Memoria de Práctica 2

Julio A. Fresneda

April 20, 2018

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 la clase de funciones más adecuada. Haremos uso de tres funciones ya programadas:

simula_unif

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.

```
simula_unif = function (N=2,dims=2, rango = c(0,1)){
  m = matrix(runif(N*dims, min=rango[1], max=rango[2]),
             nrow = N, ncol=dims, byrow=T)
  m
}
```

simula_gaus

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.

```
simula_gaus = function(N=2,dim=2,sigma){

  if (missing(sigma)) stop("Debe dar un vector de varianzas")
  sigma = sqrt(sigma)  # para la generación se usa sd, y no la varianza
  if(dim != length(sigma)) stop()

  # 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

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].

```
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
}
```

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

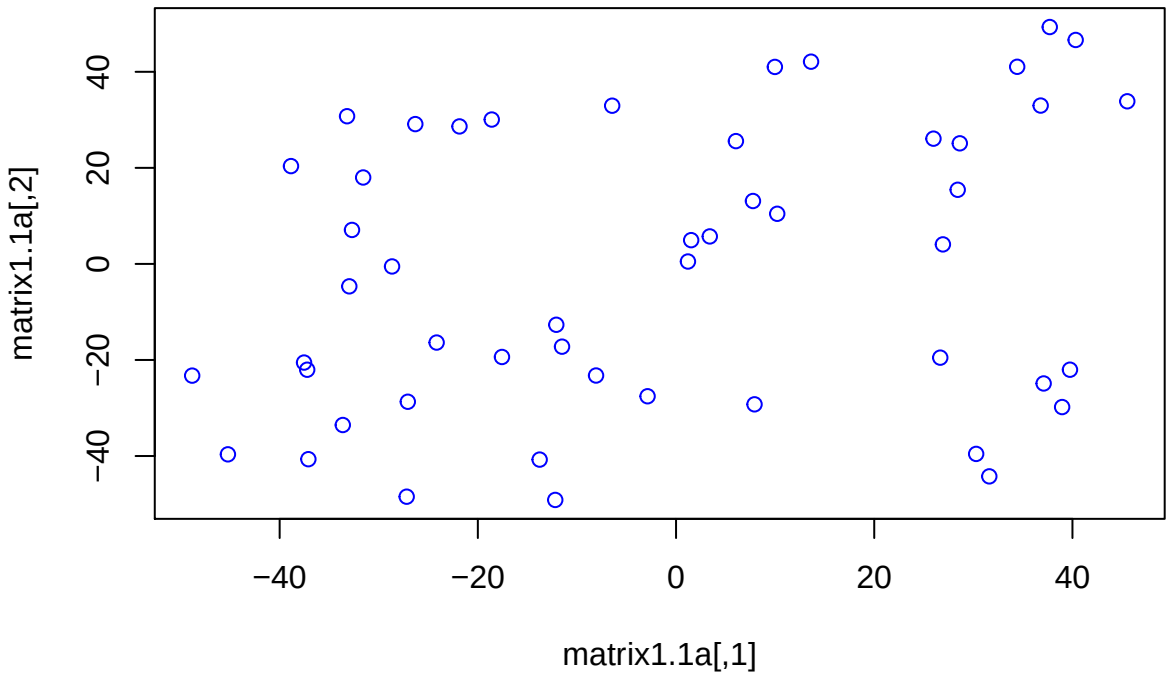
a) Considere N = 50, dim = 2, rango = [-50, +50] con simula_unif (N, dim, rango).

Vamos a llamar a simula_unif:

```
matrix1.1a = simula_unif(50,2,c(-50,50))
```

La matriz resultante se puede dibujar gráficamente así:

```
plot(matrix1.1a,col='blue')
```



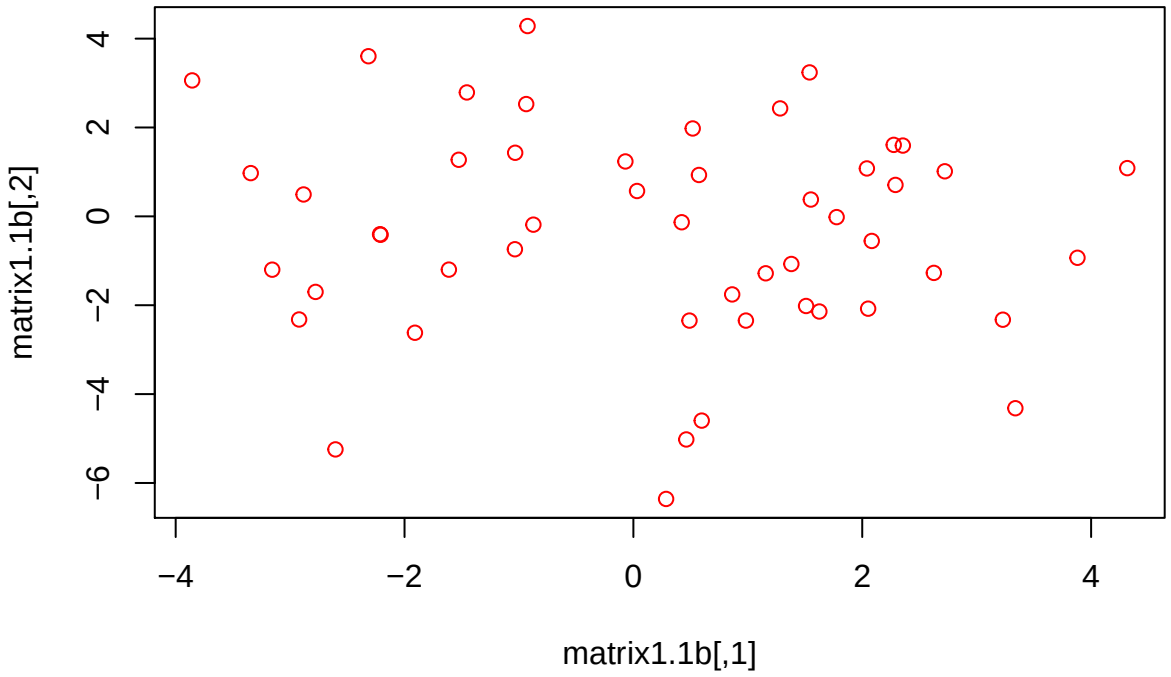
b) Considere N = 50, dim = 2 y sigma = [5, 7] con simula_gaus(N, dim, sigma).

Vamos a llamar a simula_gaus:

```
matrix1.1b = simula_gaus(50,2,c(5,7))
```

La matriz resultante se puede dibujar gráficamente así:

