Práctica 1

Juan José Sierra González

16 de marzo de 2017

Funciones aportadas para el Ejercicio 1

```
## ------------------------------------------------------------------------
# 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 especifica
  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
}

# Para el apartado 3 del Ejercicio 1
#-------------------------------------------------------------------------------
## 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 = 400)
   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")
}
```

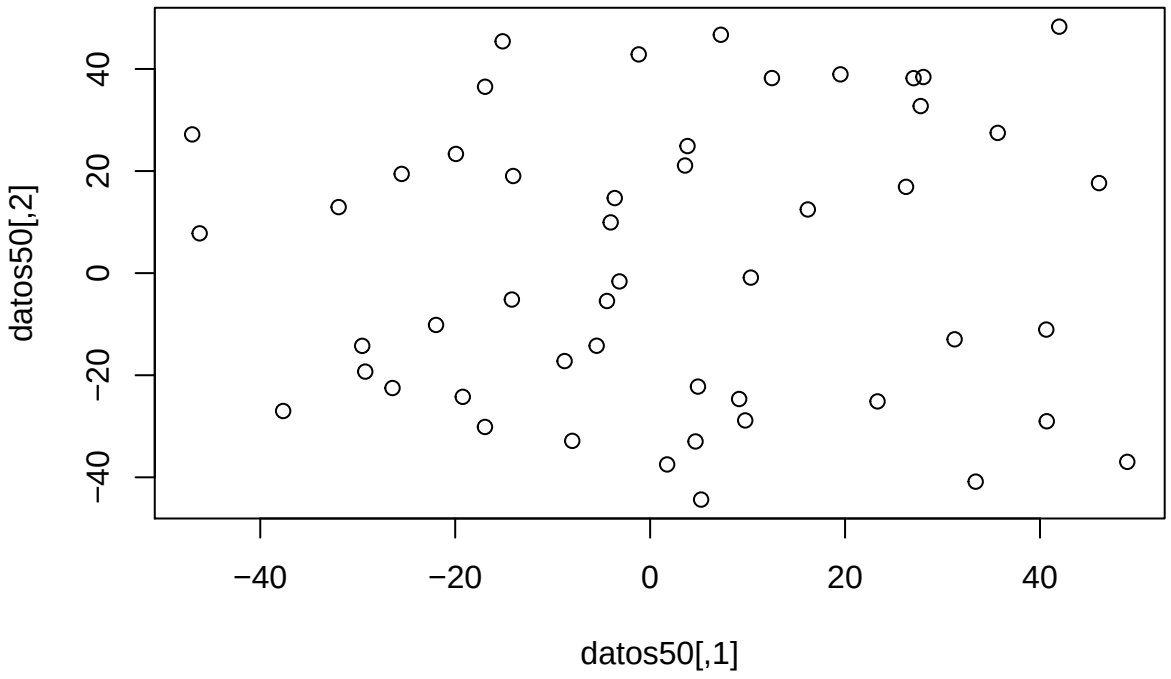
Ejercicio 1

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

Este ejercicio únicamente consistirá en llamar a dos funciones que nos son aportadas para la práctica, y cuyos datos generados pintamos en una gráfica.

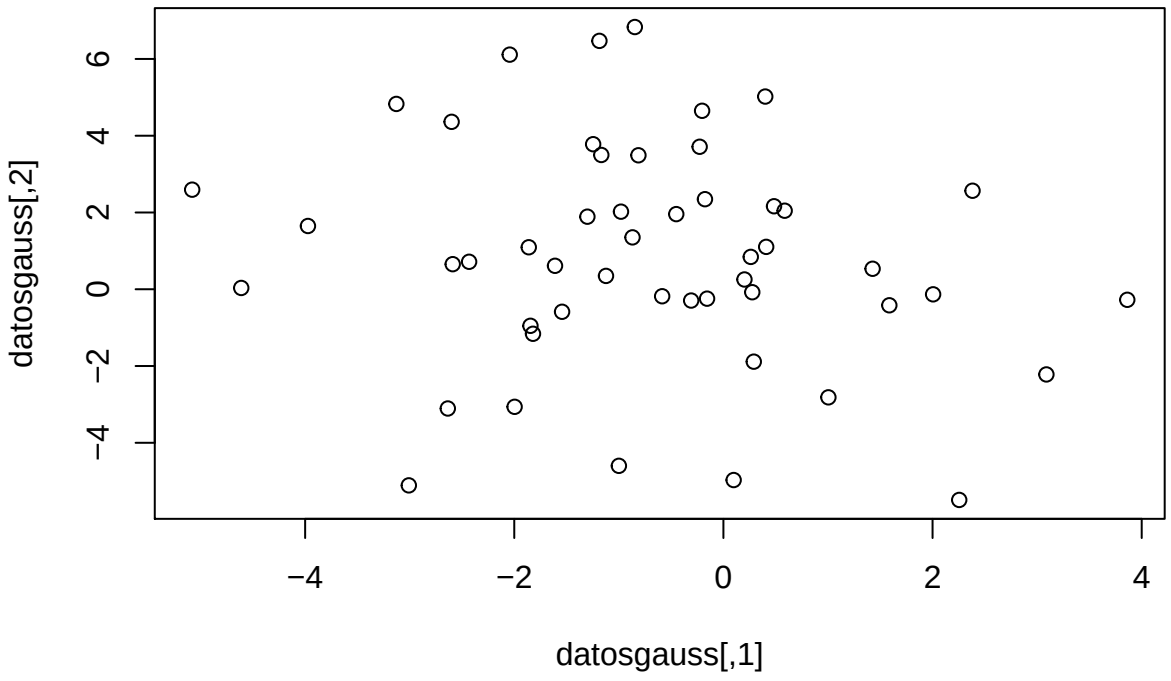
a) Considere N = 50, dim = 2, rango = [-50, +50] con simula_unif (N, dim, rango).

```
set.seed(100)
datos50 = simula_unif(N=50, dim=2, rango=c(-50,50))
plot(datos50)
```



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

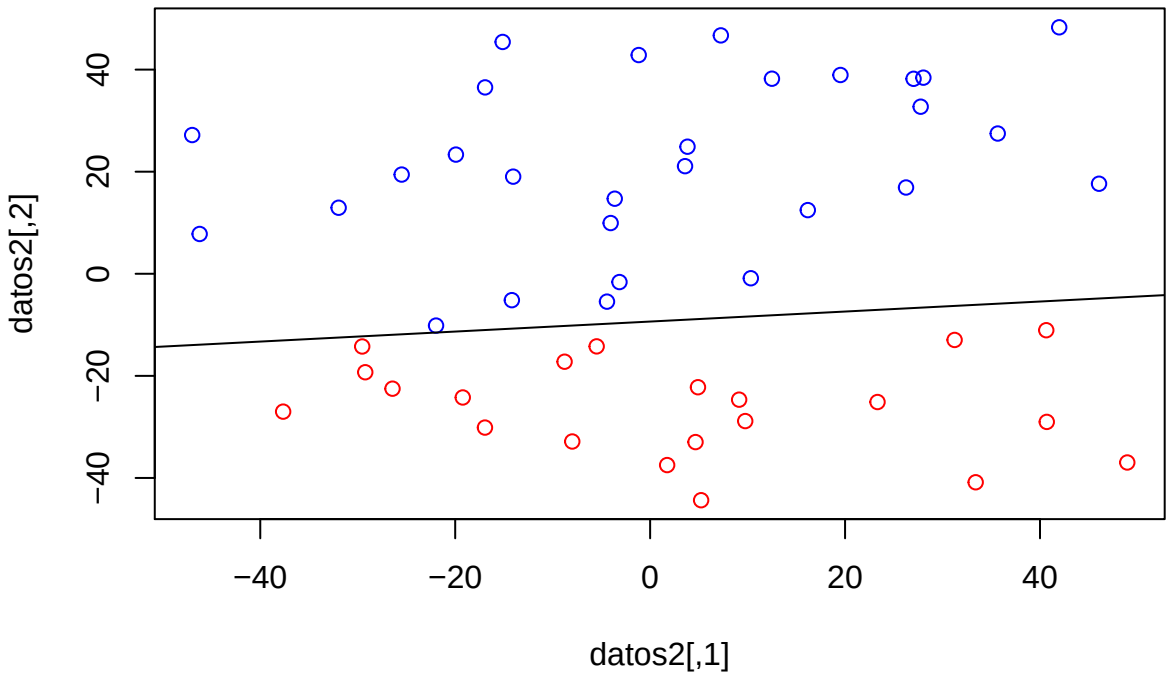
```
set.seed(100)
datosgauss = simula_gaus(N=50, dim=2, sigma=c(5,7))
plot(datosgauss)
```



1.2 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)

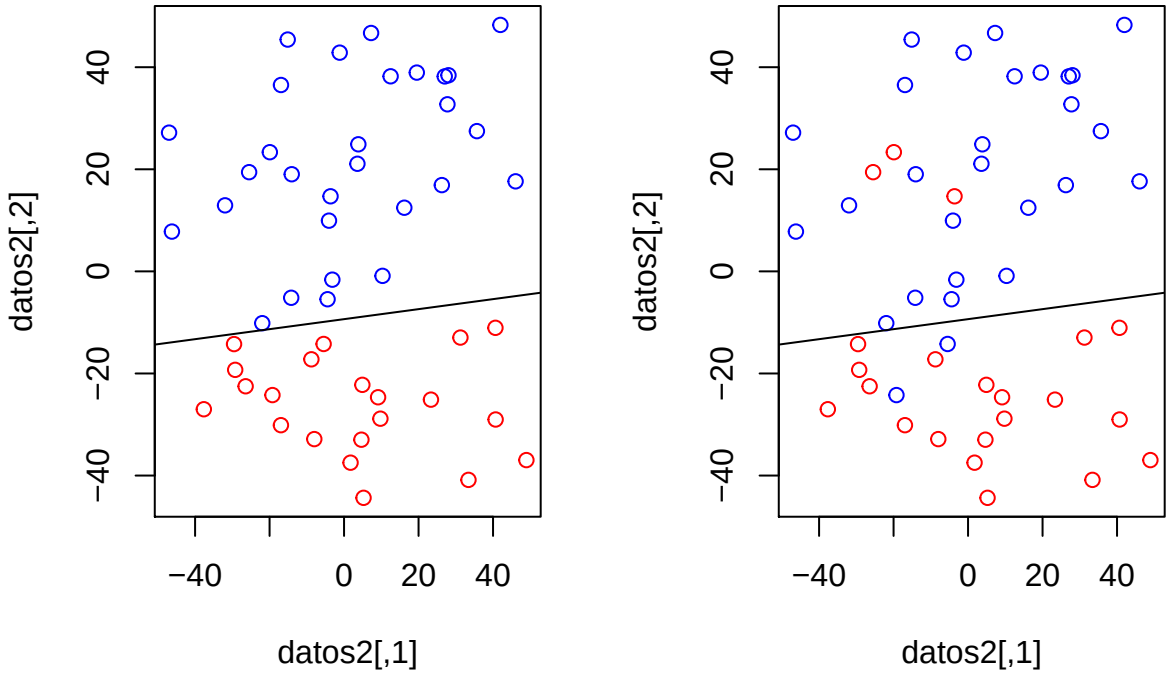
```
# Asigna etiquetas a unos puntos en relación a si están por encima o por debajo de una recta,
# es decir, si la y para ese punto es mayor o menor que la y de la recta en la x del punto.
asignarEtiquetasSegunRecta = function(recta,puntos){
    etiquetas = sign(puntos[,2] - (recta[1]*puntos[,1]+recta[2]))
}

set.seed(100)
datos2 = simula_unif(50, 2, c(-50,50))
recta = simula_recta(c(-50,50))

etiquetas2 = asignarEtiquetasSegunRecta(recta, datos2)
plot(datos2, col=etiquetas2+3)
abline(recta[2], recta[1])
```



