Práctica 1 - Aprendizaje Automático

Arthur M. Rodríguez Nesterenko - DNI:Y1680851W

27 de marzo de 2017

Ejercicio 1: Ejercicio sobre la complejidad de H y el ruido

En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir clases de funciones simples.Haremos uso de tres funciones R ya programadas:

- simula_unif(N,dim,rango), que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango.

```
## Funcion simula_unif(N,dim,rango) : por defecto genera 2 puntos entre
## el intervalo [0,1] de 2 dimensiones
simula_unif = function (N=2,dim=2, rango = c(0,1)){

  m = matrix(runif(N*dim, min=rango[1], max=rango[2]),
             nrow = N, ncol=dim, byrow=T)
  m
}
```

- simula_gaus(N, dim, sigma), que calcula una lista de longitud N de vectores de dimen- sión dim, donde cada posición del vector contiene un número aleatorio extraido de una distribucción Gaussiana de media 0 y varianza dada, para cada dimension, por la posición del vector sigma.

```
## 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_gauss = 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")

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

- simula_recta(intervalo) , que simula de forma aleatoria los parámetros, v = (a, b) de una recta, y = ax + b, que corta al cuadrado [-50, 50]x[-50, 50].

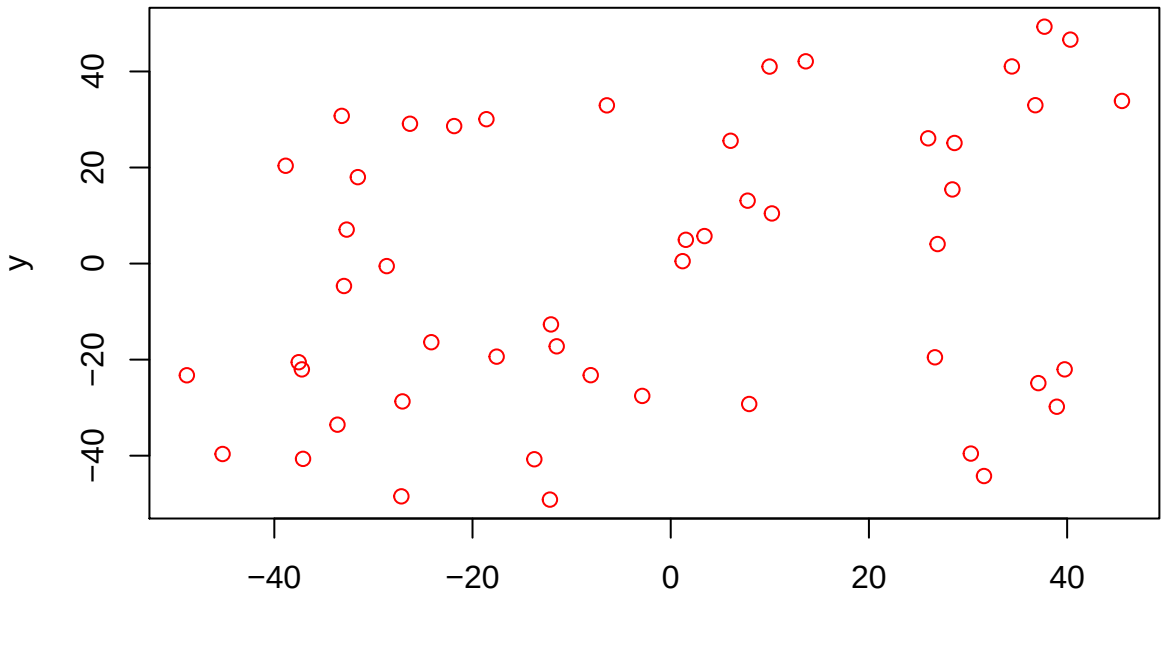
```
##  Función simula_recta(intervalo) una funcion que calcula los parámetros
##  (Para calcular la recta se simulan las coordenadas de 2 ptos dentro del
##  de una recta aleatoria, y = ax + b, que corte al cuadrado
## [-50,50]x[-50,50] y se calcula la recta que pasa por ellos),
##  se pinta o no segun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

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).

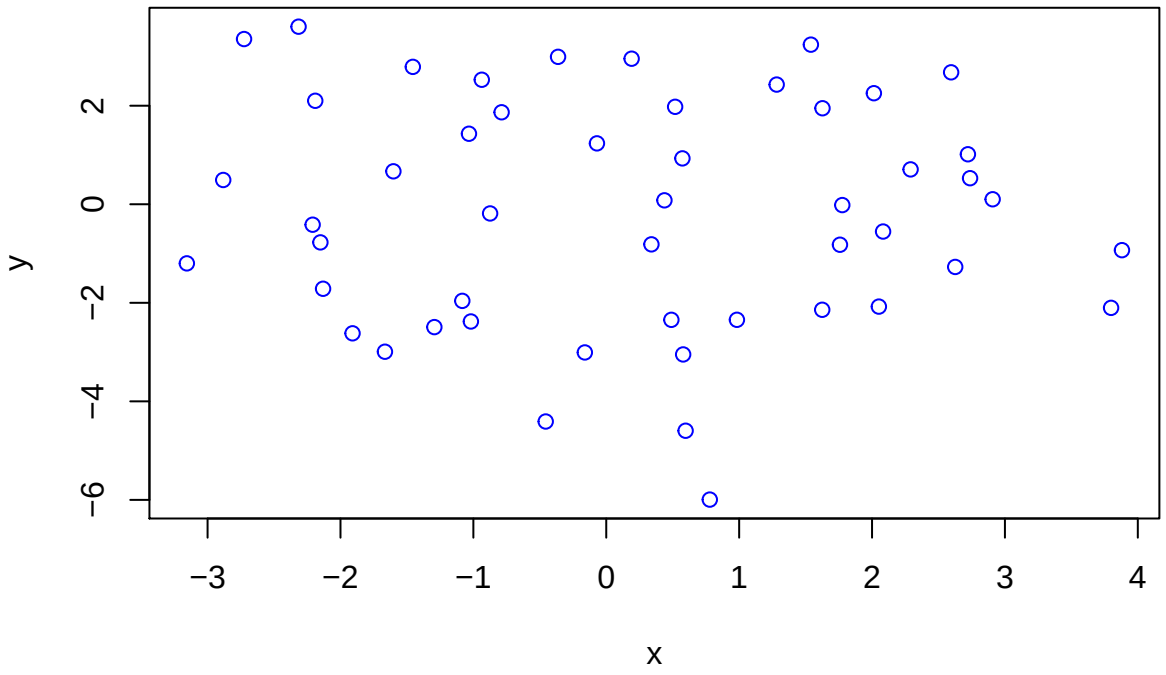
```
set.seed(3)
ejercicio1_1a=simula_unif(50,2,c(-50,50))
plot(ejercicio1_1a,xlab = "x",ylab = "y", col=2)
```



###

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

```
set.seed(3)
ejercicio1_1b=simula_gauss(50,2,c(5,7))
plot(ejercicio1_1b,xlab = "x",ylab = "y",col=4)
```



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().

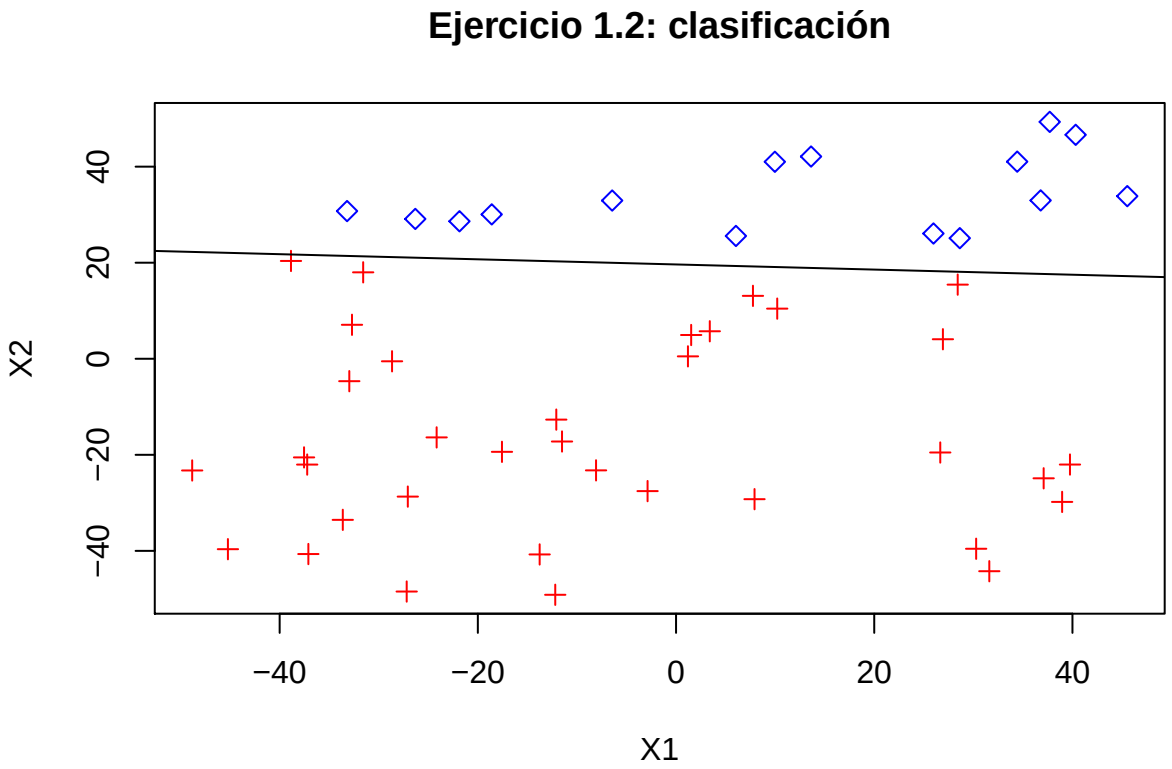
Vamos a utilizar una función que se llama asignarEtiquetas que genera las etiquetas siguiendo la función que se nos proporciona en el apartado 2.

```
## Funcion generarEtiquetas(muestra,rectaSimulada)
## para generar las etiqueta teniendo en cuenta los valores de una muestra y
## una recta simulada siguiendo la función f(x,y)= y - ax - b

generarEtiquetas=function(muestra,rectaSimulada=simula_recta(c(-50,50)) ){

  #Comprobamos si tenemos datos de muestra
  if(missing(muestra)) stop("Error: no ha proporcionado un conjunto de muestras")

  #Ahora para cada componente de la muestra 2D comprobamos su signo segun
  #la recta que le hemos pasado por parametros
  etiquetas= sign(muestra[,2] - (rectaSimulada[1]*muestra[,1]) -rectaSimulada[2])
  etiquetas

}
```

