Practica 2

Adrián Sánchez Cerrillo, Miguel Ángel López Robles

18 de abril de 2018

1. Ejercicio sobre la búsqueda iterativa de óptimos

1.1 Generar una nube de puntos

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

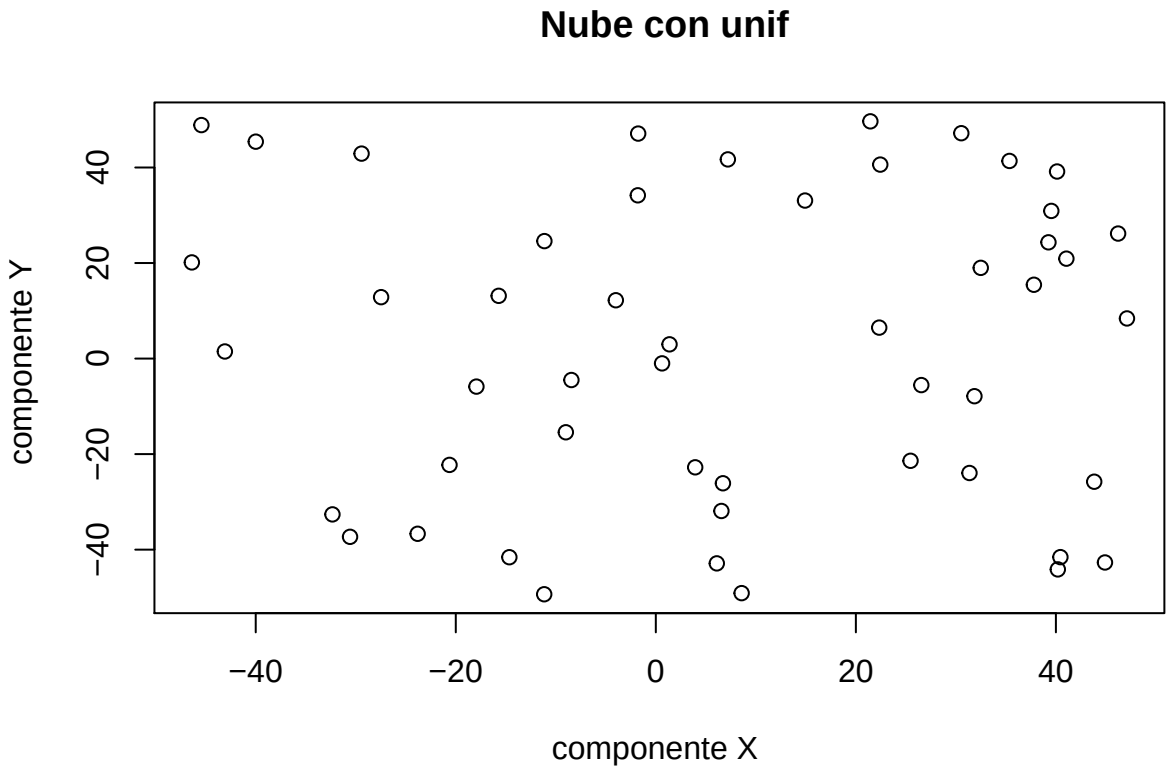
Para generar una nube de puntos con N=50,dim=2 rango[-50,50] vamos a usar dos funciones distintas. La primera genera una distribuición uniforme y la segunda para usar una distribución de gauss. Ádemas en primer lugar establecemos la semilla a 4 para poder repetir el experimento posteriormente y obtener los mismos resultados. Usaremos las siguientes funciones:

```
set.seed(4) # se establece la semilla
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
}

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 especifica
  m = t(replicate(N,simula_gauss1())) # repite N veces, simula_gauss1 y se hace la traspuesta
  m
}
```

Ahora ya podemos usar estas funciones con lo que generaremos nuestra nube y las mostraremos con un plot:

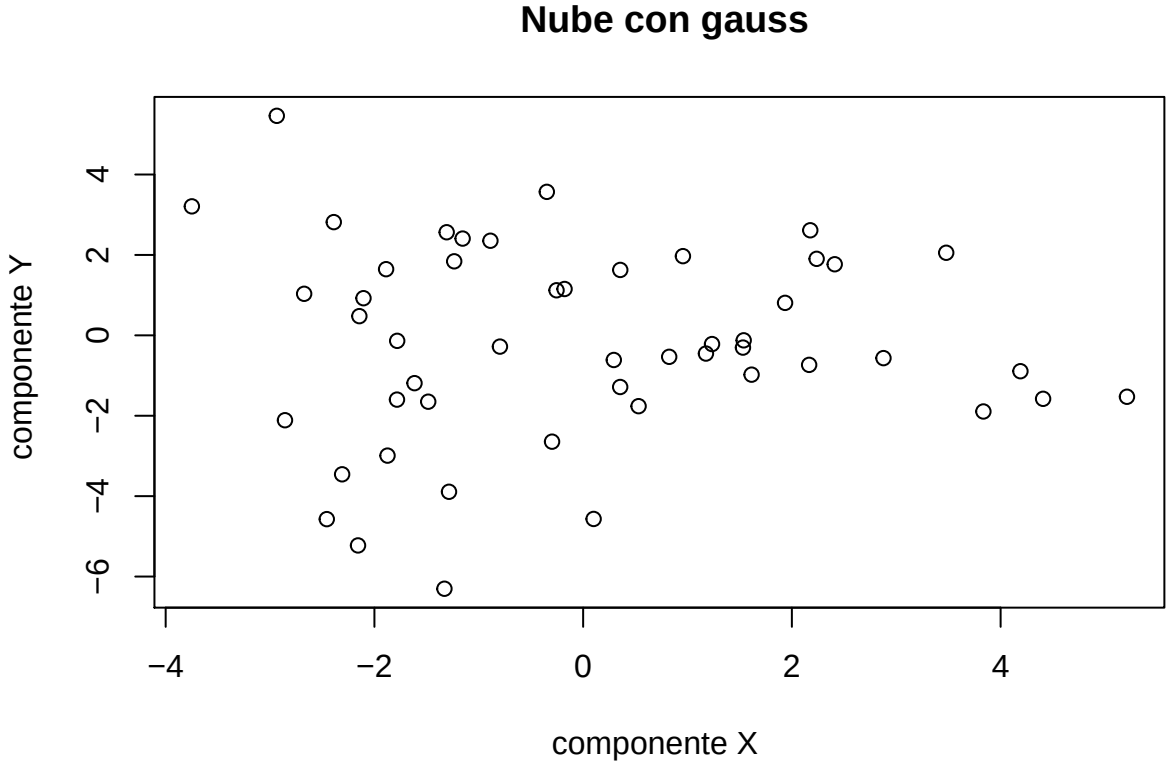
a) Considere N = 50, dim = 2, rango = [-50, +50] con *simula_unif(N, dim, rango).*

```
datosU = simula_unif(50,2,c(-50,50))

plot(datosU,main = " Nube con unif",xlab = "componente X ", ylab = "componente Y ")
```



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

```
datosG = simula_gaus(50,2,c(5,7))

plot(datosG,main = " Nube con gauss",xlab = "componente X ", ylab = "componente Y ")
```



1.2 Etiquetar los puntos

Ahora vamos a usar la función $f(x, y) = y - ax - b$ para etiquetar cada punto. Como podemos ver esta función va a etiquetar según la distancia de cada punto con una recta. Para ello vamos a necesitar generar una recta de forma aleatoria usando la siguiente función:

```
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
}
```

Lo siguiente que necesitamos por tanto es definir $f(x, y)$ la cual, a partir de ciertos valores de x e y, nos devolvería la clasificacion. Mencionar que en este caso vamos a tener en cuenta que si el lenguaje R no encuentra el nombre de una variable dentro de la función, lo buscará fuera en el entorno. Por tanto la variable *rectaS* la vamos a definir más tarde. De este modo ya veremos como podemos definir una función para evaluar a todas las funciones incluido las del apartado 3.

```
f = function(x,y){
  z = y - rectaS[1]*x + rectaS[2]
}
```

Como hemos mencionado anteriormente, definiremos una función que usaremos durante toda la practica para clasificar los puntos.

```
clasifica = function(datos, funcion){
  etiqueta = diag(sign(outer(datos[,1],datos[,2],funcion)))
  etiqueta
}
```

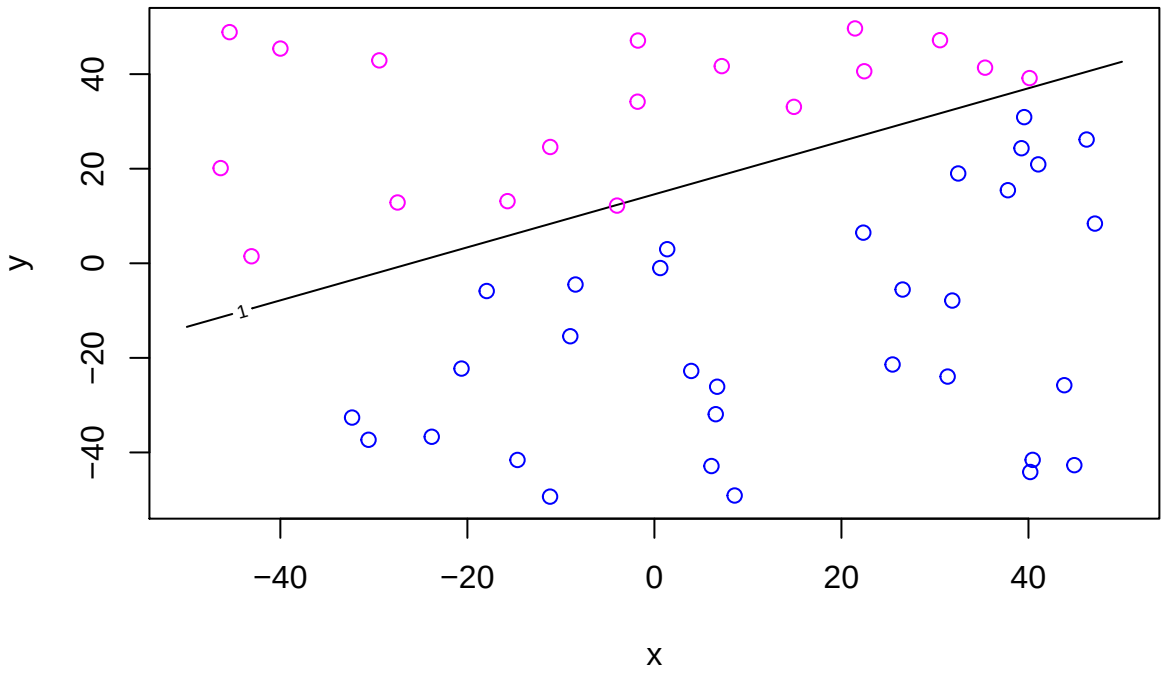
Por último antes de poder realizar los apartados necesitamos una función que nos permita visualizar el comportamiento de las funciones $f(x, y)$ para ver cómo están clasificando los puntos.

```
pintar_frontera = function(f,rango=c(-50,50)) {
  x=y=seq(rango[1],rango[2],length.out = 500)
  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 = 1, xlim =rango, ylim=rango, xlab = "x", ylab = "y")
}
```

