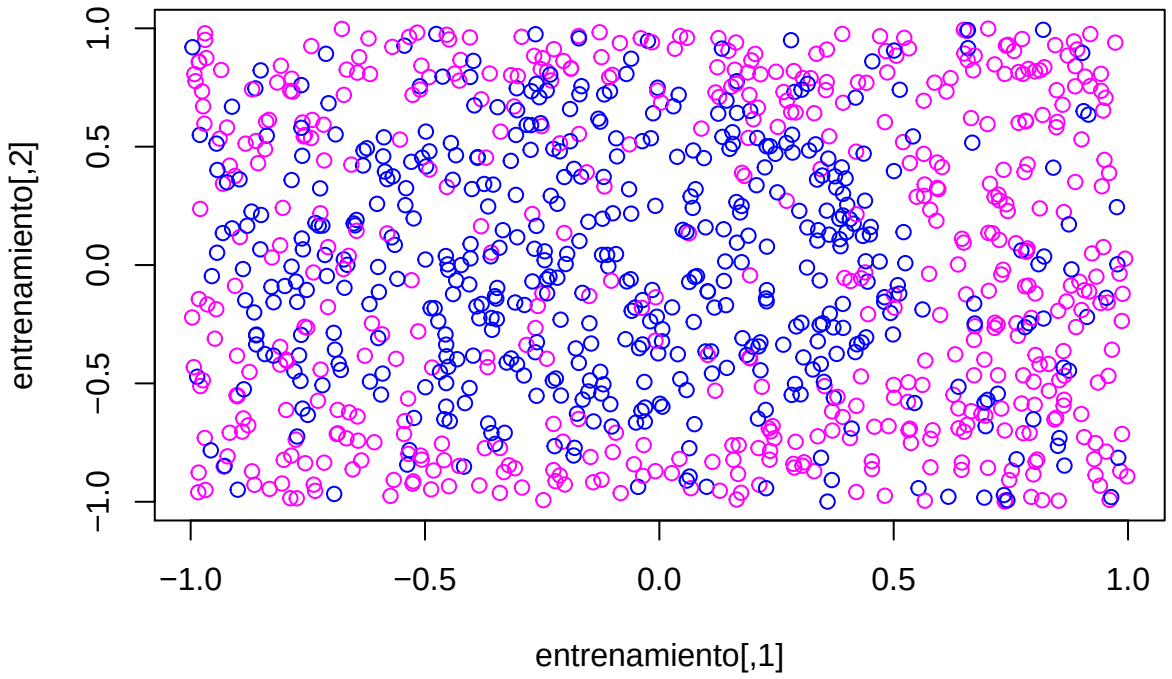
a) Generar una muestra de entrenamiento de N = 1000 puntos en el cuadrado X = [-1, 1] × [-1, 1]. Pintar el mapa de puntos 2D.

```
entrenamiento = simula_unif(1000, 2, c(-1,1))
plot(entrenamiento)
```



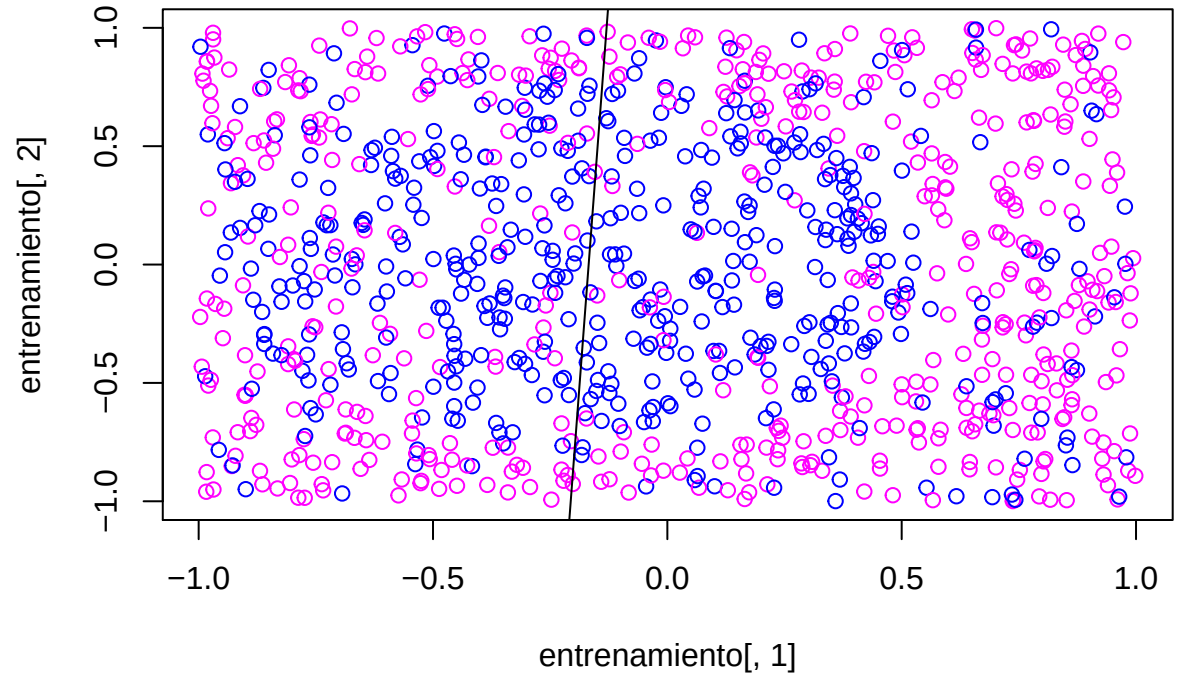
b) Consideremos la función f (x1, x2) = sign((x1 + 0,2)ˆ2 + x2ˆ2 - 0,6) que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas final.

```
f = function(x1, x2) (x1+0.2)^2 + (x2)^2 - 0.6
e = sign(f(entrenamiento[,1], entrenamiento[,2]))
e.R = asignarRuido(e, 10)
plot(entrenamiento, col=e.R+5)
```