Una vez que hemos asignado las etiquetas basándonos en si la componente Y del punto es mayor que la componente Y de la recta para el valor X del punto, podemos pintar los puntos dándole el color correspondiente a sus etiquetas. Además, gracias a la función abline podemos dibujar la recta que nos ha servido para delimitarlas sobre la gráfica y comprobamos que efectivamente están separadas correctamente.

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 que asigna un ruido del porcentaje dado a las etiquetas pasadas como argumento, de forma
# que algunas queden desclasificadas con respecto a la distribución que seguían inicialmente.
# Primero obtiene las posiciones de valores positivos y negativos. Sobre ellas, obtiene el porcentaje
# de valores de cada tipo a modificar y los extrae con un sample. Por último, esos valores positivos
# se cambian a -1 y viceversa.
asignarRuido = function(etiquetas, porcentaje=10){
  positivos = which(etiquetas==1)
  negativos = which(etiquetas==-1)
  cambiarPositivos = sample(positivos, round(length(positivos)*0.01*porcentaje))
  cambiarNegativos = sample(negativos, round(length(negativos)*0.01*porcentaje))
  etiquetas[cambiarPositivos]=-1
  etiquetas[cambiarNegativos]=1
  etiquetas
}

etiquetasRuido = asignarRuido(etiquetas2, 10)
par(mfrow=c(1,2))
plot(datos2, col=etiquetas2+3)
abline(recta[2], recta[1])
plot(datos2, col=etiquetasRuido+3)
abline(recta[2], recta[1])
```



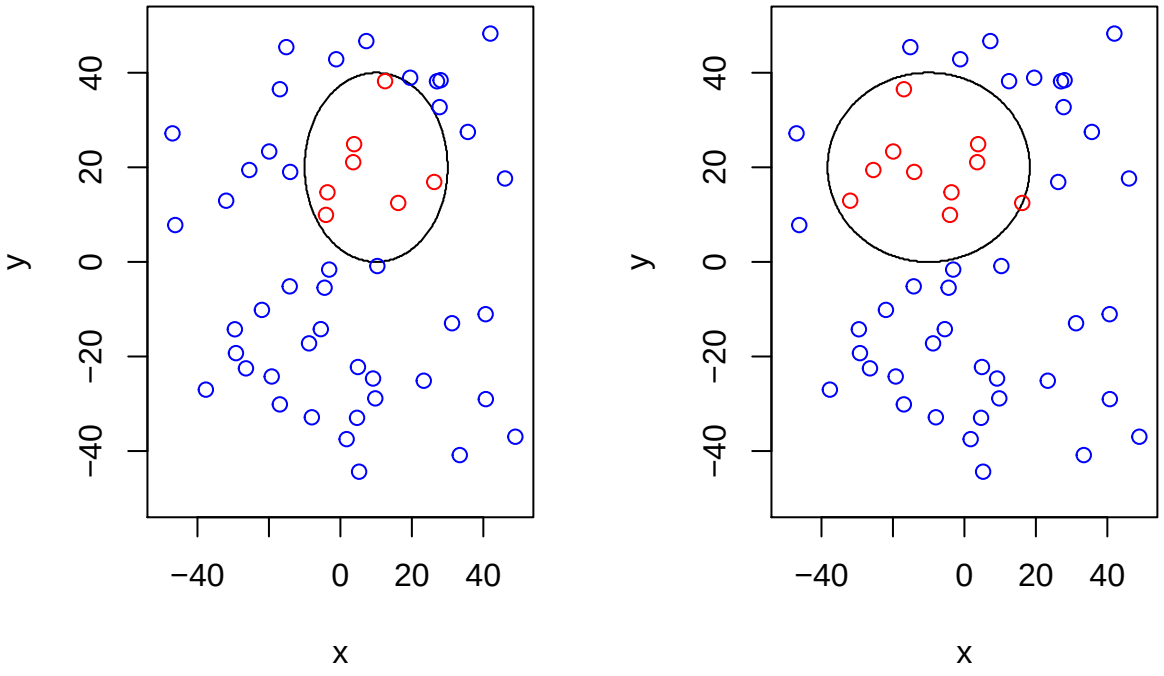
Aquí podemos observar una comparativa de las etiquetas correctas junto a las etiquetas con ruido. Vemos que son casi iguales pero algunas están cambiadas, y si hiciéramos las cuentas comprobaríamos que hay un 10% de etiquetas que eran rojas y ahora son de color azul, y viceversa.

1.3 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.

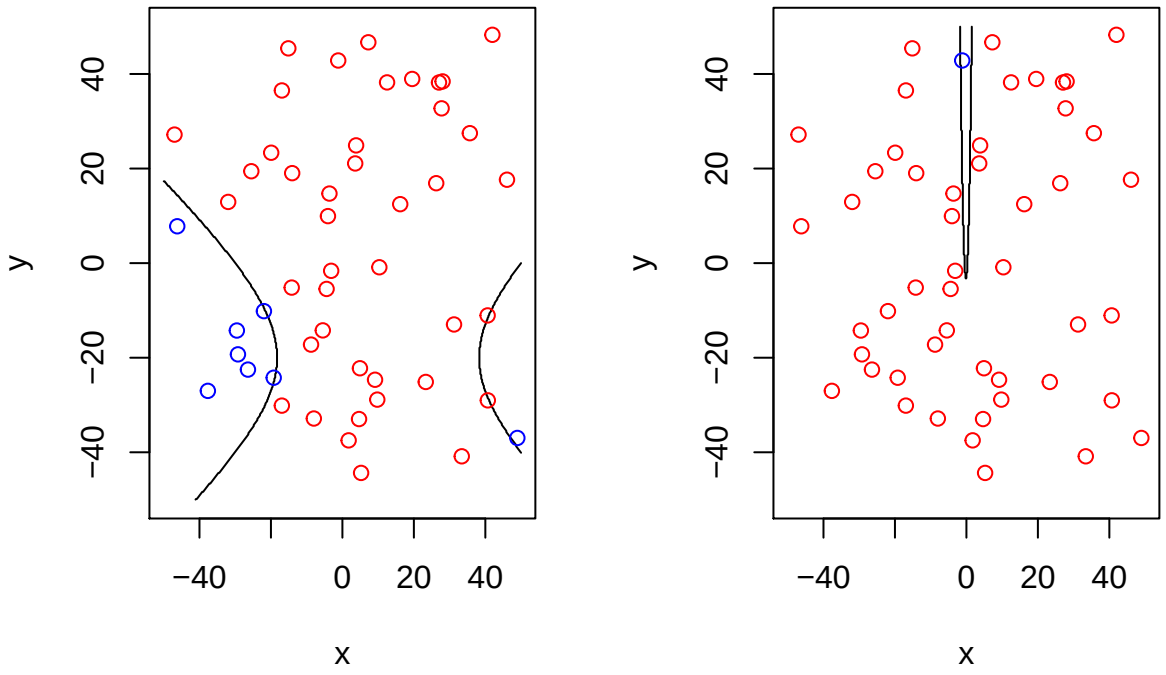
```
f1 = function(x,y){etiquetas = sign((x-10)^2 + (y-20)^2 - 400)}
f2 = function(x,y){etiquetas = sign(0.5*(x+10)^2 + (y-20)^2 - 400)}
f3 = function(x,y){etiquetas = sign(0.5*(x-10)^2 - (y+20)^2 - 400)}
f4 = function(x,y){etiquetas = sign(y - 20*x^2 - 5*x + 3)}

etiquetasf1 = f1(datos2[,1], datos2[,2])
etiquetasf2 = f2(datos2[,1], datos2[,2])
etiquetasf3 = f3(datos2[,1], datos2[,2])
etiquetasf4 = f4(datos2[,1], datos2[,2])
```

Muestro en primer lugar las etiquetas asignadas a cada función para los puntos creados en el apartado anterior, en sus respectivas gráficas.