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)

A continuación, vamos a pintar los resultados obtenidos para ver como se clasifican los datos

```
rectaS = simula_recta(c(-50,50))
etiquetas = clasifica(datosU, f)
pintar_frontera(f)
points(datosU, col= etiquetas +5)
```

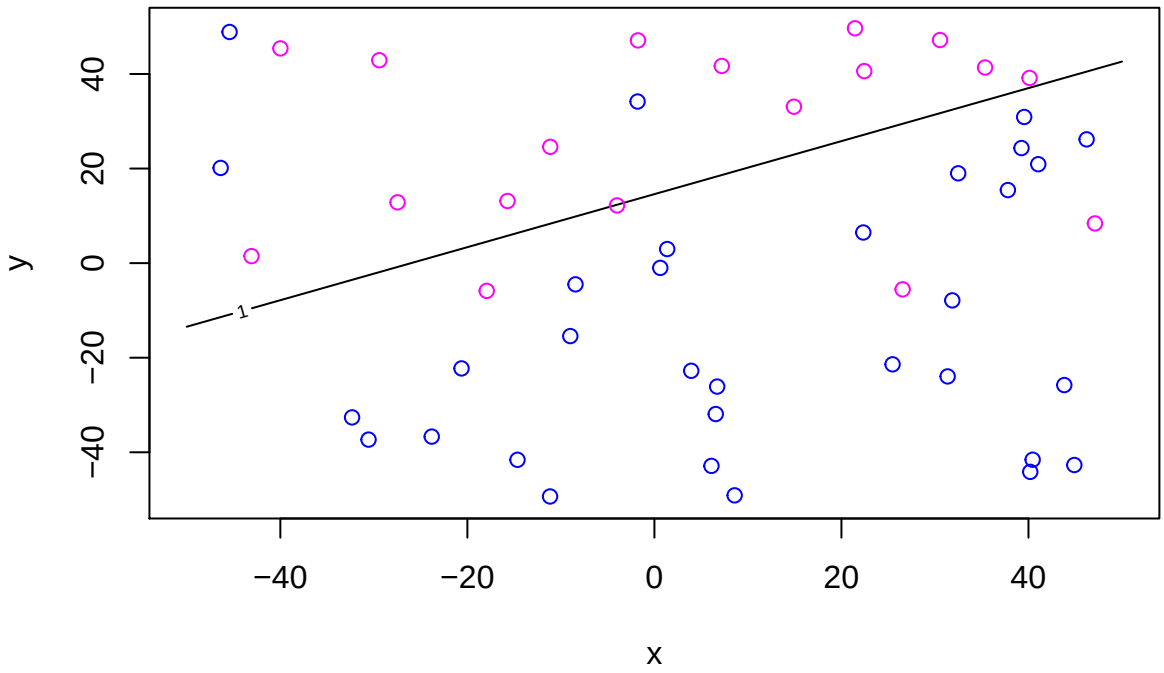


####

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

En este apartado se nos pide que introduzcamos un 10% de ruido en el conjunto de datos. Para ello vamos a usar la función genera_ruido. En esta función vamos a modificar un 10% de las positivas y otro 10% de las negativas. El procedimiento va ser seleccionar los indices positivos y negativos y seleccionar aleatoriamente un diez porciento de estos y cambiarle el signo a la etiqueta.

```
genera_ruido = function(etiquetas){
  n = length(etiquetas)
  indices = c(1:n)

  positivos = indices[etiquetas > 0]
  indices_pos = sample(positivos,round(n*0.1))
  etiquetas[indices_pos] = -etiquetas[indices_pos]

  negativos = indices[etiquetas < 0]
  indices_neg = sample( negativos,round(n*0.1))
  etiquetas[indices_neg] = -etiquetas[indices_neg]

  etiquetas
}
etiquetas = genera_ruido(etiquetas)
pintar_frontera(f)
points(datosU, col= etiquetas +5)
```



Como podemos ver ahora tenemos datos mal clasificados respecto a la recta y como veremos más adelante puede influir en como los algoritmos clasificarán los datos al tratarse de ruido.

1.3 Funciones más complejas

En este apartado vamos a definir 4 nuevas funciones mas complejas, de orden 2. Vamos a ver como la frontera que dibujan son funciones como el circulo o una elipse. Para ello las definimos a continuación:

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

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

$f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$

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

$f(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$

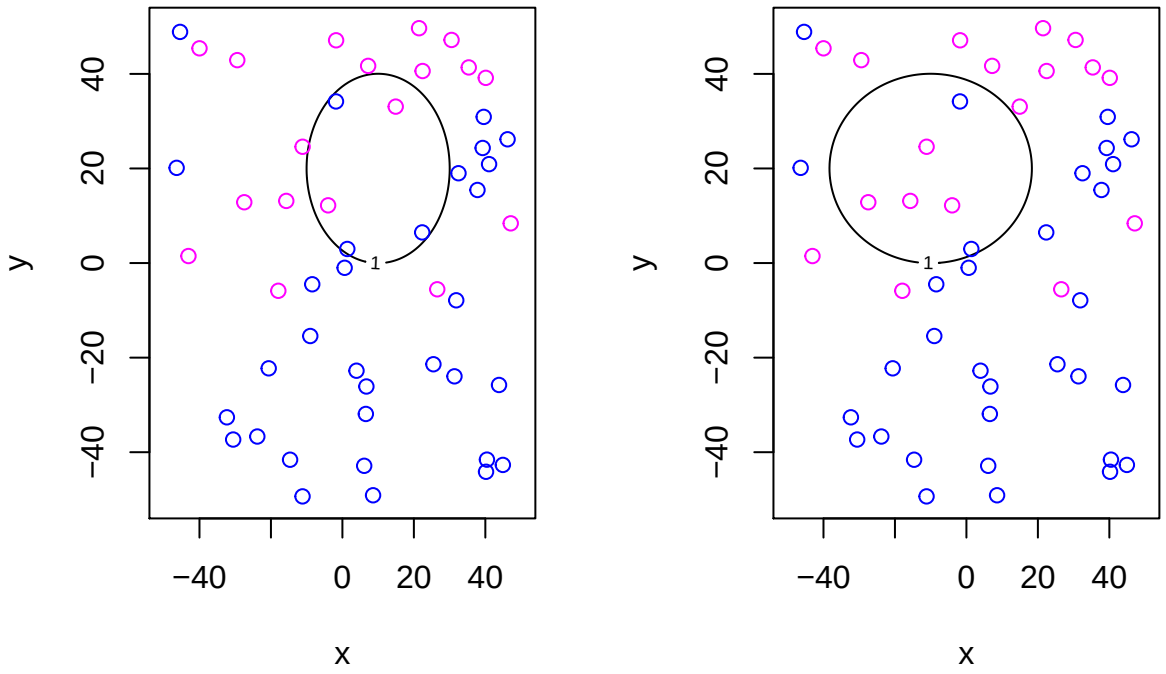
```
f3 = function(x,y){
  z = 0.5*(x-10)^2 -(y + 20)^2 -400
}
```

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

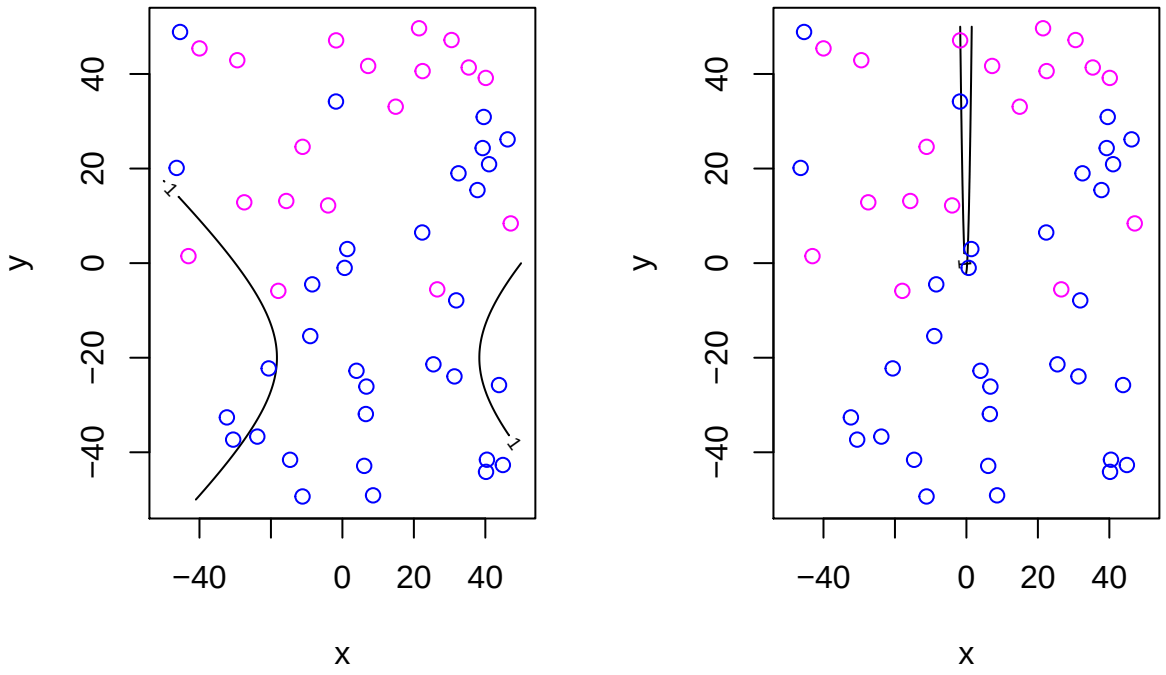
```
f4 = function(x,y){
  z = y -20*(x^2 ) -5*x +3
}
```

Ahora vamos a representar la clasificación del apartado 2b en el cual habíamos introducido ruido y vamos a ver si con estas funciones más complejas podemos conseguir resolver el problema de los puntos que no se clasifican de forma correcta al meter ruido.

```
par(mfrow=c(1,2))
pintar_frontera(f1)
points(datosU, col= etiquetas +5)

pintar_frontera(f2)
points(datosU, col= etiquetas +5)
```




```
pintar_frontera(f3)
points(datosU, col= etiquetas +5)

pintar_frontera(f4)
points(datosU, col= etiquetas +5)
```




```
par(mfrow = c(1,1))
```

Al visualizar los puntos que teníamos con estas nuevas funciones como podemos ver no hemos resuelto el problema. Estas funciones tendrían un error mayor a la hora de clasificar ya que el modelo subyacente con el cual se clasifican los datos es una recta. El error que se producen se debe al ruido introducido anteriormente. Estas clases más complejas se deberían usar para cuando tenemos datos que no son separables linealmente, por lo tanto, esto nos muestra que el hecho de usar funciones más complejas no implica que resulte en una mejor clasificación. Cuando tenemos ruido no podemos intentar resolver este error aumentando la clase de funciones ya que lo que podemos conseguir es un sobreajuste, que aunque no se produce en este caso, podría no favorecernos fuera del train.
Intentar clasificar de forma correcta el ruido, aunque a priori no lo conozcamos, nos causará grandes problemas fuera de la muestra. Por tanto la manera de solucionar esto sería intentando reducir el ruido, el cual, tal y como hemos visto en teoría, nunca podemos asegurar que no exista y por lo tanto, reducirlo no será una tarea sencilla.