c) Usando como vector de características (1, x1, x2) ajustar un modelo de regresion lineal al conjunto de datos generado y estimar los pesos w. Estimar el error de ajuste Ein .

```
datosExperimento = matrix(c(rep(1, length(entrenamiento[,2])), entrenamiento[,1],
                            entrenamiento[,2]), nrow = length(entrenamiento[,2]))
modelo = Regress_Lin(datosExperimento, e.R)
plot(entrenamiento[,1], entrenamiento[,2], col=e.R+5)
abline(-modelo[1]/modelo[3], -modelo[2]/modelo[3])
```




```
calculadas = sign(t(modelo)%*%t(datosExperimento))
diferencia = calculadas == e.R
```

```
Ein = length(diferencia[diferencia == FALSE]) / nrow(datosExperimento)
Ein
```

```
## [1] 0.458
```

d) Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes) y - Calcular el valor medio de los errores E in de las 1000 muestras. - Generar 1000 puntos nuevos por cada iteración y calcular con ellos el valor de Eout en dicha iteración. Calcular el valor medio de Eout en todas las iteraciones.

```
experimento1 = function(i) {
  f = function(x1, x2) (x1+0.2)^2 + (x2)^2 - 0.6
  entrenamiento = simula_unif(1000, 2, c(-1,1))
  eTrain = sign(f(entrenamiento[,1], entrenamiento[,2]))
  eTrain.R = asignarRuido(eTrain, 10)

  datosTrain = matrix(c(rep(1, length(entrenamiento[,2])), entrenamiento[,1],
                        entrenamiento[,2]), nrow = length(entrenamiento[,2]))
  modelo = Regress_Lin(datosTrain, eTrain.R)

  calculadas = sign(t(modelo)%*%t(datosTrain))
  diferencia = calculadas == eTrain.R
  Ein = length(diferencia[diferencia == FALSE]) / nrow(datosTrain)

  test = simula_unif(1000, 2, c(-1,1))
  eTest = sign(f(test[,1], test[,2]))
  eTest.R = asignarRuido(eTest, 10)

  datosTest = matrix(c(rep(1, length(test[,2])), test[,1],
                       test[,2]), nrow = length(test[,2]))

  calculadas = sign(t(modelo)%*%t(datosTest))
  diferencia = calculadas == eTest.R
  Eout = length(diferencia[diferencia == FALSE]) / nrow(datosTest)
  c(Ein, Eout)
}

resultadosExperimento1 = replicate(1000, experimento1())
Ein.medio = mean(resultadosExperimento1[1,])
Eout.medio = mean(resultadosExperimento1[2,])
Ein.medio
```

```
## [1] 0.42175
```

```
Eout.medio
```

```
## [1] 0.42554
```

Podemos ver que el ajuste no es muy bueno debido a que los valores están muy alejados de 0, es decir, hay muchos errores en la clasificación. Hemos conseguido que Ein y Eout se parezcan pero no nos ha servido de nada. Desde mi punto de vista, el ajuste con este modelo lineal no es bueno.