```r
plot(matrix1.1b,col='red')
```



1.2 - Con ayuda de la función simula_unif() generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función f (x, y) = y - ax - b, es decir el signo de la distancia de cada punto a la recta simulada con simula_recta().

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

Suponemos que el rango está en [-50,50] x [-50,50] .

Vamos a obtener los valores a y b de la recta con forma y = ax + b con la función simula_recta:

```r
pesos_recta = simula_recta(c(-50,50))
```

Los pesos son los siguientes:

```r
print(pesos_recta)
```

```
## [1] 0.4532805 0.1555027
```

Una vez tenemos los pesos, vamos a declarar la función clasificadora para nuestros datos.

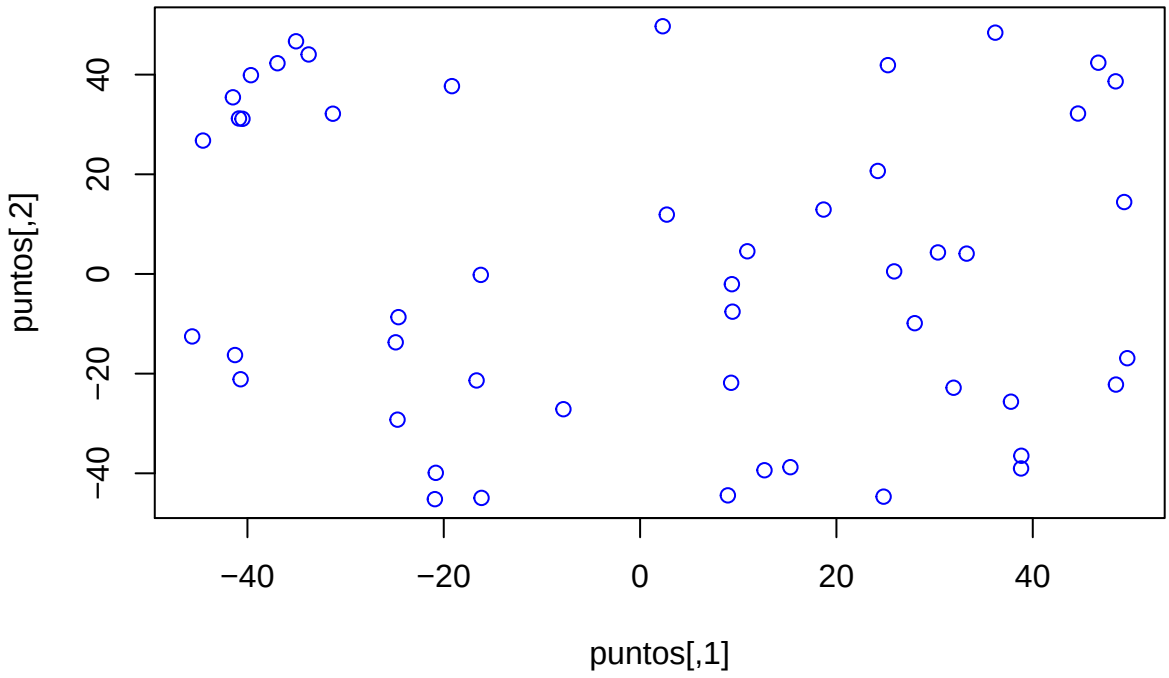
```r
funcion_clasificadora = function(x,y)
{
  r = sign(y - pesos_recta[1]*x - pesos_recta[2])
  r
}
```

Vamos ahora a generar nuestros datos para clasificar.

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

Los puntos son los siguientes.

```
plot(puntos,col='blue')
```

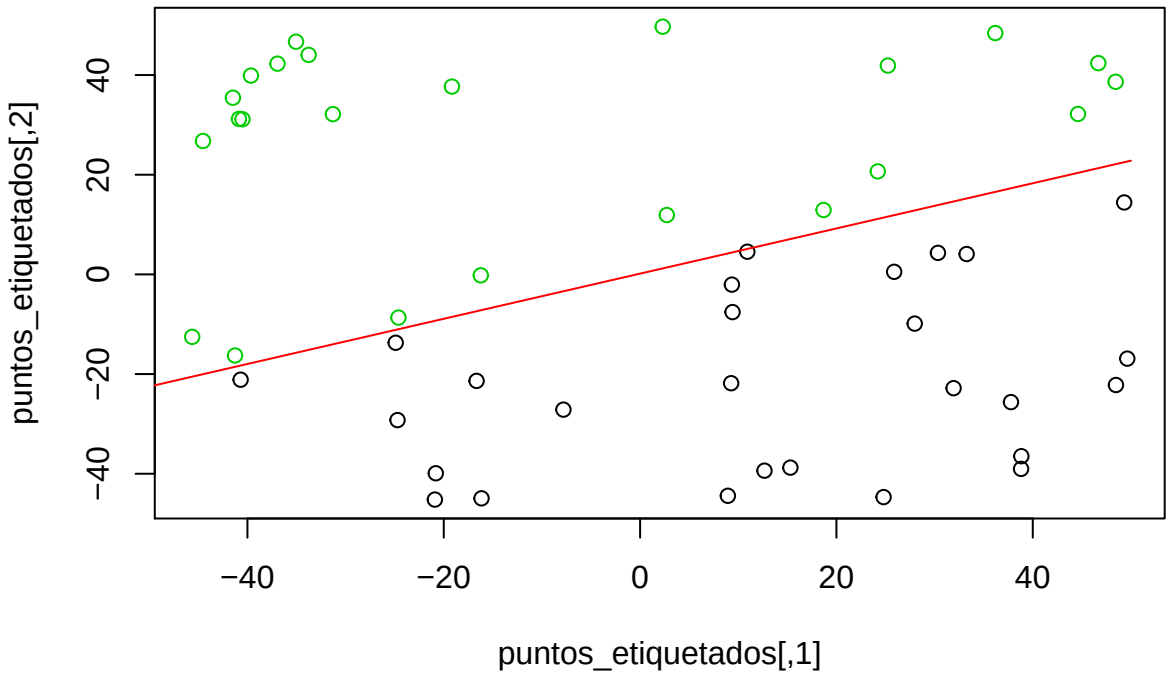


Vamos a clasificarlos con ayuda de la función obtenida.

```
puntos_etiquetados = matrix(c(puntos[,1],puntos[,2],funcion_clasificadora(puntos[,1],puntos[,2])),nrow=5
```

Los puntos clasificados, junto a la función clasificadora, son los siguientes.

