Aprendizaje Automático: Práctica 1

Braulio Vargas López

7 de marzo de 2016

Índice

1. Ejercicio de Generación y Visualización de datos

1.1. Construir una función *lista = simula_unif(N,dim,rango)* que calcule una lista de longitud N de vectores de dimensión dim conteniendo números aleatorios uniformes en el intervalo rango.

```
simula_unif <- function(N=5,dim=2,rango=5:20){
    # Generamos una matriz de datos aletorios distribuidos uniformemente
    # en el rango de valores introducido. Esta matriz tendrá el un número
    # de filas N, y un número de columnas dim.
    num = matrix(runif(N*dim, min=rango[1], max=rango[length(rango)]),
      nrow = N, ncol = dim)
    num
}
```

La función crea una matriz de tamaño N filas y dim columnas. Esta matriz se crea directamente pasándole un vector de datos aleatorios que siguen una distribución uniforme de tamaño $N * dim$, donde el constructor *matrix* se encarga de meterlos en la matriz por columnas.

```
##          [,1]      [,2]
## [1,] 18.15394  8.426853
## [2,] 16.39476  8.262196
## [3,] 14.79633 10.944363
## [4,] 19.61767 19.726716
## [5,] 10.49669  7.431019
```

1.2. Construir una función *lista = simula_gaus(N,dim,sigma)* que calcule una lista de longitud N de vectores de dimensión dim conteniendo números aleatorios gaussianos de media 0 y varianzas dadas por el vector sigma.

```
simula_gaus <- function(N=5,dim=2,sigma=1){
    media = 0
    # Al igual que antes, generamos una matriz,pero esta vez sigue una
    # distribución gaussiana, con media 0, inicializada más arriba y sigma = 1
```

```
    # por defecto. La matriz tendrá N filas y dim columnas.
    num = matrix(rnorm(N*dim, media, sigma), nrow = N, ncol = dim)
    num
}
```

Al igual que antes, generamos una matriz de forma similiar a como lo hacíamos en el apartado anterior, pero usando la función *rnorm*. En este caso, se generan valores aleatorios dentro de la función gaussiana definida por *media = 0* y *sigma*, donde este valor por defecto es 1. Un ejemplo de ejecución es el siguiente.

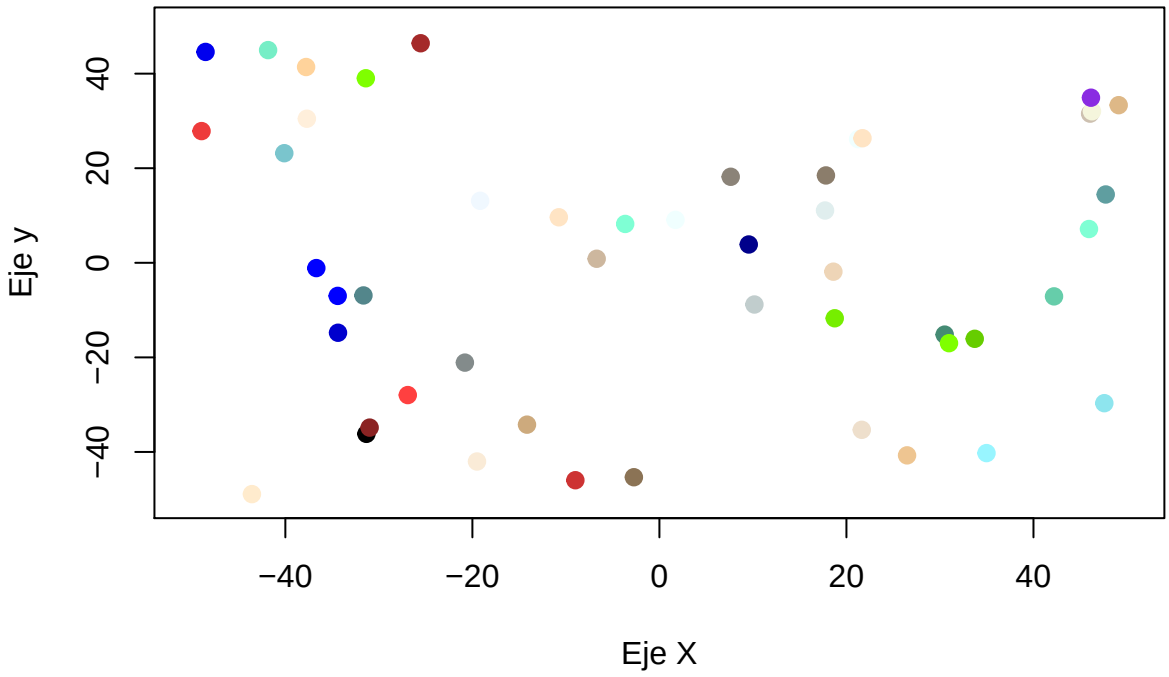
```
##            [,1]       [,2]
## [1,] -0.6372545 -1.1045690
## [2,]  0.6117197  0.1938106
## [3,] -0.1384711 -0.8665786
## [4,] -2.1871423  0.1497181
## [5,] -1.7083902 -1.1566523
```

1.3. Suponer N = 50, dim = 2, rango = [-50,+50] en cada dimensión. Dibujar una gráfica de la salida de la función correspondiente

```
pintar <- function(puntos, funcion, intervalo, nombreGrafica,
    colores = colors(), verFuncion = FALSE){
    if(verFuncion){
        x <- y <- seq(range(puntos[,1])[1],range(puntos[,1])[2],length=100)
        z <- outer(x,y,funcion)
        contour(
            x=x, y=x, z=z,
            levels=0, las=1, drawlabels=FALSE, lwd=3, xlab="Eje X",
            ylab="Eje y",main = nombreGrafica
        )
    }
    else{
        plot(intervalo, intervalo, xlab="Eje X", ylab="Eje y", type="n",
        main = nombreGrafica)
    }

    points(puntos, col = colores, pch=19, lwd = 2)
}
```

```
pintaUnif <- function(N = 50, dim = 2, rangoV = -50:50){
  pintar(simula_unif(N,dim,rangoV), intervalo = rangoV,
    nombreGrafica = "EJERCICIO 3", verFuncion = F)
}
```

```
pintaUnif()
```

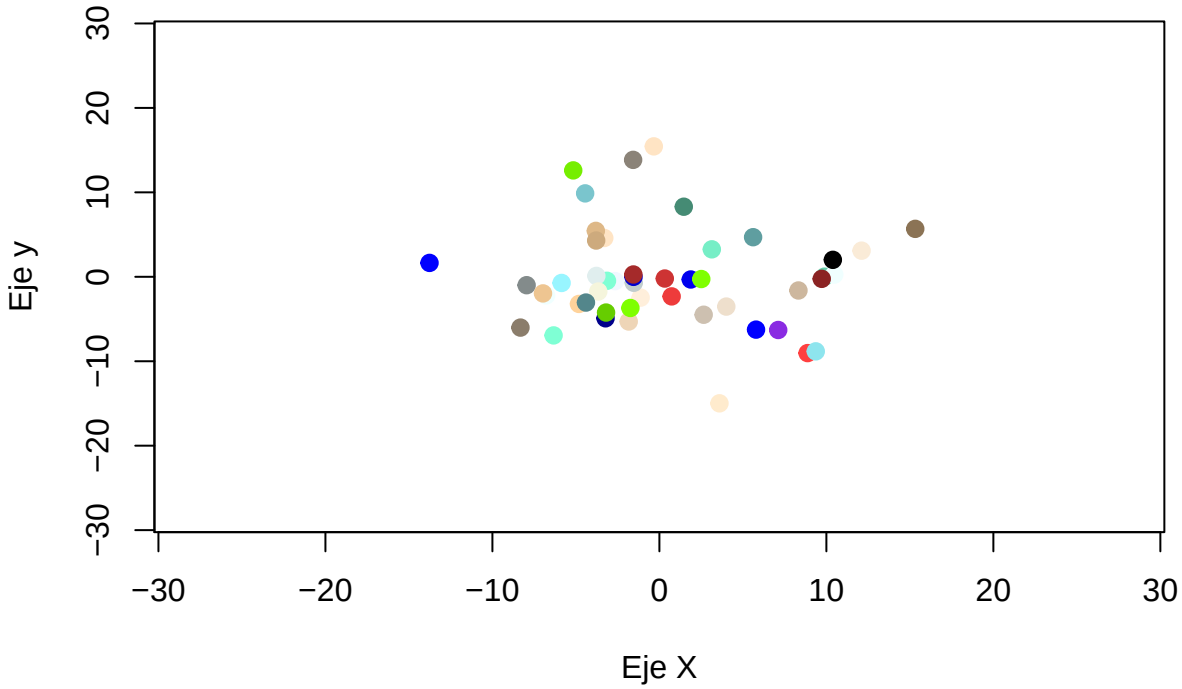


En la gráfica podemos ver cómo los puntos que hemos obtenido, están más o menos distribuidos de forma uniforme dentro de los intervalos que se ha establecido. Cada uno de los puntos está asociado a un color.

1.4. Suponer N = 50, dim = 2, rango = [5,7] en cada dimensión. Dibujar una gráfica de la salida de la función correspondiente.

```
pintaGauss <- function(N = 50, dim = 2, sigma = 5:7){
    pintar(simula_gaus(N,dim,sigma), intervalo = (sigma[3]*(-4)):(sigma[3]*4),
        nombreGrafica = "EJERCICIO 4", verFuncion = F)
}
```

EJERCICIO 4



En este caso, a diferencia de la gráfica anterior, los puntos se concentran en la zona central de la gráfica, donde es más probable que aparezcan los valores, mientras que conforme más se acerca a los valores extremos de la gráfica, es más difícil que se encuentren estos valores.

1.5. Construir la función *v=simula_recta(intervalo)* que calcula los parámetros, v=(a,b) de una recta aleatoria, y = ax + b, que corte al cuadrado [-50,50] x [-50,50] (Ayuda: Para calcular la recta simular las coordenadas de dos puntos dentro del cuadrado y calcular la recta que pasa por ellos)

```
simula_recta <- function(dim = -50:50, punto_1 = c(sample(dim, 2)),
    punto_2 = c(sample(dim, 2)), verPunto = T){
    # Por defecto, obtenemos dos puntos aleatorios dentro del intervalo,
    # pero en caso de que
    if(verPunto){
        print("#####Punto 1")
        print(punto_1)

        print("#####Punto 2")
        print(punto_2)
    }

    # Para calcular la recta que pasa por dos puntos, usaremos la expresión:
    #
    #          x - x1       y - y1
    #        --------- = ----------
    #         x2 - x1     y2 - y1
    #
```

```
    # Con lo que si despejamos obtenemos
    #
    #   => (y2-y1)x - x1(y2-y1) = y(x2-x1) - y1(x2-x1)
    #   => y = (y2-y1)x/(x2-x1) - (x1(y2-y1)+y1(x2-x1))/(y2-y1)

    punto_aux = c(punto_2[1]-punto_1[1], punto_2[2]-punto_1[2])

    ecuacion = c(punto_aux[2]/punto_aux[1],
        (((-1*punto_1[1]*punto_aux[2])+(punto_1[2]*punto_aux[1]))/punto_aux[1]))
    ecuacion
}
```

En este ejercicio obtengo dos puntos dentro del intervalo [-50:50] de forma aleatoria, tanto la componente X, como la componente Y del punto. Tras esto, uso la ecuación de la recta que pasa por dos puntos para calcular los parámetros de la recta.

$$\frac{X - x_1}{x_2 - x_1} = \frac{Y - y_1}{y_2 - y_1}$$

Despejando la ecuación obtenemos lo siguiente:

$$(y_2 - y_1) \cdot X - x_1 \cdot (y_2 - y_1) = Y \cdot (x_2 - x_1) - y_1 \cdot (x_2 - x_1)$$
$$Y = \frac{(y2 - y1)}{(x2 - x1)} \cdot X - \frac{(x1(y2 - y1) + y1(x2 - x1))}{(y2 - y1)}$$

Un ejempo de ejecución de esta función es el siguiente

```
## [1] "#####Punto 1"
## [1] -25  16
## [1] "#####Punto 2"
## [1]  38 -32
```

1.6. Generar una muestra 2D de puntos usando *simula_unif()* y etiquetar la muestra usando el signo de la función f (x, y) = y - ax - b de cada punto a una recta simulada con *simula_recta()*. Mostrar una gráfica con el resultado de la muestra etiquetada junto con la recta usada para ello.

```
funciones = c(function(x,y) y - ecu[1]*x - ecu[2],
              function(x,y) (x - 10)^2 + (y - 20)^2 - 400,
              function(x,y) 0.5*(x + 10)^2 + (y - 20)^2 - 400,
              function(x,y) 0.5*(x - 10)^2 - (y + 20)^2 - 400,
              function(x,y) y - 20*x^2 - 5*x + 3,
              function(x,y) y - rectaEJ7[1]*x - rectaEJ7[2],
              function(x,y) x^2+y^2 - 25
              )

funciones_eval = c(function(eval) eval[2] - ecu[1]*eval[1] - ecu[2],
                   function(eval) (eval[1] - 10)^2 + (eval[2] - 20)^2 - 400,
                   function(eval)  0.5*(eval[1] + 10)^2 + (eval[2] - 20)^2 - 400,
                   function(eval) 0.5*(eval[1] - 10)^2 - (eval[2] + 20)^2 - 400,
```