Experimento-2

a) Ahora vamos a repetir el mismo experimento anterior pero usando caracteristicas no lineales. Ahora usaremos el siguiente vector de características: PHI_2(x) = (1, x1, x2, x1x2, x1^2, x2^2). Ajustar el nuevo

modelo de regresión lineal y calcular el nuevo vector de pesos w. Calcular el error Ein

```
entrenamiento = simula_unif(1000, 2, c(-1,1))
eTrain = sign(f(entrenamiento[,1], entrenamiento[,2]))
eTrain.R = asignarRuido(eTrain, 10)

datosTrain = matrix(c(rep(1, length(entrenamiento[,2])), entrenamiento[,1],
                      entrenamiento[,2], entrenamiento[,1]*entrenamiento[,2],
                      entrenamiento[,1]^2, entrenamiento[,2]^2),
                      nrow = length(entrenamiento[,2]))

modelo = Regress_Lin(datosTrain, eTrain.R)
calculadas = sign(t(modelo)%*%t(datosTrain))
diferencia = calculadas == eTrain.R
Ein = length(diferencia[diferencia == FALSE]) / nrow(datosTrain)
Ein
```

```
## [1] 0.237
```

```
experimento2 = function(i) {
  f = function(x1, x2) (x1+0.2)^2 + (x2)^2 - 0.6
  entrenamiento = simula_unif(1000, 2, c(-1,1))
  eTrain = sign(f(entrenamiento[,1], entrenamiento[,2]))
  eTrain.R = asignarRuido(eTrain, 10)

  datosTrain = matrix(c(rep(1, length(entrenamiento[,2])), entrenamiento[,1],
                      entrenamiento[,2], entrenamiento[,1]*entrenamiento[,2],
                      entrenamiento[,1]^2, entrenamiento[,2]^2),
                      nrow = length(entrenamiento[,2]))
  modelo = Regress_Lin(datosTrain, eTrain.R)

  calculadas = sign(t(modelo)%*%t(datosTrain))
  diferencia = calculadas == eTrain.R
  Ein = length(diferencia[diferencia == FALSE]) / nrow(datosTrain)

  test = simula_unif(1000, 2, c(-1,1))
  eTest = sign(f(test[,1], test[,2]))
  eTest.R = asignarRuido(eTest, 10)

  datosTest = matrix(c(rep(1, length(test[,2])), test[,1],
                      test[,2], test[,1]*test[,2],
                      test[,1]^2, test[,2]^2),
                      nrow = length(test[,2]))

  calculadas = sign(t(modelo)%*%t(datosTest))
  diferencia = calculadas == eTest.R
  Eout = length(diferencia[diferencia == FALSE]) / nrow(datosTest)
  c(Ein, Eout)
}

resultadosExperimento2 = replicate(1000, experimento2())
Ein.medio = mean(resultadosExperimento2[1,])
Eout.medio = mean(resultadosExperimento2[2,])
Ein.medio
```

```
## [1] 0.232708
```

```
Eout.medio
```

```
## [1] 0.23535
```

c) Valore el resultados de este EXPERIMENTO-2 a la vista de los valores medios de los errores Ein y Eout

Podemos ver que el procentaje de error ha bajado considerablemente y que se sigue manteniendo la similitud entre Ein y Eout pero que no llegan a ser cero en ninguno de los casos.

A la vista de los resultados de los errores promedios E in y E out obtenidos en los dos experimentos ¿Que modelo considera que es el más adecuado? Justifique la decisión.

Como era de esperar, al aumentar el vector de características se han reducido los errores, tanto dentro como fuera de la muestra, ya que podemos afinar más en la clasificación de un nuevo dato al tener mayor información sobre el mismo. Como en el segundo modelo los errores son menores podemos decir que dicho modelo es mejor que el primero, ya que produce una mejor tasa de acierto.

Bonus

En este ejercicio exploramos cómo funciona regresión lineal en problemas de clasificación. Para ello generamos datos usando el mismo procedimiento que en ejercicios anteriores. Suponemos X=[-10, 10]x[-10, 10] y elegimos muestras aleatorias uniformes dentro de X. La función f en cada caso será una recta aleatoria que corta a X y que asigna etiqueta a cada punto de X con el valor del signo de f en dicho punto. En cada apartado generamos una muestra y le asignamos etiqueta con la función f generada. En cada ejecución generamos una nueva función f