```
plot(puntos_etiquetados,col=puntos_etiquetados[,3]+2)
lines(c(-50,50),c(-50*pesos_recta[1]+pesos_recta[2],50*pesos_recta[1]+pesos_recta[2]),col='red')
```



Como vemos, todos los puntos están bien clasificados.

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

Para esto, usaremos la función noise:

```
#Función para añadir ruido
noise <- function(vect, p){
  label = vect
  for( i in 1:((length(label))*p) )
  {
    x = sample(length(label),1)
    if( label[x] == 1 ) label[x] = -1
    else label[x] = 1
  }

  label
}
```

Primero vamos a modificar el 10% de las etiquetas positivas.

```
puntos_etiquetados_positivos = puntos_etiquetados[puntos_etiquetados[,3]==1,]
etiquetas_positivas = c(puntos_etiquetados_positivos[,3])
etiquetas_positivas_con_ruido = noise(etiquetas_positivas,0.1)
puntos_etiquetados_positivos[,3] = etiquetas_positivas_con_ruido
```

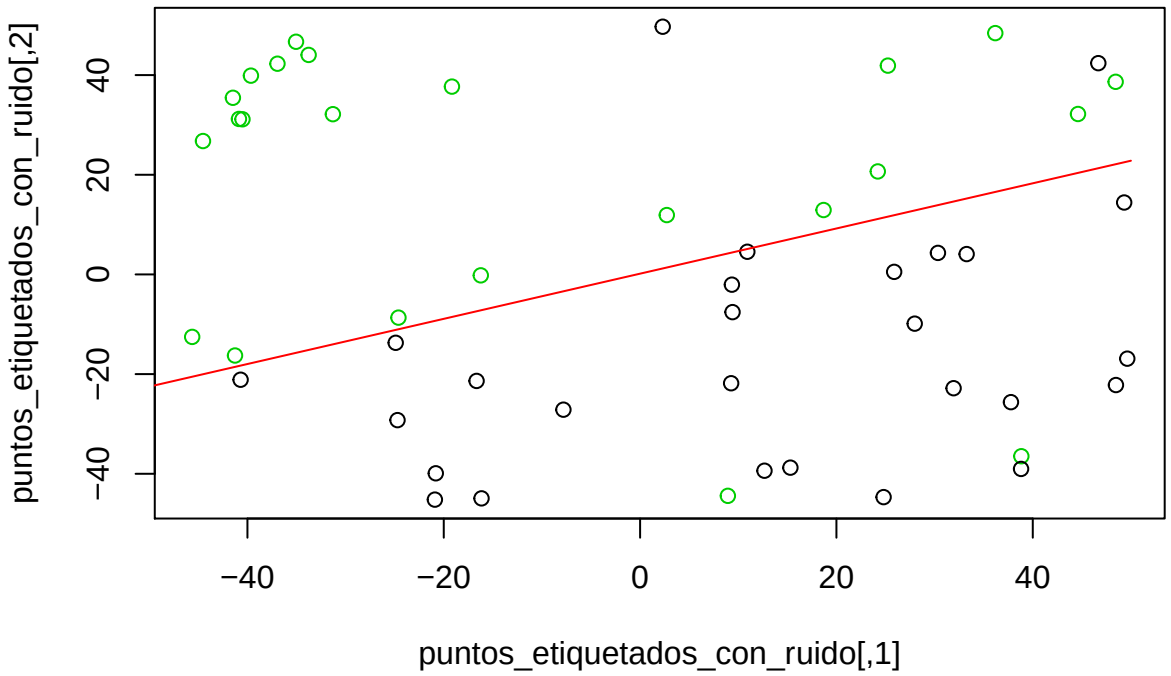
Ahora vamos a modificar el 10% de las etiquetas negativas.

```
puntos_etiquetados_negativos = puntos_etiquetados[puntos_etiquetados[,3]==-1,]
etiquetas_negativas = c(puntos_etiquetados_negativos[,3])
etiquetas_negativas_con_ruido = noise(etiquetas_negativas,0.1)
puntos_etiquetados_negativos[,3] = etiquetas_negativas_con_ruido
```

Vamos a unir los puntos etiquetados con +1 y los puntos etiquetados con -1, y a dibujarlos.

```
puntos_etiquetados_con_ruido = matrix(c(puntos_etiquetados_positivos[,1],puntos_etiquetados_negativos[,

plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
lines(c(-50,50),c(-50*pesos_recta[1]+pesos_recta[2],50*pesos_recta[1]+pesos_recta[2]),col='red')
```



Como vemos, ahora hay algunos puntos mal clasificados.

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 ¿Son estas funciones más complejas mejores clasificadores que la función lineal? ¿En que ganan a la función lineal? Explicar el razonamiento.

Vamos a dibujar las siguientes funciones sobre los puntos anteriores. Para ello usaremos la función pintar_frontera.

```
pintar_frontera = function(f,rango=c(-50,50)) {
  x=y=seq(rango[1],rango[2],length.out = 500)
  z = outer(x,y,FUN=f)
  if (dev.cur()==1) # no esta abierto el dispositivo lo abre con plot
```

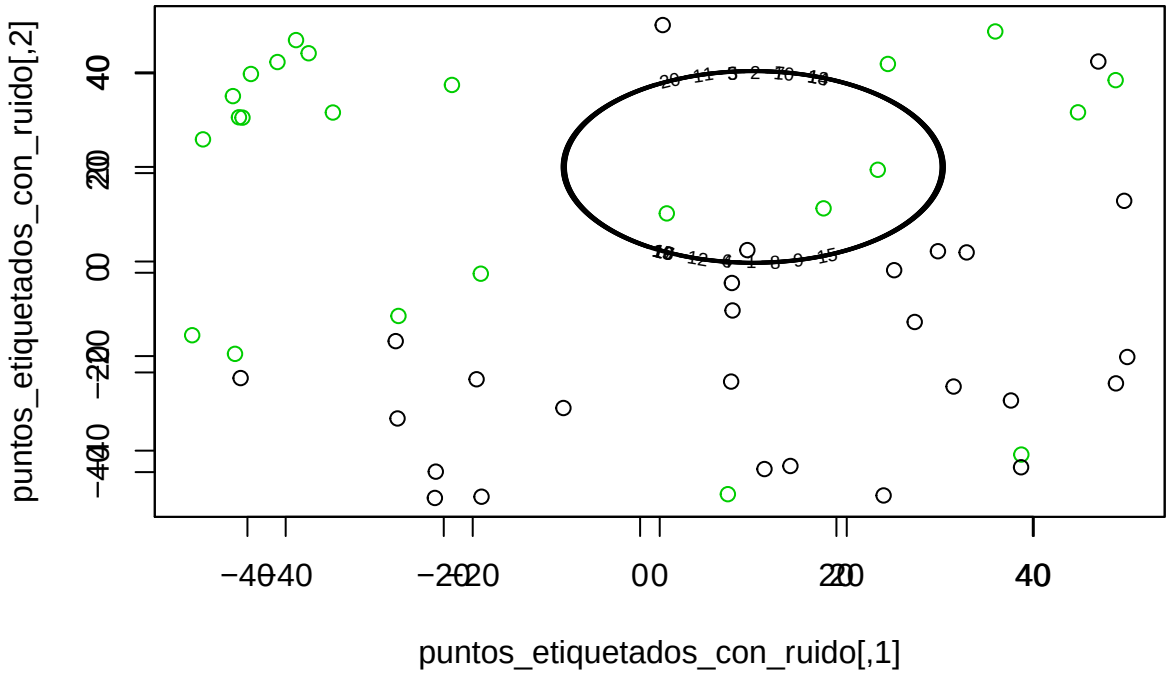
```
    plot(1, type="n", xlim=rango, ylim=rango)
  contour(x,y,z, levels = 1:20)
}
```

Función

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