```
par(mfrow=c(1,2))
pintar_frontera(f1)
points(datos2[,1], datos2[,2], col=etiquetasf1+3)
pintar_frontera(f2)
points(datos2[,1], datos2[,2], col=etiquetasf2+3)
```

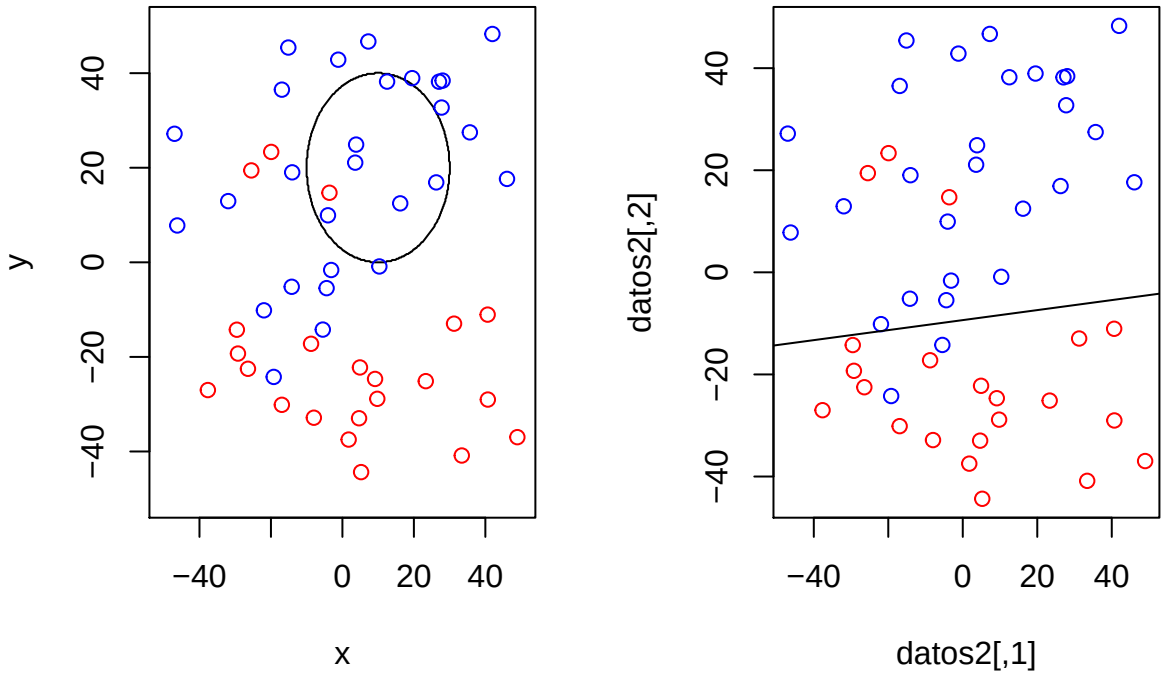



```
pintar_frontera(f3)
points(datos2[,1], datos2[,2], col=etiquetasf3+3)
pintar_frontera(f4)
points(datos2[,1], datos2[,2], col=etiquetasf4+3)
```



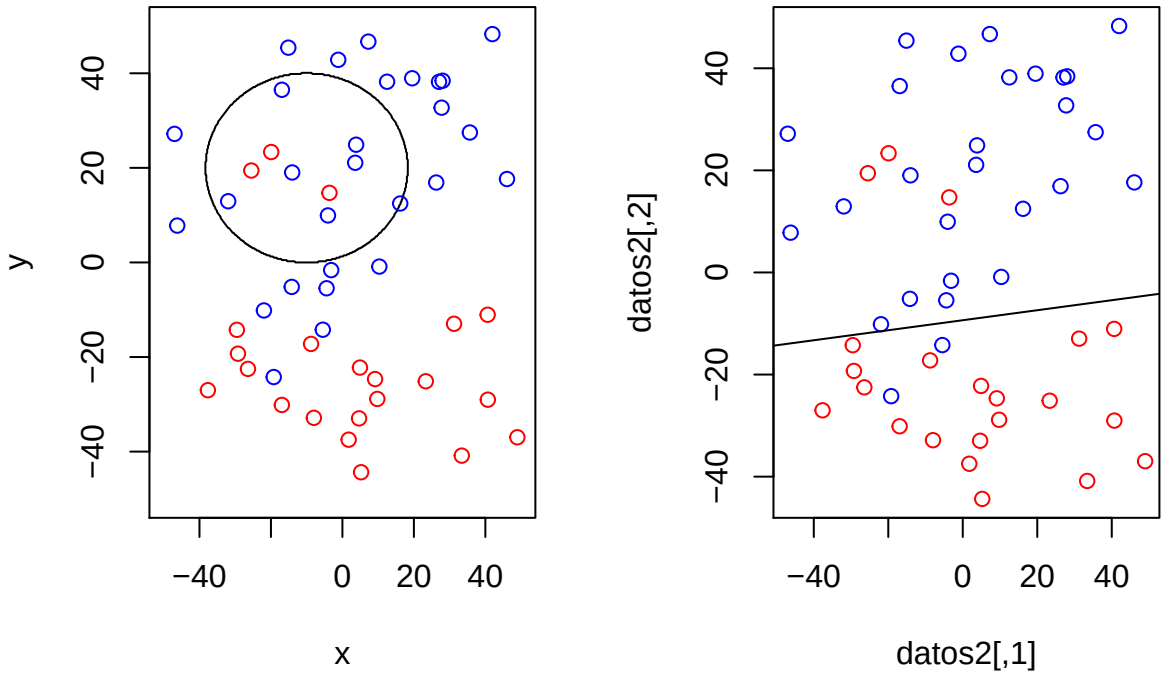
A continuación, se mostrarán las etiquetas con ruido y con la frontera de cada una de las funciones. Al lado situo la gráfica que pinta la recta anterior junto a las mismas etiquetas, para hacer la comparación.

```
par(mfrow=c(1,2))
pintar_frontera(f1)
points(datos2[,1], datos2[,2], col=etiquetasRuido+3)
plot(datos2, col=etiquetasRuido+3)
abline(recta[2], recta[1])
```

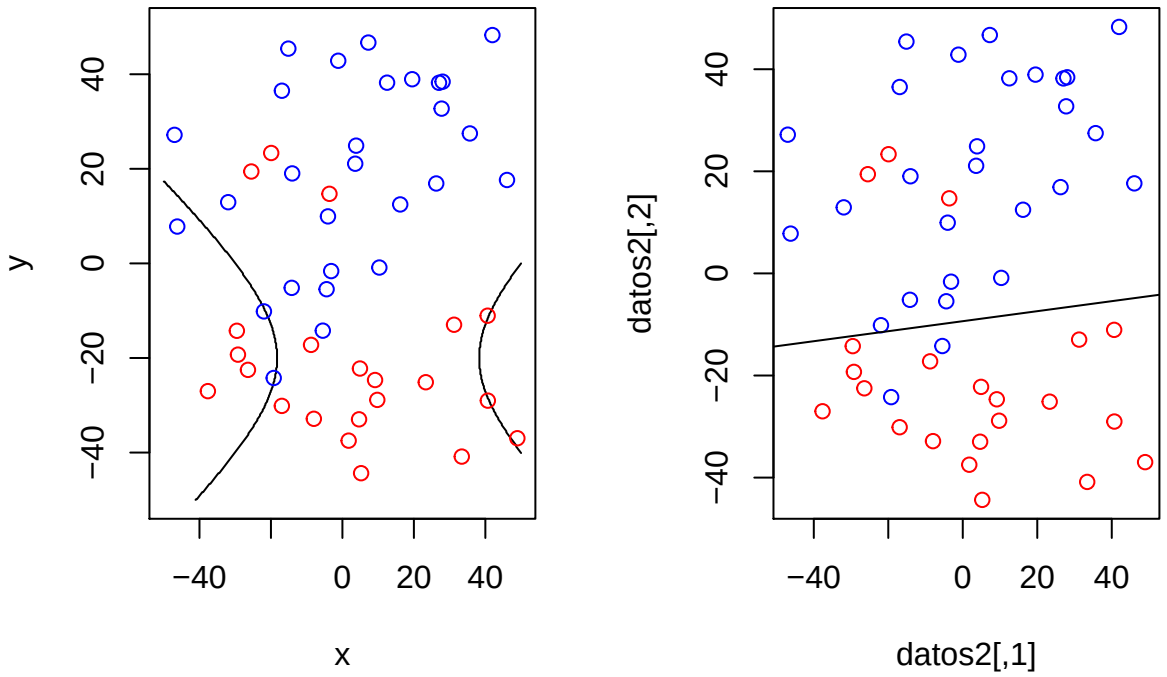



```
pintar_frontera(f2)
points(datos2[,1], datos2[,2], col=etiquetasRuido+3)
plot(datos2, col=etiquetasRuido+3)
```

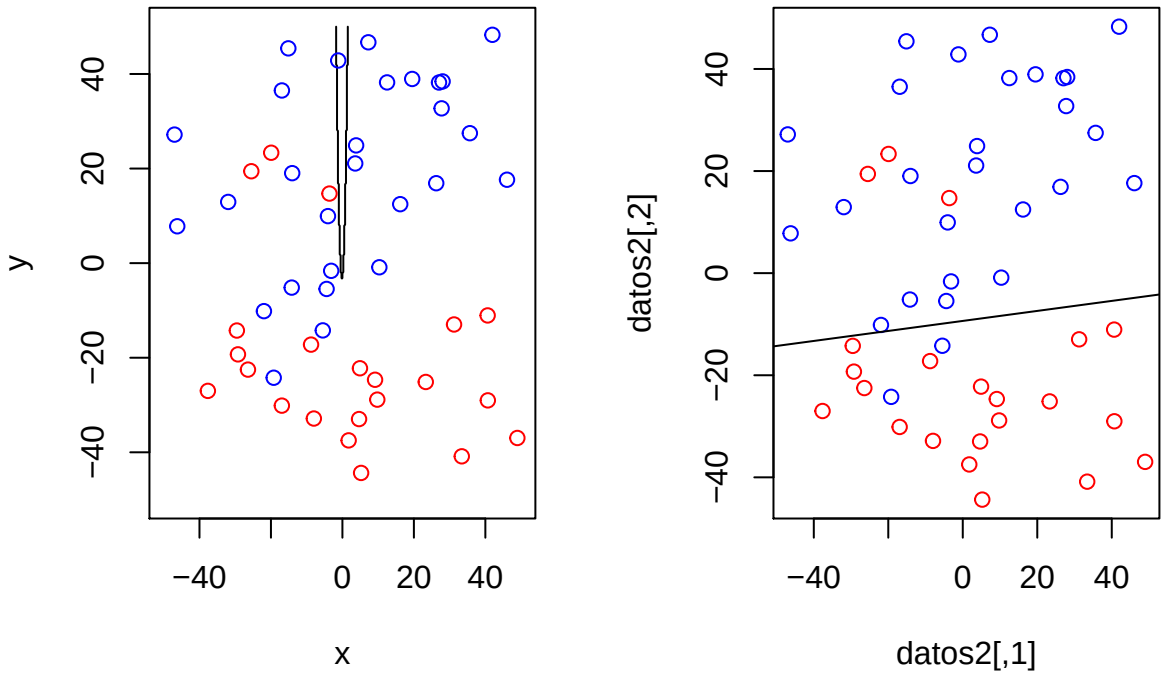
```
abline(recta[2], recta[1])
```




```
pintar_frontera(f3)
points(datos2[,1], datos2[,2], col=etiquetasRuido+3)
plot(datos2, col=etiquetasRuido+3)
abline(recta[2], recta[1])
```




```
pintar_frontera(f4)
points(datos2[,1], datos2[,2], col=etiquetasRuido+3)
plot(datos2, col=etiquetasRuido+3)
abline(recta[2], recta[1])
```



Como podemos observar, a simple vista ninguna de las funciones más complejas con las que hemos trabajado en este apartado ofrecen mejores soluciones que la recta. Esto es porque las etiquetas que han sido generadas con una función lineal no van a poderse separar de mejor manera que con una recta.

Ejercicio 2

2.1 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.