2. Modelos Lineales

2.1 Implementar el algoritmo Perceptrón (PLA)

El algoritmo de Perceptrón es un método de regresión lineal básico, el cuál hemos implementado de la siguiente forma:
*** Parámetros**
- *datos*: Matriz de datos que corresponden al conjunto de entrenamiento o data train.
- *label*: Conjunto de etiquetas asociadas a la fila i-ésima de nuestro data train.
- *maxiter*: Número máximo de iteraciones del algoritmo.
- *vini*: Vector inicial de pesos.

*** Algoritmo**
- Con las iteraciones, vamos actualizando el vector de pesos cuando clasifiquemos los datos de forma errónea con el vector de pesos actual.
- En cada iteración conocemos el valor del vector de pesos actual y el anterior, con lo cual comprobaremos que la diferencia entre dichos vectores sea significativa.

- Cuando el algoritmo finaliza, realizamos un control que imprime la razón de su finalización, aportándonos información relativa.

*** Salida**

- El algoritmo propocionará una lista conteniendo vector de pesos w, y el número de iteraciones.

```
ajusta_PLA = function(datos,label,max_iter, vini){
  datos = cbind(datos,1)

  w = vini
  wanterior = vini
  it = 0
  umbral = 10^-6
  igual = F

  while ((it < max_iter) & !igual){

    for (i in 1:nrow(datos)){
      if (sign(datos[i,] %*% t(w)) != label[i]){
        w = w + label[i]* datos[i,]
      }
    }
    igual = all(abs(wanterior-w) < umbral)
    it = it+1
    wanterior = w
  }

  if(igual){
    cat("El perceptron para por que no hay diferencia entre los pesos anteriores, tras ", it, " iteraci
  }
  else{
    print("El perceptron para por el limite de iteraciones.")
  }

  w <- list(w, it)
  names(w) <- c("w", "iteraciones")

  w
}
```

a) 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.

Vamos a probar el Perceptrón con los datos separables.
Inicializamos el vector de pesos a 0:

```
w = matrix(c(0,0,0),nrow = 1)
print(dim(w))
```

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

Usaremos también la función de paso a recta de la practica anterior para pintar la recta que esta usando para clasificar

```
pasoARecta= function(w){
  if(length(w)!= 3)
    stop("Solo tiene sentido con 3 pesos")
  a = -w[1]/w[2]
  b = -w[3]/w[2]
  c(a,b)
}
```

Obtenemos las etiquetas que clasifican nuestra función de apartados anteriores, usando una recta simulada como en el ejercicio1.

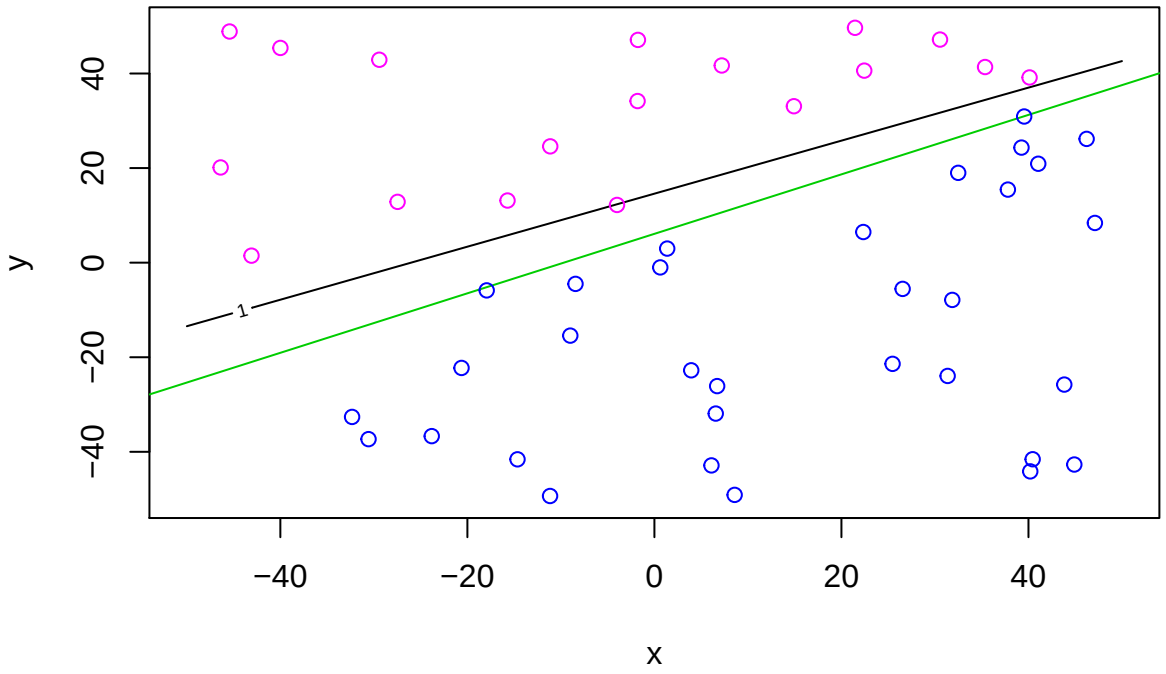
```
etiquetas = clasifica(datosU, f)
```

Obtenemos el vector de pesos, y pintamos la recta junto con los datos, para ver el resultado del perceptron.

```
PLA_ceros = ajusta_PLA(datosU, etiquetas, 600,w )
```

```
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  474  iteraciones.
```

```
ab = pasoARecta(PLA_ceros$w)

pintar_frontera(f)
abline(ab[2],ab[1], col = 3)
points(datosU, col = sign(cbind(datosU,1)%*%t(PLA_ceros$w))+5)
```



Número de iteraciones para vector de ceros: 474
Como podemos ver en el perceptron, representado por la linea verde, ha acabado antes del número de iteraciones máximo y además, ha encontrado una solución que separa sin error todos los datos. Esto era de esperar ya que partíamos de unos datos separables por lo que podemos asegurar que el perceptron converge a una solución

Ahora probamos con 10 vectores de números aleatorios y vamos a ver si esto se mantiene.
Primero generamos los vectores de pesos aleatorios que usaremos, para ello haremos uso de la función *simula_unif*.

```
  pesos <- matrix(c(simula_unif(10, 3)),ncol = 3)
```

Una vez tenemos los pesos, procedemos a calcular de forma iterativa el número medio de iteraciones que necesitan para converger.

```
iteraciones <- 0

for (i in 1:10){
  w = matrix(pesos[i,],ncol=3)
  PLA = ajusta_PLA(datosU, etiquetas, 600,w)
  iteraciones = iteraciones + PLA$iteraciones
}
```

```
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  479  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  478  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  484  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  486  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  424  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  498  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  504  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  425  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  480  iteraciones.
## El perceptron para por que no hay diferencia entre los pesos anteriores, tras  472  iteraciones.
```

```
it_media = iteraciones/10.0
```

Número medio de iteraciones vector de aleatorios: 473

Como podemos ver en todos los casos el perceptron no ha llegado a usar todas las iteraciones y en todos ellos converge. El número de iteraciones usadas dependerá del vector inicial del que parte pero finalmente en todos los casos converge a la solución.

b) 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.

Ahora vamos a generar un ruido del 10% para cada tipo en el conjunto de datos.

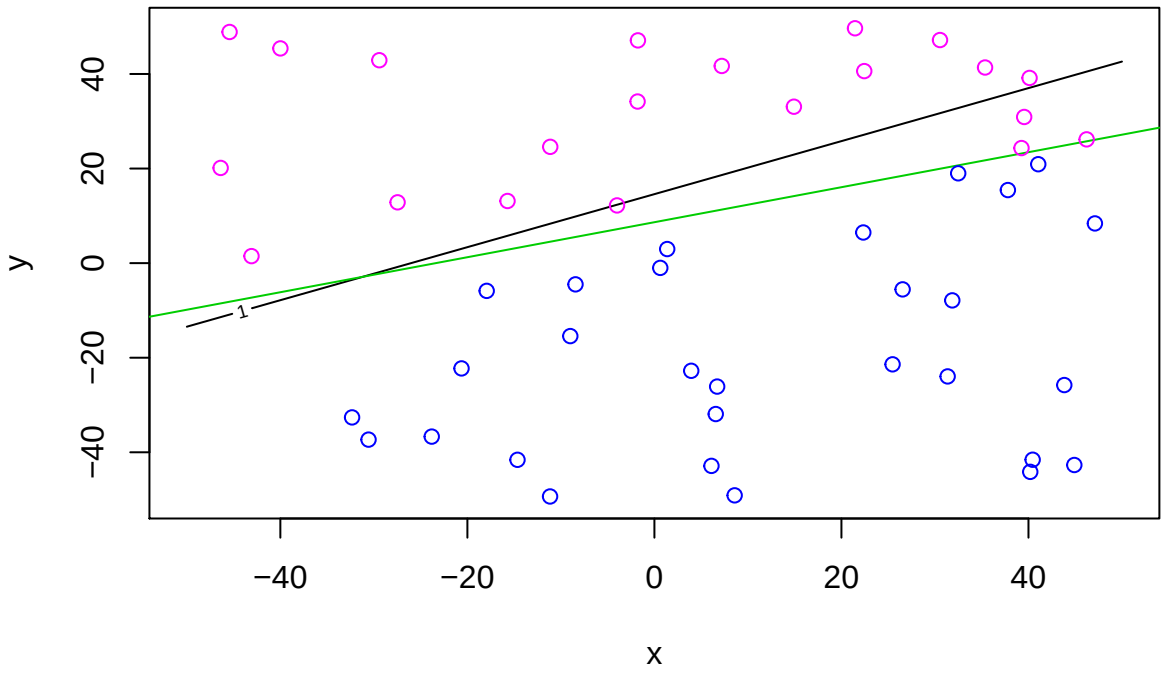
```
etiquetas = genera_ruido(etiquetas)
```