a) Fijar el tamaño de muestra N = 100. Usar regresión lineal para encontrar una primera solución g y evaluar Ein, (el porcentaje de puntos incorrectamente clasificados). Repetir el experimento 1000 veces y promediar los resultados ¿Qué valor obtiene para Ein?

```
apartadoA = function(i) {
  bonus = simula_unif(100, 2, c(-10,10))
  recta = simula_recta(c(-10,10))
  bonus.e = lapply(1:nrow(bonus), function(i) {
      d = bonus[i,]
      sign(d[2] - recta[1]*d[1] - recta[2])
  })
  bonus.e = unlist(bonus.e)

  bonusTrain = matrix(c(rep(1, length(bonus[,2])), bonus[,1],
                        bonus[,2]), nrow = length(bonus[,2]))

  g = Regress_Lin(bonusTrain, bonus.e)

  calculadas = sign(t(g)%*%t(bonusTrain))
  diferencia = calculadas == bonus.e
  Ein = length(diferencia[diferencia == FALSE]) / nrow(bonusTrain)
  Ein
}

EinApartadoA = replicate(1000, apartadoA())
Ein.medio = mean(EinApartadoA)
Ein.medio
```

```
## [1] 0.03817
```

b) Fijar el tamaño de muestra N = 100. Usar regresión lineal para encontrar g y evaluar Eout. Para ello generar 1000 puntos nuevos y usarlos para estimar el error fuera de la muestra, Eout (porcentaje de puntos mal clasificados). De nuevo, ejecutar el experimento 1000 veces y tomar el promedio. ¿Qué valor obtiene de Eout? Valore el resultado.

```
apartadoB = function(i) {
  bonus = simula_unif(100, 2, c(-10,10))
  recta = simula_recta(c(-10,10))
  bonus.e = lapply(1:nrow(bonus), function(i) {
      d = bonus[i,]
      sign(d[2] - recta[1]*d[1] - recta[2])
  })
  bonus.e = unlist(bonus.e)

  bonusTrain = matrix(c(rep(1, length(bonus[,2])), bonus[,1],
                        bonus[,2]), nrow = length(bonus[,2]))

  g = Regress_Lin(bonusTrain, bonus.e)

  bonus = simula_unif(1000, 2, c(-10,10))
  bonusTest.e = lapply(1:nrow(bonus), function(i) {
      d = bonus[i,]
      sign(d[2] - recta[1]*d[1] - recta[2])
  })
  bonusTest.e = unlist(bonusTest.e)

  bonusTest = matrix(c(rep(1, length(bonus[,2])), bonus[,1],
                       bonus[,2]), nrow = length(bonus[,2]))

  calculadas = sign(t(g)%*%t(bonusTest))
  diferencia = calculadas == bonusTest.e
  Eout = length(diferencia[diferencia == FALSE]) / nrow(bonusTest)
  Eout
}

EoutApartadoB = replicate(1000, apartadoB())
Eout.medio = mean(EoutApartadoB)
Eout.medio
```

```
## [1] 0.049793
```

Nos da un error fuera de la muestra bastante bajo, lo cual nos quiere decir que la mayoría de los datos están bien clasificados.

c) Ahora fijamos N = 10, ajustamos regresión lineal y usamos el vector de pesos encontrado como un vector inicial de pesos para PLA. Ejecutar PLA hasta que converja a un vector de pesos final que separe completamente la muestra de entrenamiento. Anote el número de iteraciones y repita el experimento 1.000 veces ¿Cual es valor promedio de iteraciones que tarda PLA en converger? (En cada iteración de PLA elija un punto aleatorio del conjunto de mal clasificados). Valore los resultados

```
apartadoC = function(i) {
  bonus = simula_unif(10, 2, c(-10,10))
  recta = simula_recta(c(-10,10))
  bonus.e = lapply(1:nrow(bonus), function(i) {
      d = bonus[i,]
      sign(d[2] - recta[1]*d[1] - recta[2])
```

```
  })
  bonus.e = unlist(bonus.e)

  bonusTrain = matrix(c(rep(1, length(bonus[,2])), bonus[,1],
                        bonus[,2]), nrow = length(bonus[,2]))

  g = Regress_Lin(bonusTrain, bonus.e)

  solucion = ajusta_PLA(bonus, bonus.e, 1000, c(g[1],g[2],g[3]))
  solucion[2]
}

iteraciones = replicate(1000, apartadoC())
iteraciones.media = mean(unlist(iteraciones))
iteraciones.media
```

```
## [1] 5.696
```

Podemos ver que el número de iteraciones se reduce bastante en comparación a cuando se le pasa un vector de pesos inicialiazado a cero o un vector de pesos aletario, ya que el Perceptron parte de un vector de pesos muy ajustado y bueno (obtenido por la regresion lineal).