```
funcion3.1 <- function(x,y){
  (x-10)^2+(y-20)^2-400
}
```

```
plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
par(new=T)
pintar_frontera((funcion3.1))
```

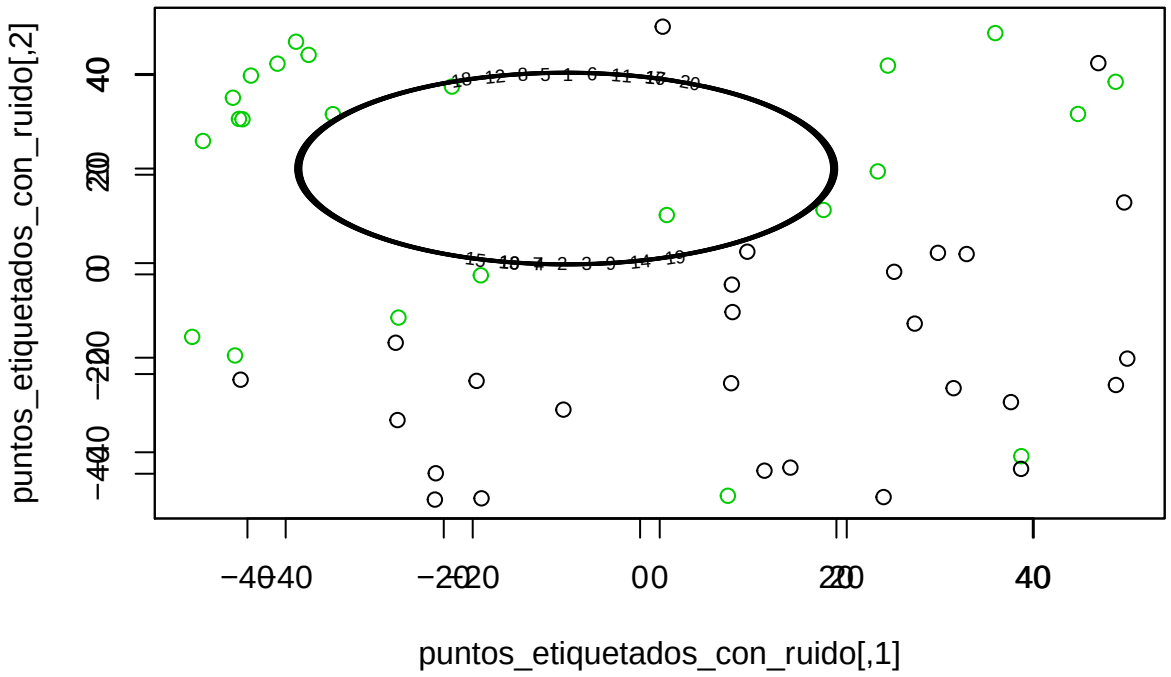


Función

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

```
funcion3.2 <- function(x,y){
  0.5*(x + 10)^2 + (y - 20)^2 - 400
}
```

```
plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
par(new=T)
pintar_frontera((funcion3.2))
```

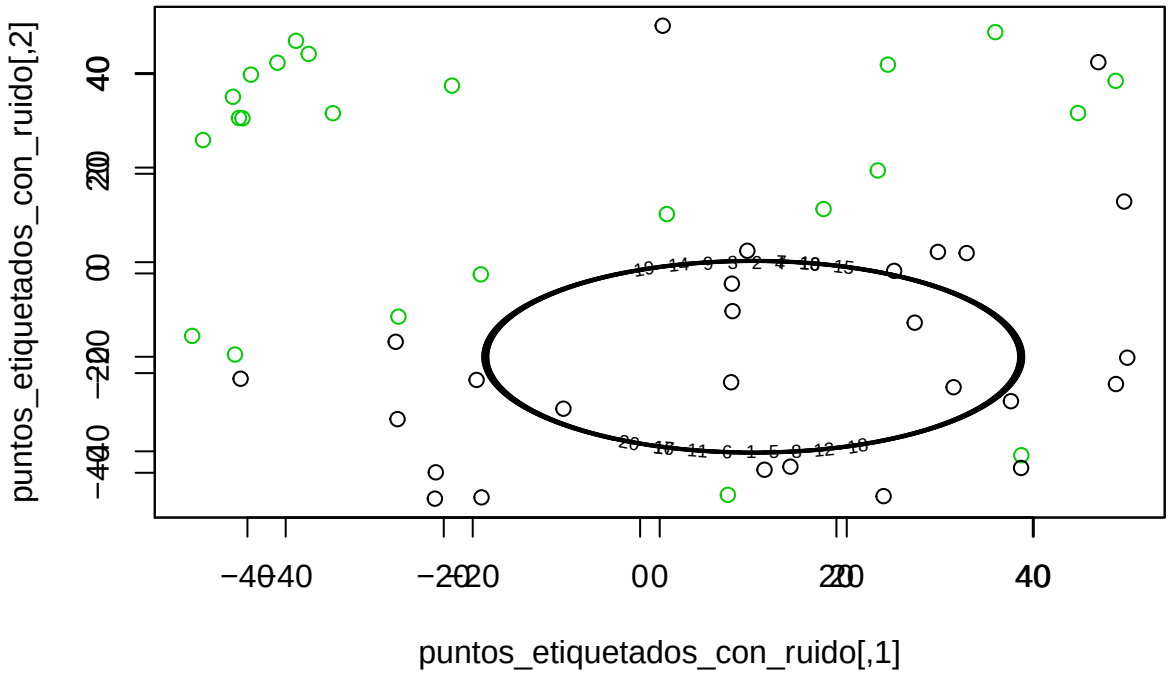


Función

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