Ajustamos con el algoritmo PLA, e inicializamos el vector de pesos a cero.

```
w = matrix(c(0,0,0),nrow = 1)
PLA = ajusta_PLA(datosU, etiquetas, 600,w )
```

```
## [1] "El perceptron para por el limite de iteraciones."
```

```
ab = pasoARecta(PLA$w)

pintar_frontera(f)
abline(ab[2],ab[1], col = 3)
points(datosU, col = sign(cbind(datosU,1)%*%t(PLA$w))+5)
```



Número medio de iteraciones con vector de ceros: 600

Como podemos ver en este caso el perceptron ha necesitado todas las iteraciones y la solución que nos devuelve no es muy buena. Esto se debe a que el PLA no almacena la mejor solución encontrada y por tanto nos devuelve la solución que tiene en la última iteración. Este hecho es algo que podiamos intuir, ya que en este caso, partíamos de un conjunto de datos no separables al introducir ruido.
Por tanto, el perceptron nunca va a converger ya que siempre va a tener datos mal clasificados y aunque encontrara la solución como en el caso anterior, tendría mal clasificado el ruido. Para solucionar el problema de devolver la última solución, usaremos en el bonus el algoritmo PLA pocket que nos devuelve la mejor solución encontrada en las distintas iteraciones.

Ahora realizamos el procedimiento con el vector de números aleatorios de los apartados anteriores. Comprobaremos que lo expuesto anteriormente se mantiene a pesar de empezar desde vectores de pesos diferentes.

```
iteraciones <- 0
for (i in 1:10){
    w = matrix(pesos[i,],ncol=3)
  PLA = ajusta_PLA(datosU, etiquetas, 600, w)
  iteraciones <- iteraciones + PLA$iteraciones
}
```

```
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
## [1] "El perceptron para por el limite de iteraciones."
```

```
it_media <- iteraciones/10.0
```

Número medio de iteraciones vector de aleatorios: 600

El número medio de iteraciones como hemos visto es el total de iteraciones que habiamos puesto como limite, 600. Esto nos demuestra que en el caso de que los datos no sean totalmente separables el PLA no va a converger y por tanto podría ciclar de forma infinita si no imponemos un limite de iteraciones.

En conclusión, podemos decir que el PLA es un algoritmo bastante fácil de implementar, sencillo e intuitivo y que además proporciona resultados aceptables en pocas iteraciones.
En contraposición, este algoritmo tiene una gran restricción y es que los datos deben ser separables, en caso contrario no asegura que el resultado ofrecido sea bueno. Además en caso de no converger, devolvería la solución encontrada en la última iteración.

2.2 Implementar el algoritmo de Regresión Logística

El algoritmo de Regresión Logística es un método probabilístico, el cuál hemos implementado de la siguiente forma:
*** Parámetros**
- *datos*: Matriz de datos que corresponden al conjunto de entrenamiento o data train.
- *etiquetas*: Conjunto de etiquetas asociadas a la fila i-ésima de nuestro data train.
- *maxit*: Número máximo de iteraciones del algoritmo. Por defecto 10000 iteraciones.
- *mu*: Tasa de aprendizaje. Por defecto 0.01.
- *dif*: Diferencia significativa para los pesos.

*** Algoritmo**
- Con las iteraciones, vamos contando las que llevamos actualmente, pues será una de nuestras condiciones de parada. Por otro lado, estableceremos la diferencia de parada entre los vectores de pesos, como la norma de la diferencia de dichos vectores.
- Utilizaremos el Gradiente Descendente Estocástico sobre la función sigmoidal y así luego actualizar los pesos.
- Cuando el algoritmo finaliza, realizamos un control que imprime la razón de su finalización, aportándonos información relativa.

*** Salida**
- El algoritmo propocionará como salida el vector de pesos *w_new*.

```
Regresion_Logistica =function(datos,etiquetas,mu=0.01,maxit = 10000, dif = 0.01){
  w_new = matrix(c(0,0,0),nrow = 1)

  datos = cbind(datos,1)
  it = 0
  diferencia = 10000

  while (it < maxit & diferencia > dif) {
    w_old = w_new
    indices = sample(1:nrow(datos),nrow(datos))

    for(i in indices){
      gradiente =  (-etiquetas[i]*datos[i,])/c(1+exp( etiquetas[i]*datos[i,]%*%t(w_new)))
      w_new = w_new - mu *gradiente
    }

    w = w_old - w_new
    diferencia = sqrt(sum(w^2))

    it = it+1
  }
  if(it == maxit){
    print("La regresión finaliza por numero de iteraciones")
```

```
  }
  else{
    print("La regresión acaba por la diferencia del vector w entre iteraciones")
  }

  w_new
}
```

Definiremos dos funciónes auxiliares para facilitar la tarea en este apartado.
- *error*: Función que calcula el error de la regresión Logística.
- *sigmoide*: Función que calcula h(x) para la Regresión Logística, obteniendo así la probabilidad.

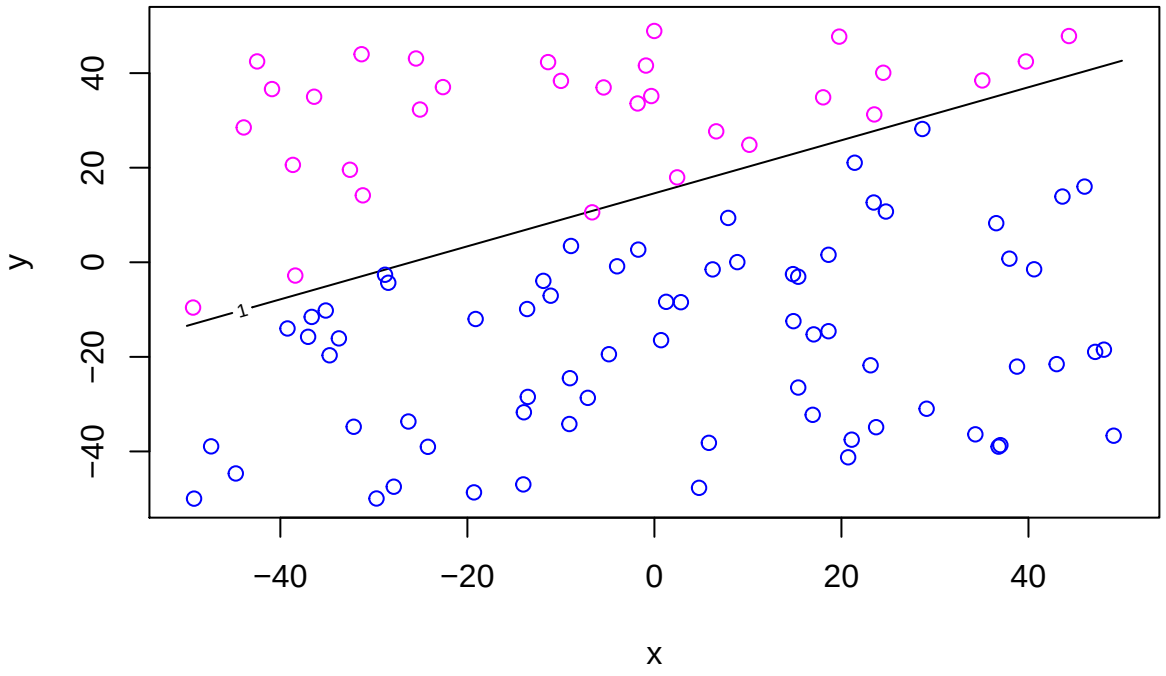
```
error = function(datos, etiquetas,w){
  datos = cbind(datos,1)
  (1/nrow(datos)) *sum(log(1+exp( -etiquetas*datos%*%t(w))))
}

sigmoide = function(datos,w){
  datos = cbind(datos,1)
  exp(datos%*%t(w)) / ( 1+ exp(datos%*%t(w)))
}
```

Definimos los nuevos datos y las etiquetas:

```
datosX = simula_unif(100,2,c(-50,50))
etiquetas = clasifica(datosX, f)
```

Vemos que los datos generados quedan de la siguiente forma, clasificados por una recta aleatoria:

```
pintar_frontera(f)
points(datosX, col = etiquetas+5)
```