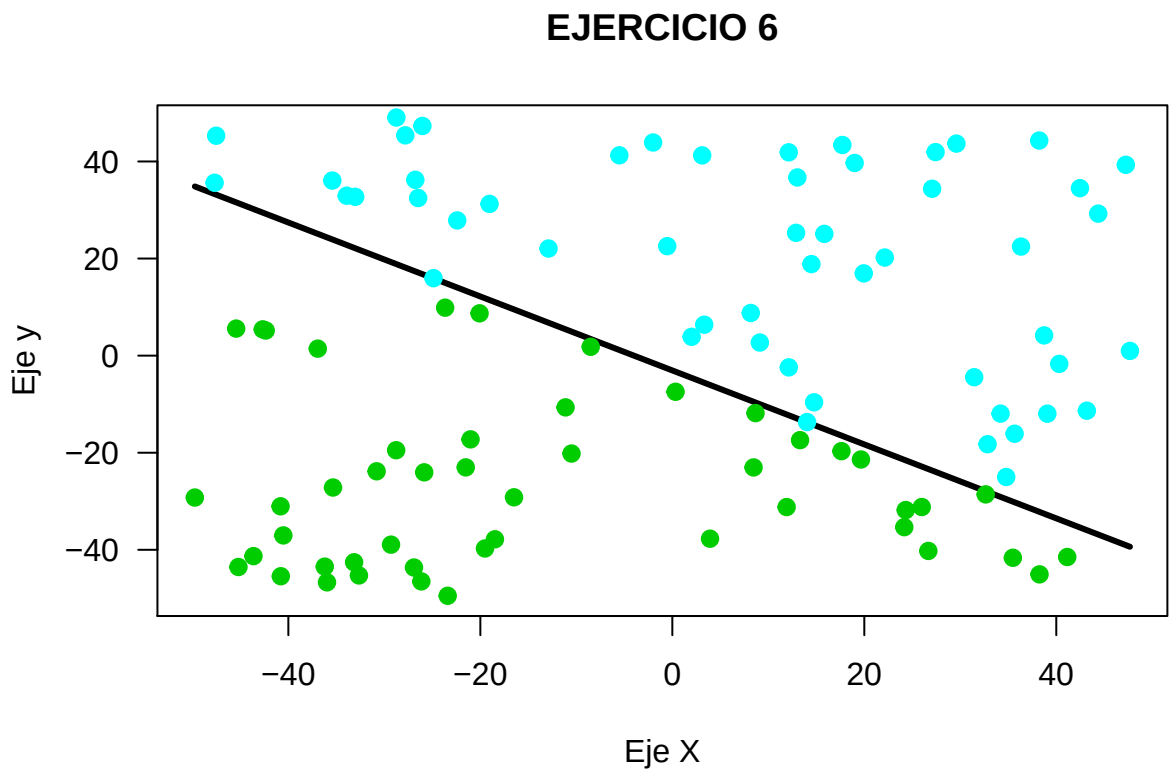
```
                    function(eval) eval[2] - 20*eval[1]^2 - 5*eval[1] + 3,
                    function(eval) eval[2] - rectaEJ7[1]*eval[1] - rectaEJ7[2],
                    function(eval) eval[1]^2+eval[2]^2 - 25
                    )
```

```
evaluaPuntos <- function(puntosAEvaluar, intervalo, apartado){
    etiquetas = apply(X=puntosAEvaluar, FUN=funciones_eval[[apartado]], MARGIN=1)
    sign(etiquetas)
}
```

Para evaluar los puntos de una función, se declara un vector de funciones que lo que recibe es un array de dos posiciones, y la evalua siguiendo la función original. Este array es *funciones_eval* y las funciones originales se guardan en el array *funciones* que se usará para pintar las gráficas más adelante.

Esta función lo que hace es aplicarle a la matriz de puntos con *apply* la función que deseamos evaluar, obteniendo un vector de etiquetas. Este vector de etiquetas lo que hacemos es obtener el signo de los valores del array y sumarle 4 para obtener un color, que pasaremos como etiqueta.

```
range = -50:50
puntos = simula_unif(N=100,dim=2,rango=range)

etiquetas6 = evaluaPuntos(puntos, range, apartado = 1)
pintar(puntos, funciones[[1]], range, "EJERCICIO 6",
    colores =  etiquetas6 + 4, verFuncion = T)
```



A continuación, declaramos el rango en el que se sacarán puntos, y obtenemos los puntos en el array *puntos*. Tras esto, llamamos a la funcion *pintar* con los puntos, la función que usaremos (en este caso la función de la recta que obtendremos con la función *simula_recta*), el intervalo, el nombre que le pondremos al gráfico, y los colores que se le pondrán a los puntos serán aquellos que obtendremos como etiquetas al evaluar los puntos

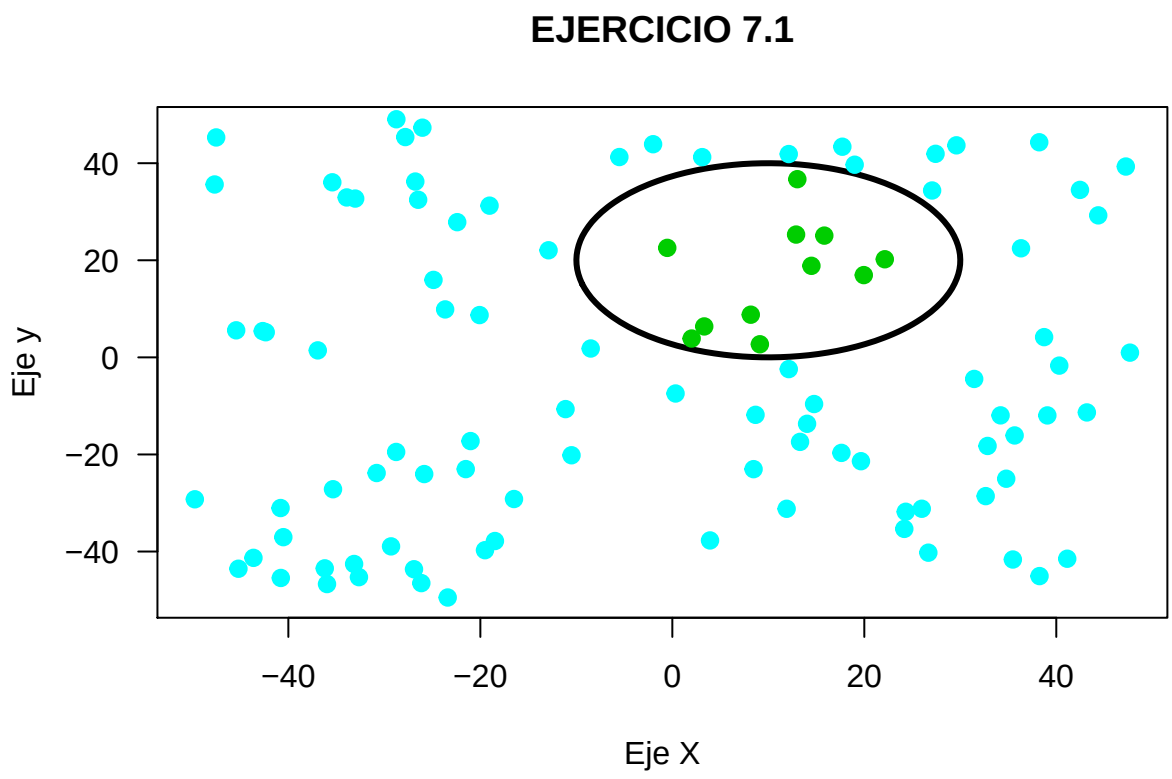
con *evaluaPuntos*. Además, el parámetro *verFuncion* estará a *TRUE* para que pinte la gráfica de la recta, usando la función de la recta almacenada en *funciones*. Esto da como resultado el gráfico visto anteriormente.

1.7. Usar la muestra generada en el apartado anterior y etiquetarla con +1,-1 usando el signo de cada una de las siguientes funciones: *f (x, y) = (x - 10)ˆ2 + (y - 20)ˆ2 - 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 - 20x2 - 5x + 3*. Visualizar el resultado del etiquetado de cada función junto con su gráfica y comparar el resultado con el caso lineal. ¿Qué consecuencias extrae sobre la forma de las regiones positiva y negativa?

Para etiquetar la muestra obtenida en el apartado anterior, lo que haremos será usar la función *evaluaPuntos* creada antes, y lo único que se hará será llamar a la función *pintar* con la función correspondiente y los colores serán el resultado de la función *evaluaPuntos*.

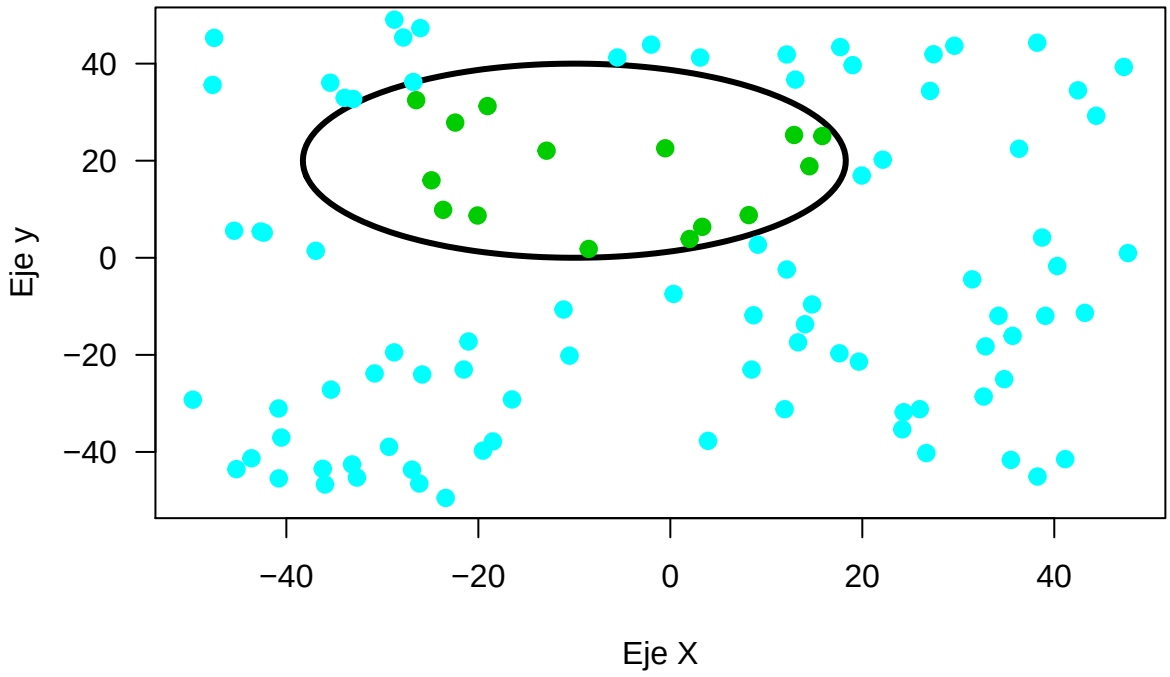
El resultado para cada una de las funciones se puede ver a continuación.

```
etiquetas7_1 = evaluaPuntos(puntos, range, apartado = 2)
pintar(puntos, funciones[[2]], range, "EJERCICIO 7.1",
    colores = etiquetas7_1+4, verFuncion = T)
```



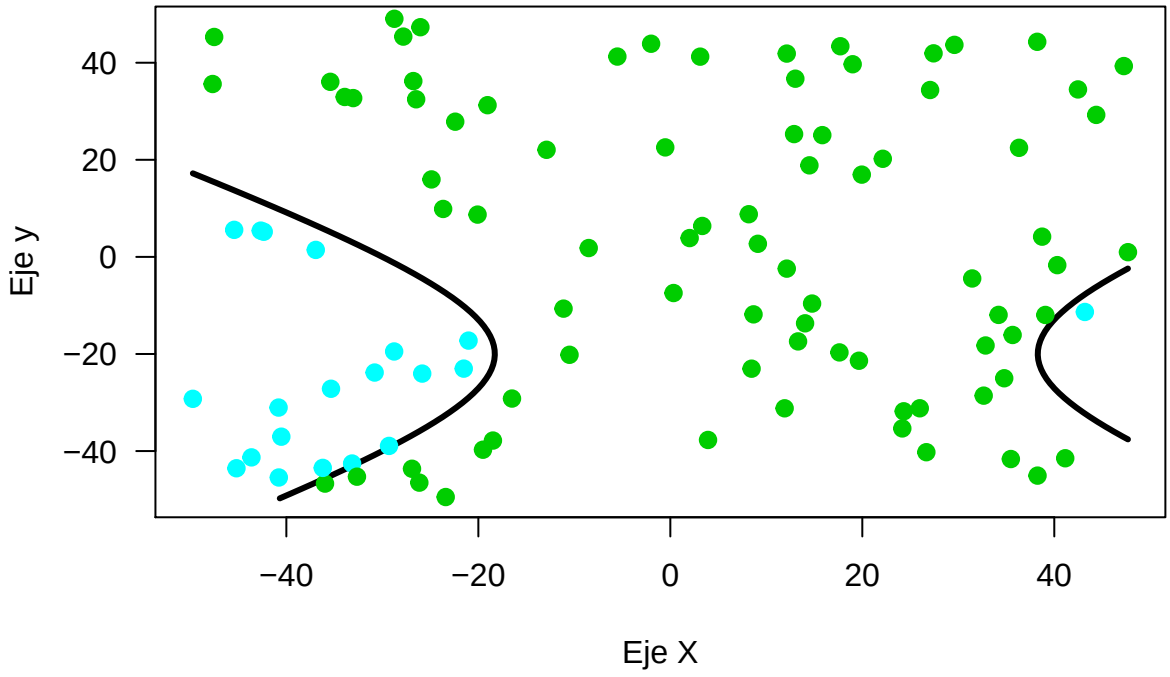

```
etiquetas7_2 = evaluaPuntos(puntos, range, apartado = 3)
pintar(puntos, funciones[[3]], range, "EJERCICIO 7.2",
    colores = etiquetas7_2+4, verFuncion = T)
```