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)

```
## Para este ejercicio necesitamos generar una muestra de puntos 2D previo
## al calculo de las etiquetas
```

```r
set.seed(3)
ejercicio1_2=simula_unif(50,2,c(-50,50))

## Simulamos una recta aleatoria que corte al cuadrado [-50,50]x[-50,50]
rectaSim=simula_recta(c(-50,50))

## Generamos el conjunto de etiquetas asociadas a la recta
etiquetas=generarEtiquetas(ejercicio1_2,rectaSim)

## A continuación pintamos la recta junto con los puntos etiquetados
plot(ejercicio1_2,main= "Ejercicio 1.2: clasificación",xlab="X1",
     ylab="X2",pch=etiquetas+4, col=etiquetas+3)
abline(rectaSim[2],rectaSim[1])
```



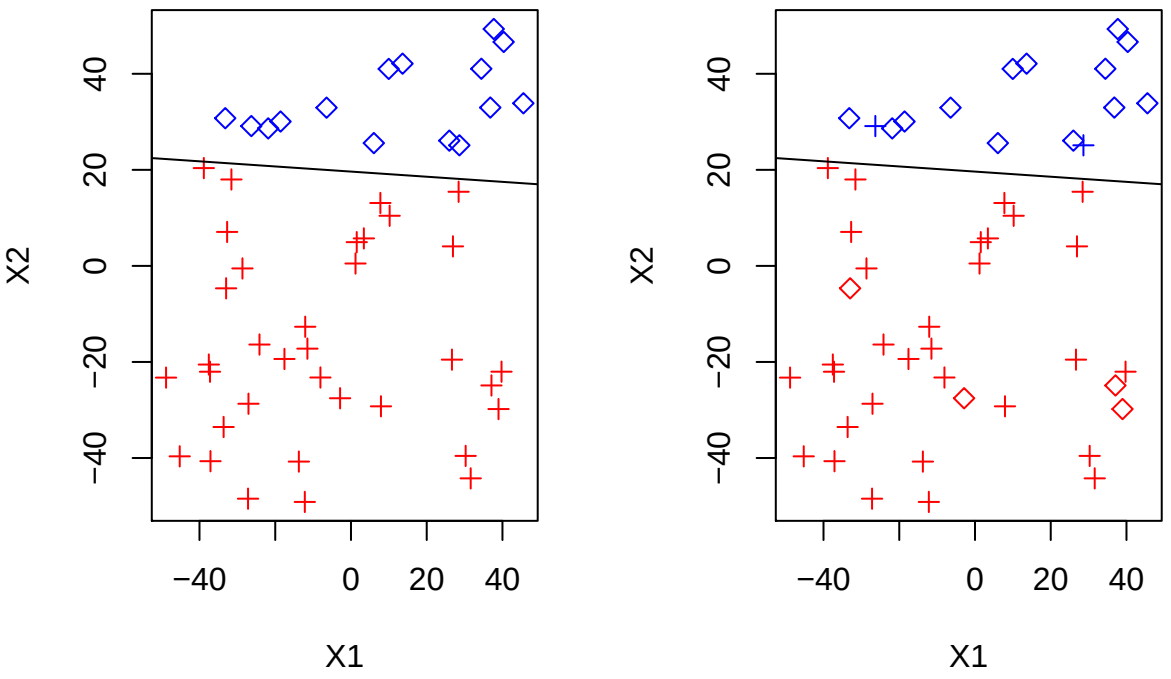
Es fácil comprobar de forma visual como los datos están bien clasificados. Pasamos con el apartado b.

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)

Para este apartado necesitamos de una función que nos asigne ruido a un determinado numero de muestras etiquetas de forma aleatoria. La función se puede ver a continuación y básicamente localiza los índices donde las etiquetas valen 1 y -1 para luego seleccionar un porcentaje de índices de forma aleatoria (redondeando para el caso de que tenga parte decimal) y finalmente sustituir esos los valores que se indexan por los nuevos valores deseados

```r
## Función asignarRuido(etiquetas,porcentaje) que a partir de un conjunto de
## etiquetas cambia un porcentaje de éstas; un porcentaje de etiquetas "1"
## ahora pasarán a ser "-1" y viceversa
asignarRuido=function(etiquetas,porcentaje){
```

```
  #Guardamos los indices de las etiquetas positivas y negativas
  positivas = which(etiquetas>0)
  negativas = which(etiquetas<0)

  #Del conjunto de indices que corresponden a etiquetas positivas
  #seleccionamos un porcentaje de forma aleatoria (sin reemplazamiento)
  #para evitar que se repitan
  positivas = sample(positivas,round(length(positivas)*0.01*porcentaje))
  negativas = sample(negativas,round(length(negativas)*0.01*porcentaje))

  #Aquellos indices de las etiquetas que han sido elegidos
  #se modifican por sus nuevos valores
  etiquetas[positivas]=-1
  etiquetas[negativas]=1

  etiquetas
}

## Para comprobar que se modifican de forma correcta vamos a establecer
## el parametro gráfico para permitir que se vean ambas graficas a la vez
par(mfrow = c(1,2))

## Cambiamos un 10% de etiquetas positivas y negativas
set.seed(3)
etiquetasRuido=asignarRuido(etiquetas,10)

## Volvemos a pintar la recta junto con los puntos BIEN etiquetados
plot(ejercicio1_2,main= "Ejercicio 1.2: clasificación sin ruido",xlab="X1",
     ylab="X2",pch=etiquetas+4, col=etiquetas+3)
abline(rectaSim[2],rectaSim[1])

## Nuevamnete pintamos la recta junto con las nuevas etiquetas con RUIDO
plot(ejercicio1_2,main= "Ejercicio 1.2b: clasificación con ruido",xlab="X1",
     ylab="X2",pch=etiquetasRuido+4, col= etiquetas+3)
abline(rectaSim[2],rectaSim[1])
```



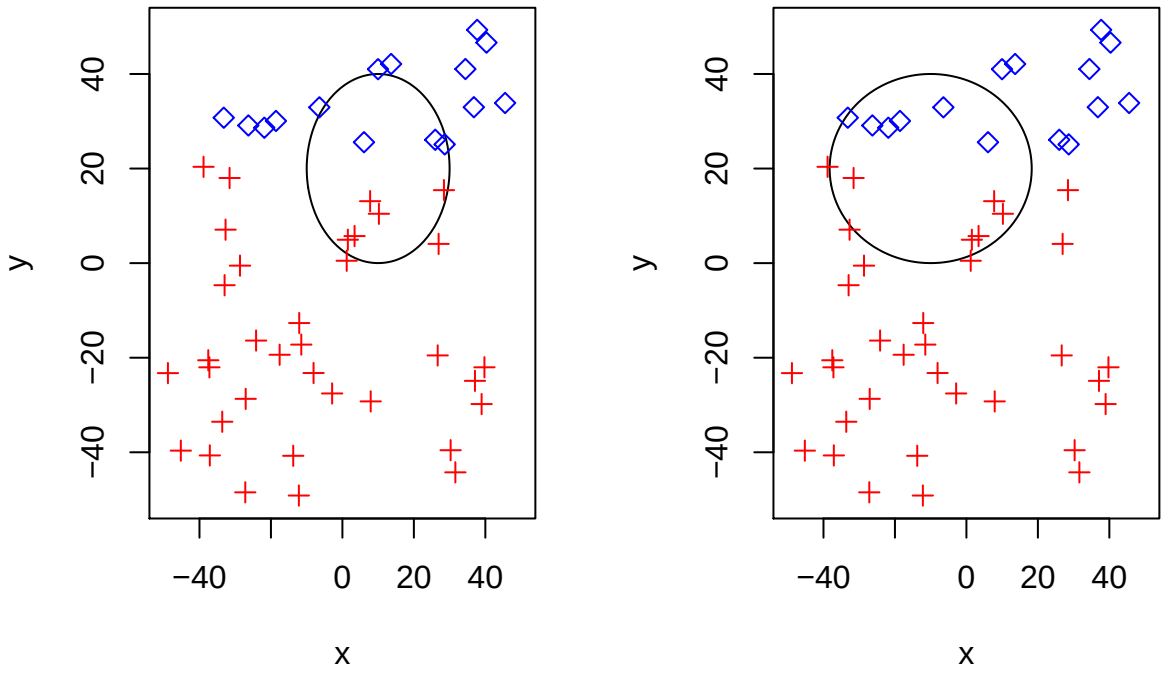
3. Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