```
# Función que ajusta el algoritmo Perceptron para unos datos y unas etiquetas pasadas
# como argumento, hasta un máximo de iteraciones también dado, y con un vector de pesos
# inicial como último argumento.
# Su funcionamiento es el siguiente:
# En primer lugar, sabiendo que tendremos que hacer un producto vectorial entre el vector
# de pesos y el vector de características, como el vector de pesos tiene 3 valores
# (un peso para cada característica, y el umbral), hay que añadir una columna de 1 a los
# datos para que pueda hacerse este producto. El algoritmo itera hasta que no se haya
# llegado al máximo de iteraciones, o no se haya cambiado el vector de pesos en una
# iteración. Cada iteración comprueba los vectores de características en un orden diferente,
# esto lo he conseguido con un sample de su longitud. Cambia los pesos si alguna etiqueta
# está situada incorrectamente, según los cálculos del conocido algoritmo.
# En mi implementación del algoritmo, si encuentra un punto en mala posición no sale del
# bucle for en el que está iterando, llega hasta el final evaluando los valores que le quedan.
# Por último, devuelve una lista con el vector de pesos final y el número de iteraciones
# que ha necesitado para converger.
```

```
ajusta_PLA = function(datos, label, max_iter, vini){
  w = vini
  cambio = T
  datos = cbind(datos,1)
  iteraciones = 0

  while(iteraciones < max_iter & cambio){
    cambio = F
    for(j in sample(1:dim(datos)[1])){
      if (sign(crossprod(datos[j,],w)) != label[j]){
        w = w + datos[j,]*label[j]
        cambio = T
      }
    }

    iteraciones = iteraciones+1
  }

  list(w=w, iter=iteraciones)
}

# Función que obtiene la pendiente y el punto de corte del hiperplano de un vector
# de pesos. Esta ecuación se puede despejar de w1x1 + w2x2 + umbral = 0,
# siendo xn la característica n de nuestro vector de características, y siendo
# w1, w2 y umbral las componentes 1, 2 y 3 del vector de pesos, respectivamente.
obtenerRectaPesos = function(pesos){
  c(-pesos[1]/pesos[2], -pesos[3]/pesos[2])
}
```

2.2 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.

a)

```
# Función que hace lo pedido en el ejercicio 2.2 a), a fin de poder llamarla con replicate
ejercicio22a = function(datos, etiquetas, max_iter){
  modelo = ajusta_PLA(datos, etiquetas, max_iter, c(0,0,0))
  modelo$iter
}

set.seed(100)
resultado = replicate(10, ejercicio22a(datos2, etiquetas2, 200))
mean(resultado)
```

```
## [1] 62.4
```

El objetivo de haber metido las operaciones requeridas en esta función ha sido poder extraer las iteraciones de la lista devuelta por mi algoritmo Perceptron y que automáticamente se incluyan en un vector como resultado del replicate. Como se puede observar, el vector inicial de pesos es (0,0,0). La media de iteraciones

obtenidas en estos 10 experimentos ha sido 62.4. Comprobemos el valor para el apartado b.

b)

```
# Función que hace lo pedido en el ejercicio 2.2 a), a fin de poder llamarla con replicate.
# Genera los valores del vector inicial con la función runif entre 0 y 1.
ejercicio22b = function(datos, etiquetas, max_iter){
  vectorIni = runif(3, 0, 1)
  modelo = ajusta_PLA(datos, etiquetas, max_iter, vectorIni)
  modelo$iter
}

set.seed(100)
resultado = replicate(10, ejercicio22b(datos2, etiquetas2, 200))
mean(resultado)
```

```
## [1] 62.7
```

La función de este apartado es similar a la anterior pero introduce un factor de aleatoriedad con el vector de pesos inicial, ya que se generan 3 valores entre 0 y 1 para cada ejecución de la función. Reunimos las iteraciones necesarias para converger y en este caso vemos que asciende a 62.7.

En este caso no podemos sacar una conclusión en claro sobre si una opción es mejor que otra, ya que son valores muy próximos, y al ser datos linealmente separables tarde o temprano el Perceptron va a acabar por encontrar una recta que cumpla el cometido. Por tanto, lo único que podemos esperar interpretar de este experimento es que no importa el valor inicial que tenga el vector de pesos, ya que el número de iteraciones no parece verse afectado.

2.3 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.

```
resultado2a = replicate(10, ejercicio22a(datos2, etiquetasRuido, 200))
mean(resultado2a)
```

```
## [1] 200
```

```
resultado2b = replicate(10, ejercicio22b(datos2, etiquetasRuido, 200))
mean(resultado2b)
```

```
## [1] 200
```

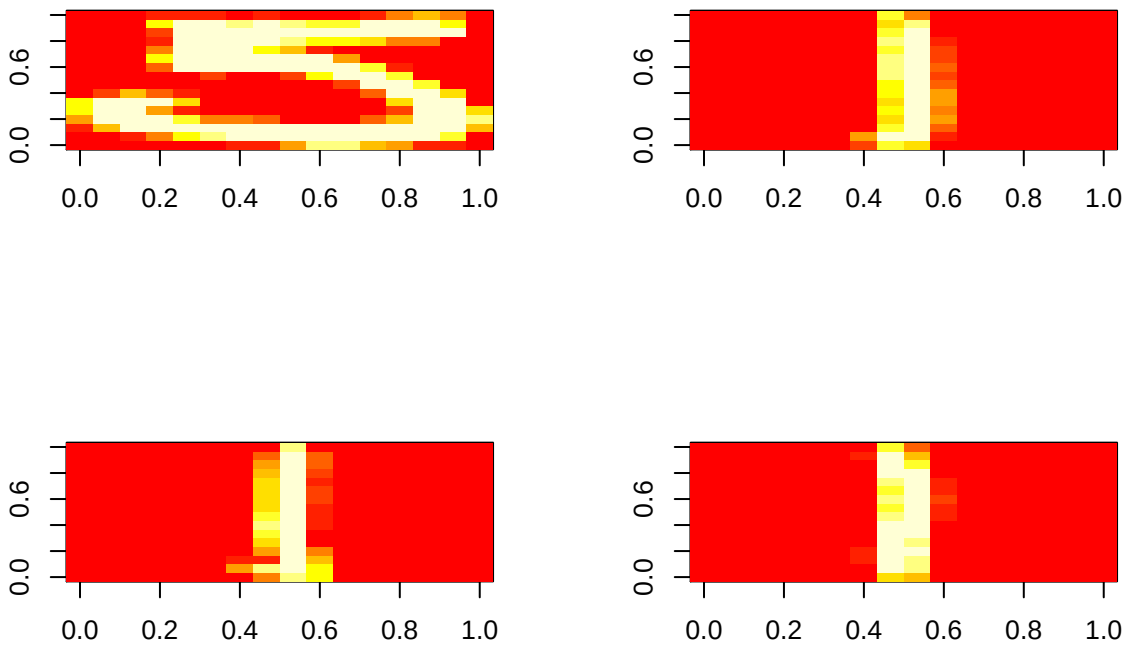
Como es de esperar, en este caso ambos terminan cuando el número de iteraciones llega al máximo. Al utilizar las etiquetas con ruido, nos queda un conjunto que no es separable linealmente, es decir, en cada pasada el Perceptron encontrará al menos un dato que estará mal situado. Esto implica que nunca tendrá una pasada sin modificar los pesos, y que por tanto iterará hasta el valor máximo que se le ha dado, por eso todos los experimentos en ambos casos convergen en 200.

Ejercicio 3

3.1 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.

Esto se consigue utilizando el código que se nos facilita para la práctica y que incluyo a continuación.

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

digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]
digitos = digitos15.train[,1]  # etiquetas
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)
}
```



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

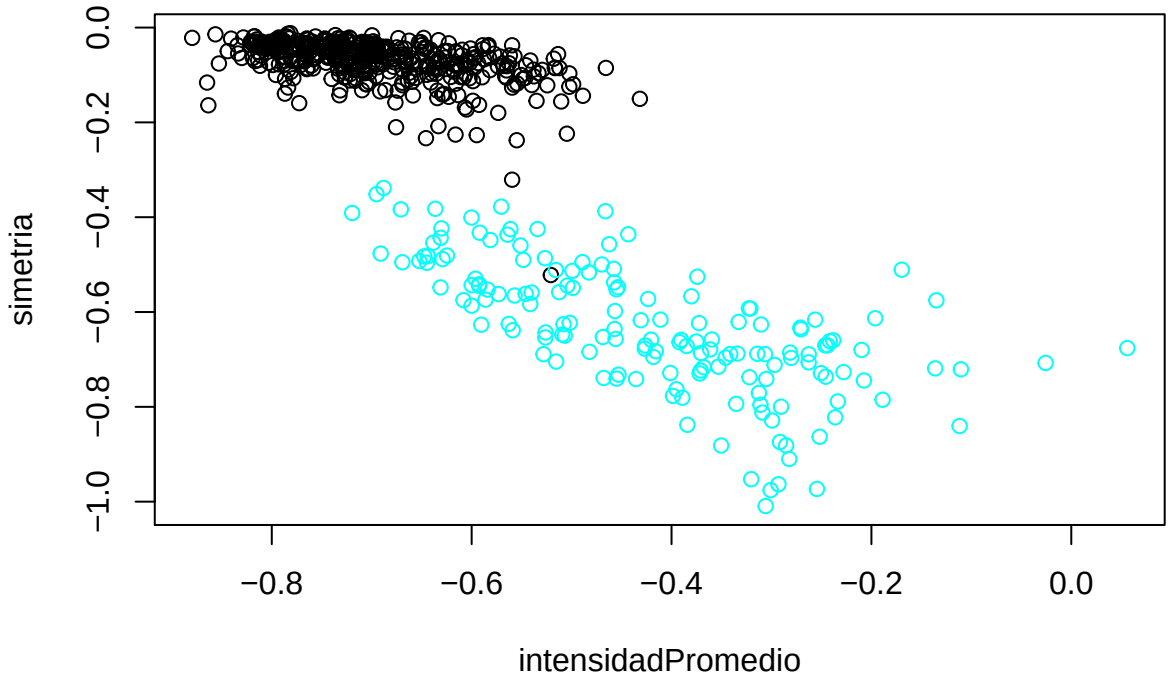
```
## [1] 5 1 1 1
```

3.2 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.

```
# Función simetría aportada por la práctica, pero cambiando sum por mean para que calcule los valores m
fsimetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -mean(A)
}

simetria = apply(grises, 1, fsimetria)
intensidadPromedio = apply(grises, 1, mean)
plot(x=intensidadPromedio, y=simetria, col=digitos+8)
```