EJERCICIO 7.2

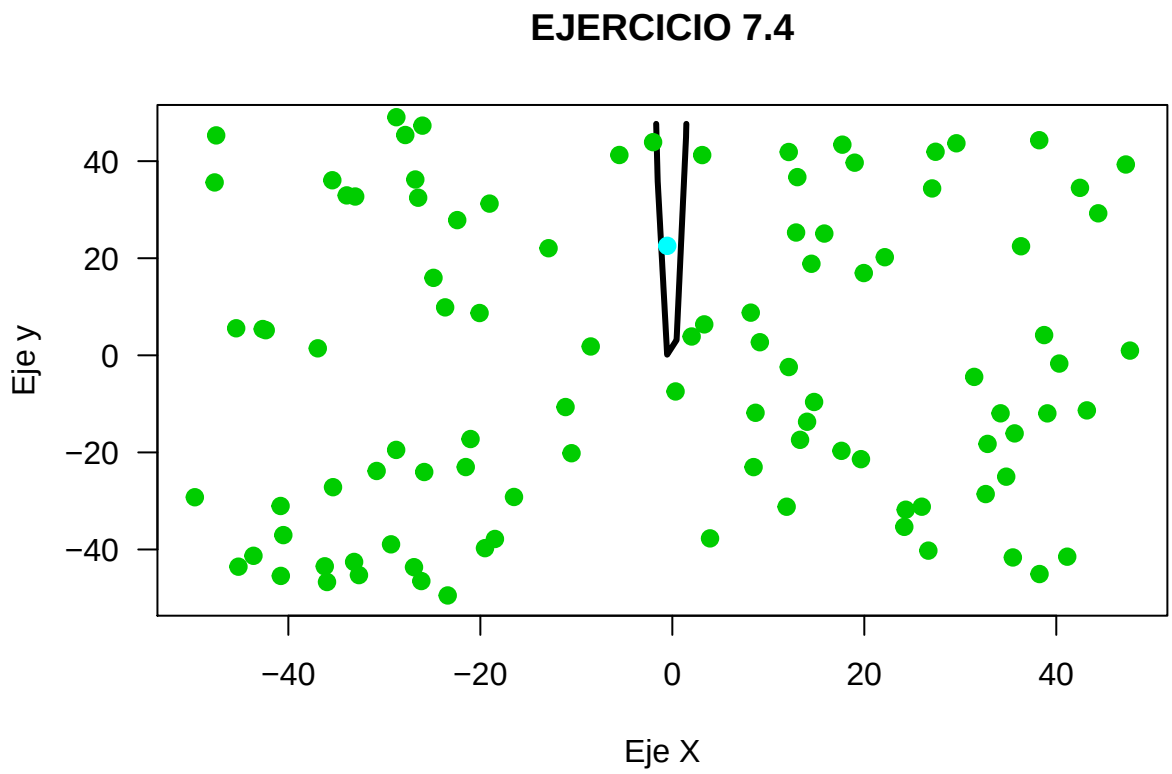



```
etiquetas7_3 = evaluaPuntos(puntos, range, apartado = 4)
pintar(puntos, funciones[[4]], range, "EJERCICIO 7.3",
    colores = etiquetas7_3+4, verFuncion = T)
```

EJERCICIO 7.3



```
etiquetas7_4 = evaluaPuntos(puntos, range, apartado = 5)
pintar(puntos, funciones[[5]], range, "EJERCICIO 7.4",
    colores = etiquetas7_4+4, verFuncion = T)
```




```
etiquetasFunc = list(etiquetas6, etiquetas7_1,
    etiquetas7_2, etiquetas7_3, etiquetas7_4)
```

En estas gráfica, se puede ver cómo a partir de una función (una circunferencia en el apartado 1, una elipse en el apartado 2, una hipérbola y una parábola en los siguientes), podemos clasificar puntos sin tener que usar un modelo lineal, que en estos casos, no puede clasificar bien los puntos, con lo que no se podría aprender nada.

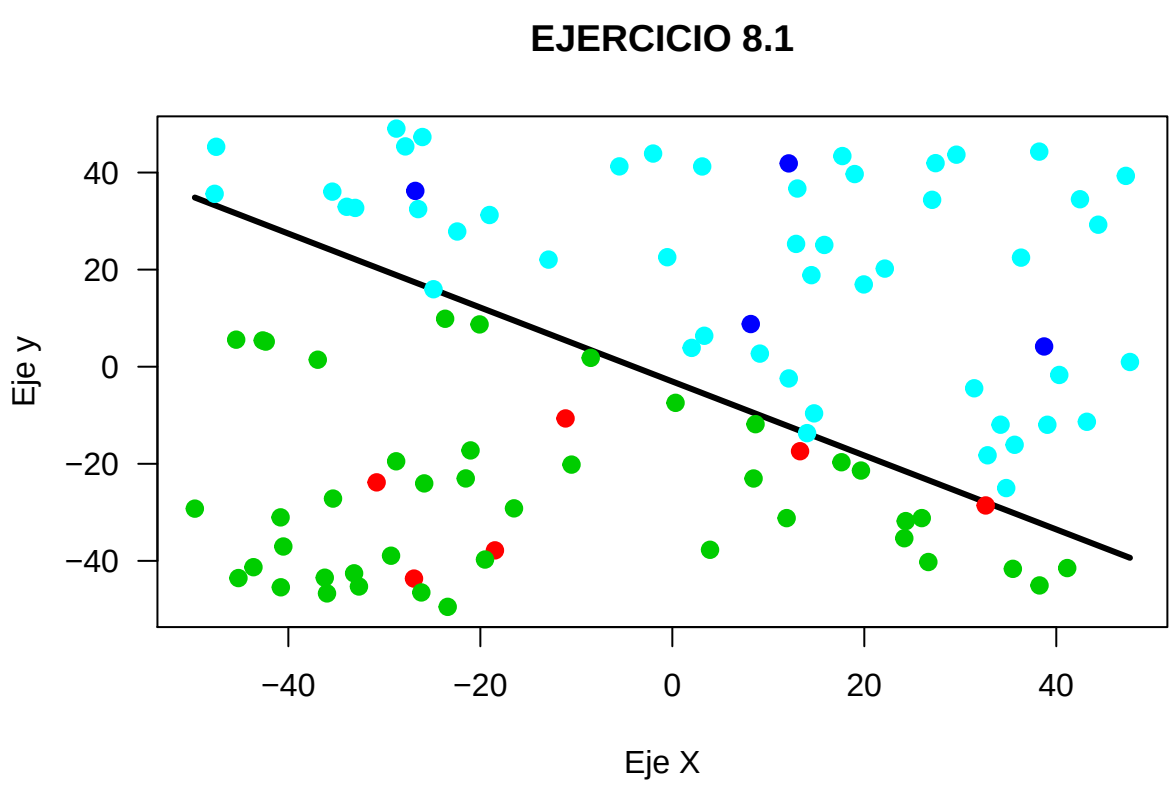
1.8. Considerar de nuevo la muestra etiquetada en el apartado.6. Modifique las etiquetas de un 10 % aleatorio de muestras positivas y otro 10 % aleatorio de negativas.# 1. Visualice los puntos con las nuevas etiquetas y la recta del apartado 6. 2. En una gráfica aparte visualice de nuevo los mismos puntos pero junto con las funciones del apartado 7. Observe las gráficas y diga que consecuencias extrae del proceso de modificación de etiquetas en el proceso de aprendizaje.

Para cambiar las etiquetas que obtenemos de los puntos, se ha creado la función *cambiarEtiqueta* que recibe las etiquetas obtenidas para una clasificación, y el porcentaje que queremos cambiar de ellas. Para cambiar estas etiquetas, obtiene un array de números aleatorios desde 1 hasta la longitud del array de etiquetas, con una longitud igual a la longitud de las etiquetas por el tanto por ciento del argumento. Con este array de números aleatorios, accedemos al array de etiquetas y le restamos uno a esos índices.

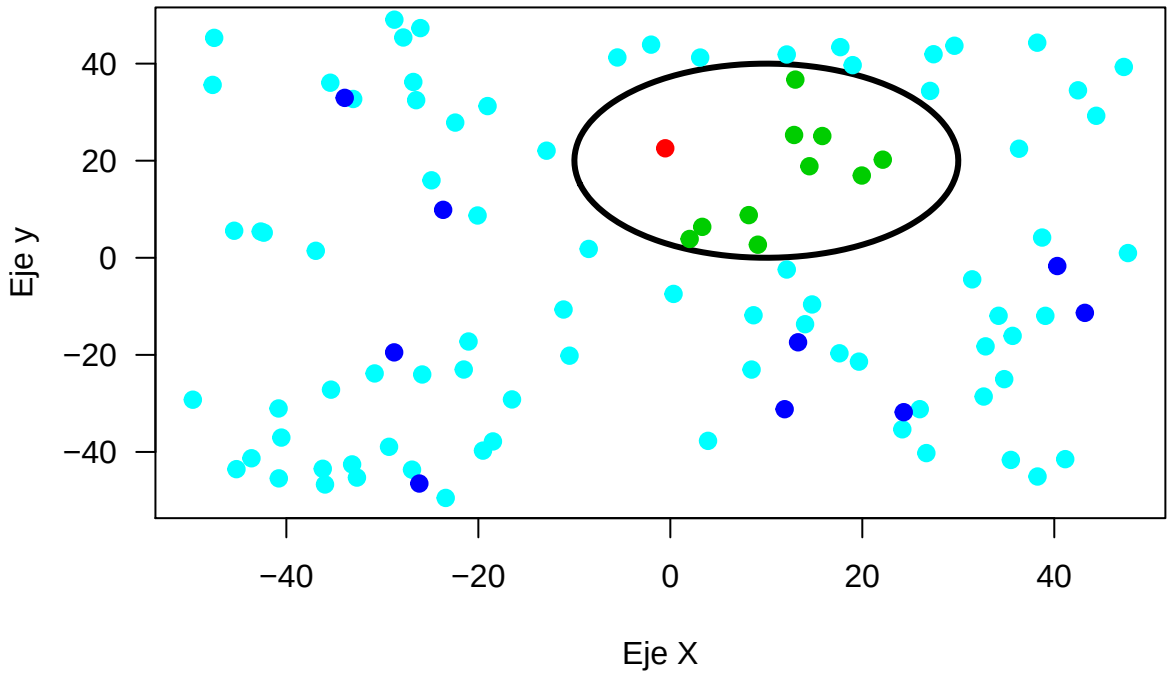
```r
cambiarEtiqueta <- function(etiquetas, porcentaje = 0.1){
    i = 1
    num = sample(1:length(etiquetas),length(etiquetas)*0.1)
    etiquetas[num] = -etiquetas[num]
    etiquetas
}

# Función para asignar colores distintos a las etiquetas bien clasificasdas
# y a las mal clasificadas
asignaColorEtiquetasErroneas <- function(etiquetasOriginales, etiquetasErroneas){
    coloresEtiquetas = etiquetasOriginales
    coloresEtiquetas[coloresEtiquetas != etiquetasErroneas] =
        coloresEtiquetas[coloresEtiquetas != etiquetasErroneas] -1
    coloresEtiquetas + 4
}
```

El resultado de modificar las etiquetas lo podemos ver en las siguientes gráficas.

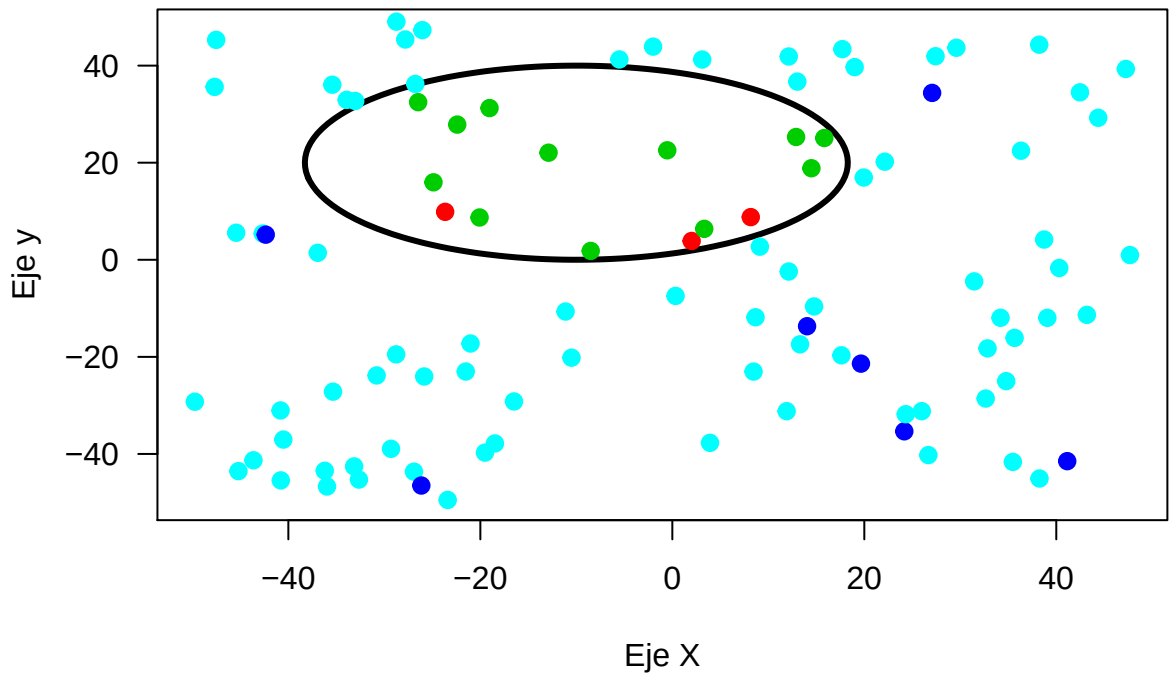
```r
etiquetasErroneas8_1 = cambiarEtiqueta(etiquetas6)

pintar(puntos, funciones[[1]], range, "EJERCICIO 8.1",
    colores = asignaColorEtiquetasErroneas(etiquetas6,etiquetasErroneas8_1),
    verFuncion = T)
```




```r
etiquetasErroneas8_2 = cambiarEtiqueta(etiquetas7_1)

pintar(puntos, funciones[[2]], range, "EJERCICIO 8.2",
    colores = asignaColorEtiquetasErroneas(etiquetas7_1,etiquetasErroneas8_2),
    verFuncion = T)
```