- 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

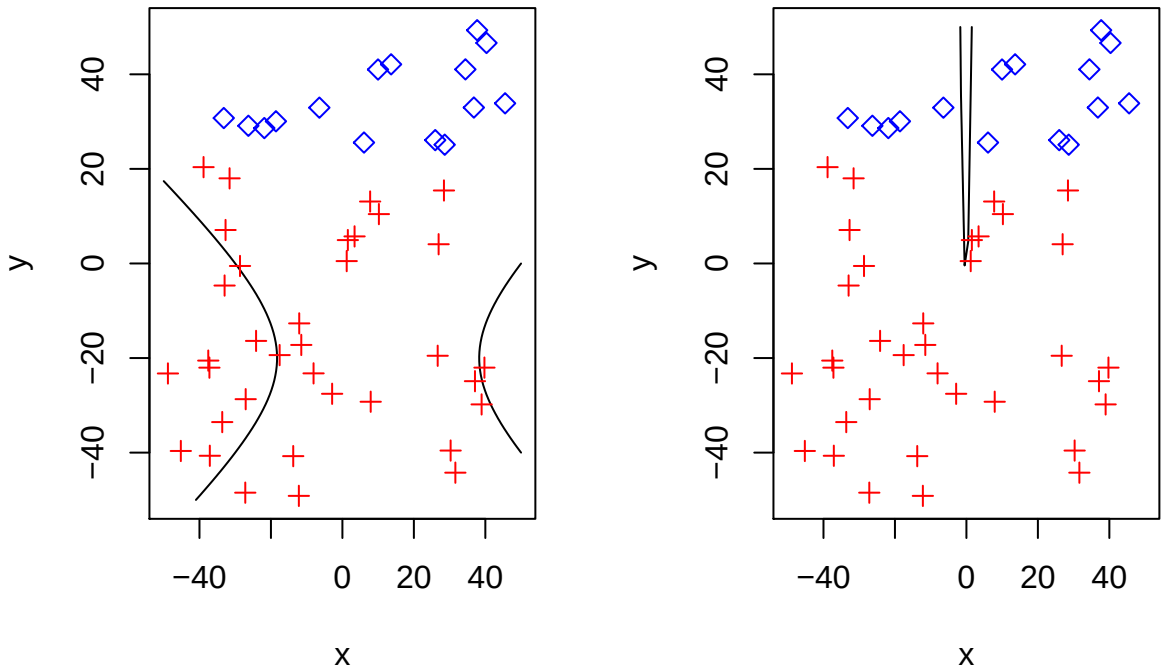
Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Hemos ganado algo en mejora de clasificación al usar funciones más complejas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

```
## Hacemos uso de la funcion proporcionada en el fichero "paraTrabajo1.R"
## Funcion pintar_fronter(f,rango) para pintar la frontera de la función
# a la que se pueden añadir puntos y etiquetas
pintar_frontera = function(f,rango=c(-50,50)) {
  x=y=seq(rango[1],rango[2],length.out = 100)
  z = outer(x,y,FUN=f)
  if (dev.cur()==1) # no esta abierto el dispositivo lo abre con plot
    plot(1, type="n", xlim=rango, ylim=rango)
  contour(x,y,z, levels = 0, drawlabels = FALSE,xlim =rango,
          ylim=rango, xlab = "x", ylab = "y")
}

## A continuación definimos las 4 funciones que nos van a servir para
## clasificar las muestras del ejercicio 2b.

## Funcion 1
```

```
f1_xy = function(x,y){
  (x-10)^2 + (y-20)^2 - 400
}


## Funcion 2
f2_xy = function(x,y){
  0.5*(x+10)^2 + (y-20)^2 - 400
}

## Funcion 3
f3_xy = function(x,y){
  0.5*(x-10)^2 - (y+20)^2 - 400
}

## Funcion 4
f4_xy = function(x,y){
  y - 20*(x^2) - 5*x + 3
}

## Para realizar las evaluaciones establecemos el parametro gráfico
## para poder ver las 4 gráficas a la vez
par(mfrow = c(1,2))
## Finalmente pintamos las 4 funciones que definen nuestra frontera
## junto con las etiquetas del ejercicio 1.2b

## Funcion 1
pintar_frontera(f1_xy,c(-50,50))
points(ejercicio1_2,pch=etiquetas+4, col=etiquetas+3)

## Funcion 2
pintar_frontera(f2_xy,c(-50,50))
points(ejercicio1_2,pch=etiquetas+4, col=etiquetas+3)
```




```
par(mfrow = c(1,2))
## Funcion 3
pintar_frontera(f3_xy,c(-50,50))
points(ejercicio1_2,pch=etiquetas+4, col=etiquetas+3)

## Funcion 4
pintar_frontera(f4_xy,c(-50,50))
points(ejercicio1_2,pch=etiquetas+4, col=etiquetas+3)
```



Es facil comprobar que a pesar de utilizar funciones mucho mas complejas (orden cuadrático) no se ha obtenido mejora sustancial en cuanto a la capacidad de clasificacion. Dado que las etiquetas han sido generadas a partir de una función lineal, ninguna otra función será capaz de ajustar mejor los datos. Independientemente

de las regiones positivas y negativas de cada una de las funciones, es muy fácil comprobar como nunca serán capaces de ajustarse a un etiquetado "lineal", por lo tanto no hemos ganado nada en mejora en cuanto a utilizar este tipo de funciones más complejas para este problema de clasificación.

Ejercicio 2: Ejercicio sobre el Algoritmo Perceptron

1. Implementar la función ajusta_PLA(datos,label,max_iter,vini) que calcula el hiper- plano 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.

Vamos a implementar la función ajusta_PLA(datos, label, max_iter, vini) donde calcularemos el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA, devolviendo los coeficientes del hiperplano Entradas: - datos: matriz donde cada item (y su etiqueta) se representa por una fila - label: vector de etiquetas (rango (-1,+1)) - max_iter: numero maximo de iteraciones permitidas (por defecto 100) - vini: valor inicial del vector de pesos (Por defecto 0,0,1) El hiperplano solucion que calcula el perceptron es de la forma W1$x1$ + $W2$x2 - bias = 0 donde W1,W2 son los pesos del vector y bias el 1 que colocamos al final (normalización).

Describir el funcionamiento del algoritmo del Perceptron es sencillo:

1. Realizamos una comprobación previa para evitar que no hayamos proporcionado algun dato para la ejecución del algoritmo.
2. Añadimos un 1 al final de cada fila (normalización)
3. Mientras que el número de iteraciones no supere el máximo de iteraciones (max_iter) y el algoritmo detecte en una pasada completa al menos una muestra mal clasificada: 3.1 Recorremos de forma aleatoria el conjunto de vector de características y seleccionamos una muestra. 3.2 Miramos si el signo de XiW *es igual al de la etiqueta i-ésima 3.3.1 Si es distinto, actualizamos* $W = W_Old + Yi$Wi
4. Devolvemos como salida una lista que contiene los pesos calculados y el número de iteraciones consumidas por el Perceptron.

```
ajusta_PLA = function(datos, label,max_iter=300, vini=c(0,0,1)){

  #Comprobamos si tenemos datos
  if(missing(datos)) stop("Error: no ha proporcionado el conjunto de datos")

  #Comprobamos si tenemos etiquetas
  if(missing(label)) stop("Error: no ha proporcionado las etiquetas")

  # Agregamos al final de la matriz de el 1 como
  #tercera componente en todas las filas
  datos = cbind(datos,1)

  #Booleano para controlar si en alguna de las pasadas
  #completas hemos modificado los pesos
  modificacionPesos=TRUE

  #Variable auxiliar
  tamDatos= length(datos[,1])

  #Mientras no superemos el numero de iteraciones
```

```
  iteraciones = 0

  while(iteraciones < max_iter & modificacionPesos ){

    modificacionPesos=FALSE

    #Recorremos la matriz de datos, accediendo a cada
    #fila (vector de características)
    for(i in sample(1:length(datos[,1]))){

      #Calculamos el producto escalar
      valor = crossprod(datos[i,],vini)

      #Solo modificaremos el vector de pesos cuando
      # los signos sean distintos (mal clasificado)
      if(sign(valor) != label[i]){

        vini = vini + label[i]*datos[i,]
        #Contabilizamos, al menos, una modificacion
        modificacionPesos=TRUE

      }
    }

    iteraciones=iteraciones+1

  }

  print(sprintf("El algoritmo Perceptron ha terminado tras %d iteraciones",iteraciones))
  #Damos como salida los pesos para luego calcular
  # los coeficientes de la recta
  list(pesos = vini,iters = iteraciones)
}
```