```
funcion3.3 <- function(x,y){
  0.5*(x - 10)^2 + (y + 20)^2 - 400
}
```

```
plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
par(new=T)
pintar_frontera((funcion3.3))
```

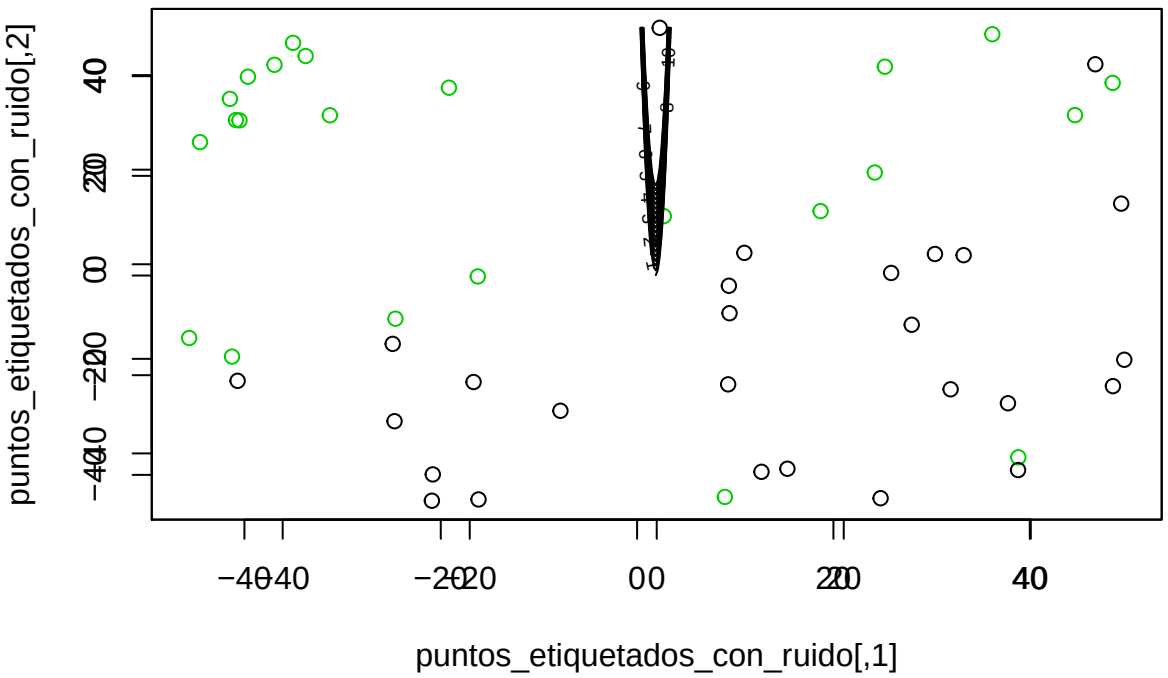


Función

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

```
funcion3.4 <- function(x,y){
  y - 20*x^2 - 5*x + 3
}
```

```
plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
par(new=T)
pintar_frontera((funcion3.4))
```



En estas funciónes, excepto en la última, los puntos que queden dentro del "círculo", deberían tener como etiqueta -1. Los puntos que queden fuera, tendrían etiqueta +1.

En la última función, los puntos que queden dentro de la "v" que forma la función, deberían tener +1 como etiqueta. Los puntos que queden fuera, deberían tener -1 como etiqueta.

Estas funciones son más complejas que las funciones lineales, pero, en este caso, son peor clasificadoras. Ésto es por que las funciones complejas clasifican bien puntos con distribuciones muy específicas, por ejemplo, todos los puntos +1 acumulados en el centro. En este caso, al estar los puntos separados en dos bloques, una función lineal clasifica mucho mejor.

2. Modelos Lineales

2.1 - Algoritmo Perceptron: Implementar la función ajusta_PLA(datos, label, max_iter, vini) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada datos es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, label el vector de etiquetas (cada etiqueta es un valor +1 o -1), max_iter es el número máximo de iteraciones permitidas y vini el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

El algoritmo implementado es el siguiente.

```
ajusta_PLA <- function(datos, label, max_iter = 10000, vini=c(0,0,0)){
 changes = T
 w = vini
 it = 0

 #mientras haya cambios en un recorrido completo y el número de iteraciones no llege al máximo
 while( changes && it < max_iter )
 {
   changes = F

   for( i in 1:length(label) )
   {

     # Si la etiqueta no coincide con la eitqueta predicha por nuestros pesos, corrige
     if( sign(t(w) %*% datos[i,]) != label[i] ) {
       w = w + label[i]*datos[i,]
       changes = T
     }
   }
   it = it + 1

 }
 cat('\n',"Iteraciones: ",it,'\n')

 w
}
```

a) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección 1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en [0, 1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

Primero vamos a obtener los datos necesarios.

```
datos = matrix(c(puntos_etiquetados[,1],puntos_etiquetados[,2]),nrow=50,ncol=2,byrow=F)
```

Para usar PLA, necesitamos añadir una columna de 1 a los datos. Ésto es para que en el cálculo de los pesos podamos obtener una variable independiente.

```
datos = cbind(1,datos)
```

Ahora vamos a calcular pesos.

Desde el vector cero:

```
pesos = ajusta_PLA(datos,puntos_etiquetados[,3])
```

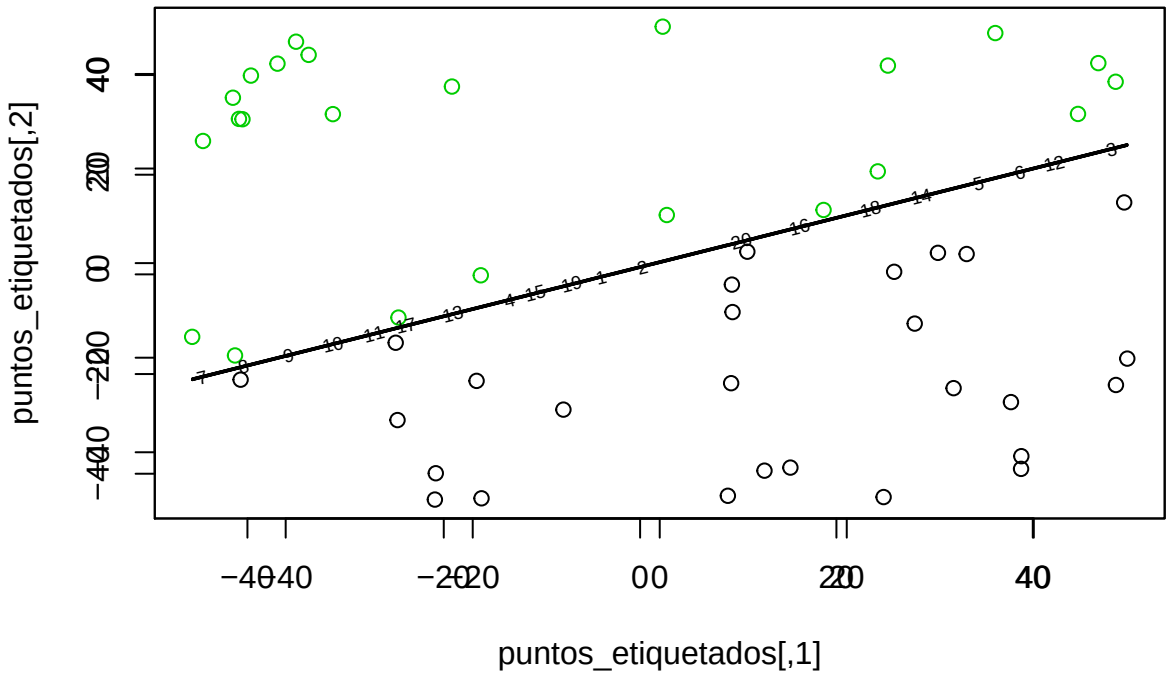
```
##
##  Iteraciones:  3
```

Hemos obtenido los pesos. Como vemos, ajusta_PLA ha necesitado 3 iteraciones desde el vector cero.

Vamos a ver la función clasificadora que obtenemos con estos pesos.