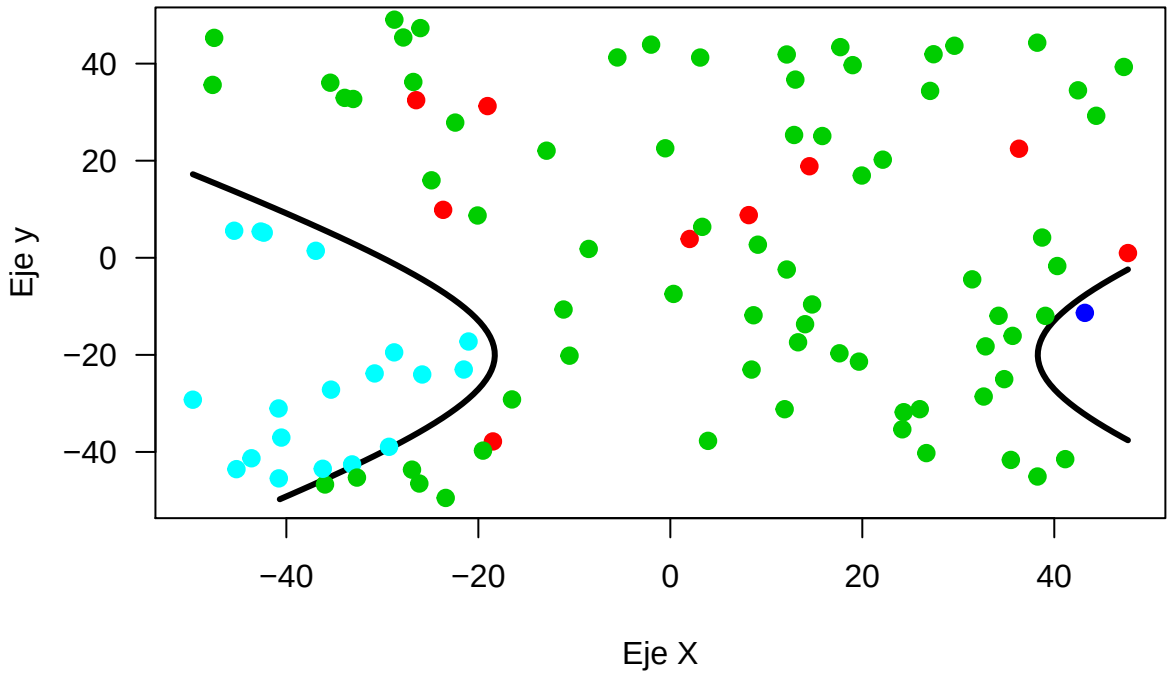



```
etiquetasErroneas8_3 = cambiarEtiqueta(etiquetas7_2)

pintar(puntos, funciones[[3]], range, "EJERCICIO 8.3",
    colores = asignaColorEtiquetasErroneas(etiquetas7_2,etiquetasErroneas8_3),
    verFuncion = T)
```

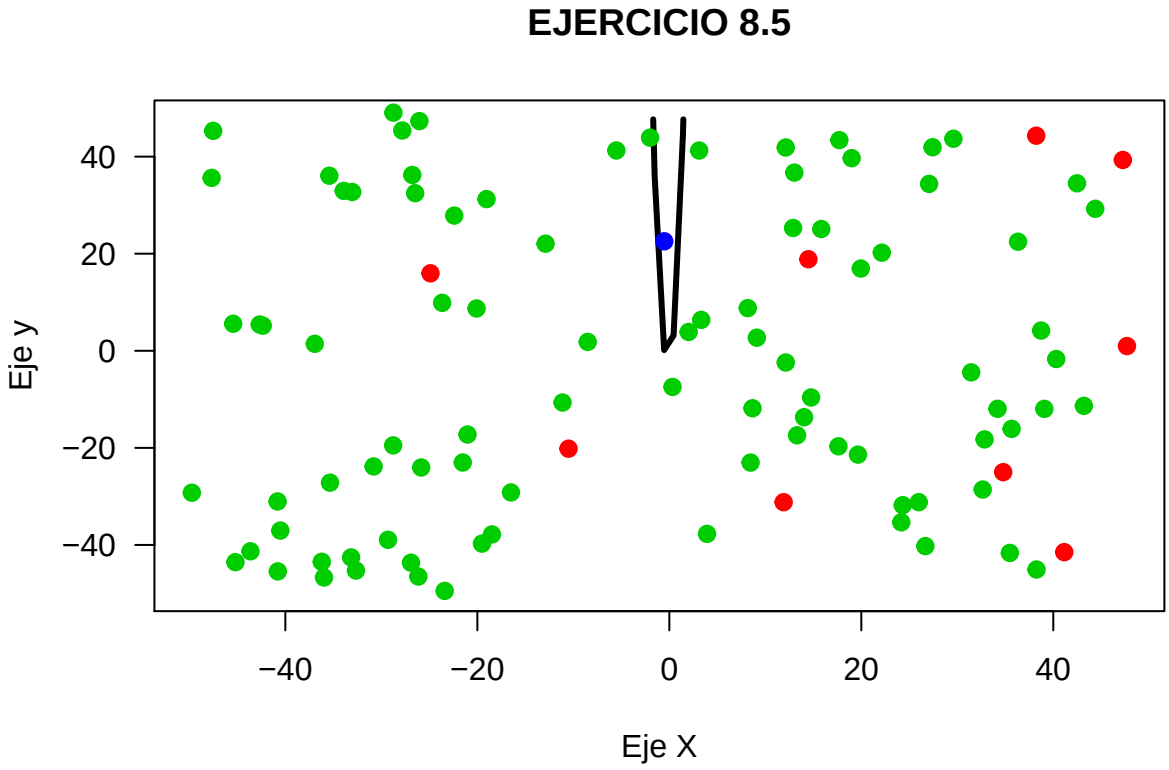



```
etiquetasErroneas8_4 = cambiarEtiqueta(etiquetas7_3)

pintar(puntos, funciones[[4]], range, "EJERCICIO 8.4",
    colores = asignaColorEtiquetasErroneas(etiquetas7_3,etiquetasErroneas8_4),
    verFuncion = T)
```

EJERCICIO 8.4




```
etiquetasErroneas8_5 = cambiarEtiqueta(etiquetas7_4)

pintar(puntos, funciones[[5]], range, "EJERCICIO 8.5",
    colores = asignaColorEtiquetasErroneas(etiquetas7_4,etiquetasErroneas8_5),
    verFuncion = T)
```



Con esta modificación, vemos que los puntos no son claramente separables, con lo que las posibilidades de aprendizaje se reducen.

2. Ejercicio de Ajuste del Algoritmo Perceptron

2.1. Implementar la función *sol = 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 salida *sol* devuelve los coeficientes del hiperplano.

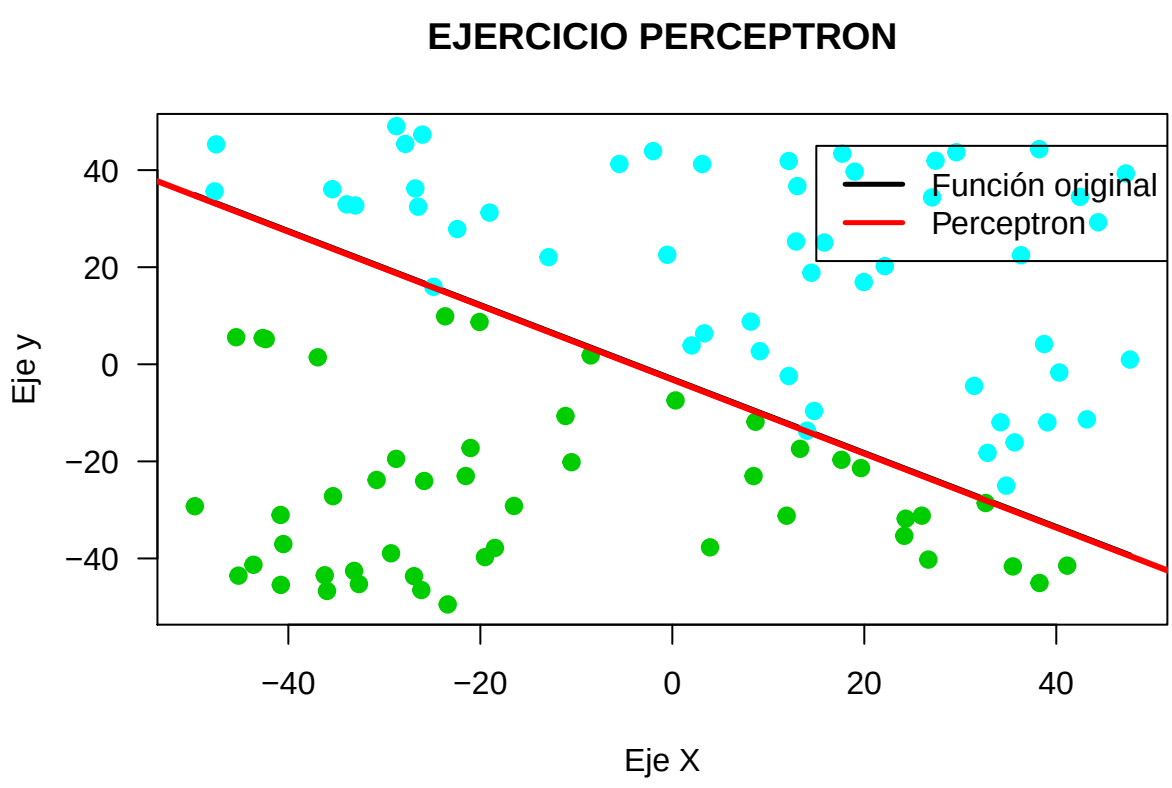
```
PLA <- function(datos, label, max_iter = 3000, vini = c(0,0,0), verRecorrido = F){

    # Añadimos un 1 por la izquierda a la matriz de puntos para que los puntos
    # de la matriz sean (x0, x1, x2) donde x1 será un 1, y x1 y x2 las
    # coordenadas del punto.
    datos = cbind(rep(1, nrow(datos)), datos)
    i = 1 # Contador para saber el número de iteraciones
    converge = FALSE # Variable para saber si el perceptron cambia o no. Es decir,
                     # cuando deje de cambiar, el algoritmo para

    # Empezamos el bucle del perceptron
    while ((i <= max_iter) & (!converge)){
        # while (i <= max_iter){
```

```
        cambiado = FALSE
        # iteramos sobre los puntos de cada dato
        for(j in 1:nrow(datos)){
            # Calculamos el signo en función de los pesos realizando el producto
            # del punto y el vector de pesos
            signo = sign(datos[j,]%*% vini)

            # En caso de que el signo sea 0 por la función sign, se pone a 1
            if (signo == 0){
                signo = signo + 1
            }

            # Cuando las etiquetas no coinciden, se actualiza el vector
            # de pesos y se cambia la variable "cambiado" a true para indicar
            # que se ha equivocado y por lo tanto no converge
            if(label[j] != signo){
                cambiado = TRUE
                vini = vini + datos[j,]*label[j]
                if(verRecorrido)
                  abline(a=(-vini[1]/vini[3]),b=(-vini[2]/vini[3]), col="grey");
            }
        }

        i = i + 1 # incrementamos el contador de iteraciones

        if(!cambiado){
            # en caso de que no se "equivoque" en ningún punto, decimos que
            # el perceptron ha convergido y finaliza el bucle
            converge = TRUE
        }
    }

    # Se devuelve la recta que ha calculado el perceptron normalizada
    # y el número de iteraciones hechas por el algoritmo
    pla_result = c((-vini[2]/vini[3]),(-vini[1]/vini[3]), i)
    pla_result
}
```

Un ejemplo de ejecución para los valores del apartado 6 del ejercicio anterior es el siguiente, donde pprimero se genera el gráfico y los puntos con las etiquetas, se genera una leyenda para el gráfico y después se dibuja con *abline* el resultado del perceptron.

```
pintarPLA <- function(puntosAPintar, rango, apartado, nombreGrafica,
    verF = T, verR = F, verLeyenda = F, leyenda, coloresLeyenda,
    usarPocket = F){


    pintar(puntosAPintar, funciones[[apartado]], rango, nombreGrafica,
           colores = (etiquetasFunc[[apartado]] + 4), verFuncion = verF)

    if(verLeyenda){
        # Leyenda de la función
        legend(x=15,y=45,legend=leyenda, lty=c(1,1),
```

```
            lwd=c(2.5,2.5),col=coloresLeyenda)
    }

    if(!usarPocket)
        perceptron = PLA(datos = puntos, label = etiquetasFunc[[apartado]],
            verRecorrido = verR)
    else{
        perceptron = PocketPLA(datos = puntos, label = etiquetasFunc[[apartado]],
            verRecorrido = verR)

        print("ERROR OBTENIDO CON PLAPOCKET")
        print(perceptron[4])
    }

    abline(a=perceptron[2],b=perceptron[1], col="red", lwd=3)
}

pintarPLA(puntos, range, apartado = 1, nombreGrafica = "EJERCICIO PERCEPTRON",
    leyenda = c("Función original","Perceptron"), verLeyenda = T,
    coloresLeyenda=c("black","red"))
```



La función *pintarPLA* recibe un los elementos necesarios para pintar la gráfica, los puntos, la función original y el resultado del perceptrón. También podemos configurar una leyenda para la función, y establecer si queremos ver el recorrido del perceptrón, cosa que se utilizará en los siguientes apartados.