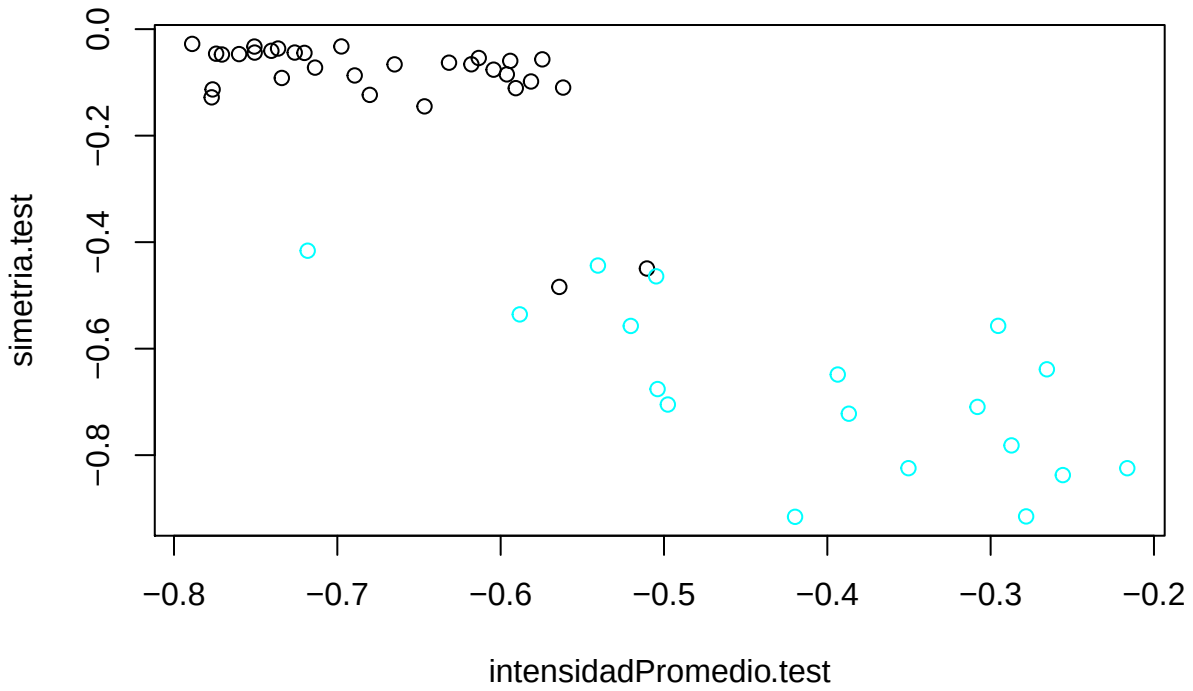


La simetría de cada matriz puede obtenerse ejecutando la función fsimetria con las matrices cuadradas de grises. Grises es una matriz tridimensional, es decir, es un vector que contiene matrices 16x16. Con esos argumentos de la función apply, podemos hacer que fsimetria se ejecute una vez por cada una de las matrices 16x16 de grises, ya que va aumentando de 1 en 1 en ese vector de matrices (de ahí el argumento 1 de apply). El caso de la intensidadPromedio es similar solo que en este caso únicamente queremos calcular la media de valores que tienen los puntos de cada matriz 16x16, por lo que en lugar de fsimetria utilizamos mean.

3.3 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).

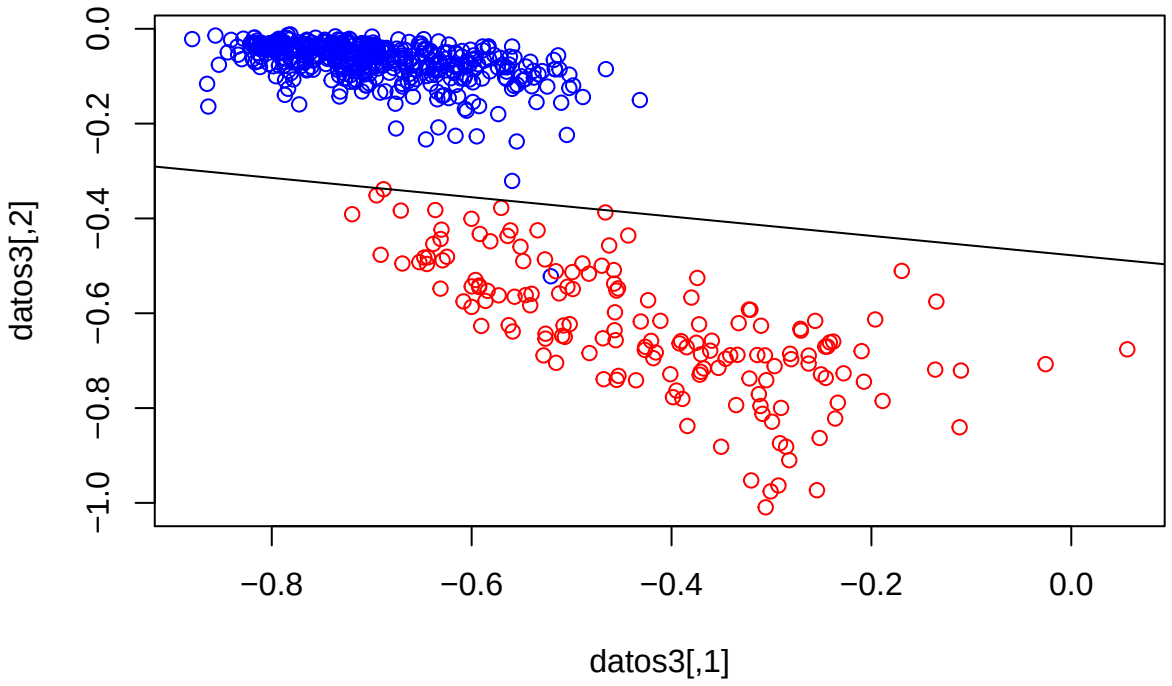
En primer lugar para poder calcular el error fuera de la muestra (Eout) vamos a leer los datos de zip.test de forma análoga a como lo hicimos con los datos del train. Los muestro en gráfica para comprobar que se han leído correctamente.

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

digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,]
digitos.test = digitos15.test[,1]  # etiquetas
ndigitos.test = nrow(digitos15.test)

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

simetria.test = apply(grises.test, 1, fsimetria)
intensidadPromedio.test = apply(grises.test, 1, mean)
plot(x=intensidadPromedio.test, y=simetria.test, col=digitos.test+8)
```



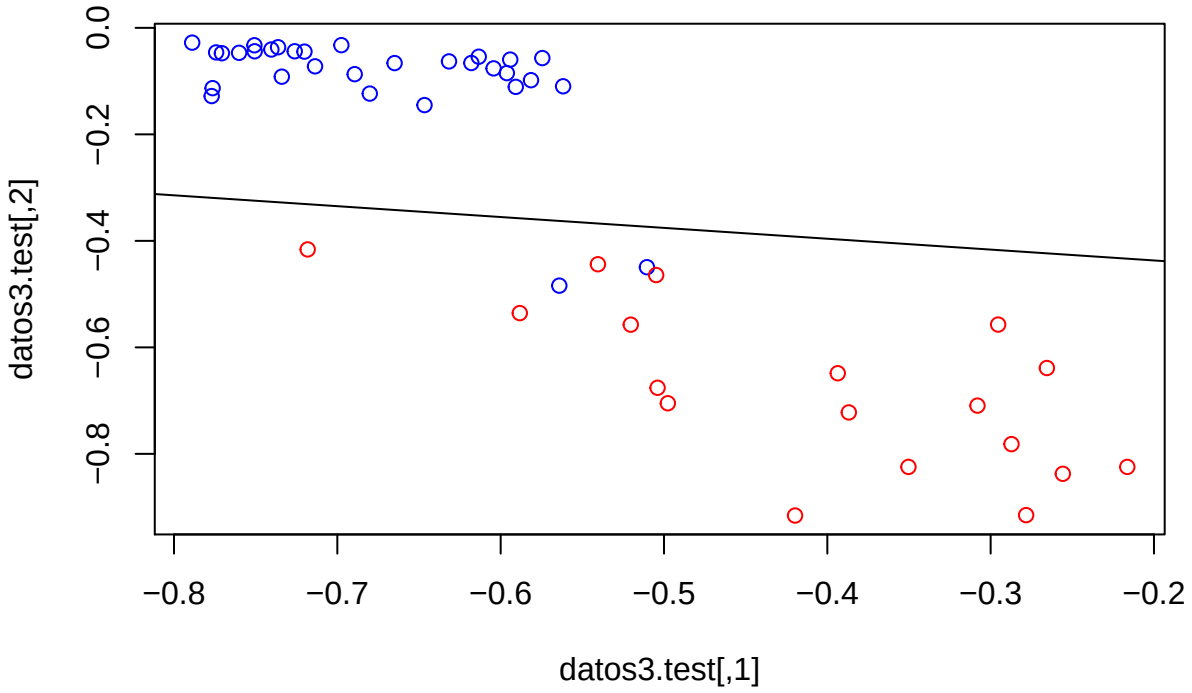
Para ajustar la regresión lineal hace falta la siguiente función, que he definido en función de los argumentos que se indican en el guión de la práctica. Para probar que funciona, voy a calcular la regresión sobre la matriz compuesta por las características IntensidadPromedio y Simetría del apartado anterior, y la dibujaré sobre la gráfica de los puntos. Tal y como se indica en el enunciado, he cambiado las etiquetas que eran 5 por etiquetas -1.