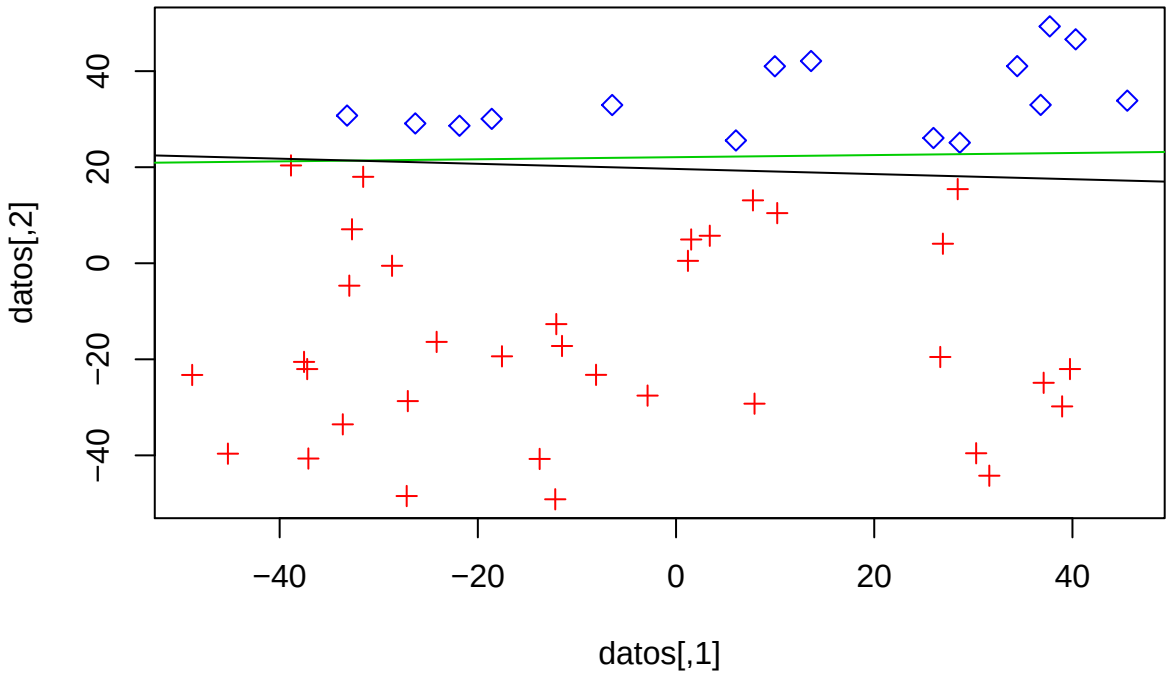
Ahora implementamos una función que a partir del vector de pesos me calcule los coeficientes de la recta que separa los datos para finalmente ejecutar un ejemplo del Perceptron sobre un conjunto de datos y etiquetas generados de forma aleatoria.

```
## Funcion calcularCoeficientes(vini)que calcula los coeficientes
## de la recta (pendiente y punto de corte)
calcularCoeficientes = function(vini){

  #Calculamos la pendiente y el punto de corte según la formula
  # W1*x1 + W2*x2 -bias = 0
  puntoCorte = -vini[3]/vini[2]
  pendiente = -vini[1]/vini[2]

  coeficientes=c(puntoCorte,pendiente)
  coeficientes


}
set.seed(3)
## Para comprobar que nuestro PLA funciona, simulamos un conjunto de datos
```

```r
datos = simula_unif(50,2, c(-50,50))

## Simulamos la recta sobre la que se establecerán las etiquetas
recta1=simula_recta(c(-50,50))
label= generarEtiquetas(datos,recta1)

## Inicializamos el vector de pesos a (0,0,1)
vini=c(0,0,1)

## Ejecutamos PLA y obtenemos los pesos, para calcular los coeficientes despues
resultados = ajusta_PLA(datos,label, 1000, vini)
```

```
## [1] "El algoritmo Perceptron ha terminado tras 48 iteraciones"
```

```r
coeficientes= calcularCoeficientes(resultados$pesos)

## Parametrizamos el plot
par(mfrow = c(1,1))

## Pintamos los puntos con el vector de pesos
## y la recta sobre la que establecemos las etiquetas
## Nota: el hiperplano solucion se pinta de verde y
## y la recta de negro
plot(datos, pch=label+4, col=label+3)
abline(coeficientes[1],coeficientes[2], col= 3)
abline(recta1[2],recta1[1])
```



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.

Para este ejercicio basta con generar un vector de pesos estándar (vini=0,0,1) con el que ejecutaremos 10 veces el algoritmo del Perceptron. Acto seguido realizaremos el mismo procedimiento pero en cada una de las 10 iteraciones ejecutaremos el PLA con un vector de pesos aleatorios entre 0 y 1 para las componentes w1 y w2. Anotaremos el número de ejecuciones media para cada uno de estos experimentos para luego valorar el resultado.

```
## Recuperamos los datos del ejercicio 2a
datosEj2_2 = ejercicio1_2
labelEj2_2 = etiquetas
```

```
## Variables utiles
pesos = c(0,0,1)
pasosConverger_2A=0


## Funcion para ejecutar el PLA devolviendo unicamente las iteraciones utilizadas
funcionEjercicio2_2A = function(datos, label, iters, pesos){

  resultados= ajusta_PLA(datos,label,iters,pesos)
  resultados$iters
}

##Ejecutar el algoritmo 10 veces para el mismo vector de pesos
## produce los mismos resultados, por lo que basta con ejecutarlo
## una unica vez y ese será el valor medio
resultados=replicate(10,funcionEjercicio2_2A(datosEj2_2,labelEj2_2,500,pesos))
```

```
## [1] "El algoritmo Perceptron ha terminado tras 113 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 131 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 199 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 94 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 112 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 50 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 50 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 33 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 46 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 67 iteraciones"
```

```
pasosConverger_2A=round(mean(resultados))
print(sprintf("Numero de iteraciones medio para converger (pesos constantes): %d ",
              pasosConverger_2A))
```

```
## [1] "Numero de iteraciones medio para converger (pesos constantes): 90 "
```

```
## Funcion para ejecutar el PLA devolviendo unicamente las iteraciones utilizadas
## con un vector de pesos aleatorio
funcionEjercicio2_2B=function(datos, label){

  pesos=runif(3,0,1)
  resultados= ajusta_PLA(datos,label,500,pesos)
```

```
  resultados$iters

}

## Obtenemos la media de los pasos para converger
pasosConverger_2B=replicate(10,funcionEjercicio2_2B(datosEj2_2,labelEj2_2))
```

```
## [1] "El algoritmo Perceptron ha terminado tras 35 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 114 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 47 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 73 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 69 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 69 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 100 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 36 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 81 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 104 iteraciones"
```

```
pasosConverger_2B=round(mean(pasosConverger_2B))
print(sprintf("Numero de iteraciones medio para converger (pesos aleatorios): %f ",
              pasosConverger_2B))
```

```
## [1] "Numero de iteraciones medio para converger (pesos aleatorios): 73.000000 "
```

Dado que ambos experimentos cuentan con aleatoriedad, el primero en la elección del xi a evaluar y el segundo igual pero sumado a que tiene un vector de pesos aleatorio entre 0 y 1, ambos experimentos pueden dar resultados parecidos tanto en mayor como menor número de iteraciones, incluso dependiendo de los valores que tome el vector aleatorio de pesos, el segundo experimento puede llega a oferecer un número iteraciones de media tanto mayor como menor que el primero (aunque ambos con el mismo grado de magnitud), pero dada la componente aleatoria implícita en el, no podemos saber cuando será así, por lo que no se puede establecer una relación directa entre el número de iteraciones y el punto de inicio y por ende ambas configuraciones pueden arrojar resultados parecidos.

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.

```
## Recuperamos los datos del ejercicio 2a
datosEj2_3 = ejercicio1_2
labelEj2_3 = etiquetasRuido

## Variables utiles
pesos = c(0,0,1)
pasosConverger_3A = 0
set.seed(3)
## Ejecutamos el algoritmo 10 veces y vemos cuanto tarda de media en
## converger, aunque es predecible lo que sucederá
resultados=replicate(10,funcionEjercicio2_2A(datosEj2_3,labelEj2_3,500,pesos))
```

```
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
```

```
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
```

```
pasosConverger_3A = mean(resultados)
print(sprintf("Numero de iteraciones medio para converger (pesos constantes): %f ",
              pasosConverger_3A))
```

```
## [1] "Numero de iteraciones medio para converger (pesos constantes): 500.000000 "
```

```
## Obtenemos la media de los pasos para converger en el caso B
pasosConverger_3B=replicate(10,funcionEjercicio2_2B(datosEj2_3,labelEj2_3))
```

```
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
## [1] "El algoritmo Perceptron ha terminado tras 500 iteraciones"
```

```
pasosConverger_3B=mean(pasosConverger_3B)
print(sprintf("Numero de iteraciones medio para converger (pesos aleatorios): %f ",
              pasosConverger_3B))
```

```
## [1] "Numero de iteraciones medio para converger (pesos aleatorios): 500.000000 "
```

El comportamiento de ambos experimentos es el esperado teniendo en cuenta que tras asignar ruido de forma aleatoria a un 10% de las etiquetas, la muestra deja de ser separable linealmente y por lo tanto ambos experimentos tardarán en converger de media la cantidad de iteraciones que le hemos indicado (en este caso por defecto 500), puesto que el Perceptron agota todas las iteraciones antes de ser capaz de encontrar un hiperplano que separe a la muestra, hiperplano que naturalmente, no existe.

Ejercicio 3: Ejercicio sobre Regresión Lineal

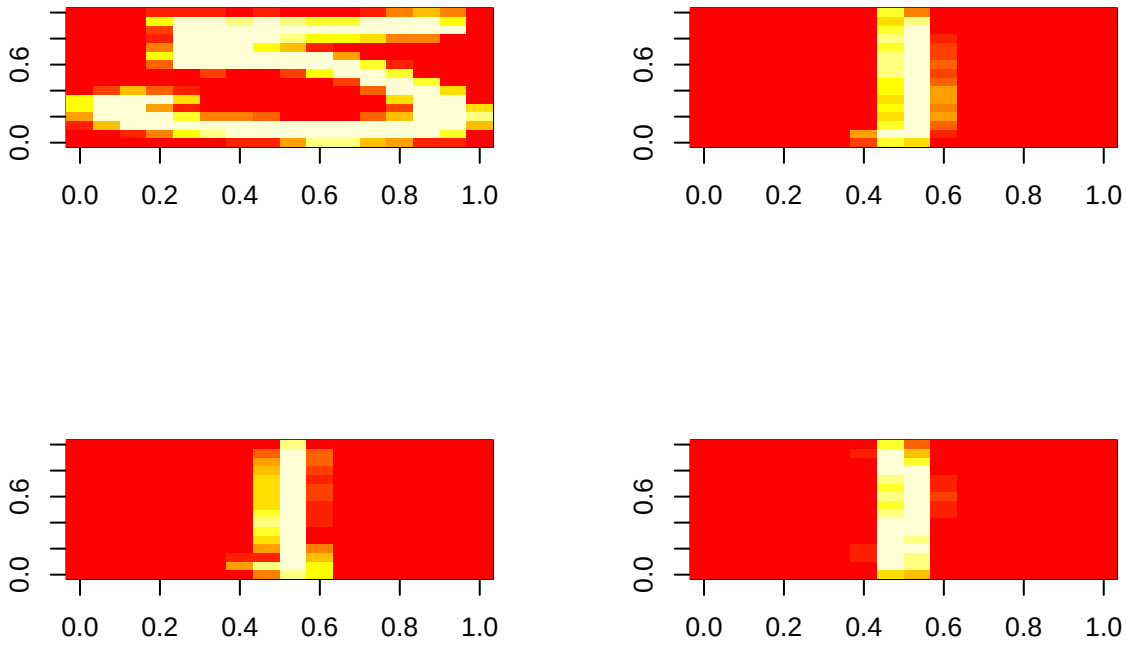
En la web del curso se encuentran disponibles la descripción