```
funcion_clasificadora = function(x1,x2)
{
  x1*pesos[2]+x2*pesos[3]+pesos[1]
}

plot(puntos_etiquetados,col=puntos_etiquetados[,3]+2)
par(new=T)
pintar_frontera(funcion_clasificadora)
```



Aparentemente la función obtenida clasifica a la perfección. Vamos a contabilizar fallos. Para ello usaremos la siguiente función:

```
contabilizar_fallos <- function(puntos,label,pesos)
{
  f = 0
  for( i in 1:length(label))
  {
    if( sign(puntos[i,1]*pesos[1]+puntos[i,2]*pesos[2]+puntos[i,3]*pesos[3]) != label[i] ) {
      f = f +1
    }
  }

  f
}
```

Los fallos que tiene la función obtenida desde el vector cero son:

```
print(contabilizar_fallos(datos,puntos_etiquetados[,3],pesos))
```

```
## [1] 0
```

Como vemos, no obtenemos ningún fallo.

Vamos a calcular los pesos, pero esta vez partiendo de pesos aleatorios entre [0,1]

```
for( i in 1:10 )
{
  pesos = runif(3,0,1.0)
  cat('\n',"Pesos iniciales: ",pesos,'\n')

  pesos = ajusta_PLA(datos,puntos_etiquetados[,3],vini = pesos)
  cat("Pesos ajustados: ",pesos,'\n')
  cat("Fallos: ")
  cat(contabilizar_fallos(datos,puntos_etiquetados[,3],pesos),'\n','\n')
}
```

```
##
##  Pesos iniciales:  0.3237707 0.5678706 0.9280159
##
##  Iteraciones:  3
## Pesos ajustados:  -1.676229 -34.85529 67.18827
## Fallos: 0
##
##
##  Pesos iniciales:  0.7764013 0.04720967 0.9508571
##
##  Iteraciones:  4
## Pesos ajustados:  -1.223599 -35.94615 72.06949
## Fallos: 0
##
##
##  Pesos iniciales:  0.9407495 0.6231595 0.0676701
##
##  Iteraciones:  4
## Pesos ajustados:  -1.059251 -35.37021 71.1863
## Fallos: 0
##
##
##  Pesos iniciales:  0.5296653 0.1185138 0.583886
##
##  Iteraciones:  4
## Pesos ajustados:  -1.470335 -35.87485 71.70252
## Fallos: 0
##
##
##  Pesos iniciales:  0.3607006 0.319596 0.1799278
##
##  Iteraciones:  4
## Pesos ajustados:  -1.639299 -35.67377 71.29856
## Fallos: 0
##
```

```
##
##  Pesos iniciales:  0.8122641 0.8021944 0.5436312
##
##  Iteraciones:  3
## Pesos ajustados:  -1.187736 -34.62096 66.80389
## Fallos: 0
##
##
##  Pesos iniciales:  0.3122059 0.1634479 0.7666881
##
##  Iteraciones:  4
## Pesos ajustados:  -1.687794 -35.82992 71.88532
## Fallos: 0
##
##
##  Pesos iniciales:  0.5594434 0.267855 0.08105204
##
##  Iteraciones:  4
## Pesos ajustados:  -1.440557 -35.72551 71.19968
## Fallos: 0
##
##
##  Pesos iniciales:  0.6344774 0.5259455 0.9651323
##
##  Iteraciones:  3
## Pesos ajustados:  -1.365523 -34.89721 67.22539
## Fallos: 0
##
##
##  Pesos iniciales:  0.3766457 0.9269196 0.7459828
##
##  Iteraciones:  3
## Pesos ajustados:  -1.623354 -34.49624 67.00624
## Fallos: 0
##
```

El número de iteraciones partiendo desde pesos con el vector cero es de 3 iteraciones.

El número medio de iteraciones partiendo desde pesos con números aleatorios entre [0,1] es de 3.7 iteraciones. En algunos casos ha tardado 3 iteraciones y en otros 4.

Como hemos visto, parece que el algoritmo PLA no necesita muchas iteraciones, independientemente del punto de inicio. Ésto puede ser por que los puntos de inicio elegidos no difieren demasiado de los pesos finales, y tarda poco en ajustar.

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

Vamos a probar con los datos con ruido, a ver cómo ajusta el PLA. Primero obtenemos los datos.