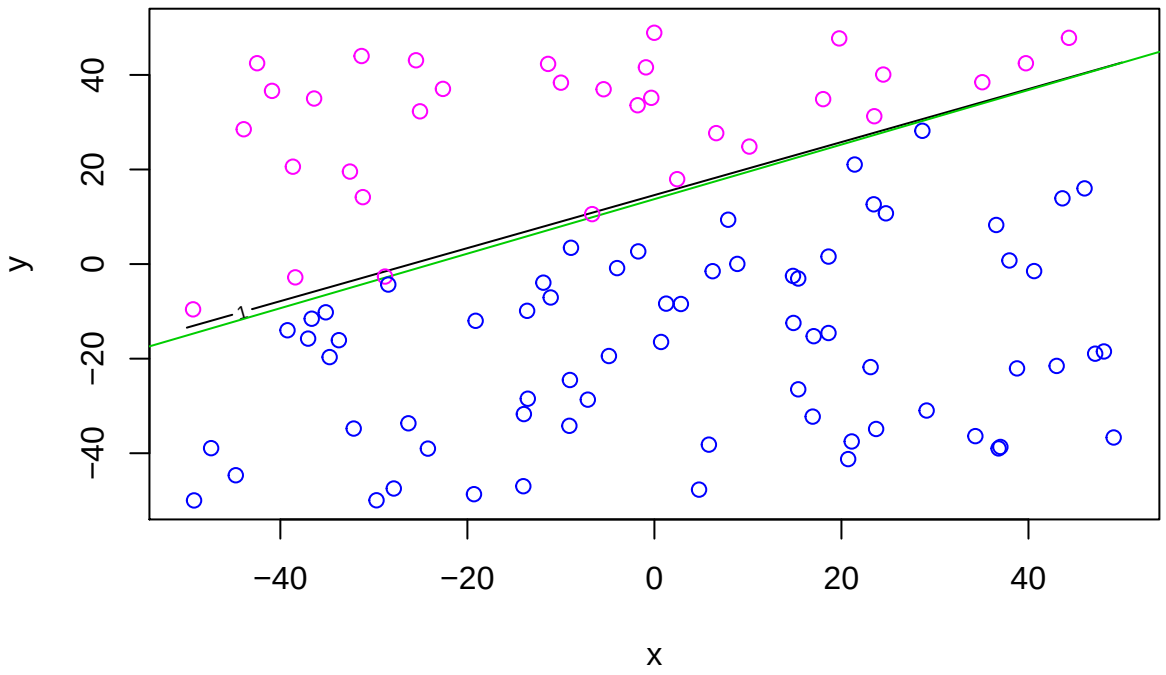


Calculamos el vector de pesos y el error asociado a dicho vector. Una vez realizados los cálculos, procedemos a comprobar los valores erróneos, extrayendo la etiqueta y la probabilidad obtenida por nuestro modelo probabilístico.

```
w = Regresion_Logistica(datosX, etiquetas)
```

```
## [1] "La regresión acaba por la diferencia del vector w entre iteraciones"
```

```
ab = pasoARecta(w)
pintar_frontera(f)
abline(ab[2],ab[1], col = 3)

points(datosX, col = sign(cbind(datosX,1)%*%t(w))+5)
```




```
e = error(datosX,etiquetas,w)
probabilidad = sigmoide(datosX,w)

etiquetas[etiquetas<0] = 0

et_error = etiquetas[etiquetas != round(probabilidad)]
prob_error = probabilidad[etiquetas != round(probabilidad)]
```

El error para la regresión logística es 0.0169479.

Como podemos ver, la regresión logística consigue un buen ajuste y un error cercano a cero. En el posterior bucle mostraremos si algún punto se ha clasificado de forma incorrecta, y su probabilidad.

```
for (i in 1:length(et_error)){
  cat("Error al clasificar. Etiqueta = ", et_error[i], ". Probabilidad asignada: ", prob_error[i], "\n")
}
```

```
## Error al clasificar. Etiqueta =  0 . Probabilidad asignada:  0.5584608
```

Como podemos observar, únicamente hemos clasificado de forma incorrecta un punto, y su probabilidad según la función sigmoide es muy cercana a 0.5. Cuanto más nos acercamos a la línea que genera la regresión lineal y que separa nuestros datos, más nos introducimos en una región de indecisión, en esta parte clasificamos los datos con una etiqueta pero la seguridad que tenemos para esto es muy baja, y las probabilidades de que se puedan clasificar con una etiqueta u otra similares. Cuanto más lejos estemos de esta zona, más seguras serán nuestras respuestas y por tanto menos probabilidad de que el etiquetado sea incorrecto.

Por lo tanto, cuando estamos en zona de indecisión, deberíamos clasificar los datos pero indicar que no podemos realizar un etiquetado con mucha seguridad.

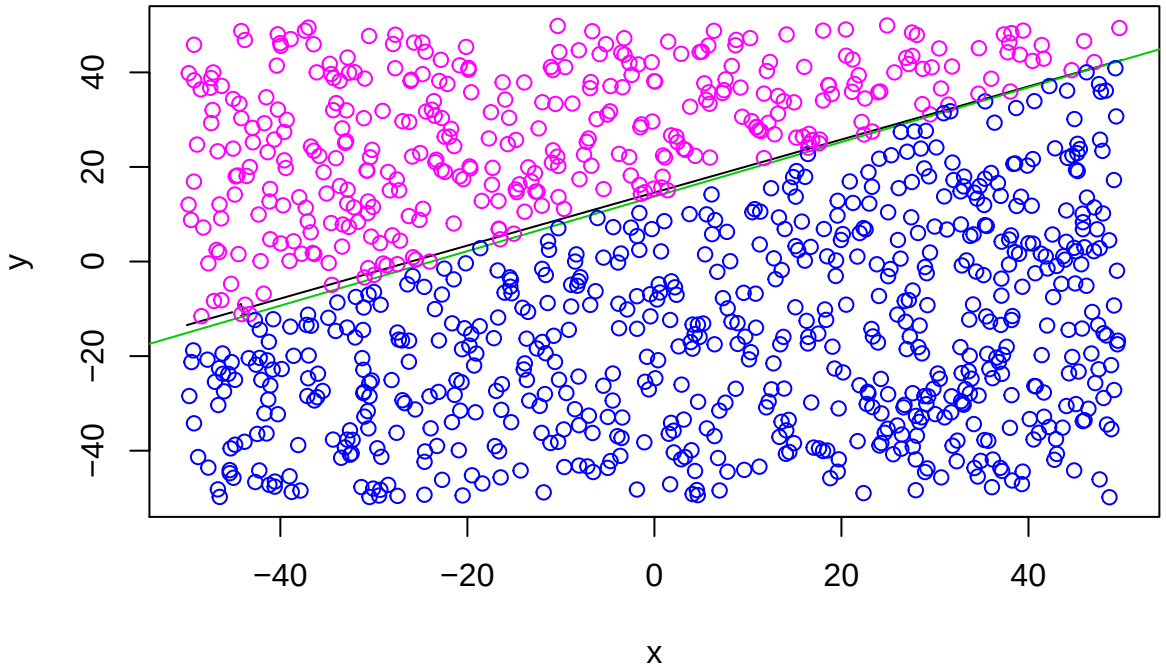
b) Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar Eout usando para ello un número suficientemente grande de nuevas muestras (>999)

Generamos un conjunto de muestras de tamaño 1000 junto con sus etiquetas:

```
datosTest = simula_unif(1000,2,c(-50,50))
etiquetasTest = clasifica(datosTest, f)
```

Pintamos los resultados

```
pintar_frontera(f)
abline(ab[2],ab[1], col = 3)

points(datosTest, col = sign(cbind(datosTest,1)%*%t(w))+5)
```



Calculamos el error:

```
e = error(datosTest,etiquetasTest,w)
```

El error en el test es de: 0.018539.

Hemos probado el modelo que habíamos generado anteriormente, con 1000 datos nuevos, y hemos obtenido un error *Eout* casi igual al que habíamos obtenido en *Ein*. Esto nos indica que nuestro modelo funciona igual de bien dentro y fuera de la muestra. Como hemos mencionado anteriormente, los errores que se han producido en este conjunto de datos han sido en puntos muy cercanos a la recta y por tanto con probabilidad de ser una etiqueta u otra muy similares.

3. BONUS

Clasificación de Dígitos.

1. Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

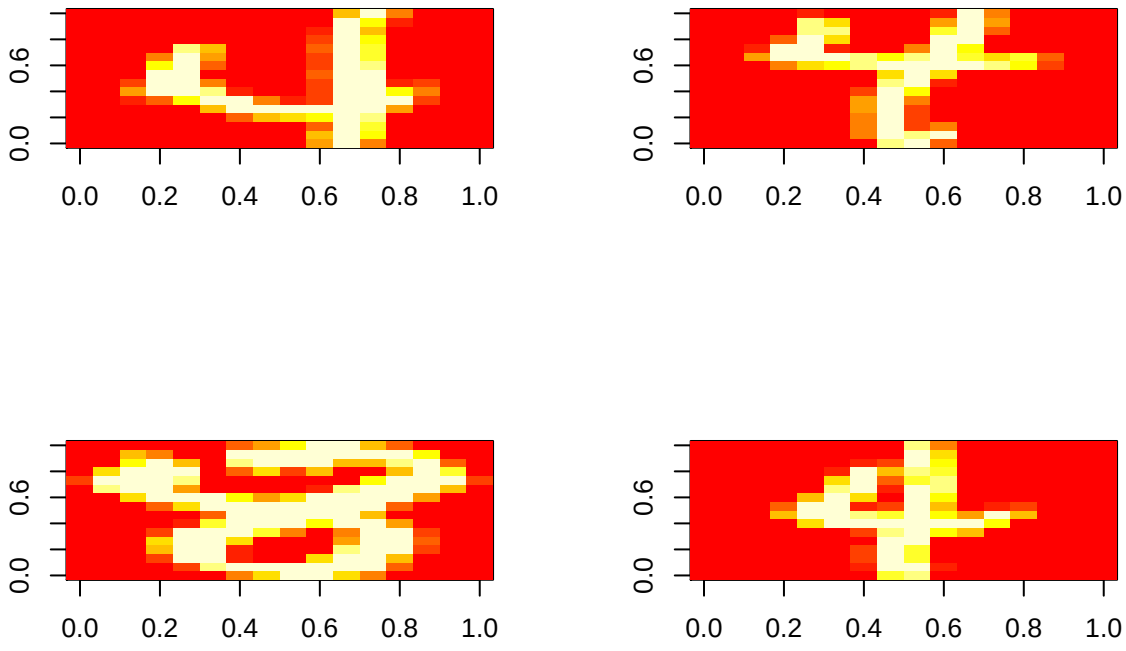
Para este ejemplo, consideraremos el conjunto de muestras de dígitos de la práctica anterior. Extraeremos las muestras correspondientes a los dígitos 4 y 8 respectivamente y procederemos extrayendo la intensidad promedio y la simetría.

Leemos el fichero zip.train y cargamos los digitos 4 y 8. Seguiremos los mismos pasos que en la practica anterior para obtener los datos.