Esta función también tiene la opción de utilizar el algoritmo del perceptrón versión *pocket* en vez de la versión clásica. Para ello, el parámetro *usarPocket* debe valer *TRUE*. Este parámetro se usará más adelante

2.2. Ejecutar el algoritmo PLA con los valores simulados en apartado.6 del ejercicio.4.2, inicializando el algoritmo con el vector cero y con vectores de números aleatorios en [0, 1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado.

```
iteraciones = vector(length = 10)
aux = c(0,0,0,0,0,1,0,1,0,0,1,1,1,0,0,1,0,1,1,1,0,1,1,1,0,0,1,1,0,1)
pesosM = matrix(aux, nrow = 10, ncol = 3, byrow=T)

for(i in 1:10){
    perceptron = PLA(datos = puntos, label = etiquetas6, vini=pesosM[i,])
    iteraciones[i] = perceptron[3]
}

tabla = cbind(pesosM, iteraciones)
print(tabla)
```

```
##             iteraciones
##  [1,] 0 0 0         285
##  [2,] 0 0 1         271
##  [3,] 0 1 0         274
##  [4,] 0 1 1         296
##  [5,] 1 0 0         256
##  [6,] 1 0 1         288
##  [7,] 1 1 0         274
##  [8,] 1 1 1         277
##  [9,] 0 0 1         271
## [10,] 1 0 1         288
```

Es importante elegir bien los pesos de la función, ya que puede hacer variar en mucho, el número de iteraciones que necesita el perceptrón para converger y finalizar el algoritmo dando un resultado. En la tabla que se genera anteriormente se pueden ver las diferencias en el número de iteraciones dándole al vector inicial unos pesos u otros.

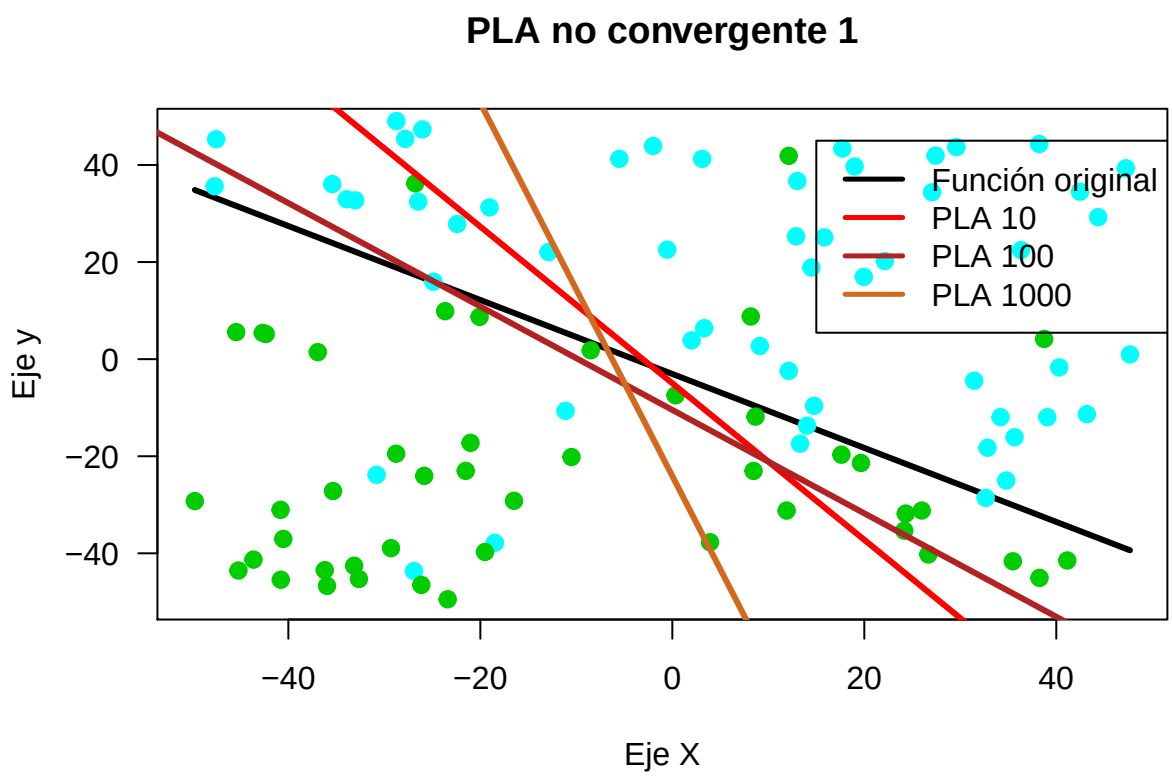
2.3. Ejecutar el algoritmo PLA con los datos generados en el apartado.8 del ejercicio.4.2, usando valores de 10, 100 y 1000 para *max_iter*. Etiquetar los datos de la muestra usando la función solución encontrada y contar el número de errores respecto de las etiquetas originales. Valorar el resultado.

Para realizar este ejercicio, he creado una lista que contiene las etiquetas erroneas del apartado 8 del ejercicio 5.3.3, junto con la siguiente función, que recibe el apartado en el que se encuentra la función y el rango. Tras esto, genera el gráfico inicial según el apartado que sea, usando la función *pintar*. Tras esto, añade una leyenda y con un bucle *for*, ejecuta el *PLA* con un *max_iter* con valores 10, 100 y 1000 iteraciones. Estas iteraciones se calculan de forma procedural dentro del bucle, gracias al uso de *10^i*.

```
etiqErr = list(etiquetasErroneas8_1,etiquetasErroneas8_2,etiquetasErroneas8_3,
    etiquetasErroneas8_4,etiquetasErroneas8_5)

funcionesNoConvergentes <- function(apartado, nombre, rango=range){
    # Se pinta el gráfico inicial, junto con la función original, junto con la
```

```
    # leyenda del gráfico
    pintar(puntos, funciones[[apartado]], rango, nombreGrafica=nombre,
           colores = (etiqErr[[apartado]] + 4), verFuncion = T)

    legend(x=15,y=45,legend=c("Función original","PLA 10","PLA 100","PLA 1000"),
        lty=c(1,1),lwd=c(2.5,2.5,2.5,2.5),
        col=c("black","red","firebrick","chocolate"))

    coloresEj533 = c("red","firebrick","chocolate")

    for(i in 1:3){
        plaRecta = PLA(datos = puntos, label = etiqErr[[apartado]], max_iter = 10^i)
        abline(a=plaRecta[2],b=plaRecta[1], col=coloresEj533[i], lwd=3)
    }
}
```

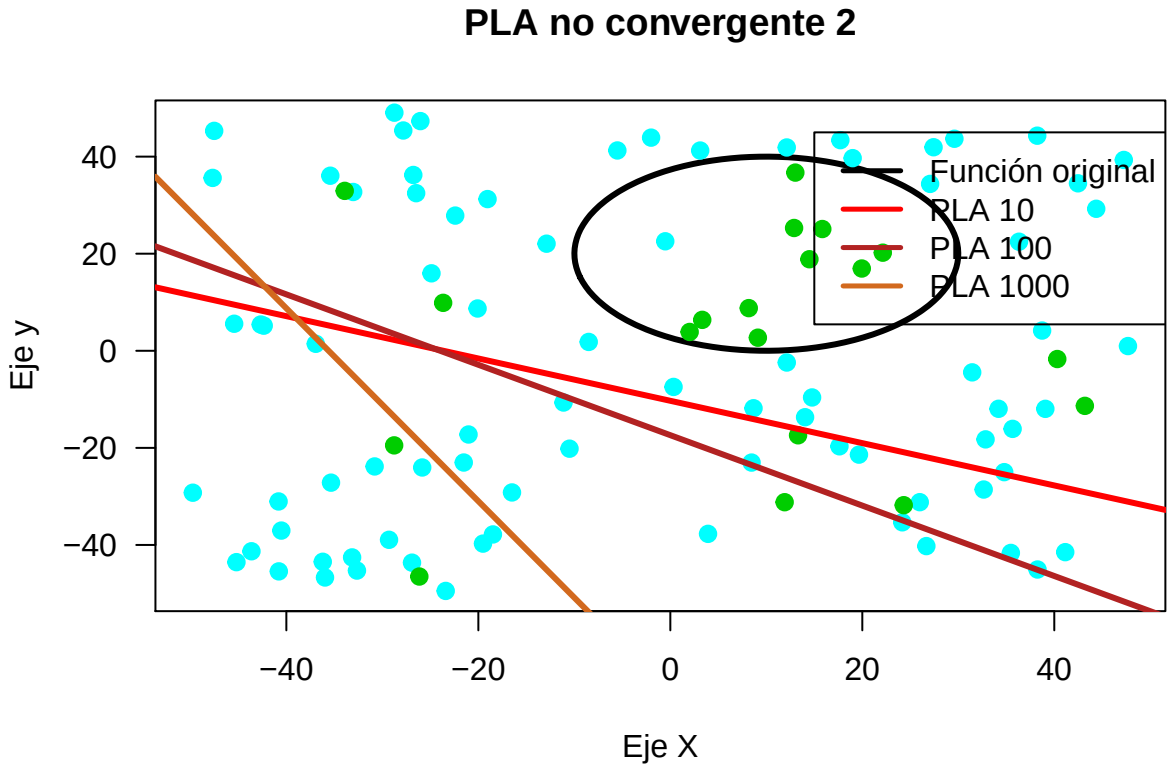
```
funcionesNoConvergentes(apartado = 1, nombre = "PLA no convergente 1")
```



Como se puede ver, en ninguno de los casos el algoritmo es incapaz de converger ni de separar los datos, ya que los datos no son linealmente separables, por eso, por más iteraciones que haga el algoritmo, no es capaz de converger.

2.4. Repetir el análisis del punto anterior usando la primera función del apartado.7 del ejercicio.4.2

```
funcionesNoConvergentes(apartado = 2, nombre = "PLA no convergente 2")
```

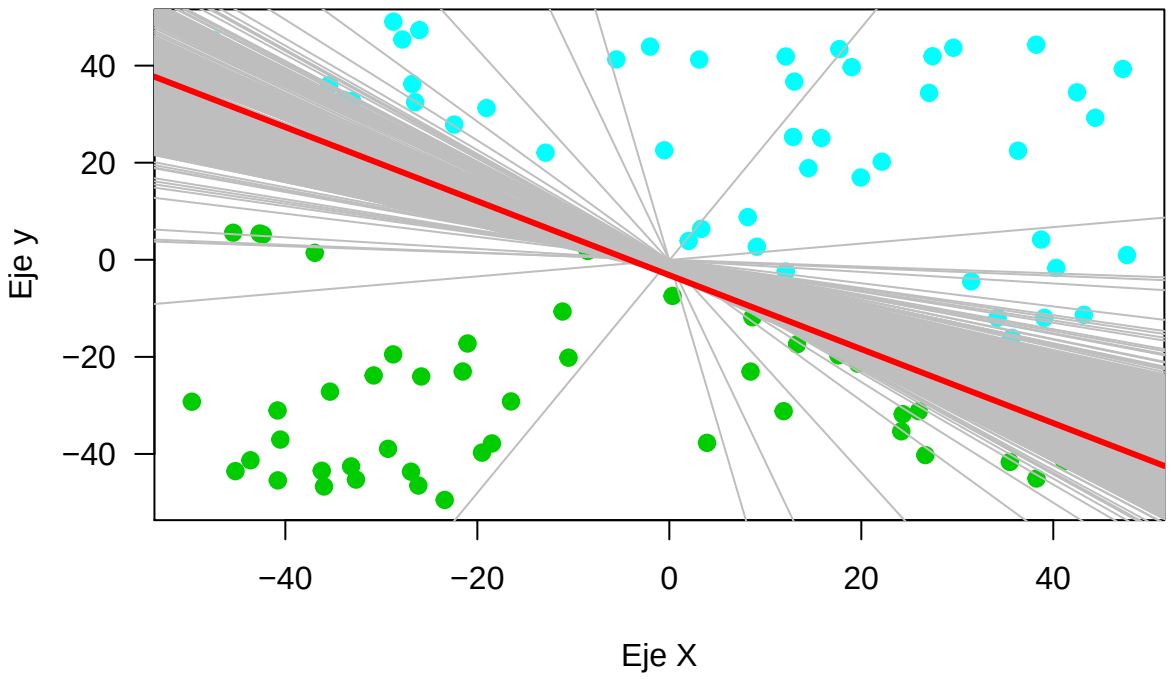


Al igual que en el apartado anterior, el perceptrón no es capaz de converger, ya que intenta clasificar de forma lineal datos que no son separables linealmente.

2.5. Modifique la función *ajusta_PLA* para que le permita visualizar los datos y soluciones que va encontrando a lo largo de las iteraciones. Ejecute con la nueva versión el apartado.3 del ejercicio.4.2.