```
# Función que calcula los pesos por regresión lineal de unos datos y unas etiquetas
# aportadas. Se hace uso de la fórmula para calcular la matriz pseudoinversa de
# los datos que aparece en las diapositivas de teoría. Según esto, la matriz
# pseudoinversa de X se puede obtener como la multiplicación de la inversa de
# la multiplicación de X traspuesta por X, por la traspuesta de X. En fórmula
# matemática, pseudoX = (XT*X)-1 * XT.
# El resto de operaciones son las mismas que aparecen en las transparencias,
# empleando la función svd para extraer las matrices U, D y V de X, y sabiendo
# que (XT*X)-1 = V * (pseudoD)^2 * VT (en las transparencias no aparece el cuadrado
# por error, pero lo corregimos en clase) y que la pseudoD es la diagonal de 1/D.
# Finalmente, los pesos se obtienen de multiplicar la pseudoinversa de X por las
# etiquetas.
Regress_Lin = function(datos,label){

  datos=cbind(datos,1)

  descomposicion = svd(datos)
  pseudoinversaD = diag(1/descomposicion$d)
  inversadatosTdatos = (descomposicion$v)%*%(pseudoinversaD^2) %*%t(descomposicion$v)
  pseudoinversa = inversadatosTdatos%*%(t(datos))

  pesos = (pseudoinversa%*%label)
}

datos3 = matrix(c(intensidadPromedio,simetria),nrow=length(intensidadPromedio),ncol=2)
digitos[digitos==5]=-1

pesos3 = Regress_Lin(datos3,digitos)
plot(datos3, col=digitos+3)
coeficientespesos3 = obtenerRectaPesos(pesos3)
abline(coeficientespesos3[2], coeficientespesos3[1])
```



Aquí mostramos que la regresión lineal nos da un vector de pesos con una recta que ofrece una solución buena, pero que no es perfecta al tener unos datos no linealmente separables. A continuación, creamos la función que calcula el error y lo evaluamos para los datos de train (Ein) y test (Eout). Por supuesto, en el cálculo de Eout emplearemos los pesos calculados para los datos de train, ya que si no no estamos comprobando realmente lo bueno que es nuestro vector de pesos calculado.

```
# Función que calcula el error de unas etiquetas de unos datos para un vector de pesos
# en base a cuántas etiquetas están mal situadas con respecto a la recta que formaría
# dicho vector.
errores = function(datos,label,pesos){
  datos = cbind(datos,1)
  (sum(sign(datos%*%pesos) != label))/length(label)
}

datos3.test = matrix(c(intensidadPromedio.test,simetria.test),nrow=length(intensidadPromedio.test),ncol=
digitos.test[digitos.test==5]=-1

plot(datos3.test, col=digitos.test+3)
abline(coeficientespesos3[2], coeficientespesos3[1])
```




```
Ein = errores(datos3, digitos, pesos3)
Eout = errores(datos3.test, digitos.test, pesos3)

print(Ein)
```

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

```
print(Eout)
```

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

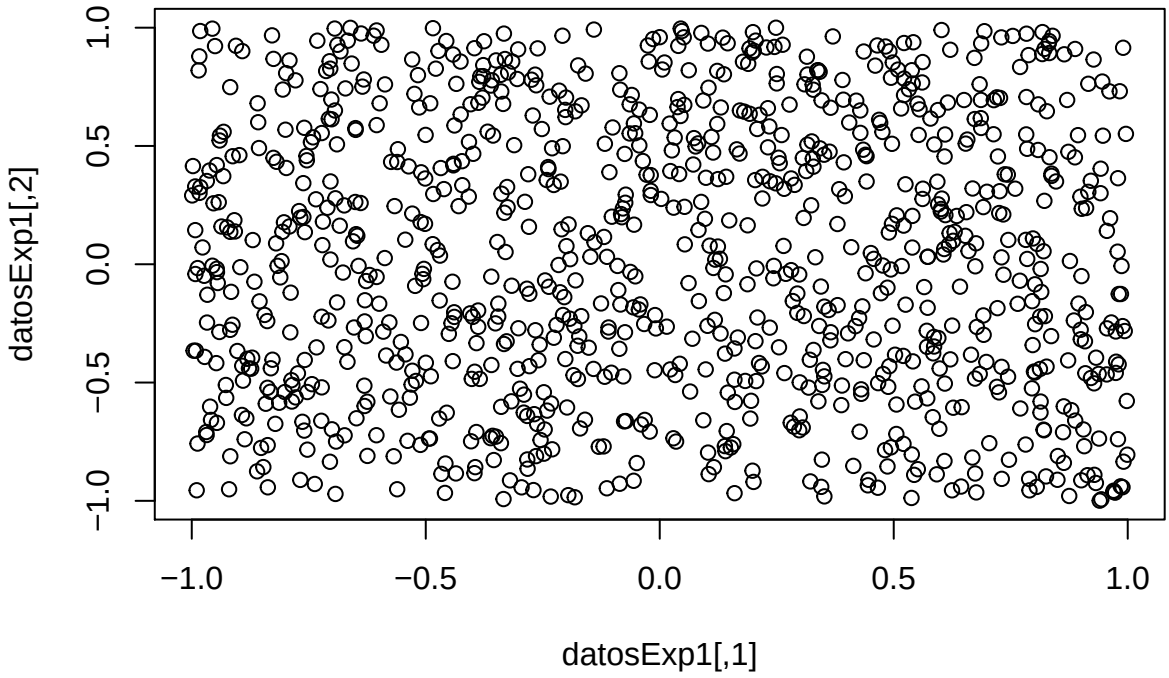
El error de la muestra es muy pequeño, ya que consigue encontrar un vector de pesos que sólo deja una etiqueta mal colocada. En los datos test aparecen dos etiquetas mal colocadas, y además son muchos menos datos, por lo que el error aumenta considerablemente. Por tanto, pese a que parece ser que el error aumenta

mucho de dentro de la muestra hacia fuera, no es un valor alto (sólo un 4% de etiquetas mal colocadas) y además en una muestra con un tamaño mucho más grande obtuvimos mejores resultados. Salvo que la distribución de los datos en general sea muy distinto a la muestra, en este caso considero que la regresión ha obtenido una solución muy buena.

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.

```
set.seed(100)
datosExp1 = simula_unif(N=1000, dim=2, rango=c(-1,1))
plot(datosExp1)
```

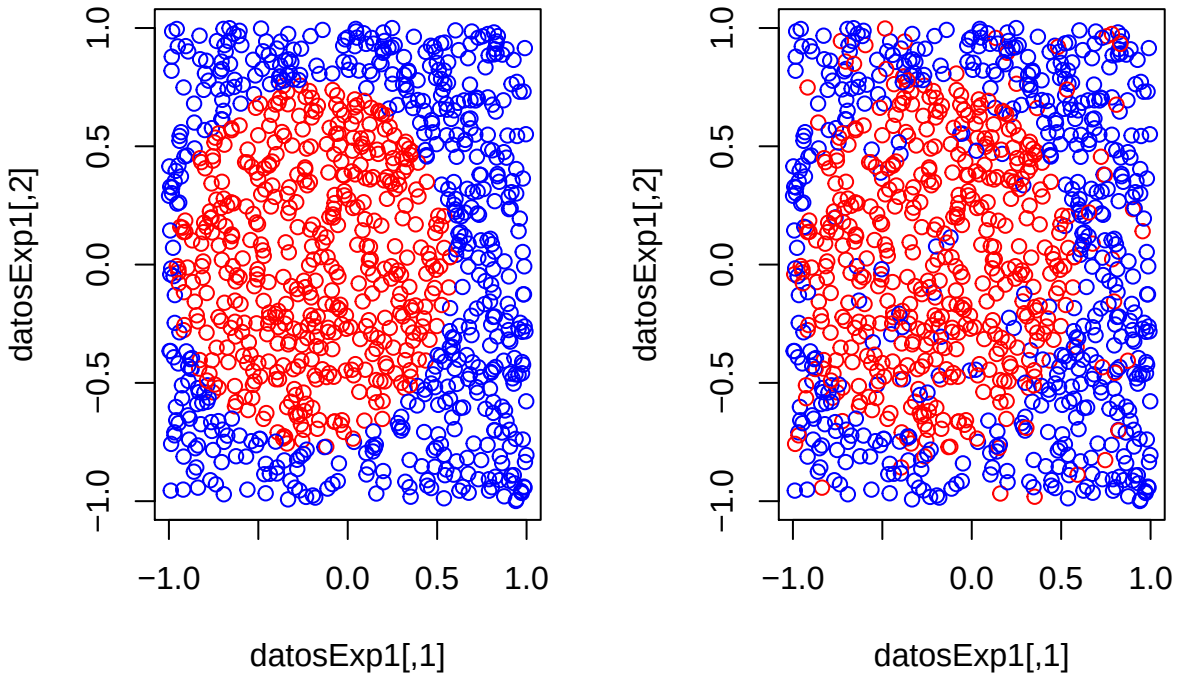


b) Consideremos la función f (x1 , x2) = sign((x1 + 0,2)2 + x22 - 0,6) que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas final.

Voy a definir una función que devuelva el signo de la función del enunciado para los valores x e y de un punto de nuestra matriz. Genero las etiquetas aplicando la matriz y genero el ruido con la función diseñada en el ejercicio 1. Pinto las dos etiquetas para comprobar cómo varían las etiquetas.

```
fexp1 = function(x,y){
  sign((x+0.2)^2 + y^2-0.6)
}

etiquetasexp1 = fexp1(datosExp1[,1], datosExp1[,2])
etiquetasexp1ruido = asignarRuido(etiquetasexp1, 10)
par(mfrow=c(1,2))
```

```
plot(datosExp1, col=etiquetasexp1+3)
plot(datosExp1, col=etiquetasexp1ruido+3)
```



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

El vector de características está internamente diseñado así dentro de la función de regresión lineal, por lo que no tengo que añadir la columna de 1 a mi matriz de puntos de forma previa. Llamo a la función Regress_Lin para obtener los pesos y calculo el Ein en base a esos pesos.