1. Leemos datos:

Abra el fichero Zip.info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el 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. También está disponible el fichero Zip.test que deberemos usar más adelante.

```
## En una funcion que llamaremos leerDatosYEstructurar(fichero)
## englobaremos el funcionamiento de las lineas de codigo proporcionadas
## para utilizarla sobre distintos ficheros de datos

leerDatosYEstructurar=function(fichero){
```

```
  # Lectura de fichero datos.train (2780 muestras con 256 valores en escala de gris)
  # o datos.test
  datos = read.table(fichero, quote="\"", comment.char="", stringsAsFactors=FALSE)

  ## Nos limitamos a seleccionar las instancias de los numeros
  ## 1 y 5 (aquellos en los que la columna 1 - V1 tengan un 1 o 5)
  digitos15 = datos[datos$V1==1 | datos$V1==5,]

  digitos = digitos15[,1]  # nombre de las etiquetas (si es 1 o 5)
  ndigitos = nrow(digitos15) # numero de muestras de 1 y 5

  ## Se retira la clase y se monta una matriz 3D: 599*16*16
  ## Para cada instancia de 1 o 5, dimesionamos los 256 valores de
  ## escala de gris en una matriz 16x16
  grises = array(unlist(subset(digitos15,select=-V1)),c(ndigitos,16,16))

  ## Eliminamos los dataframes que ya no vamos a utilizar
  rm(datos)
  rm(digitos15)

  list(datos=grises,labels=digitos)
}

imagenesTrain=leerDatosYEstructurar("datos/zip.train")
## Para visualizar los 4 primeros
## ------------------------------------------------------------------------
par(mfrow=c(2,2))

for(i in 1:4){
  imagen = imagenesTrain$datos[i,,16:1] # se rota para verlo bien
  image(z=imagen)
}
```



```
## Etiquetas correspondientes a las 4 imágenes (los numeros que hemos dibujado)
imagenesTrain$labels[1:4]
```

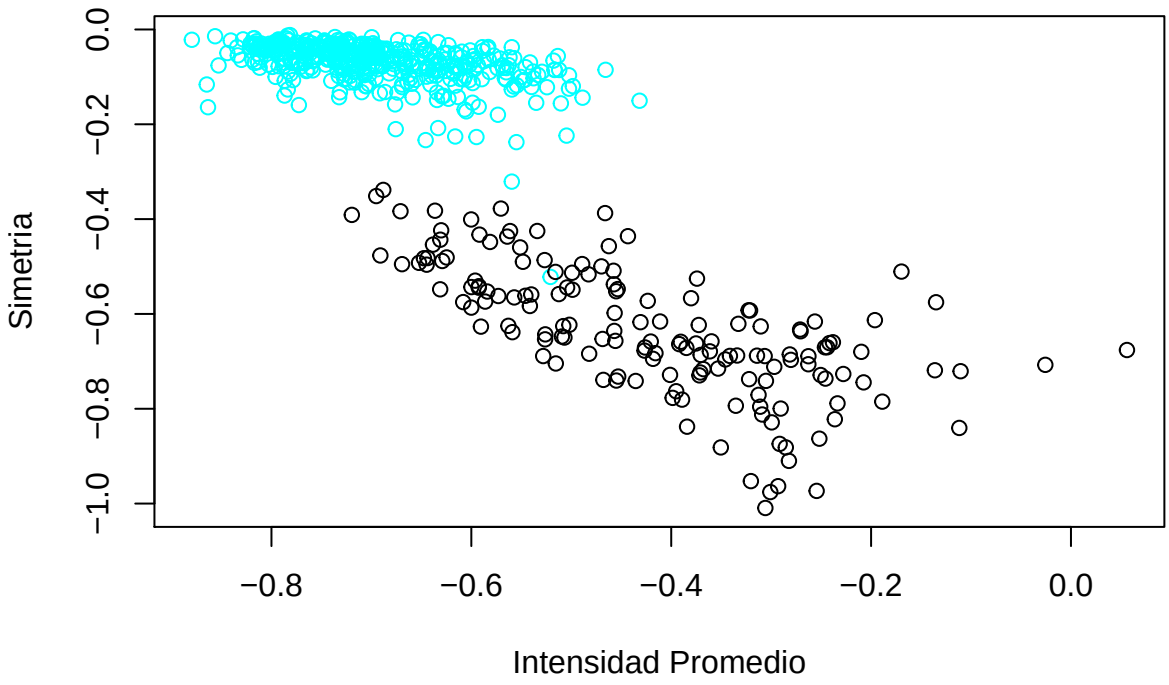
```
## [1] 5 1 1 1
```

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 selec- cionadas de 1's y 5's.

```
## Para calcular el grado de simetria nos valemos de la funcion
## fsimetria(A) que calcula la diferencia en valor absoluto de la
## matriz A con la propia matriz A pero con las columnas invertidas
fsimetria = function(A){
  A = abs(A-A[,ncol(A):1])
  -mean(A)
}

## Calculamos el grado de simetria y el valor de intensidad media
gradoSimetriaTrain = apply(imagenesTrain$datos,1,fsimetria)
intensidadMediaTrain = apply(imagenesTrain$datos,1,mean)

par(mfrow=c(1,1))
## Pintamos las dos características
plot(intensidadMediaTrain,gradoSimetriaTrain,xlab = "Intensidad Promedio",
     ylab = "Simetria",col=imagenesTrain$labels+4)
```



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).

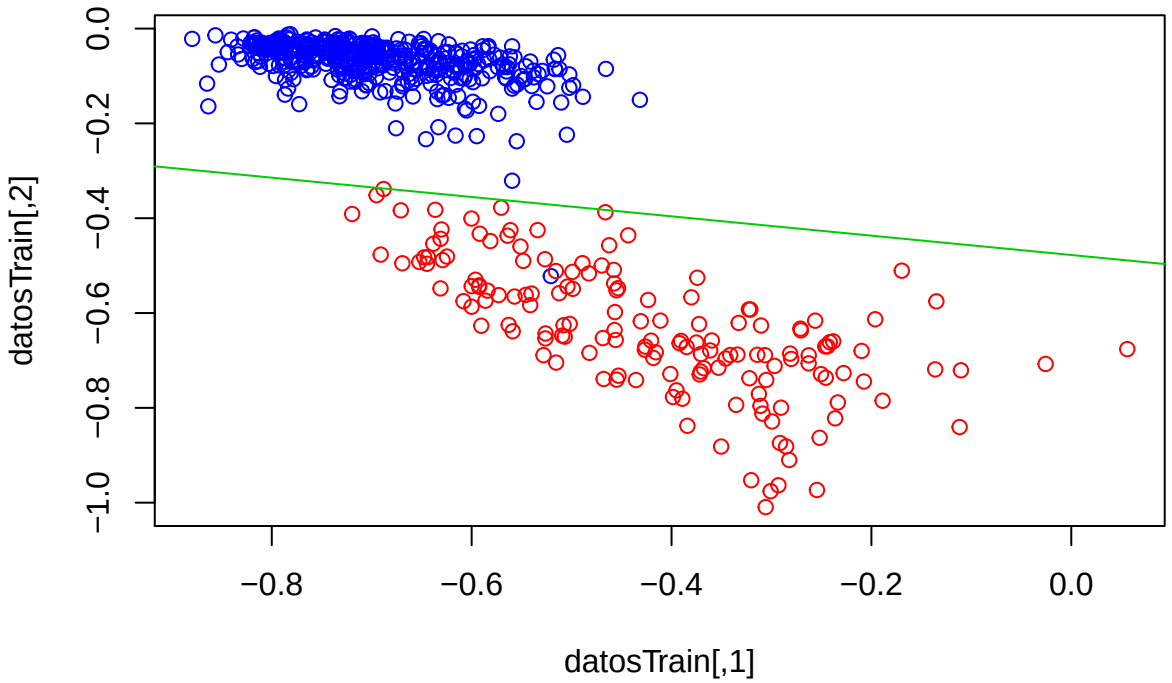
```
## Modificamos las etiquetas de tal forma que los 1s se queden con etiqueta
## 1 y los 5s cambiarán a -1.
etiquetasTrain=imagenesTrain$labels
etiquetasTrain[etiquetasTrain==5]=-1

## Creacion de la matriz de datos a partir de la intensidad promedio y el valor medio
datosTrain=matrix(c(intensidadMediaTrain,gradoSimetriaTrain),length(etiquetasTrain),2)
# Anidimos un 1 al final de nuestro vector de caracteristicas
datosTrain=cbind(datosTrain,1)
```

A continuación pasamos a explicar el algoritmo Regress_Lin. Este algoritmo simple utiliza como entrada unos datos y unas etiquetas para ajustar un modelo de regresión lineal utilizando la transformación SVD (Descomposición en valores singulares). La transformación SVD permite expresar X como X= UDV' (con V' traspuesta de V), con U y V matrices ortogonales y D una matriz diagonal. Utilizando la función svd() que nos devuelve estas tres matrices y aplicando lo que aparece en las transparencias de clase de teoría, calculamos la pseudo inversa de D como la matriz diagonal de 1/D, posteriormente calculamos (X'X)^-1 = VD^2V, que es el paso previo a calcular la pseudoinversa de X. El paso final es el cálculo del vector de pesos W como el producto matricial de la pseudoinversa de X por la traspuesta de X. Este vector de pesos será con el que calcularemos los coeficientes de la recta que nos clasificará los datos.

```
## Funcion Regress_Lin(datos, label) para ajustar un modelo de regresión
## lineal usando la transformación por SVD sobre la matriz de datosUnosCincos
## y pintar la solucion obtenida con los datos usados en el ajuste
## Vamos a utilizar la funcion svd(datos) para obtener la descomposición
## en valores singulares de la matriz de datos y obtener la matriz U y D
## para computar (X'X)^-1

Regress_Lin = function(datos,label){
  # Calculamos la descomposicion en valores singurales
  # y recuperamos sus valores
  descompValSing = svd(datos)
  D = descompValSing$d
  V = descompValSing$v

  #Calculamos la pseudoinversa de D
  pInversaD = diag(1/D)

  #Calculamos (X'X)^-1 = V*D^2*V'
  traspuestaXInversa = V%*%(pInversaD^2) %*%t(V)

  # Finalmente obtenemos la pseudoinversa
  pseudoInversa = traspuestaXInversa%*%(t(datos))

  # Nuestro vector de pesos final será PseudoInvX*label
  W = (pseudoInversa%*%label)

}
```