La función *PLA* del apartado 1 de este ejercicio, contiene el parámetro *verRecorrido*. Si este parámetro está activo (tiene valor *TRUE*), cada vez que la recta se corrija, pintará la nueva línea que ha generado el perceptrón con un color gris. Para ello, la función *PLA* se llamará igual que se llamaba antes, pero añadiendo *verRecorrido* = *TRUE*. Un ejemplo de ejecución es el siguiente:

```
pintarPLA(puntos, range, apartado = 1, nombreGrafica = "EJERCICIO PERCEPTRON",
    leyenda = c("Función original","Perceptron"), verLeyenda = F,
    coloresLeyenda=c("black","red"), verR = T)
```



2.6. **A la vista de la conducta de las soluciones observada en el apartado anterior, proponga e implemente una modificación de la función original *sol = ajusta_PLA_MOD(...)* que permita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando los datos del apartado.7 del ejercicio.4.2**

```
Ein <- function(datosEval, labels, resultPLA){
    # Esta función de evaluación se puede ver en la página 80 del libro
    # "Learning From Data: A Short Course", donde se minimiza el error dentro
    # de la muestra, calculando el porcentaje de error
    ein = sign(datosEval%*%resultPLA)
    ein[ein==0]=1
    error = length(ein[ein != labels])/nrow(datosEval)
    error
}

PocketPLA <- function(datos, label, max_iter = 3000, vini = c(0,0,0), verRecorrido = F){

    # Añadimos un 1 por la izquierda a la matriz de puntos para que los puntos
    # de la matriz sean (x0, x1, x2) donde x1 será un 1, y x1 y x2 las
    # coordenadas del punto.
    datos = cbind(rep(1, nrow(datos)), datos)
    i = 1 # Contador para saber el número de iteraciones

    mejorSol = vini
    porcentajeError = Ein(datos=datos, labels=label, resultPLA=mejorSol)
```

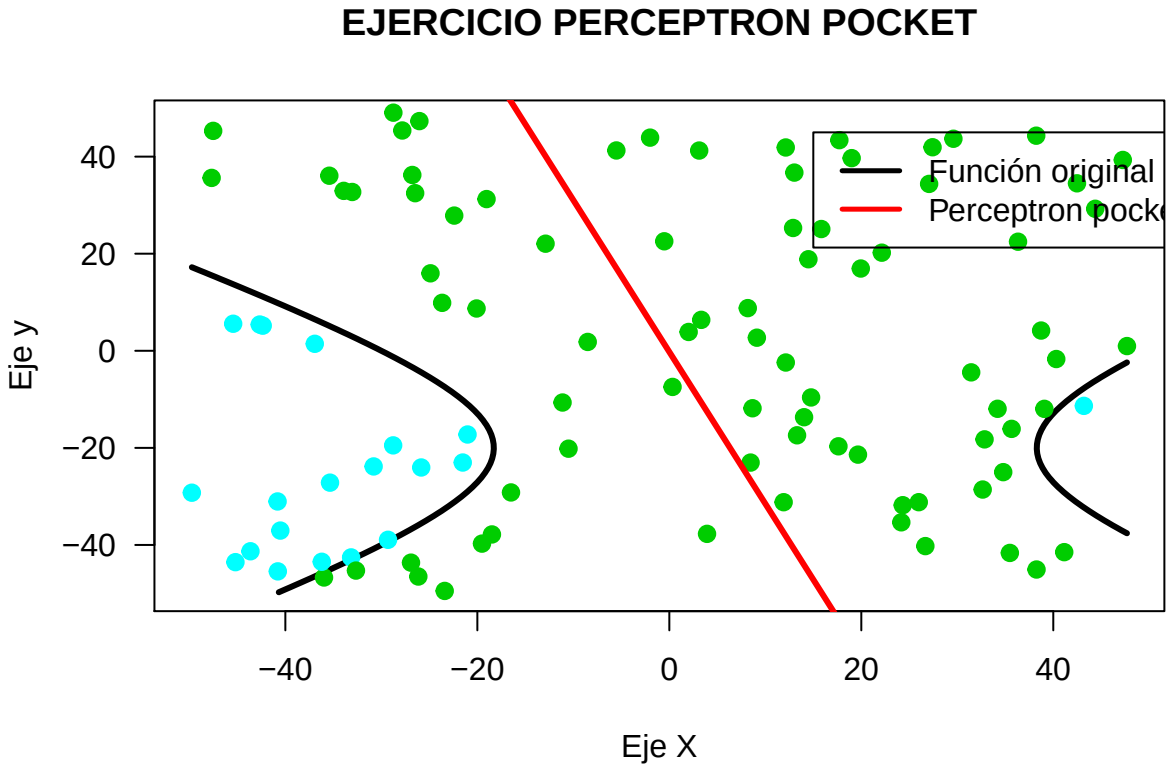
```
    # Empezamos el bucle del perceptron
    while (i <= max_iter){
        # iteramos sobre los puntos de cada dato
        for(j in 1:nrow(datos)){
            # Calculamos el signo en función de los pesos realizando el producto
            # del punto y el vector de pesos
            signo = sign(datos[j,]%*% vini)

            # En caso de que el signo sea 0 por la función sign, se pone a 1
            if (signo == 0){
                signo = signo + 1
            }
            # Cuando las etiquetas no coinciden, se calcula w(t+1) y se guarda
            # en v_aux. Tras esto, se calcula el error dentro de la muestra
            # (nuevoPorcentaje) y se compara con el que teníamos antes. Si es
            # menor, se actualiza la mejor solucion y el porcentaje minimo. Si
            # no, se pasa a la siguiente iteración con el w(t)
            if(label[j] != signo){
                v_aux = vini + datos[j,]*label[j]
                nuevoPorcentaje = Ein(datos=datos, labels=label, resultPLA=v_aux)
                if(porcentajeError > nuevoPorcentaje){
                    # se muestran los
                    cat("Antiguo pocentaje de error = ",porcentajeError,"\n")
                    cat("Nuevo porcentaje de error = ",nuevoPorcentaje,"\n")
                    vini = v_aux
                    porcentajeError = nuevoPorcentaje
                }
                if(verRecorrido)
                    abline(a=(-vini[1]/vini[3]),
                        b=(-vini[2]/vini[3]), col="grey")
            }
        }

        i = i + 1 # incrementamos el contador de iteraciones
    }

    # Se devuelve la recta que ha calculado el perceptron normalizada
    pla_result = c((-vini[2]/vini[3]),(-vini[1]/vini[3]), i, porcentajeError)
    pla_result
}
```

En estas dos funciones puede verse la función que evalúa el error dentro de la muestra *Ein*, que es una implementación del la función de la página 80 del libro "Learning From Data: A Short Course", donde se intenta minimizar E_{in} de la siguiente forma:

$$min_{\mathbb{R}^{d+1}} \frac{1}{N} \sum_{n=1}^{N} [[sign(w^T x_n) \neq y_n]]$$

Esto quiere decir que va minimizando el número de etiquetas erroneas dentro de la muestra. En caso de que la nueva recta calculada tenga un menor número de etiquetas erróneas que la anterior, se actualiza la mejor recta guardad a la nueva calculada, es decir $w(t+1)$.

Un ejemplo de ejecución para las etiquetas del apartado en el que aparece la hipérbola del ejercicio 7.




```
## Antiguo pocentaje de error =  0.8
## Nuevo porcentaje de error =  0.33
## Antiguo pocentaje de error =  0.33
## Nuevo porcentaje de error =  0.32
## Antiguo pocentaje de error =  0.32
## Nuevo porcentaje de error =  0.3
## [1] "ERROR OBTENIDO CON PLAPOCKET"
## [1] 0.3
```

3. Ejercicio sobre Regresión Lineal

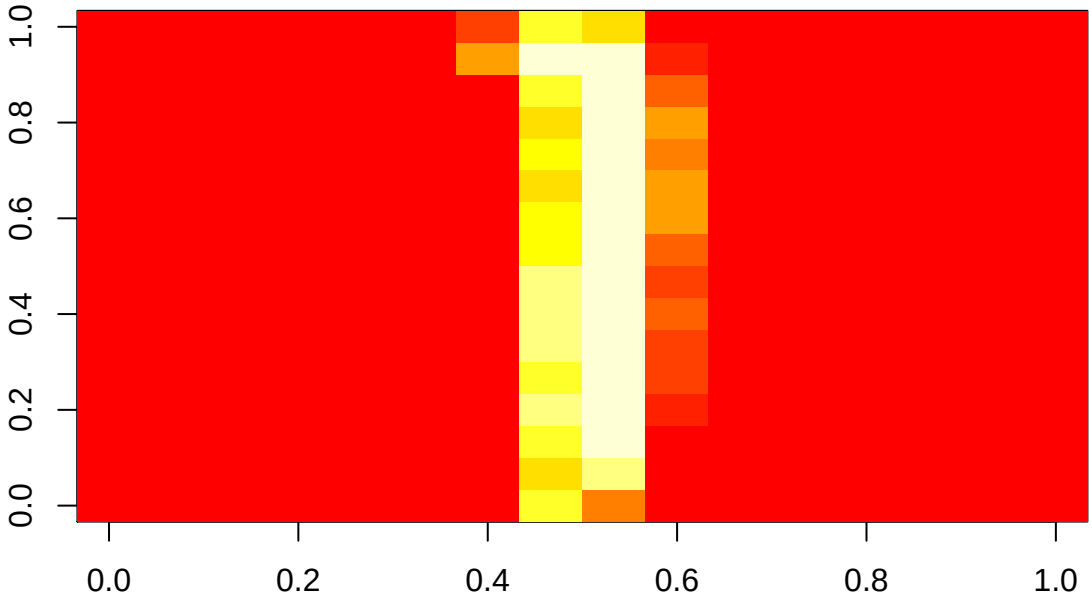
3.1. Abra el fichero ZipDigits.info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el fichero ZipDigits.train.

El fichero contiene una serie de valores distribuidos por filas, donde el primer número es el dígito que se está representando, seguido por 256 valores que representan el nivel de gris por píxel, en una matriz de 16x16 píxeles.

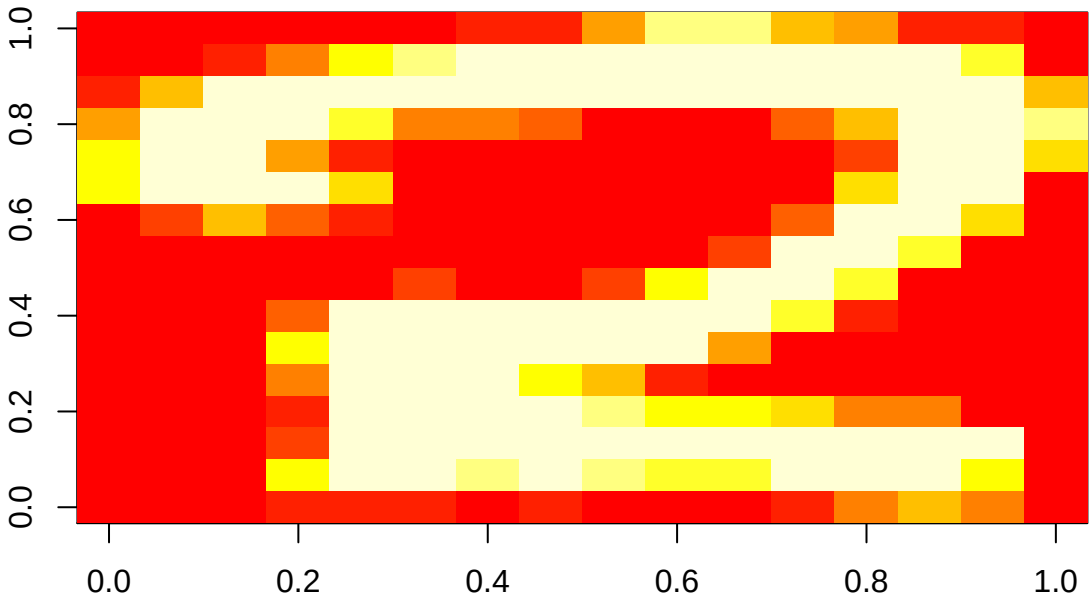
3.2. Lea el fichero ZipDigits.train dentro de su código y visualize las imágenes. Seleccione solo las instancias de los números 1 y 5. Guárdelas como matrices de tamaño 16x16.

```
leerDatos <- function(nombreFichero="./DigitosZip/zip.train"){
    zip = read.table(file=nombreFichero, sep=" ", stringsAsFactors=F)
    # Como la última columna de la tabla que genera son valores NA, la
    # eliminamos igualándola a NULL.
    unos = zip[which(zip$V1==1.0000),2:257]
    cincos = zip[which(zip$V1==5.0000),2:257]
    list(unos,cincos)
}

pintarNumero <- function(numeros){
    # print(numeros)
    numero = numeros[[1]][1,]
    matriz = as.matrix(numero)
    dim(matriz)=c(16,16)
    image(matriz)
}

datosLeidos = leerDatos()
```

```
pintarNumero(datosLeidos[1])
```




```
pintarNumero(datosLeidos[2])
```



En estas dos funciones se hace lo siguiente:

- La primera función, *leerDatos*, se encarga de leer los datos que hay en un fichero, que es el parámetro de la función y encajarlos en un *data.frame* con la función *read.table*. Tras esto, como sólo nos interesan los datos de los números 1 y los números 5, metemos aquellas filas cuyo primer valor coincidan con 1 y 5 y se guardan en vectores separados. Tras esto se devuelve una lista que contiene estos vectores.
- Esta función se utiliza para representar un número de uno de las listas que hemos calculado, transformando los 256 valores del número en una matriz de 16x16, y representándola con la función *image*.