```
pesosExp1 = Regress_Lin(datosExp1,etiquetasexp1ruido)
EinExp1 = errores(datosExp1,etiquetasexp1ruido,pesosExp1)
print(pesosExp1)
```

```
##             [,1]
## [1,] 0.47798232
## [2,] 0.01295278
## [3,] 0.05588164
```

```
print(EinExp1)
```

```
## [1] 0.4
```

El error es cercano al 0.5 dado que es muy complicado separar linealmente de forma mejor una serie de etiquetas generadas en base a una función cuadrática y añadiendo un ruido.

d) Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes) y • Calcular el valor medio de los errores Ein 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.

He creado una función que realiza todo el experimento, utilizando el código de los apartados anteriores y generando 1000 puntos aleatorios con etiquetas con ruido como test para cada iteración. La función devuelve la media de Ein y Eout generados en las 1000 iteraciones.

```
# Función que realiza la funcionalidad pedida por el apartado d) del experimento 1.
# Realiza 1000 iteraciones de los apartados a) al c) de este mismo experimento
# y calcula el Ein en base a ello. Para calcular el Eout creo una muestra aleatoria
# de 1000 datos, y asigno ruido a sus etiquetas, y las utilizo como test.
# La función devuelve una lista que contiene la media de Ein y de Eout que se han
# calculado en las 1000 iteraciones.
experimento1d = function(){
  EinTotal = 0
  EoutTotal = 0
  for (i in 1:1000){
    datosExp1 = simula_unif(N=1000, dim=2, rango=c(-1,1))
    etiquetasexp1 = fexp1(datosExp1[,1], datosExp1[,2])
    etiquetasexp1ruido = asignarRuido(etiquetasexp1, 10)
    pesosExp1 = Regress_Lin(datosExp1,etiquetasexp1ruido)
    EinExp1 = errores(datosExp1,etiquetasexp1ruido,pesosExp1)

    datostest = simula_unif(N=1000, dim=2, rango=c(-1,1))
    etiquetastest = fexp1(datostest[,1], datostest[,2])
    etiquetastestruido = asignarRuido(etiquetastest, 10)
    EoutExp1 = errores(datostest, etiquetastestruido, pesosExp1)

    EinTotal = EinTotal + EinExp1
    EoutTotal = EoutTotal + EoutExp1
  }
  EinMedia = EinTotal / 1000
  EoutMedia = EoutTotal / 1000
  list(Ein=EinMedia, Eout=EoutMedia)
}

set.seed(100)
erroresExp1 = experimento1d()
print(erroresExp1$Ein)
```

```
## [1] 0.397556
```

```
print(erroresExp1$Eout)
```

```
## [1] 0.399439
```

e) Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de Ein y Eout

Los errores son muy próximos al 40% tanto en la muestra como fuera de ella, lo que es un muy mal resultado, por consiguiente, derivado un mal ajuste. Como he dicho antes, esto es debido a que con características lineales no se puede ajustar bien una serie de etiquetas que han sido generadas en base a una función cuadrática.

Como los datos de test van a ser similarmente distribuidos a los de entrenamiento, es comprensible que el error se parezca mucho.

Experimento 2

a) Ahora vamos a repetir el mismo experimento anterior pero usando características no lineales. Ahora usaremos el siguiente vector de características: o2 (x) = (1, x1, x2, x1*x2, x1², x2²). Ajustar el nuevo modelo de regresión lineal y calcular el nuevo vector de pesos ŵ. Calcular el error Ein.

Como en el apartado anterior, la columna de 1 ya va añadida de forma interna en la función Regress_Lin por lo que se añadirá ahí. Por lo demás, hay que añadir columnas con las nuevas características que se especifican, multiplicando las dos columnas entre sí y por ellas mismas. Calculo la regresión lineal igual que en el caso anterior, pero ahora obtengo un vector de pesos de tamaño 6, el tamaño del vector de características.

```
set.seed(100)
datosExp2 = cbind(datosExp1,datosExp1[,1]*datosExp1[,2], (datosExp1[,1])^2, (datosExp1[,2])^2)
etiquetasexp2 = fexp1(datosExp2[,1], datosExp2[,2])
etiquetasexp2ruido = asignarRuido(etiquetasexp2, 10)
pesosExp2 = Regress_Lin(datosExp2, etiquetasexp2ruido)
EinExp2 = errores(datosExp2, etiquetasexp2ruido, pesosExp2)

print(pesosExp2)
```

```
##             [,1]
## [1,]  0.39311726
## [2,]  0.01427830
## [3,]  0.05709863
## [4,]  1.20553563
## [5,]  1.38845788
## [6,] -0.81615146
```

```
print(EinExp2)
```

```
## [1] 0.149
```

El error ahora es mucho más pequeño, porque esta vez estamos realizando al regresión lineal con un vector de características con características cuadráticas, del orden de la función que determinó las etiquetas. Se demuestra así que podemos obtener una solución mucho mejor (casi un 15% de error).

b) Al igual que en el experimento anterior repetir el experimento 1000 veces calculando con cada muestra el error dentro y fuera de la muestra, Ein e Eout respectivamente. Promediar los valores obtenidos para ambos errores a lo largo de las muestras.

He creado una nueva función que realiza todos los cálculos necesarios para el experimento, replicando prácticamente el código del experimento anterior pero añadiendo las características nuevas tanto a los datos de muestra como a los datos de test. De ahí consigo la media de Ein y Eout para las 1000 iteraciones.

```
# Función que realiza la funcionalidad pedida por el apartado b) del experimento 2.
# Realiza 1000 iteraciones en las que genera los datos de train de los que genera
# las etiquetas con ruido y los pesos. Los datos de train tienen las características
# solicitadas en el enunciado. A continuación se calcula el Ein.
# Para calcular el Eout creo una muestra aleatoria de 1000 datos, y asigno ruido a
# sus etiquetas, y las utilizo como test. Los datos test también tienen las
# características especificadas. Obtengo el Eout en base a estos datos test.
```

```r
# La función devuelve una lista que contiene la media de Ein y de Eout que se han
# calculado en las 1000 iteraciones.
experimento2b = function(){
  EinTotal = 0
  EoutTotal = 0
  for (i in 1:1000){
    datos= simula_unif(N=1000, dim=2, rango=c(-1,1))
    etiquetasdatos = fexp1(datos[,1], datos[,2])
    etiquetasdatosruido = asignarRuido(etiquetasdatos, 10)
    datosExp2 = cbind(datos,datos[,1]*datos[,2], (datos[,1])^2, (datos[,2])^2)
    pesosExp2 = Regress_Lin(datosExp2,etiquetasdatosruido)
    EinExp2 = errores(datosExp2,etiquetasdatosruido,pesosExp2)

    datosaux = simula_unif(N=1000, dim=2, rango=c(-1,1))
    etiquetastest = fexp1(datosaux[,1], datosaux[,2])
    etiquetastestruido = asignarRuido(etiquetastest, 10)
    datostest = cbind(datosaux,datosaux[,1]*datosaux[,2], (datosaux[,1])^2, (datosaux[,2])^2)
    EoutExp2 = errores(datostest, etiquetastestruido, pesosExp2)

    EinTotal = EinTotal + EinExp2
    EoutTotal = EoutTotal + EoutExp2
  }
  EinMedia = EinTotal / 1000
  EoutMedia = EoutTotal / 1000
  list(Ein=EinMedia, Eout=EoutMedia)
}

set.seed(100)
erroresExp2 = experimento2b()
print(erroresExp2$Ein)
```

```
## [1] 0.142158
```

```r
print(erroresExp2$Eout)
```

```
## [1] 0.144541
```

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

Ahora los datos de error del experimento son mucho más pequeños, por la misma razón que he explicado en el caso anterior. La inclusión de características cuadráticas es determinante a la hora de ajustar los pesos en la regresión lineal. Con una serie de características del orden de la función que sirvió para determinar las etiquetas se obtienen resultados mucho mejores, no hay más que comparar los resultados para un vector de características lineal (Experimento 1, Ein=0.39, Eout=0.39) con un vector de características cuadráticas (Experimento 2, Ein=0.14, Eout=0.14) para las mismas muestras y datos de test (generados con la misma semilla).

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?

Primero obtengo la primera solución y después ejecuto la función bonusA con un replicate 1000 veces. La función devuelve el Ein, al repetirse 1000 veces el replicate devuelve un vector de 1000 Ein, se le hace la media y se obtiene el error medio en la muestra.

```
# Función que realiza las operaciones requeridas para el apartado A del bonus.
# Crea en cada iteración una muestra de 100 datos en rango [-10,10] y a cada una
# le asigna una recta que delimita sus etiquetas. Saco su ruido y los pesos
# con la regresión lineal y extraigo el Ein.
bonusA = function(){
  datosBonus = simula_unif(N=100, dims=2, rango=c(-10,10))
  rectaBonus = simula_recta(intervalo=c(-10,10))
  etiquetasBonus = asignarEtiquetasSegunRecta(rectaBonus,datosBonus)
  etiquetasBonusRuido = asignarRuido(etiquetasBonus, 10)
  pesosBonus = Regress_Lin(datosBonus,etiquetasBonusRuido)
  EinBonus = errores(datosBonus, etiquetasBonusRuido, pesosBonus)
}

set.seed(100)
datosBonus = simula_unif(N=100, dims=2, rango=c(-10,10))
rectaBonus = simula_recta(intervalo=c(-10,10))
etiquetasBonus = asignarEtiquetasSegunRecta(rectaBonus,datosBonus)
etiquetasBonusRuido = asignarRuido(etiquetasBonus, 10)
pesosBonus = Regress_Lin(datosBonus,etiquetasBonusRuido)
EinBonus = errores(datosBonus, etiquetasBonusRuido, pesosBonus)
print(EinBonus)
```

```
## [1] 0.16
```

```
EinBonusA = replicate(1000, bonusA())
mean(EinBonusA)
```

```
## [1] 0.13379
```

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.

La función que he creado devolverá el Eout, utilizando el vector de pesos que obtenga en cada iteración de la muestra generada y creando un nuevo conjunto de datos para que haga de test. Utilizándola con un replicate de 1000 iteraciones podemos calcular el Eout medio haciendo la media del vector resultante.