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

```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

```
digitos15.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]
digitos = digitos15.train[,1]    # vector de etiquetas del train
ndigitosTr = nrow(digitos15.train)  # numero de muestras del train

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


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




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

```
## [1] 4 4 8 4
```

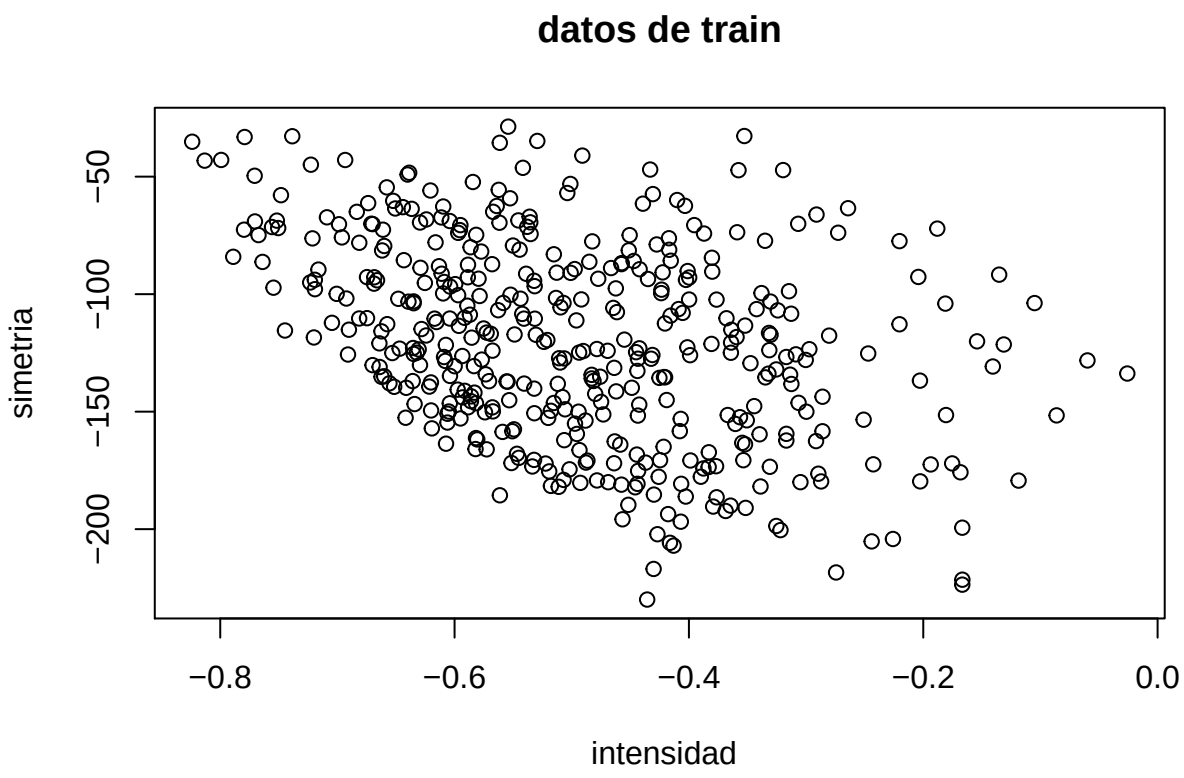
```r
par(mfrow = c(1,1))

rm(digit.train)
rm(digitos15.train)

intensidad = apply (grises,1,mean)

fsimetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -sum(A)
}
simetria = apply (grises,1,fsimetria)
```

Establecemos las etiquetas de la siguiente forma:

```r
etiquetasTr = digitos
etiquetasTr[etiquetasTr == 4 ] = -1
etiquetasTr[etiquetasTr == 8 ] = 1
datosTr = as.matrix(cbind(intensidad,simetria))
rm(grises)
```

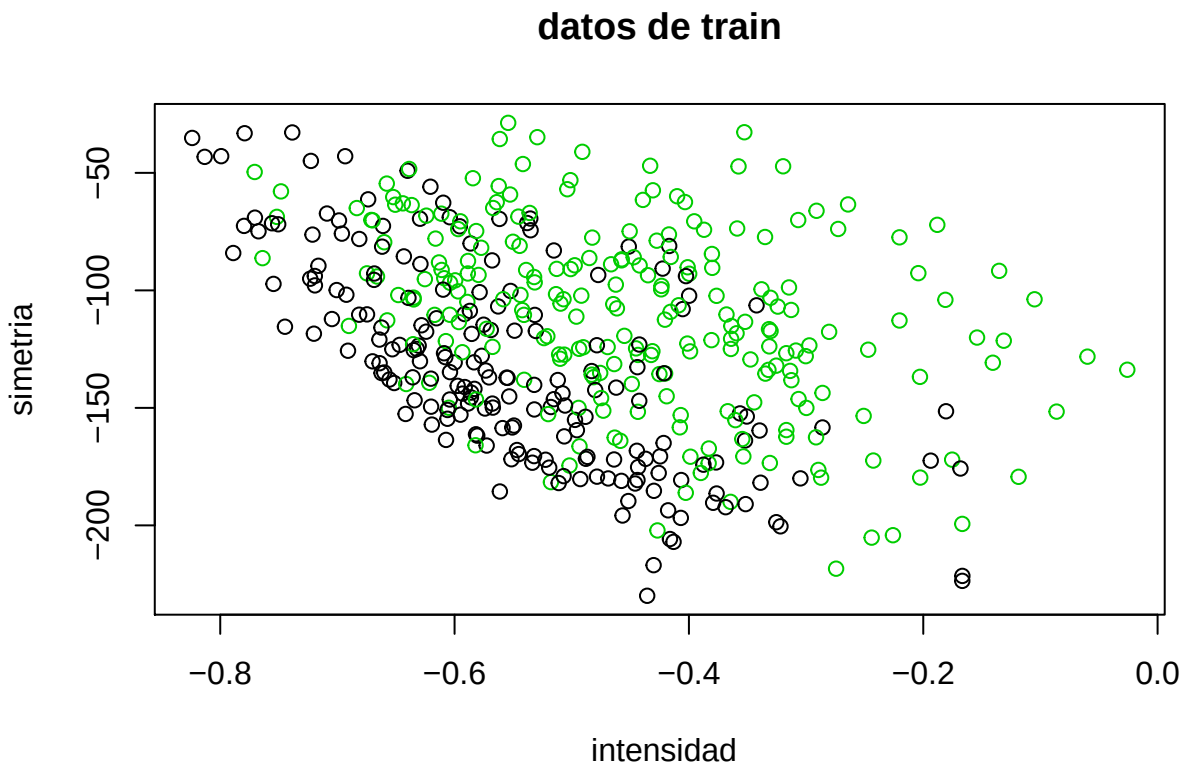
Relación entre las características

```r
plot(datosTr,xlab=" intensidad",ylab="simetria", main="datos de train")
```



Pintamos cada punto correspondiente a su clasificación.

```
plot(datosTr,xlab=" intensidad",ylab="simetria", main="datos de train", col = etiquetasTr+2)
```

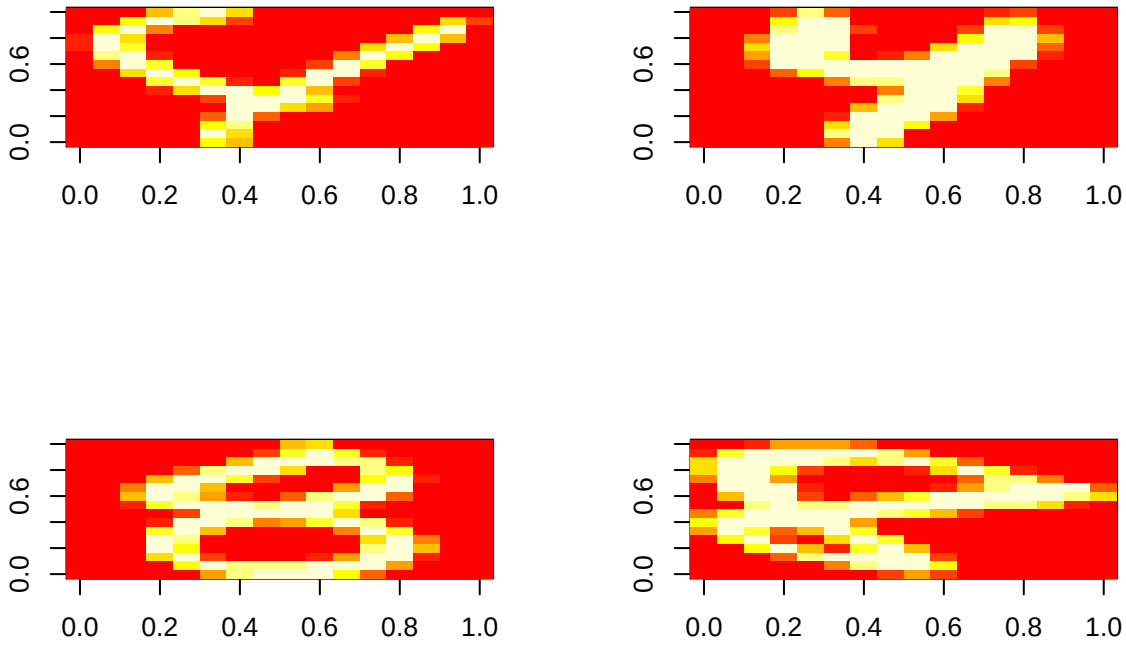


Realizamos ahora el mismo proceso para los datos del test.

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

```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

```
digitos15.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]
digitos = digitos15.test[,1]    # vector de etiquetas del test
ndigitosTst = nrow(digitos15.test)  # numero de muestras del test

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

rm(digit.test)
rm(digitos15.test)

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



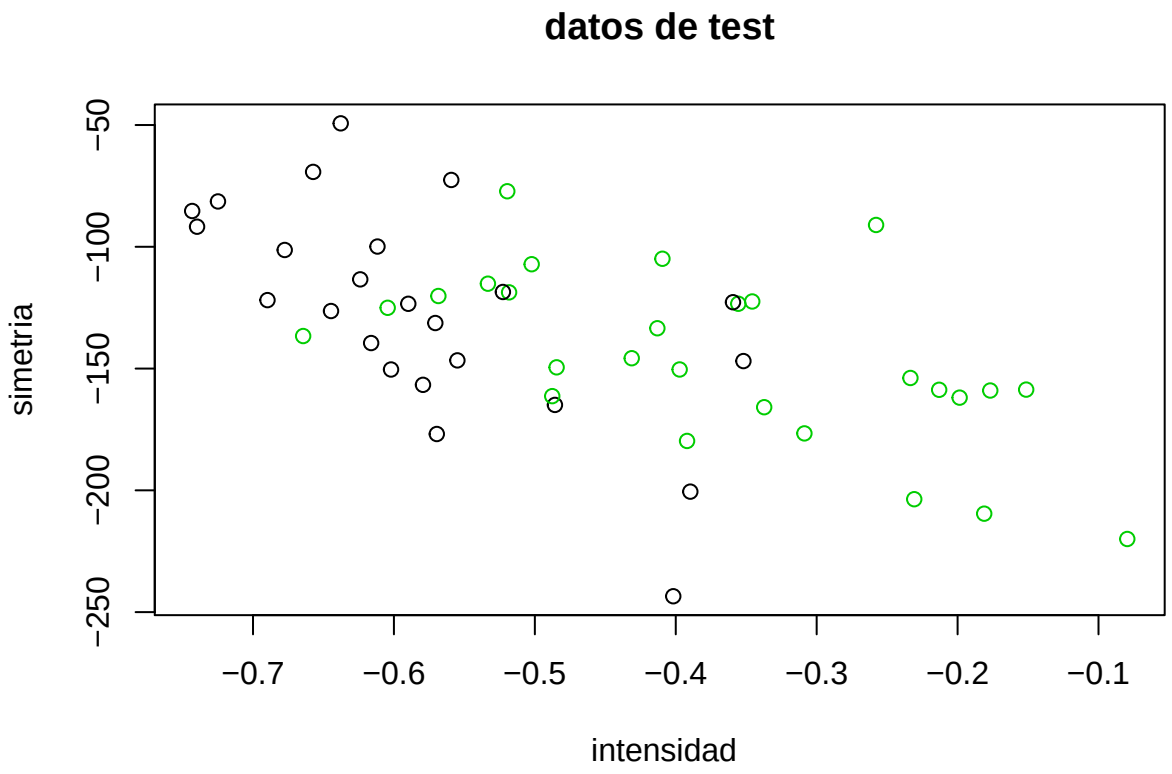

```
par(mfrow = c(1,1))

intensidad = apply (grises,1,mean)
simetria = apply (grises,1,fsimetria)
```