3.3. Para cada matriz de números calcularemos su valor medio y su grado de simetría vertical. Para calcular la simetría calculamos la suma del valor absoluto de las diferencias en cada píxel entre la imagen original y la imagen que obtenemos invirtiendo el orden de las columnas. Finalmente le cambiamos el signo.

```
# Función que calcula la media de los datos para cada píxel
media <- function(n) apply(X=n,FUN=mean,MARGIN=1)
# Función que calcula la simetría de la matriz
simetria <- function(n) apply(X=n,
    FUN=function(line)-1*sum(abs(line[1:256]-line[256:1])), MARGIN=1)

calculaSimetria <- function(numeros){
    # Calculamos la media de los píxeles para los unos y los cincos, además
    # de las simetrias de ambos, devolviendolos en distintas listas
    unos=list(media(numeros[[1]]),simetria(numeros[[1]]))
    cincos=list(media(numeros[[2]]),simetria(numeros[[2]]))
    list(unos,cincos)
}

val=calculaSimetria(datosLeidos)
```

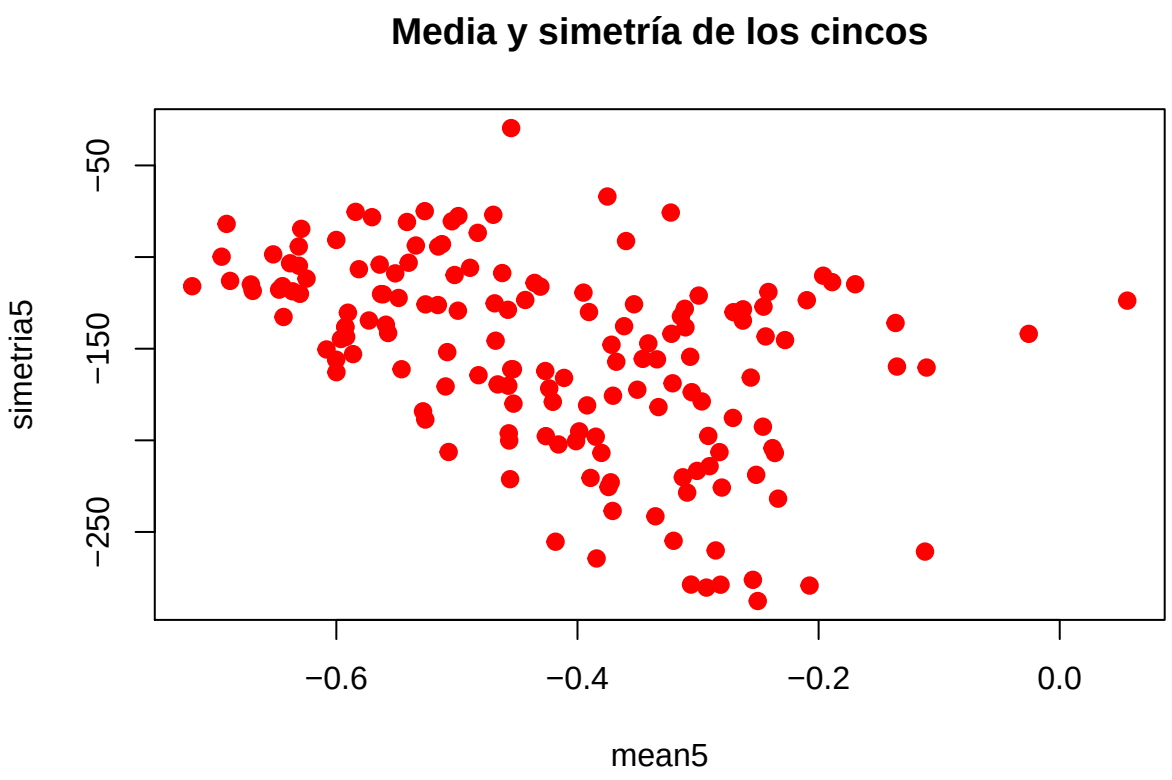
Para calcular la media y la simetría, he creado dos funciones que se encargan de ello. La primera, *media* recibe un array de puntos y calcula la media de la densidad para cada píxel, usando para ello la función *mean* que calcula la media, y la función *apply* para hacer los cálculos de forma vectorizada.

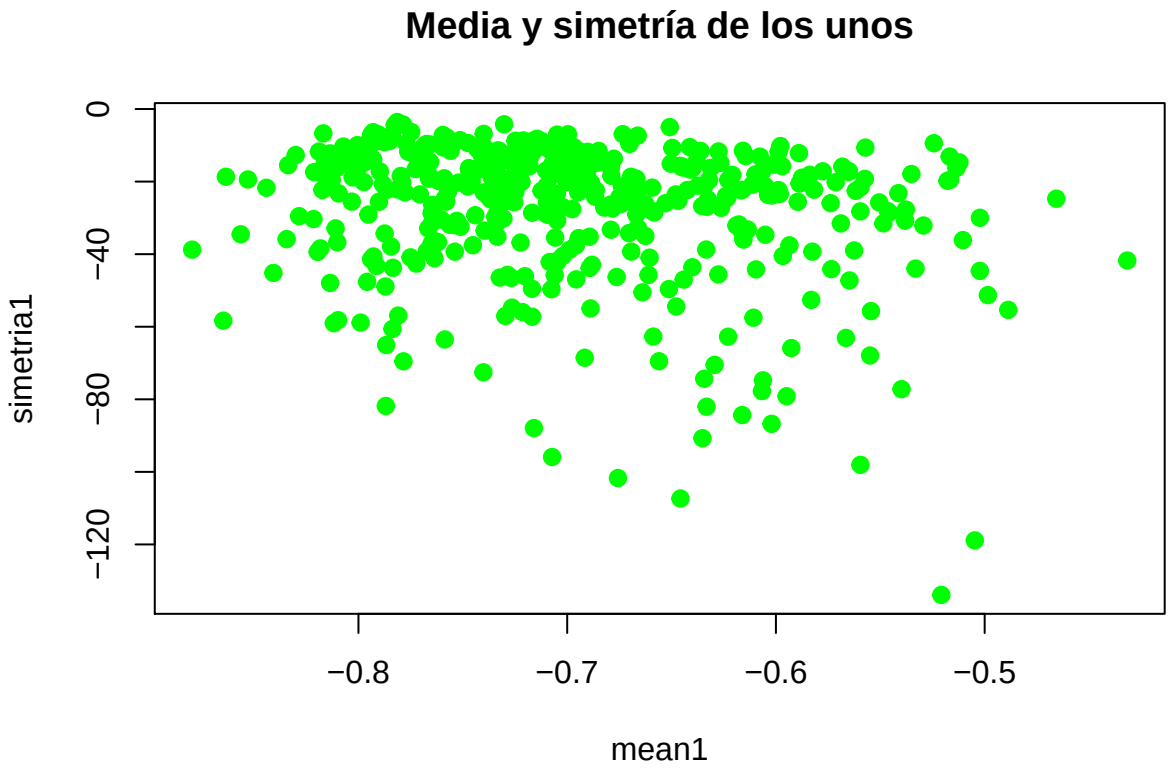
La segunda función, se encarga de realizar el cálculo de la simetría, también usando la función *apply* y la función que viene en el enunciado para calcular la simetría. Para evitar invertir las columnas de la matriz, la resta se hace restando a la matriz original, la misma matriz pero empezando desde el final.

La función *calculaSimetria* recibe los datos que hemos escogido en la función *leerDatos*, y calcula la media para cada píxel y la simetría. Para el cálculo de la media hacemos uso de la función *mean* junto con *apply*.

3.4. Representar en los ejes { X=Intensidad Promedio, Y=Simetríŋa} las instancias seleccionadas de 1's y 5's.

```
representarMediaSimetria <- function(valores){
    mean1 = valores[[1]][[1]]
    simetria1 = valores[[1]][[2]]

    mean5 = valores[[2]][[1]]
    simetria5 = valores[[2]][[2]]

    plot(x=mean5, y=simetria5, col = "red",pch=19, lwd = 2, main="Media y simetría de los cincos")
    plot(x=mean1, y=simetria1, col = "green",pch=19, lwd = 2, main="Media y simetría de los unos")
}

representarMediaSimetria(val)
```



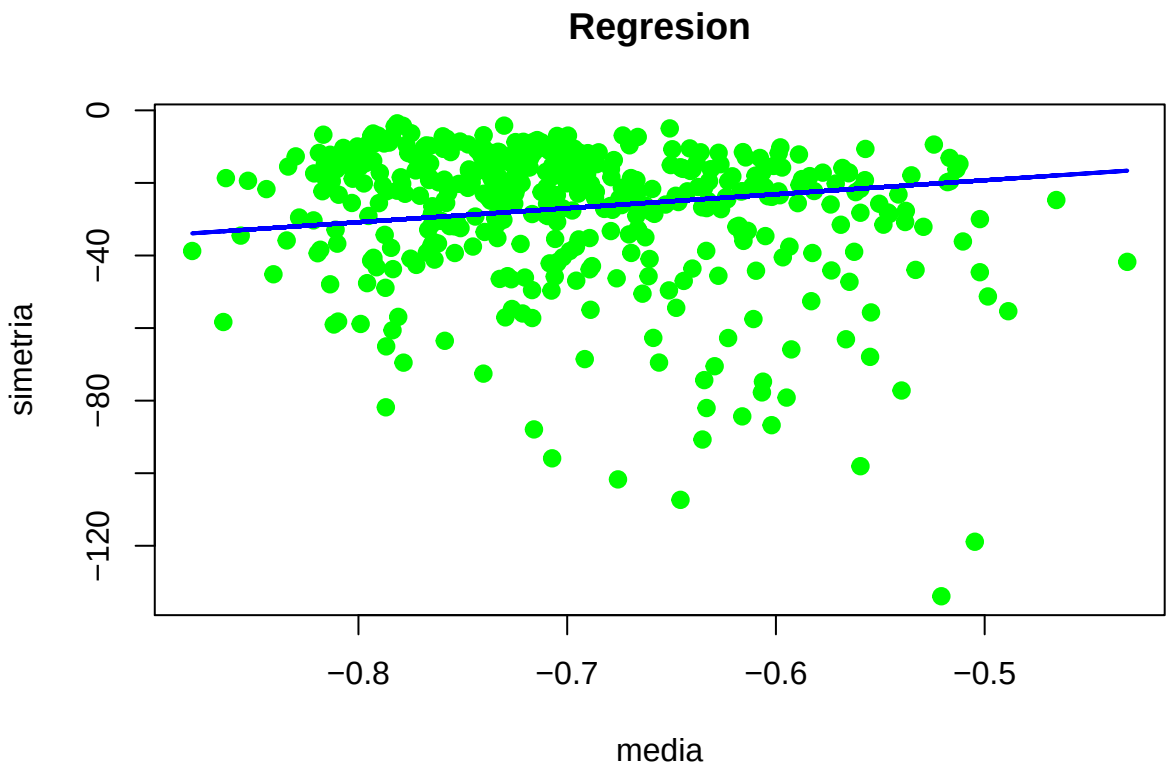


3.5. Implementar la función sol = Regress_Lin(datos, label) que permita ajustar un modelo de regresion lineal (usar SVD). Los datos de entrada se interpretan igual que en clasificación.

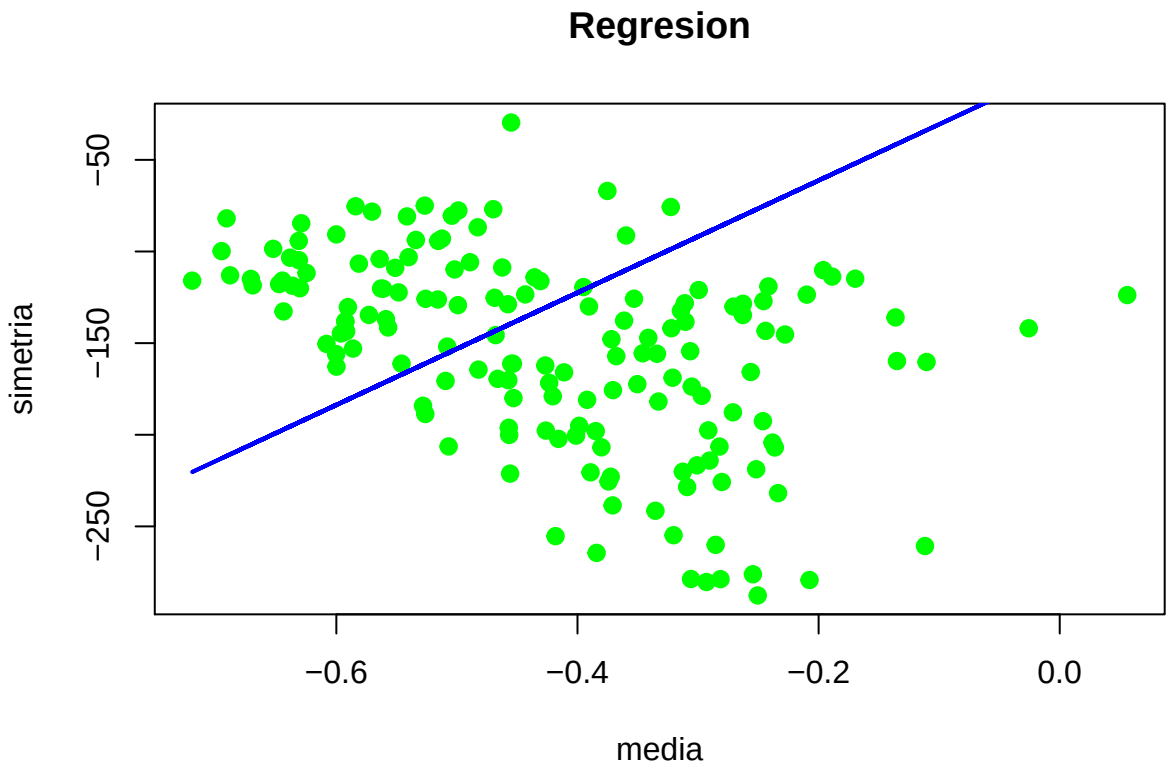
```
Regress_Lin <- function(datos, label){
    pseudoinversa = solve(t(datos)%*%datos)%*%t(datos)

    wlin=(pseudoinversa%*%label)
    y_prima=datos%*%(wlin)

    list(wlin, y_prima)
}
```

3.6. Ajustar un modelo de regresion lineal a los datos de (Intensidad promedio, Simetria) y pintar la solución junto con los datos. Valorar el resultado.

```
ajusteRegresion <- function(media, simetria){
    regresion = Regress_Lin(media, simetria)
    plot(x=media, y=simetria, col = "green",pch=19, lwd = 2, main="Regresion")
    lines(x=media, y = regresion[[2]], col="blue", lwd=2, pch=19)
}
ajusteRegresion(val[[1]][[1]], val[[1]][[2]])
```




```
ajusteRegresion(val[[2]][[1]], val[[2]][[2]])
```



3.7. **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 $X = [-10, 10] \times [-10, 10]$ 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 con el valor de su signo. 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 g y evaluar E_{in}, (el porcentaje de puntos incorrectamente clasificados). Repetir el experimento 1000 veces y promediar los resultados ¿Qué valor obtiene para Ein? b) Fijar el tamaño de muestra $N = 100$. Usar regresión lineal para encontrar g y evaluar E_{out}. Para ello generar 1000 puntos nuevos y usarlos para estimar el error fuera de la muestra, E_{out} (porcentaje de puntos mal clasificados). De nuevo, ejecutar el experimento 1000 veces y tomar el promedio. ¿Qué valor obtiene de E_{out}? Valore los resultados. 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 ¿Cuál 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.**