```
# Función que realiza las operaciones requeridas para el apartado B del bonus.
# Crea en cada iteración una muestra de 100 datos en rango [-10,10] y a cada una
# le asigna una recta que delimita sus etiquetas. Saco su ruido y los pesos
# con la regresión lineal. A continuación, con ese vector de pesos calculo
# el error Eout con un conjunto de datos test creado en cada iteración, del mismo
# modo que la muestra.
bonusB = function(){
  datosBonus = simula_unif(N=100, dims=2, rango=c(-10,10))
  rectaBonus = simula_recta(intervalo=c(-10,10))
  etiquetasBonus = asignarEtiquetasSegunRecta(rectaBonus,datosBonus)
  etiquetasBonusRuido = asignarRuido(etiquetasBonus, 10)
  pesosBonus = Regress_Lin(datosBonus,etiquetasBonusRuido)

  datosTest = simula_unif(N=100, dims=2, rango=c(-10,10))
  rectaTest = simula_recta(intervalo=c(-10,10))
  etiquetasTest = asignarEtiquetasSegunRecta(rectaTest,datosTest)
  etiquetasTestRuido = asignarRuido(etiquetasTest, 10)

  EoutBonus = errores(datosTest, etiquetasTestRuido, pesosBonus)
}

set.seed(100)
EoutBonusB = replicate(1000,bonusB())
mean(EoutBonusB)
```

```
## [1] 0.42868
```

El error se aproxima mucho al error que menos información nos aporta, el 0.5. Esto es lógico, dado que la función f (la recta, en este caso) sobre la que se delimitan las etiquetas varía también con cada iteración, por tanto no es de extrañar que en la mayoría de los casos la recta generada por los pesos obtenidos no separe de forma parecida a la que genera las etiquetas del test.

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.

En este caso, dentro de la función correspondiente al experimento el resultado a obtener es el número de iteraciones que el Perceptron necesita para converger. Dentro de la función extraigo de la lista que devuelve el PLA las iteraciones y con replicate me devuelve el vector al que puedo hacerle la media.

```
# Función que realiza las operaciones requeridas para el apartado C del bonus.
# Crea en cada iteración una muestra de 10 datos en rango [-10,10] y a cada una
# le asigna una recta que delimita sus etiquetas. Saco su ruido y los pesos
```

```
# con la regresión lineal. A continuación, utilizo ese vector de pesos como el
# vector inicial del PLA, poniendo un máximo de 200 iteraciones y pasando como
# datos la muestra y como etiquetas las de ruido. De la lista que devuelve la
# función, saco las iteraciones necesarias para converger.
bonusC = function(){
  datosBonus = simula_unif(N=10, dims=2, rango=c(-10,10))
  rectaBonus = simula_recta(intervalo=c(-10,10))
  etiquetasBonus = asignarEtiquetasSegunRecta(rectaBonus,datosBonus)
  etiquetasBonusRuido = asignarRuido(etiquetasBonus, 10)
  pesosBonus = Regress_Lin(datosBonus,etiquetasBonusRuido)

  resultadoPLA = ajusta_PLA(datosBonus,etiquetasBonusRuido,200,pesosBonus)
  resultadoPLA$iter
}

set.seed(100)
iteracionesBonusC = replicate(1000,bonusC())
mean(iteracionesBonusC)
```

```
## [1] 119.426
```

En este caso, lo que hemos podido comprobar y que nos llama la atención es que el Perceptron, a pesar de trabajar con etiquetas generadas con ruido, no alcanza siempre el número máximo de iteraciones. Si evaluamos la matriz de iteraciones (a continuación) comprobamos que hay bastantes valores de 200 (max_iter), pero también hay muchos de 1, y varios intermedios. Interpretamos esto como que en bastantes casos, aquellos en los que el Perceptron termina con 1 iteración, la regresión lineal ha funcionado muy bien y el PLA termina de una pasada. En otros tantos casos, el número oscila entre el mínimo y el máximo valor de iteraciones, esto es debido a que la regresión lineal ha aproximado bien, pero al Perceptron le han faltado algunos pasos que ha necesitado para converger. Por último, cuando llega al máximo de iteraciones seguramente tengamos unas etiquetas no separables por este algoritmo.

En resumen, el Perceptron puede funcionar muy bien si lo ejecutamos después de haber inicializado el vector inicial con un algoritmo de regresión lineal, lo que hay que tener en cuenta a la hora de evaluar futuros problemas de aprendizaje.

Muestro para finalizar los valores del vector de iteraciones que ha necesitado el PLA para converger en este experimento, a fin de demostrar el razonamiento que he seguido.

```
print(iteracionesBonusC)
```

```
##    [1]   1   5   1  12  13 200  77 200   8 200   1 200 200   6   1  30 200
##   [18] 200 200   1   1   1 200  12   1   2   1 200 200 200   1   3   6 200
##   [35] 200 200 200 200 200   4   4 200   1 200 200  35   6 200   1   1   1
##   [52] 200 200 200 200  13 200   1   1  10   1 200  19 200 102 200 200   1
##   [69] 200   1 200   1 200   1 200 200 200   1 200 200   1 200   1 200  37
##   [86]   1   1   8   9   1 200   1 200  20   1 200   6  34   9 200  53 200
##  [103] 200 200 200 200   1 200  10 200  10 200  14 200 200 200   8   1 200
##  [120] 200 200 200 200 200 200 200 200 200 200 200  15  22   1 200 200 200
##  [137]   1 200 200   1 200   1 200  16  29 200 200   1  25 200 200 200 200
##  [154]  15 177 200 200 200 200   1 200   1  20 200 200   1 200 200   7  16
##  [171] 200 200  15 200 200   1 200  10 200   1   1   1 200 200 200  98   1
##  [188]   1   1   1 200   1 200 200 200  38 200   1 200  58  59 200  16 200
##  [205] 200 200 200  36 200   1 200   1 200  34  16 200   1  21 200   1 200
##  [222] 200 200  85  20 200 200 200  19 200 200   1  48 200  14  22 200 200
##  [239]   3 200 200 200 200 200 200   1 200 200 200 200 200 200   1 200 200
##  [256]  13 200 200 200  11 200 200  14   6 200  23 200 200   1  14   1   4
```

```
##   [273] 200 200  70  12   4 200  14   1 200   1 200 200   1  25 200   1   1
##   [290] 173 200 200   1 200   1 200 200 200   2   1 200 200 200 200 200 200
##   [307] 200 200 200 200 200   1 200  99 200   1 200   7   1 200   1  29   1
##   [324]  43  55  10   7  15 200   1  69 200   1 200 200 200   1 200 200 200
##   [341] 200  48 200  47 200   1   1 141 200   8   1 200   1 200 200   7 200
##   [358]   6 164 200 177 200 200 200   9 200 200 200 200 200 200 200 200 200
##   [375] 200 200 200   1 200   1 200 200 200 200 200  20 200 200 200   1   1
##   [392] 200 140   1   1 200 200 200 200 200 200 200   1 108   1   1   4 200
##   [409] 200  12 200 200  12 200   1 200   1 200   1   1 200 200 200  18   5
##   [426]   1   1   1   8   9 200 200   1 200 200  16 200 200 200 200 200 200
##   [443]  40 200 200 200 200   1 200   1   4   1   3 152 108  12   4   1 200
##   [460]  68 200 200 200  21 200 200 200 200 200 200 200   1 200 200  30   5
##   [477]   1 200 200   1   1 200 200 200 200 200  12 200 200 115 200 200 200
##   [494] 200   1 200 200  12 200 200 200 200   5 200 200 200   1   1 200 200
##   [511]   1  36 200  13   1  29  85 200  15 200 200 200   1 200 200   1 200
##   [528]   1   1 200  16 200 200 200 200   1 200 200 200 200   5 200 200  14
##   [545]   1   3  68 200 200   1   1   1   1   1 200   1 200 200   1 200   1
##   [562]   1 200 200 200 200 200 200   1 200 200 200   1   1   1   7 200   1
##   [579] 200   1 200   6 200 200 200 200   1 200   1 200 200   1  14 200 200
##   [596] 200   1 200  43 200   1 200 200  38   1   1 200 200   1 200   1   1
##   [613] 200   1 200  32  51 200 200  16  15  44   1 200 200   1 200   5 200
##   [630] 200   3 200 200  13 200 200 200 200 200 200 200 200 200 113 200 200
##   [647] 200 149 200  60 200   1  26   1 200 200 200   1 200 200   1   1   1
##   [664]   1 200 200   1  48 200 200 200   2  31 200 200 200 200 200 200   1
##   [681]  67 200 200  99  18 200   1  23 200   1 200   1 200 200   7 200 200
##   [698]   1 200 200 200 200  44   5 200 166   1  92 200 200 200 200 200 200
##   [715]   3 200 200  14   1 200 200   1 200 200  21 200 200 200 140   1 200
##   [732] 200 171 200   1   1 200  38 116 200  32   1 200   1 200 200  20   6
##   [749]   1 200 200   1   1 200   1   1 200   6   1   2 200 200 200 188 200
##   [766]   1   3 200 200   1 200   1 200 200 200 200   1 200 200   1 200 200
##   [783]   1  26 200   1 200 200   1   8 200   4   2   8 200 200   1 200 200
##   [800] 200  20 200 200 200   1   1  39 200   9   1   1 200   1 200 200  22
##   [817]   1 200   1 200 200 200 200 200   1   4 200   4 200 200   6 200   1
##   [834] 200 200   1   1 200   1   1 200 200 200 200  10   1 200 200 200   1
##   [851] 200   1   1 200 200 200 200  50 200  19   1   1 200 200   1 200 200
##   [868]   1 200   1 186   1 200  85   1   5 119   5 200  39   1 200 200   1
##   [885]  44 200 200  39 200   1   1 200  18 200 200 200 200 200 200   1 200
##   [902] 162 200   8 200 200 200 200 200 200 200 200 200   1   1 200   6 200
##   [919] 103 200 200 200 200 200   2 200 200 200 200 200 200   1   1 200 200
##   [936] 200   3 200  21 200  41   1 200 200 200 200 200 200   1   1 200 200
##   [953] 200 200   1 200 200   1 200 200  36   1   1   1 200   1 200   5 200
##   [970]   1  23 200   1 200 200 200   1  17   1  33 200  20   7   5   1 200
##   [987] 200  10 200   1  27   9 200 200 200 200   1 200 200   1
```