Probamos que el algoritmo funciona para un determinado conjunto de datos realizando una prueba

```
## Ejecutamos el algoritmo de regresion lineal y comprobamos que funciona
regresionTrain = Regress_Lin(datosTrain,etiquetasTrain)
plot(datosTrain,col=etiquetasTrain+3)
coeficientesTrain=calcularCoeficientes(regresionTrain)
abline(coeficientesTrain[1],coeficientesTrain[2],col=3)
```



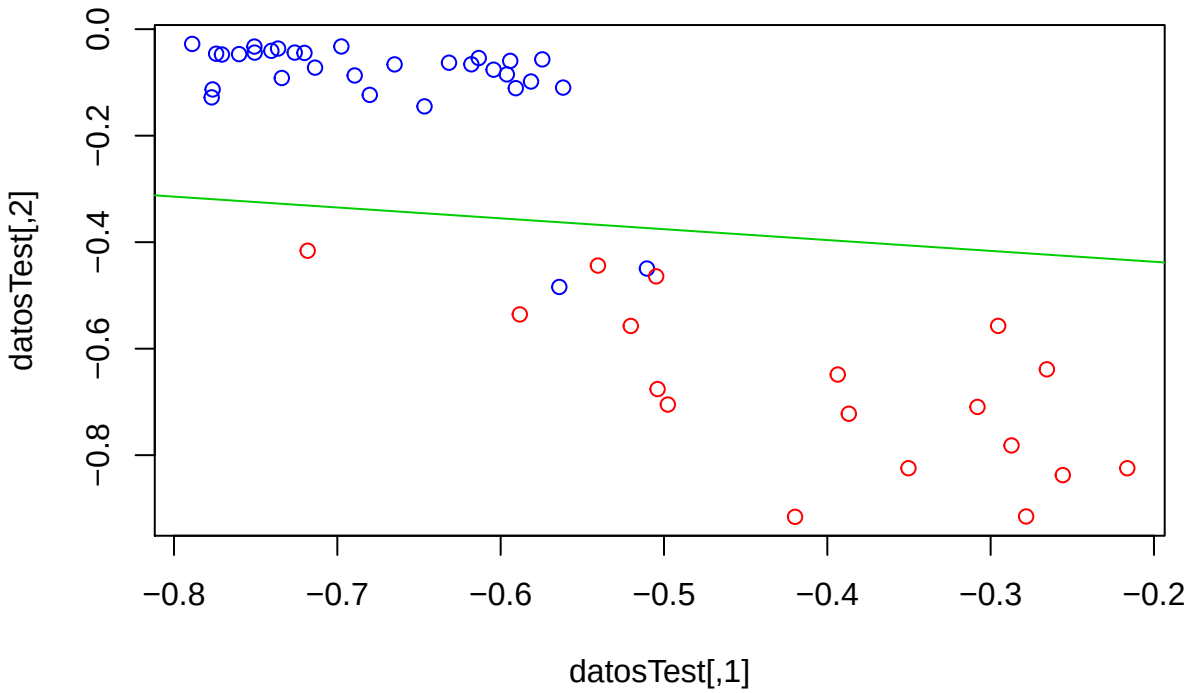
Una vez hemos visto que nuestra regresión lineal se ha ajustado lo mejor que ha podido a los datos, vamos a calcular el Ein y el Eout a partir de una función que me cuenta el número de muestras mal clasificadas.

```
## Creamos una funcion para obtener el error en funcion del numero de muestras
## mal clasificadas que se llamara calculaErrorMalClasificadas(datos,label,pesos)
calcularErrorMalClasificadas = function(datos,label,pesos){

  # Calculamos el producto de los datos por los pesos
  malClasificadas = apply(datos,1,"%*%",pesos)

  #Obtenemos cuantas etiquetas han sido mal clasificadas
  numMalClasificadas = sum((sign(malClasificadas)!=label))

  #Obtenemos el error
  error = 100*numMalClasificadas/(length(label))

}

Ein = calcularErrorMalClasificadas(datosTrain,etiquetasTrain,regresionTrain)
print(sprintf("El Ein es del %f porciento",Ein))
```

```
## [1] "El Ein es del 0.166945 porciento"
```

A continuación realizaremos el mismo procedimiento pero para los datos de test, con el objetivo de comprobar que tan bueno es el ajuste realizado.

```
## Para comparar la bondad del ajuste, utilizamos los datos de zip.test
imagenesTest=leerDatosYEstructurar("datos/zip.test")

## Calculamos el grado de simetria y el valor de intensidad media
gradoSimetriaTest = apply(imagenesTest$datos,1,fsimetria)
intensidadMediaTest = apply(imagenesTest$datos,1,mean)

## Modificamos las etiquetas de tal forma que los 1s se queden con etiqueta
## 1 y los 5s cambiarán a -1.
etiquetasTest=imagenesTest$labels
etiquetasTest[etiquetasTest==5]=-1

## Creacion de la matriz de datos a partir de la intensidad promedio y el valor medio
datosTest=matrix(c(intensidadMediaTest,gradoSimetriaTest),length(etiquetasTest),2)
# Anidimos un 1 al final de nuestro vector de caracteristicas
datosTest=cbind(datosTest,1)

## Ejecutamos el algoritmo de regresion lineal y comprobamos que funciona
regresionTest = Regress_Lin(datosTest,etiquetasTest)
plot(datosTest,col=etiquetasTest+3)
coeficientesTest=calcularCoeficientes(regresionTest)
abline(coeficientesTrain[1],coeficientesTrain[2],col=3)
```




```
## Calculamos Ein para luego comparar con el Eout de la muestra de Test
## donde Eout(W)=1/N||XW - y||^2
Eout = calcularErrorMalClasificadas(datosTest,etiquetasTest,regresionTrain)

## Calculamos la bondad del ajuste
bondadAjuste = 100 - (Eout-Ein)
print(sprintf("El Eout es del %f porciento",Eout))
```

```
## [1] "El Eout es del 4.081633 porciento"
```

```
print(sprintf("La bondad del ajuste es de un %f porciento",bondadAjuste))
```

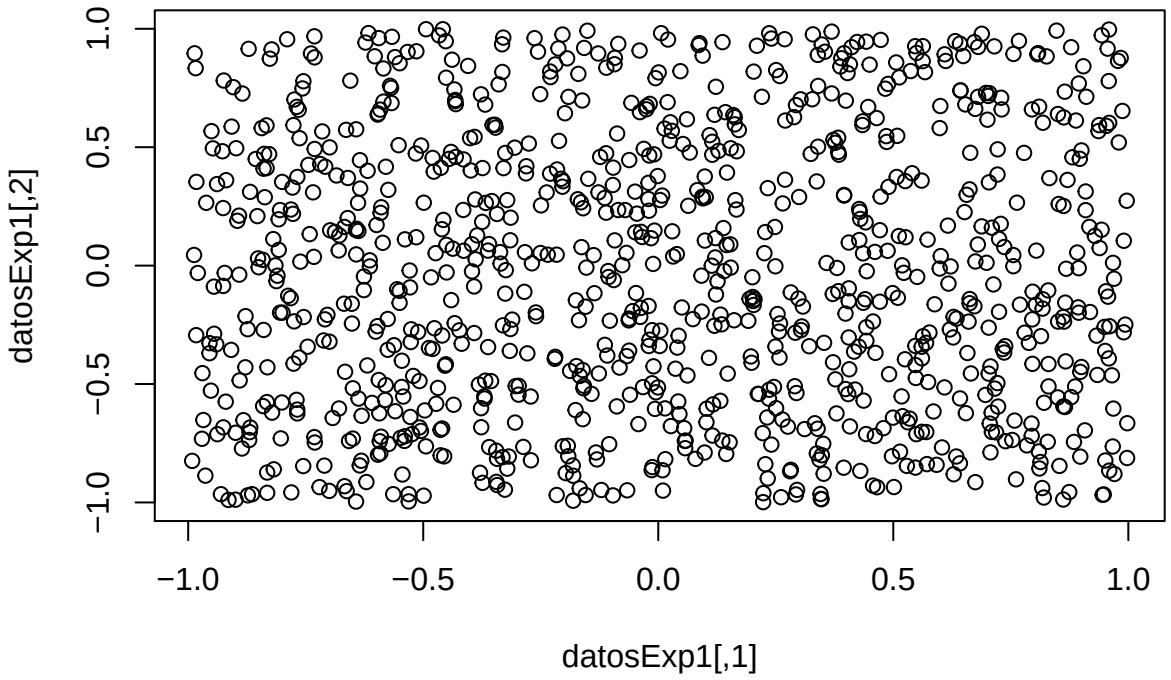
```
## [1] "La bondad del ajuste es de un 96.085312 porciento"
```

4. En este apartado exploramos como se transforman los errores Ein y Eout cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función simula_unif(N,2,size) que nos devuelve N coordenadas 2D de puntos uniformemente muestreados dentro del cuadrado definido por [size, size] x[size, size]

EXPERIMENTO-1:

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

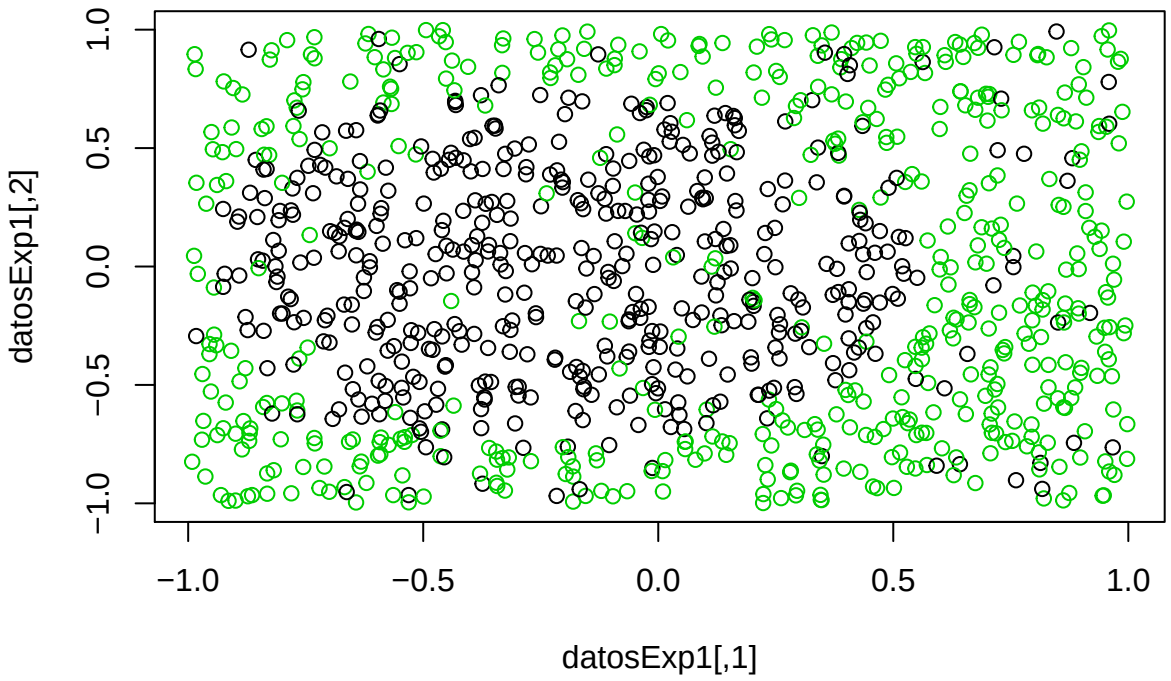
```
## Apartado a)
## Generamos la muestra de entrenamiento de N=1000
datosExp1=simula_unif(1000,2,c(-1,1))
plot(datosExp1)
```



b) Consideremos lafunción f(x1,x2)=sign((x1+0,2)$^{2+x}$2 -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.

```
 ## Apartado b)
## Definimos la funcion que nos clasificará los puntos
f_Exp1=function(x1,x2){
```

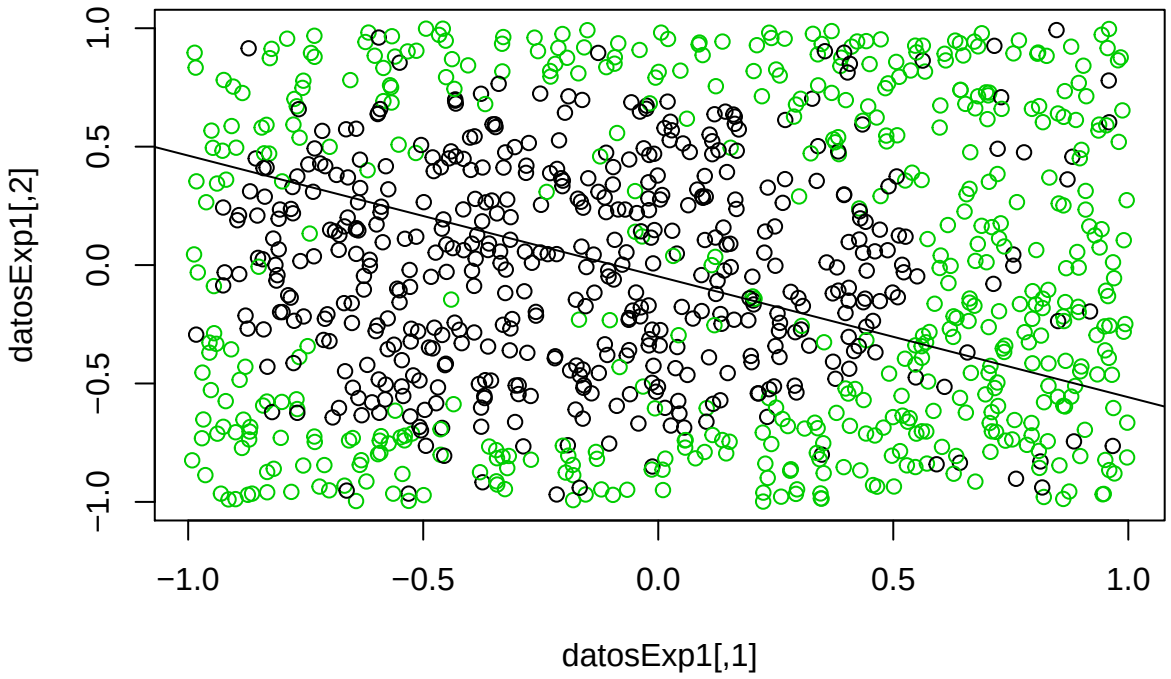
```
  sign((x1+0.2)^2 + x2^2 -0.6)
}

## Generamos las etiquetas con y sin ruido
etiquetasExp1 = f_Exp1(datosExp1[,1],datosExp1[,2])
etiquetasRuidoExp1 = asignarRuido(etiquetasExp1,10)

## Pintamos los puntos y comprobamos que funciona
par(mfrow=c(1,1))
plot(datosExp1,col=etiquetasRuidoExp1+2)
```



###

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.

```
## Apartado c)
datosExp1_Aux=cbind(1,datosExp1)
pesosExp1=Regress_Lin(datosExp1_Aux,etiquetasRuidoExp1)
plot(datosExp1, col=etiquetasRuidoExp1+2)
abline(-pesosExp1[1]/pesosExp1[2],-pesosExp1[2]/pesos[3])
```




```
Ein_Exp1 = calcularErrorMalClasificadas(datosExp1_Aux,etiquetasRuidoExp1,pesosExp1)
print(sprintf("El error Ein es del %f porciento",Ein_Exp1))
```

```
## [1] "El error Ein es del 37.400000 porciento"
```

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.

```
## Apartado d)
## Generamos 1000 muestras distintas (datos 2x1000)
funcionAuxiliar = function(datos){

  sign((datos[,1]+0.2)^2 + datos[,2]^2 -0.6)

}

## Generamos las 1000 matrices
milMatrices = replicate(1000,simula_unif(1000,2,c(-1,1)))
## Generamos los 1000 conjuntos de etiquetas y les asignamos ruido
## (apply las genera por columnas, asi que la cambio por filas para mantener
##el mismo estandar en todos los ejercicios)
etiquetasMilMatrices = apply(milMatrices,3,funcionAuxiliar)
etiquetasMilMatrices = t(etiquetasMilMatrices)
etiquetasRuidoMilMatrices = apply(etiquetasMilMatrices,1,asignarRuido,porcentaje=10)

## Para cada matriz y cada conjunto de etiquetas, recuperamos los pesos
## lo hacemos una primera vez para no hacer el rbind con NULL
```

```
pesosMilMatrices = matrix(c(0,0,0),1,3,byrow = T)
Ein_MilMatrices = 0

## Funcion para calcular el Ein

calcularEinMilMatrices= function(milMatrices,pesosMilMatrices){

  for(i in 1:dim(milMatrices)[3]){

    # Aislamos la matriz y las etiquetas a utilizar
    datos=cbind(milMatrices[,,i],1)
    label = etiquetasRuidoMilMatrices[i,]

    #Calculamos los pesos y el Ein
    pesos = Regress_Lin(datos,label)
    EinEach = calcularErrorMalClasificadas(datos,label,pesos)

    #Guardamos los valores de pesos y Ein
    pesosMilMatrices = rbind(pesosMilMatrices,t(pesos))
    Ein_MilMatrices = cbind(Ein_MilMatrices,EinEach)

  }

  list(Ein=Ein_MilMatrices,Pesos=pesosMilMatrices)
}


## Borramos la primera fila de los pesos y la de los Ein
resultadosExp1 = calcularEinMilMatrices(milMatrices,pesosMilMatrices)
Ein_MilMatrices = resultadosExp1$Ein[1,2:dim(resultadosExp1$Ein)[2]]
Ein_MilMatrices = mean(Ein_MilMatrices)
print(sprintf("El error Ein para las mil matrices de entrenamiento es del %f porciento",
              Ein_MilMatrices))
```

```
## [1] "El error Ein para las mil matrices de entrenamiento es del 47.105800 porciento"
```

Una vez hemos calculado el Ein y tenemos los pesos para cada una de las mil matrices, repetimos el proceso pero ahora al calcular el Eout tenemos en cuenta los pesos calculados para las matrices de entrenamiento por lo que es inmediato.

```
## Una vez tenemos los pesos de cada matriz, repetimos el experimento
## con nuevas muestras y etiquetas para calcular el Eout
milMatrices_Eout = replicate(1000,simula_unif(1000,2,c(-1,1)))
etiquetasMilMatrices_Eout = apply(milMatrices_Eout,3,funcionAuxiliar)
etiquetasMilMatrices_Eout = t(etiquetasMilMatrices_Eout)
etiquetasRuidoMilMatrices_Eout = apply(etiquetasMilMatrices_Eout,1,
                                       asignarRuido,porcentaje=10)

Eout_MilMatrices = 0
for(i in 1:dim(milMatrices_Eout)[3]){

  # Aislamos la matriz y las etiquetas a utilizar
  datos = cbind(milMatrices_Eout[,,i],1)
  label = etiquetasRuidoMilMatrices_Eout[i,]
```

```
  Eout_MilMatrices = cbind(Eout_MilMatrices,
                           calcularErrorMalClasificadas(datos,label,resultadosExp1$Pesos[i,]))

}

## Eliminamos la primera componenente dado que es 0
Eout_MilMatrices = Eout_MilMatrices[1,2:dim(Eout_MilMatrices)[2]]
Eout_MilMatrices = mean(Eout_MilMatrices)
print(sprintf("El error Eout para las mil matrices de test es del %f porciento",
              Eout_MilMatrices))
```

```
## [1] "El error Eout para las mil matrices de test es del 48.558900 porciento"
```

```
print(sprintf("La bondad del ajuste es de un %f porciento",100-abs(Eout_MilMatrices-Ein_MilMatrices)))
```

```
## [1] "La bondad del ajuste es de un 98.546900 porciento"
```