Para la realización de este ejercicio, he creado las siguientes funciones:

```
# Indicamos el tamaño de la muestra, el rango de los puntos
# y generamos la recta (o función f) que evaluará los puntos
tam = 100
rangoEJ7 = -10:10
rectaEJ7 = simula_recta(dim=rangoEJ7,verPunto = F)

# Función para calcular el error fuera de la muestra
calcularEout <- function(ein, datos, tam){
    Eout = ein + (ncol(datos))/tam
    Eout
}

# Función para calcular el error dentro de la muestra
calcularEin <- function(X, wlin, N, y){
    errorIN = (1/N)*(t(wlin)%*%t(X)%*%X%*%wlin -
        2*t(wlin)%*%t(X)%*%y +
        t(y)%*%y)
}

# Función para calcular la regresión lineal para la clasificación.
# En ella se devuelve el error dentro y fuera de la muestra
regresionLinealClasificacion <- function(){
  # Generamos una muestra y calculamos las etiquetas en función de
  # la recta generada anteriormente
```

```
    puntosEJ7 = simula_unif(N=tam, dim=2, rango=rangoEJ7)
    etiquetasEJ7 = evaluaPuntos(puntosEJ7, rangoEJ7, apartado = 6)

  # Ajustamos la regresión lineal para los puntos y las etiquetas
    g = Regress_Lin(puntosEJ7, etiquetasEJ7)

    wlin = g[[1]]
  # Calculamos el error dentro de la muestra
    errorIN = (1/tam)*(t(wlin)%*%t(puntosEJ7)%*%puntosEJ7%*%wlin -
        2*t(wlin)%*%t(puntosEJ7)%*%etiquetasEJ7 +
        t(etiquetasEJ7)%*%etiquetasEJ7)
  # Y el error dentro de la muestra
    Eout = calcularEout(errorIN, puntosEJ7, tam)
    c(errorIN, Eout)
}

# Esta función se encarga de realizar todo lo necesario para
# realizar el apartado c
apartadoEJ7C <- function(){
    # seleccionamos el tamaño de la muestra a 10 y generamos las muestras,
    # junto con las etiquetas
    tam = 10
    puntosEJ7C = simula_unif(N=tam, dim=2, rango=rangoEJ7)
    etiquetasEJ7C = evaluaPuntos(puntosEJ7C, rangoEJ7, apartado = 6)

    # ajustamos la regresión para los datos y las etiquetas
    regC = Regress_Lin(puntosEJ7C, etiquetasEJ7C)

    # una vez que tenemos los datos, y la regresión, calculamos los
    # parámetros de la recta que se ajustan a la regresión
    rectaEJ7C = simula_recta(punto_1 = c(puntosEJ7C[1,1],regC[[1]][1,]),
                             punto_2 = c(puntosEJ7C[2,1],regC[[2]][1,]),
                             verPunto = F)

    # lanzamos el pla con la recta que se ajusta a la regresión como vector
    # inicial del pla
    plaEJ7C = PLA(datos=puntosEJ7C, label=etiquetasEJ7C,
        max_iter = 3000, vini = c(rectaEJ7C[2],rectaEJ7C[1],1))

    plaEJ7C
}
```

La ejecución de los tres apartados es el siguiente:

```
Eins = vector(length=1000)
Eouts = vector(length=1000)
numMedioIteraciones = vector(length=1000)
for (i in 1:1000) {
    Eins[i] = regresionLinealClasificacion()[1]
    Eouts[i] = regresionLinealClasificacion()[2]
    numMedioIteraciones[i] = apartadoEJ7C()[3]

}
```

```
meanEin = mean(Eins)
meanEout = mean(Eouts)
meanIters = mean(numMedioIteraciones)
cat("Media de Ein para la regresion lineal = ",meanEin,"\n")
```

```
## Media de Ein para la regresion lineal =  0.6456573
```

```
cat("Media de Eout para la regresion lineal = ",meanEout,"\n")
```

```
## Media de Eout para la regresion lineal =  0.66224
```

```
cat("Numero medio de iteraciones para el PLA = ",meanIters,"\n")
```

```
## Numero medio de iteraciones para el PLA =  17.409
```

- Apartado 1: para este apartado, podemos ver que la media de *Ein* para los cálculos que realiza la regresión lineal, se mueve al rededor del 50 %, por lo que en algunos casos, ofrece una buena separación, y en otros, el error puede ser muy grande.
- Apartado 2: en este caso, la media de *Eout*, como depende del error obtenido dentro, variará ligeramente con el error que se obtiene dentro de la muestra, haciendo una leve generalización.
- Apartado 3: En este caso, ajustando la regresión de los datos, obtenemos una recta que calculamos con la función *simula_recta*. El resultado de esta función, nos da una recta $rectaEJ7C = a \cdot x + b$, con lo que le pasamos al *PLA* el siguiente vector de pesos: $\omega = \{rectaEJ7C[2], rectaEJ7C[1], 1\}$. Tras esto, y realizar la ejecución 1000 veces, vemos que el número de iteraciones es muy pequeño, ya que la recta que se le ha pasado es muy buena.

3.8. En este ejercicio exploramos el uso de transformaciones no lineales. Consideremos la función objetivo $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 25)$. Generar una muestra de entrenamiento de N = 1000 puntos a partir de *X = [-10, 10] x [-10, 10]* muestreando cada punto $x \in X$ uniformemente. Generar las salidas usando el signo de la función en los puntos muestreados. Generar ruido sobre las etiquetas cambiando el signo de las salidas a un 10 % de puntos del conjunto aleatorio generado. a) Ajustar regresion lineal, para estimar los pesos w. Ejecutar el experimento 1.000 y calcular el valor promedio del error de entrenamiento Ein . Valorar el resultado. b) Ahora,consideremos N = 1000 datos de entrenamiento y el siguiente vector de variables: $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Ajustar de nuevo regresión lineal y calcular el nuevo vector de pesos $\hat{w}$. Mostrar el resultado.c9. c)Repetir el experimento anterior 1.000 veces calculando en cada ocasión el error fuera de la muestra. Para ello generar en cada ejecución 1.000 puntos nuevos y valorar sobre ellos la función ajustada. Promediar los valores obtenidos Â£Qué valor obtiene?. Valorar el resultado.

```
ejercicio8<-function(){
  # Generamos la muestra y la evaluamos según la función
  # mencionada en el enunciado. Tras esto, modificamos el
  # 10% de las etiquetas
    puntosEJ8 = simula_unif(N=1000, dim=2, rango=(-10:10))
    etiquetasEJ8=evaluaPuntos(puntosAEvaluar=puntosEJ8,
        intervalo=(-10:10), apartado=7)

    etiquetasEJ8=cambiarEtiqueta(etiquetasEJ8)

  # Ajustamos la regresión
    g=Regress_Lin(puntosEJ8,etiquetasEJ8)
  # Calculamos el error de la muestra
    Ein = calcularEin(X = puntosEJ8, wlin = g[[1]], N = 1000, y = etiquetasEJ8)
    Ein
}
```

En esta función, como se puede ver, se generan la muestras, se evalúan y se les añade ruido modificando el valor del 10 % de las etiquetas calculadas. Tras esto, ajustamos la regresión lineal y se calcula el error dentro de la muestra, siendo esto último lo que se devuelve.

Para realizar lo que se pide en el apartado *a*, generamos el siguiente bucle, el cual, tras él se calcula la media de los errores obtenidos, almacenados en un vector.

```
Eins8 = vector(length=1000)
for (i in 1:1000) {
    Eins8[i]=ejercicio8()
}
ErrorMedioIn8 = mean(Eins8)
cat("Error medio en Ein del ejercicio 8",ErrorMedioIn8,"\n")
```

```
## Error medio en Ein del ejercicio 8 0.9979664
```

Como se puede ver, el error dentro de la muestra es muy alto, superando el 90 %, debido al ruido que hemos generado.

En el caso del segundo apartado, he generado el siguiente código:

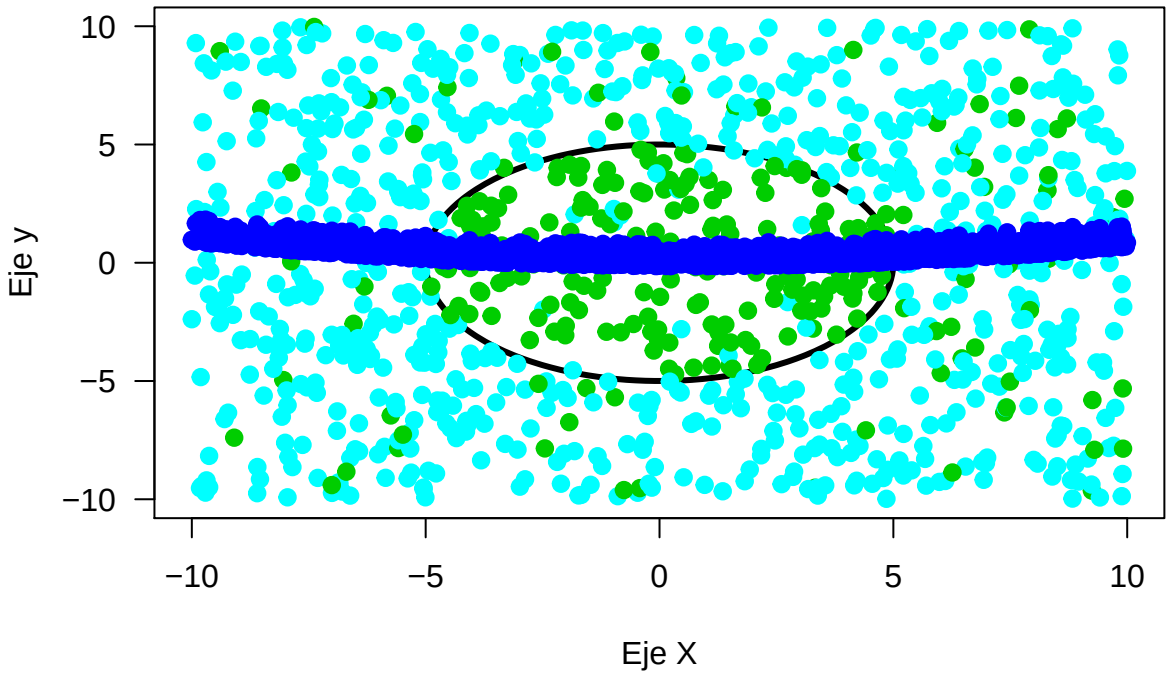
```
apartado8B <- function(mostrar=F){
  # Generamos la muestra y la evaluamos según la función
  # mencionada en el enunciado. Tras esto, modificamos el
  # 10% de las etiquetas
    puntosEJ8 = simula_unif(N=1000, dim=2, rango=(-10:10))
    etiquetasEJ8=evaluaPuntos(puntosAEvaluar=puntosEJ8,
        intervalo=(-10:10), apartado=7)

    etiquetasEJ8=cambiarEtiqueta(etiquetasEJ8)

  # Esta función sirve para construir la matriz con
  # (1,x1,x2,x1x2,x1^2,x2^2)
    f<-function(linea) c(1,linea[1],linea[2],linea[1]*linea[2],
        linea[1]*linea[1],linea[2]*linea[2])
```

```
  # Usamos la función appli para hacer el cálculo vectorizado
    nuevosPuntos=apply(X=puntosEJ8, FUN=f, MARGIN=1)
  # Como la matriz que obtenemos es de 1000 columnas y 6 filas
  # realizamos la traspuesta para obtener nuestra matriz de
  # 1000 filas y 6 columnas
    nuevosPuntos = t(nuevosPuntos)

  # Ajustamos la regresión
    g=Regress_Lin(nuevosPuntos, etiquetasEJ8)

  # En caso de querer mostrar la función, se muestra por pantalla
    if(mostrar){
        pintar(puntos=puntosEJ8, funciones[[7]], range, "EJERCICIO 8.b",
        colores =  etiquetasEJ8 + 4, verFuncion = T)
        points(x=puntosEJ8[,1], y = g[[2]], col="blue", lwd=2, pch=19)
    }
  # Calculamos el error dentro y fuera de la muestra y devolvemos Eout
    Ein = calcularEin(X = nuevosPuntos, wlin = g[[1]],
        N = 1000, y = etiquetasEJ8)
    Eout = calcularEout(Ein, nuevosPuntos, 1000)
    Eout
}
```

En esta función generamos los datos al igual que hacíamos en el apartado anterior, junto con las etiquetas. Tras esto, generamos la matriz cuyas filas son $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$, donde x_1 y x_2 son las coordenadas del punto de la matriz original. Esta matriz lo que hace es obtener en una sola matriz, todas las posibles curvas cuadráticas existentes en el espacio X ([-10,10]x[-10,10]), para así hayar la función que se ajusta a estos datos, aunque eso supone aumentar el tamaño de la matriz.

Un ejemplo de ejecución es el siguiente:

```
apartado8B(T)
```




```
##           [,1]
## [1,] 0.6163013
```

En él se puede ver cómo la regresión ha encontrado una función que se ajusta mejor a los datos, junto con el error fuera de la muestra.

En el caso del último apartado, se ha generado un bucle *for*, que ejecuta 1000 veces la función anterior y obtiene la media del error fuera de la muestra, tal y como se ve a continuación:

```
Eouts8 = vector(length=1000)
for (i in 1:1000) {
    Eouts8[i]=ejercicio8()
}
ErrorMedioOut8 = mean(Eouts8)
cat("Error medio en Eout del ejercicio 8",ErrorMedioOut8,"\n")
```

```
## Error medio en Eout del ejercicio 8 0.9980222
```

Como se puede ver, el error fuera de la muestra, se reduce frente al error.[1]

[1] Esto puede variar según la ejecución.