Obtenemos las etiquetas del Test y la matriz de datos.

```
etiquetasTst =digitos
etiquetasTst[etiquetasTst== 4 ] = -1
etiquetasTst[etiquetasTst== 8 ] = 1

datosTst = as.matrix(cbind(intensidad,simetria))
rm(grises)
```

Pintamos el test.

```
plot(datosTst,xlab=" intensidad",ylab="simetria", main="datos de test", col = etiquetasTst+2)
```



2. Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada. b) Calcular Ein y Etest (error sobre los datos de test). c) Obtener cotas sobre el verdadero valor de Eout. Pueden calcularse dos cotas, una basad en Ein y otra en Etest. Usar tolerancia de 0.05. ¿Qué cota es mejor?

Definimos la función de Regresión Lineal como en la práctica anterior. Esta la usaremos para sacar una primera aproximación a una solución buena.

```
Regress_Lin = function(datos,label){

  datos = cbind(datos,c(rep(1,length(label))))
  descom = svd(datos)
  D = descom$d
  V = descom$v
  U = descom$u

  pseudo = (V %*% diag(1/D) %*% t(U))
  w = pseudo %*% label
  t(w)
}
```

Definimos a su vez también funciones para el cálculo del error para saber como de bueno es nuestro modelo.

```
calculoE = function(w,datos, label){
  E = sum(sign( (datos%*%t(w)) ) != label)/length(label)
  E
}
```

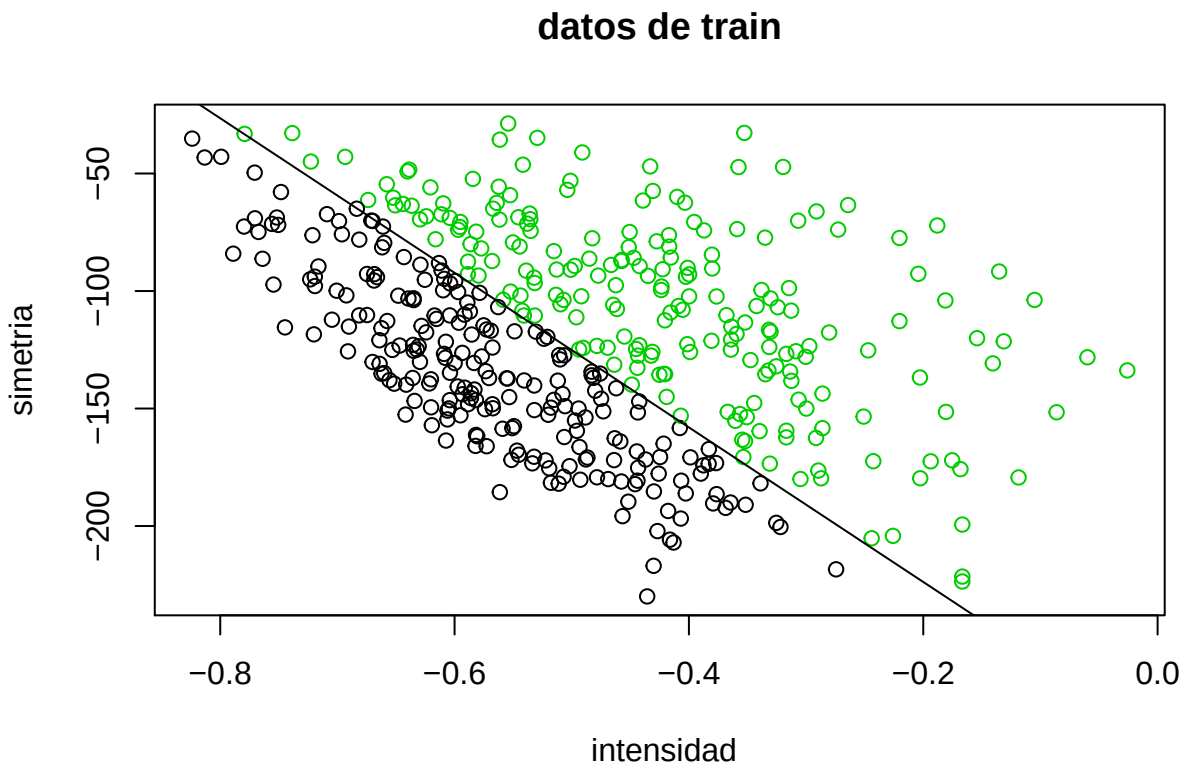
Calculamos el vector de pesos y los errores asociados.

```
w = Regress_Lin(datosTr,etiquetasTr)

Ein = calculoE(w,cbind(datosTr,1),etiquetasTr)
Eout = calculoE(w,cbind(datosTst,1),etiquetasTst)
```

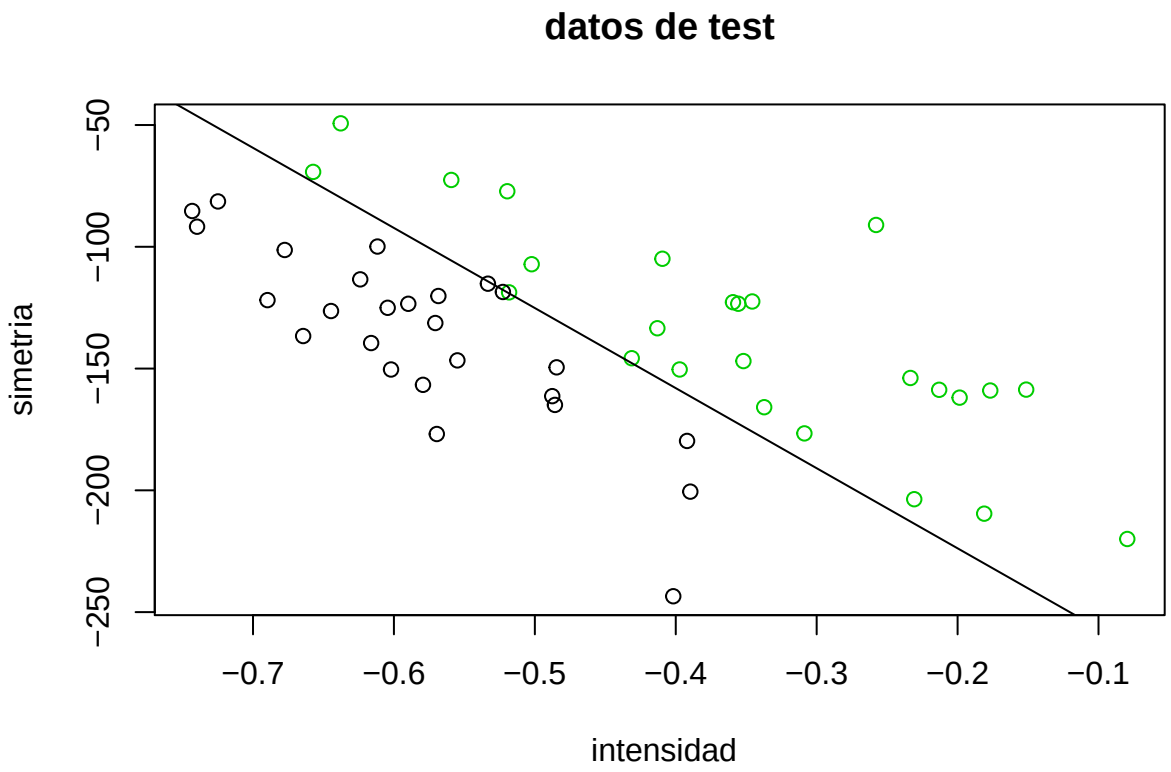
Hemos obtenido los siguientes errores:
- Ein: 0.2476852
- Eout: 0.2352941

Ahora visualizamos la clasificación de los datos en las gráficas

```
ab = pasoARecta(w)

plot(datosTr,xlab=" intensidad",ylab="simetria", main="datos de train", col = sign(cbind(datosTr,1)%*%t
abline(ab[2],ab[1])
```




```
plot(datosTst,xlab=" intensidad",ylab="simetria", main="datos de test", col = sign(cbind(datosTst,1)%*%
abline(ab[2],ab[1])
```



Como podemos ver el resultado no es demasiado bueno. Debemos pensar que los datos no eran fácil de clasificar y menos con un modelo lineal. Por tanto, el método de la pseudo inversa nos ofrece una solución que podemos considerar buena respecto a los datos que teníamos y realiza una división de forma general de forma rápida. Veremos si el PLA pocket mejora este problema.

PLA Pocket El algoritmo de Perceptrón-Pocket es un método de regresión lineal basado en Perceptrón, el cuál hemos implementado de la siguiente forma:

*** Parámetros**

- *datos*: Matriz de datos que corresponden al conjunto de entrenamiento o data train.
- *label*: Conjunto de etiquetas asociadas a la fila i-ésima de nuestro data train.
- *maxiter*: Número máximo de iteraciones del algoritmo.
- *vini*: Vector inicial de pesos.

*** Algoritmo**

- El algoritmo funciona de la misma forma que el Perceptrón original.
- Las únicas diferencias es que ahora almacenamos el vector de pesos que nos proporciona un menor error a la hora de clasificar la muestra.