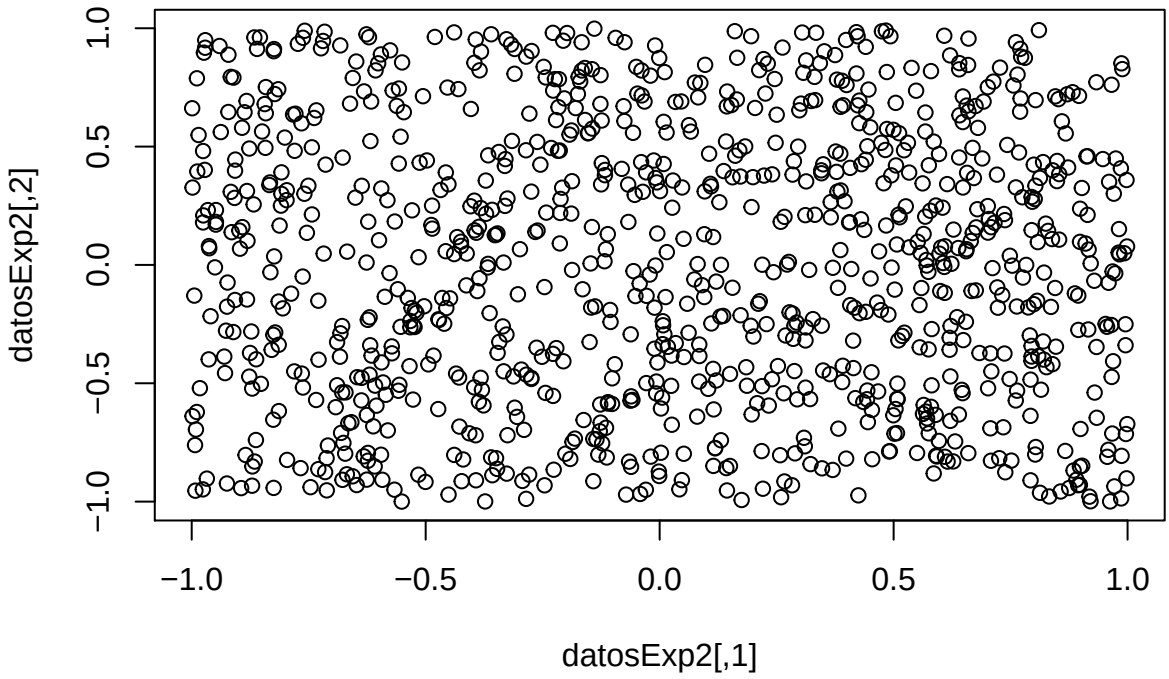
e) Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de Ein y Eout

A vista de los resultados de Ein y Eout, el ajuste puede parecer bueno ya que el Ein y el Eout tienen valores muy cercanos entre si, difieren en apenas 1 unidad de media pero realmente el ajuste es malo ya que el Eout medio se encuentra justo en medio (entre 0 y 1) por lo que no podemos considerar que es un buen ajuste en terminos de Eout.

EXPERIMENTO-2:

a) Ahora vamos a repetir el mismo experimento anterior pero usando caracteristicas no lineales. Ahora usaremos el siguiente vector de características: 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

```
## Apartado a)
## Repetimos el mismo experimento anterior pero ahora con el vector de caracte-
## risticas fi 2(x) = (1, x1, x2, x1x2, (x1)^2, (x2)^2)

## Generamos la muestra de entrenamiento de N=1000
datosExp2=simula_unif(1000,2,c(-1,1))
plot(datosExp2)
```




```
## Definimos la funcion que nos clasificará los puntos en el Experimento 2
f_Exp2=function(x1,x2){

  sign((x1+0.2)^2 + x2^2 -0.6)
}

## Generamos las etiquetas con y sin ruido
etiquetasExp2 = f_Exp2(datosExp2[,1],datosExp2[,2])
etiquetasRuidoExp2 = asignarRuido(etiquetasExp2,10)

## Generamos el vector de caracteristicas completo
datosExp2 = cbind(datosExp2, datosExp2[,1]*datosExp2[,2])
datosExp2 = cbind(datosExp2,datosExp2[,1]^2)
datosExp2 = cbind(datosExp2,datosExp2[,2]^2)
datosExp2_Aux = cbind(1,datosExp2)

## Calculamos el Ein del experimento 2 (primera parte)
pesosExp2=Regress_Lin(datosExp2_Aux,etiquetasRuidoExp2)
Ein_Exp2 = calcularErrorMalClasificadas(datosExp2_Aux,etiquetasRuidoExp2,pesosExp2)
print(sprintf("El error Ein es del %f porciento",Ein_Exp2))
```

```
## [1] "El error Ein es del 13.600000 porciento"
```

b) Al igual que en el experimento anterior repetir el experimento 1000 veces cal- culando 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.

Segunda parte del experimento 2: tal y como hemos hecho para el experimento 1, generamos 1000 matrices 1000x2, asignamos etiquetas, calculamos los pesos y el Ein para posteriormente calcular el Eout con otras 1000 matrices 1000x2 . Repetiremos el siguiente proceso 1000 veces: 1. Generaremos una matriz de entrenamiento y calcularemos sus pesos y Ein. 2. Generamos otra matriz de Test y calcularemos su Eout con los pesos de la matriz de entrenamiento inmediatamente anterior. 3. Los Ein y los Eout los guardamos para devolverlos al

final.

```
EinTotales_Exp2 = 0
EoutTotales_Exp2 = 0

for(i in 1:1000){

  #Calculamos las matrices y etiquetas de entrenamiento (con ruido)
  matrizTrain_Exp2=simula_unif(1000,2,c(-1,1))
  etiquetasTrain_Exp2 = f_Exp2(matrizTrain_Exp2[,1],matrizTrain_Exp2[,2])
  etiquetasTrain_Exp2 = asignarRuido(etiquetasTrain_Exp2,10)

  matrizTrain_Exp2=cbind(1,matrizTrain_Exp2,matrizTrain_Exp2[,1]*matrizTrain_Exp2[,2],
                         matrizTrain_Exp2[,1]^2,matrizTrain_Exp2[,2]^2)

  #Calculamos los pesos de entrenamiento y el Ein de entrenamiento
  pesosTrain = Regress_Lin(matrizTrain_Exp2,etiquetasTrain_Exp2)
  EinTrain_Exp2 = calcularErrorMalClasificadas(matrizTrain_Exp2,
                                               etiquetasTrain_Exp2,pesosTrain)

  EinTotales_Exp2 = cbind(EinTotales_Exp2,EinTrain_Exp2)


  #Calculamos las matrices y etiquetas de Train (con ruido)
  matrizTest_Exp2=simula_unif(1000,2,c(-1,1))
  etiquetasTest_Exp2 = f_Exp2(matrizTest_Exp2[,1],matrizTest_Exp2[,2])
  etiquetasTest_Exp2 = asignarRuido(etiquetasTest_Exp2,10)

  matrizTest_Exp2=cbind(1,matrizTest_Exp2,matrizTest_Exp2[,1]*matrizTest_Exp2[,2],
                         matrizTest_Exp2[,1]^2,matrizTest_Exp2[,2]^2)

  #Calculamos el Eout
  EoutTest_Exp2 = calcularErrorMalClasificadas(matrizTest_Exp2,
                                               etiquetasTest_Exp2,pesosTrain)

  EoutTotales_Exp2 = cbind(EoutTotales_Exp2,EoutTest_Exp2)

}

## Calculamos Ein de las mil matrices
EinTotales_Exp2 = EinTotales_Exp2[1,2:dim(EinTotales_Exp2)[2]]
Ein_MilMatrices_Exp2 = mean(EinTotales_Exp2)

## Calculamos Eout de las mil matrices
EoutTotales_Exp2 = EoutTotales_Exp2[1,2:dim(EoutTotales_Exp2)[2]]
Eout_MilMatrices_Exp2 = mean(EoutTotales_Exp2)

print(sprintf("El Ein del experimento 2 es del %f porciento",Ein_MilMatrices_Exp2))
```

```
## [1] "El Ein del experimento 2 es del 14.196300 porciento"
```

```
print(sprintf("El Eout del experimento 2 es del %f porciento",Eout_MilMatrices_Exp2))
```

```
## [1] "El Eout del experimento 2 es del 14.392100 porciento"
```

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

A la vista de los resultados de Ein y Eout medios, podemos concluir que el ajuste que se realiza utilizando el vector de características fi_2(x) = (1, x1, x2, x1x2, (x1)ˆ2, (x2)ˆ2) es notablemente mejor incluso con los datos que presentan ruido. Esto quiere decir que al añadir mas parametros efectivos, los pesos generados permiten que la regresion lineal considere mas elemenos a ajustar y por tanto sea capaz de clasificar mejor las muestras tanto de entrenamiento como las de test.

A la vista de los resultados de Ein y Eout en cada uno de los experimentos, el modelo que considero mas adecuado es el del experimento 2 ya que cumple en mayor grado con las dos condiciones que hacen posible que se pueda aprender

a) Ein tiende a 0: aunque el valor no sea precisamente muy cercano a 0, si que es menor que el Ein medio del experimento 1, por tanto la bondad del ajuste es mayor.

b) Ein y Eout tienen valores muy parecidos: por la desigualdad de Hoeffding. Cuando se verifican ambas condiciones entonces Eout tiende a 0, por lo tanto en este caso como el Eout medio del experimento 2 es menor que el del experimento 1, el modelo que presenta el experimento 2 es la mejor opcion para ajustar los datos etiquetados con la función que se nos proporciona.