*** Salida**

- El algoritmo propocionará un vector de pesos w.

```
PLA_Pocket = function(datos,label,max_iter, vini){
  datos = cbind(datos,1)

  w = vini
  wanterior = vini
  it = 0
  umbral = 10^-6
  igual = F
  w_mejor = vini
  error_mejor = calculoE(w,datos,label)
```

```
  while( (it < max_iter) & (!igual)){
    for(i in 1:nrow(datos)){
      if( sign(datos[i,] %*% t(w)) != label[i]){
        w = w + label[i]* datos[i,]
      }
    }
    error =calculoE(w,datos,label)
    if(error < error_mejor){
      error_mejor = error
      w_mejor = w
    }
    igual = all(abs(wanterior-w) < umbral)
    it = it+1
    wanterior = w

  }
  if(igual){
    cat("El perceptron para por que no hay diferencia entre los pesos anteriores, tras ")
    print(it)
  }
  else{
    print("El perceptron para por el limite de iteraciones")
  }

  w_mejor
}
```

Ahora visualizamos la clasificación de los datos en las gráficas

Calculamos el vector de pesos a PLA-Pocket y los errores asociados:

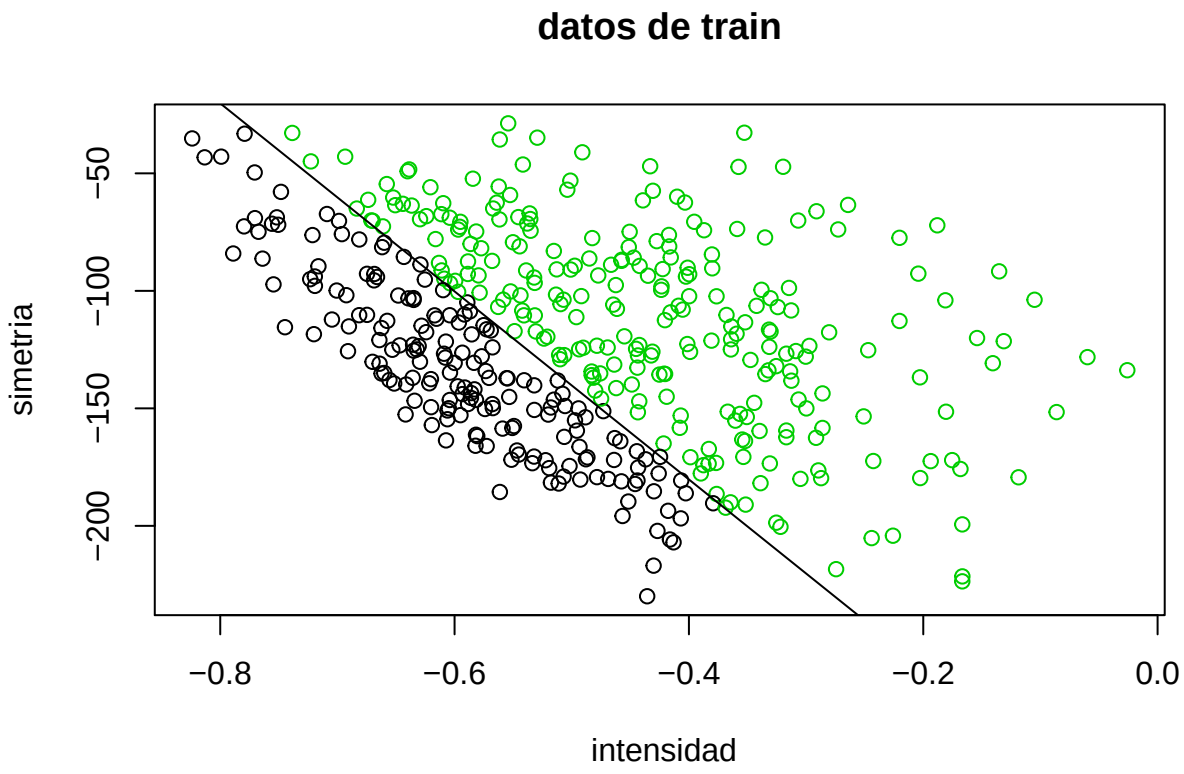
```
w = PLA_Pocket(datosTr, etiquetasTr, 4000,w )
```

```
## [1] "El perceptron para por el limite de iteraciones"
```

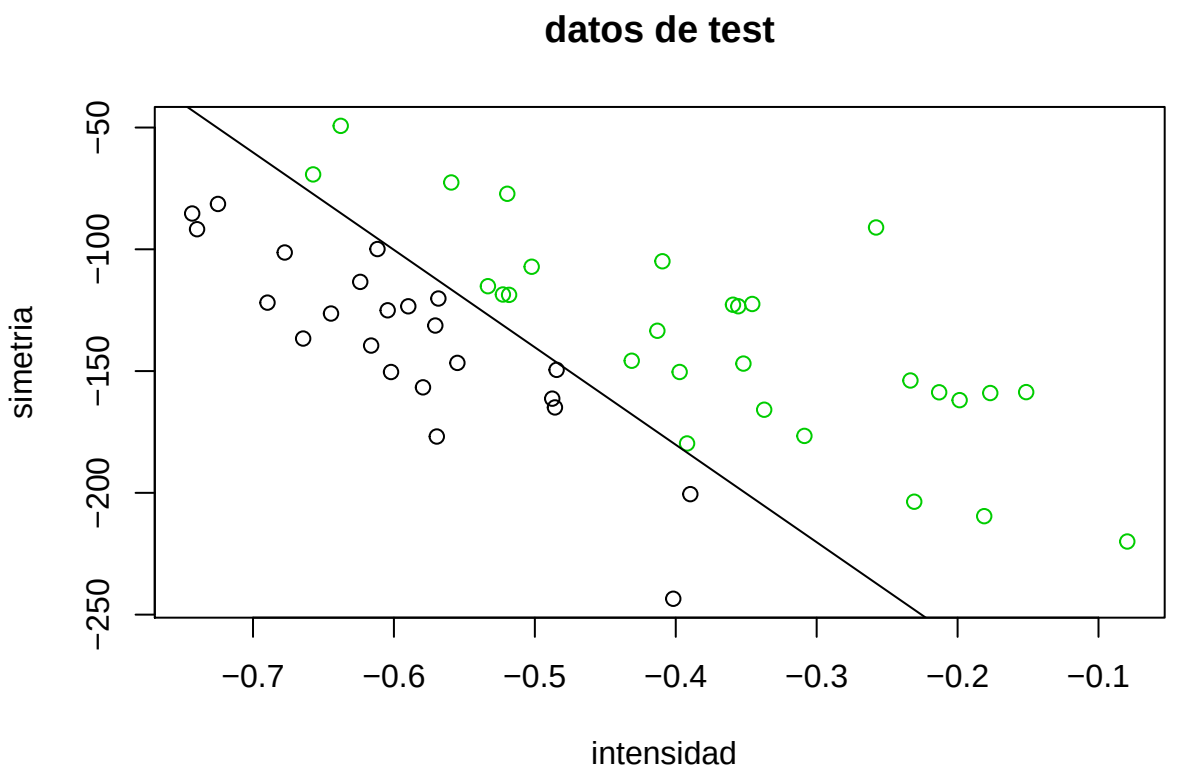
```
Ein = calculoE(w,cbind(datosTr,1),etiquetasTr)
Etest = calculoE(w,cbind(datosTst,1),etiquetasTst)
```

Ahora visualizamos los resultados

```
ab = pasoARecta(w)

plot(datosTr,xlab=" intensidad",ylab="simetria", main="datos de train", col = sign(cbind(datosTr,1)%*%t
abline(ab[2],ab[1])
```




```
plot(datosTst,xlab=" intensidad",ylab="simetria", main="datos de test", col = sign(cbind(datosTst,1)%*%
abline(ab[2],ab[1])
```



Hemos obtenido los siguientes errores:
- Ein: 0.2268519

- Etest: 0.2156863

En este caso le hemos pasado como vector inicial al PLA-Pocket el vector que obteniamos del método de la pseudo inversa. De este modo el perceptron comienza desde una buena solución. En este caso la variante pocket lo que hace es almacenar por cada iteración la mejor solución encontrada de modo que finalmente si para por el número de iteraciones, puede devolver la mejor solución y no la que en ese momento tenga como nos pasaba en el ejercicio 2. Para este ejemplo hemos establecido el límite de iteraciones a 4000, porque primeramente con esos datos es imposible que converga a una solución.
Por otro lado, hicimos pruebas con menos iteraciones y no conseguía encontrar ninguna solución mejor a la que nos proporcionaba Regress_Lin. Es realmente difícil dividir correctamente con un modelo lineal los datos que teníamos. Por eso tras varías pruebas vimos que conseguía una pequeña mejora con 4000. Llegamos a probar con 10000 iteraciones y no encontro ninguna mejora.

Cotas Para el último apartado, debemos calcular las cotas de Eout basándonos en la dimensión VC (Vapnik-Chervonenkis dimension).
Definimos esta función para calcular la cota:

```
calcula_cota = function(e, ndigitos, dim, tc){
  e + sqrt((8/ndigitos)*log((4*((2*ndigitos)^dim +1 )) /tc))
}
```

Calculamos las cotas de error que obtendremos:

```
EoutTrain = calcula_cota(Ein,ndigitosTr,3,0.5 )
EoutTest = calcula_cota(Etest,ndigitosTst,3,0.5 )
```

Hemos obtenido los siguientes errores:
- Eout Train: 0.8703976
- EoutTest: 1.7976613

Como podemos ver son cotas de error muy altas ya que tenemos varios factores que influyen en estas cotas. En primer lugar estamos intentado calcular con un alto grado de confianza, 95%. En segundo lugar podemos ver que ambas cotas son muy diferentes y la gran diferencia entre ambos conjuntos es el tamaño. La cota para el train es bastante mas pequeña debido a que tenemos un conjunto de datos mayor y por tanto puede aproximar de forma más exacta el error. En el caso del Test no tenemos muchos ejemplos, por ello la cota se nos hace casi inservible.

En definitiva podemos decir que estas cotas no nos aportan nada de forma práctica ya que son muy amplias, pero podemos extraer que el factor que influye para asegurarnos que fuera de la muestra tendremos un buen resultado, es el tamaño de la muestra, ya que la dimensión VC será la misma. Si conseguimos aumentar el número de muestras, podremos asegurar que nuestro modelo no tendrá un Eout muy diferente al Ein que hemos obtenido dentro del conjunto.