```
datos = matrix(c(puntos_etiquetados_con_ruido[,1],puntos_etiquetados_con_ruido[,2]),nrow=50,ncol=2,byrou
datos = cbind(1,datos)
```

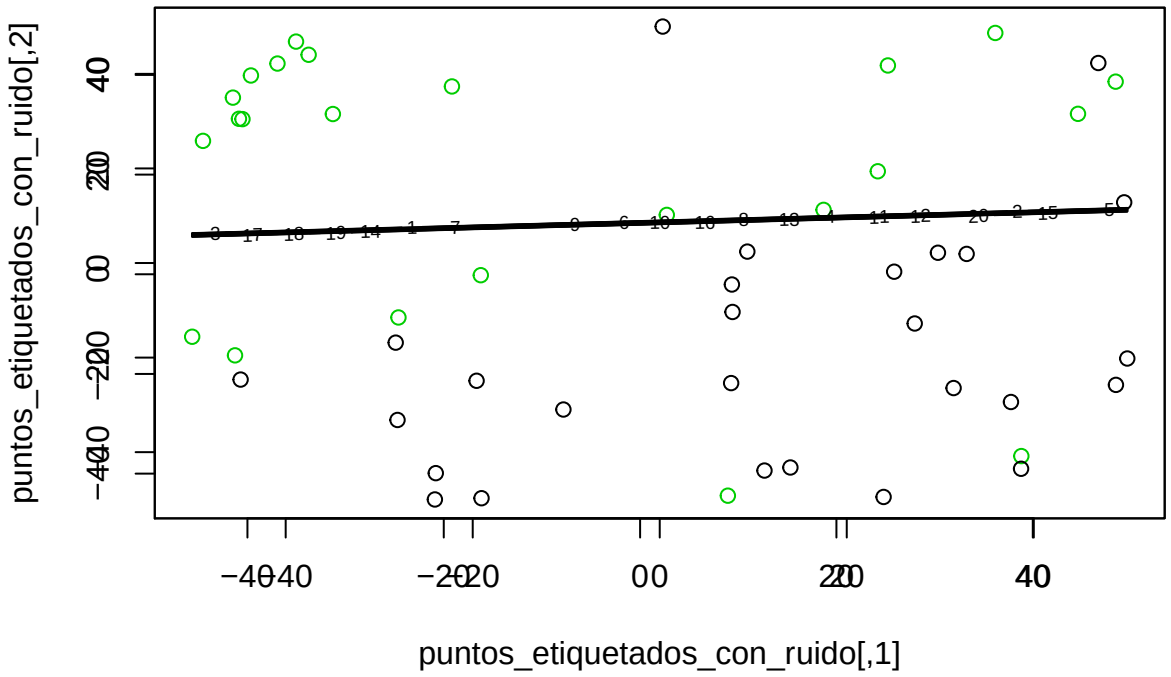
Ejecutamos el algoritmo PLA desde el vector cero:

```
pesos = ajusta_PLA(datos,puntos_etiquetados[,3])
```

```
##
##  Iteraciones:  10000
```

Vamos a ver gráficamente la función estimada.

```
plot(puntos_etiquetados_con_ruido,col=puntos_etiquetados_con_ruido[,3]+2)
par(new=T)
pintar_frontera(funcion_clasificadora)
```



Ejecutamos el algoritmo PLA desde 10 vectores aleatorios:

```
for( i in 1:10 )
{
  pesos = runif(3,0,1.0)
  cat('\n',"Pesos iniciales: ",pesos)

  pesos = ajusta_PLA(datos,puntos_etiquetados_con_ruido[,3],vini = pesos)
  cat("Pesos ajustados: ",pesos,'\n')
  cat("Fallos: ")
  cat(contabilizar_fallos(datos,puntos_etiquetados_con_ruido[,3],pesos),'\n','\n')
}
```

```
##
##  Pesos iniciales:  0.6599877 0.8103561 0.5800322
##  Iteraciones:  10000
## Pesos ajustados:  -125.34 -7.683251 10.15472
## Fallos: 10
##
##
##  Pesos iniciales:  0.5207691 0.9539427 0.6982399
##  Iteraciones:  10000
## Pesos ajustados:  -126.4792 -20.60017 56.17153
```

```
## Fallos: 6
##
##
##  Pesos iniciales:  0.1673164 0.5950642 0.7460641
##  Iteraciones:  10000
## Pesos ajustados:  -110.8327 -9.290556 24.63257
## Fallos: 6
##
##
##  Pesos iniciales:  0.7299215 0.2900139 0.2066105
##  Iteraciones:  10000
## Pesos ajustados:  -131.2701 -16.45773 43.12819
## Fallos: 6
##
##
##  Pesos iniciales:  0.9927414 0.5001998 0.380324
##  Iteraciones:  10000
## Pesos ajustados:  -121.0073 -19.6825 55.44756
## Fallos: 6
##
##
##  Pesos iniciales:  0.4433879 0.4532835 0.8555523
##  Iteraciones:  10000
## Pesos ajustados:  -136.5566 -7.145999 28.06978
## Fallos: 8
##
##
##  Pesos iniciales:  0.8338344 0.3165452 0.7227196
##  Iteraciones:  10000
## Pesos ajustados:  -124.1662 -43.69234 25.52138
## Fallos: 13
##
##
##  Pesos iniciales:  0.89709 0.1133394 0.5914758
##  Iteraciones:  10000
## Pesos ajustados:  -142.1029 -0.6136057 58.38425
## Fallos: 12
##
##
##  Pesos iniciales:  0.3813533 0.3411453 0.6558858
##  Iteraciones:  10000
## Pesos ajustados:  -145.6186 -26.43189 51.76108
## Fallos: 4
##
##
##  Pesos iniciales:  0.5673139 0.04567028 0.4161492
##  Iteraciones:  10000
## Pesos ajustados:  -142.4327 -4.945489 26.54626
## Fallos: 8
##
```

Como en estos casos al haber ruido es imposible que el PLA estime a la perfección, siempre corrige, por lo que sólo para cuando llega al límite de iteraciones.

Vemos que la función estimada no sólo falla en el ruido, si no también en algunos puntos bien clasificados.

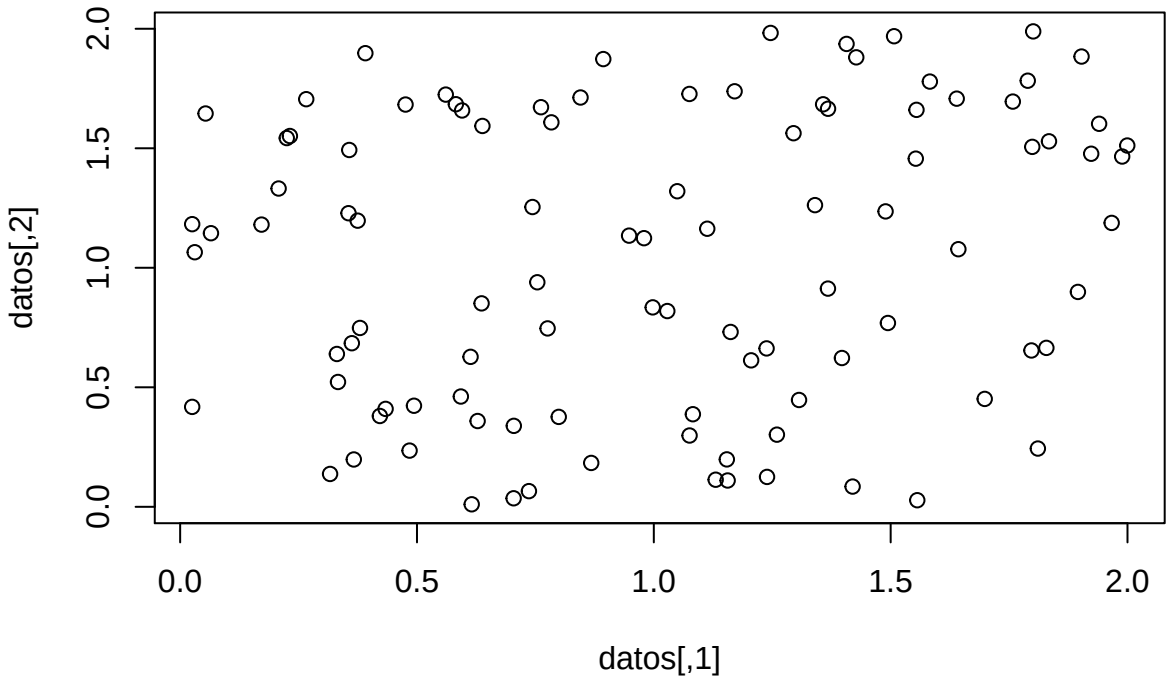
Ésto es por que el ruido ajusta en exceso los pesos.

Podemos concluir que el algoritmo PLA es útil especialmente cuando hay poco o ningún ruido, pero con un ruido significativo la función que obtenemos clasifica con un error importante.

2.2 - Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que la etiqueta y es una función determinista de x. Consideremos d = 2 para que los datos sean visualizables, y sea X = [0, 2] x [0, 2] con probabilidad uniforme de elegir cada x perteneciente a X . Elegir una línea en el plano que pase por X como la frontera entre f (x) = 1 (donde y toma valores +1) y f (x) = 0 (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar N = 100 puntos aleatorios de X y evaluar las respuestas de todos ellos respecto de la frontera elegida

Primero, vamos a generar los datos. Vamos a generar 100 puntos aleatorios entre 0 y 2:

```
datos = simula_unif(100,2,c(0,2))
plot(datos)
```



Vamos a elegir una función para asignar etiquetas a estos datos. Por ejemplo, esta:

```
funcion_clasificadora <- function(x,y)
{
  sign(x - y + 0.5)
}
```

Para ver la función gráficamente, usaremos la siguiente:

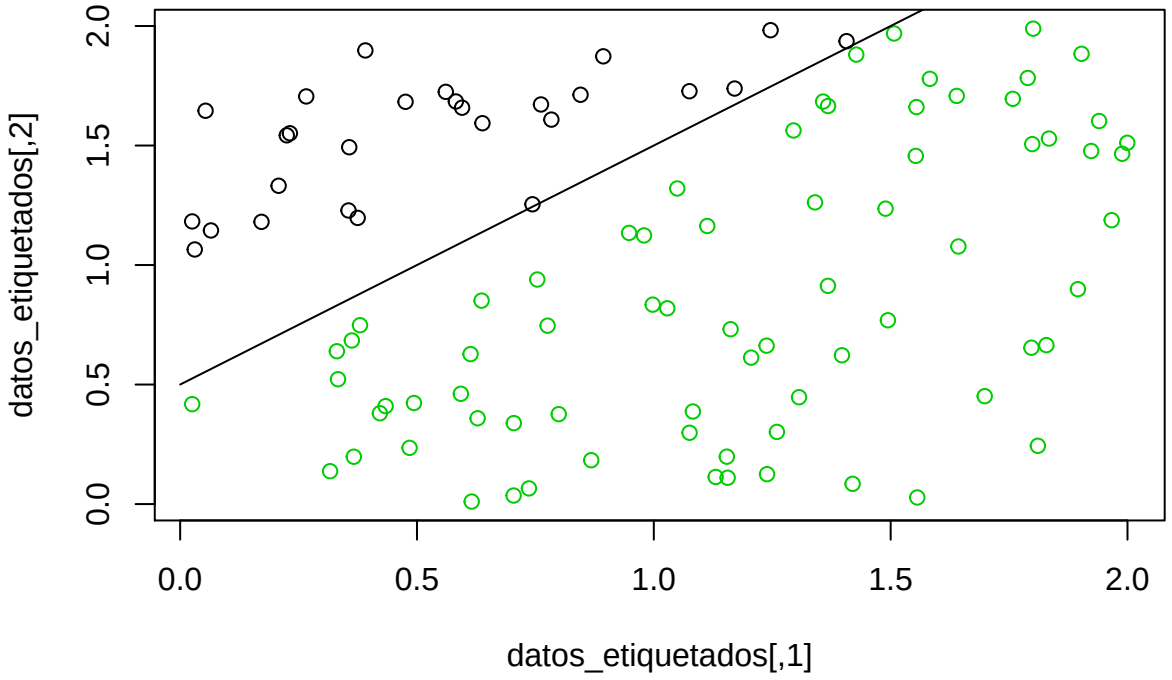
```
funcion_para_dibujar <- function(x)
{
  x + 0.5
}
```

Vamos a asignarle las etiquetas a estos datos:

```
datos_etiquetados = matrix(c(datos,funcion_clasificadora(datos[,1],datos[,2])),nrow=100,ncol=3,byrow=F)
```

El resultado sería el siguiente.

```
plot(datos_etiquetados,col=datos_etiquetados[,3]+2)
lines(c(0,2),c(funcion_para_dibujar(0),funcion_para_dibujar(2)))
```



a) Implementar Regresión Logística (RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:

Inicializar el vector de pesos con valores 0.

Parar el algoritmo cuando ||w(t-1) - w(t)|| < 0.01, donde w(t) denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.

Aplicar una permutación aleatoria, 1, 2, . . . , N , en el orden de los datos antes de usarlos en cada época del algoritmo.

Usar una tasa de aprendizaje de n = 0.01

En Regresión Logística, buscamos una función de probabilidad. A esta función se le pasa como argumentos el punto a evaluar, devolverá un número entre 1 y 0. Cuanto más se acerque el número a 1, más probabilidades tiene el punto evaluado de tener etiqueta +1. Cuanto más se acerque a 0, más probabilidades tiene el punto de tener etiqueta -1.

Pero no tenemos los pesos de esta función. ¿Qué pesos cogemos?

Debemos coger los pesos que minimicen el error de entrada. Esta función de error es la siguiente:

$$Ein(w) = \frac{1}{N}\sum_{i=1}^{N} ln(1 + e^{-y_n w^T x_n})$$

La cuestión está en qué pesos minimizan esta función.

Para calcular estos pesos, vamos a usar el Gradiente Descendiente Estocástico (SGD).

La implementación del SGD es la siguiente.

```
## - Función del algoritmo SGD
SGD <- function(df,x,y,w_inicial=c(0,0,0),num_iteraciones=10000,porcentaje=0.2,nu=0.01,umbral_dif=NaN){

  i = 0 # Contador de iteraciones

  if( !is.nan(umbral_dif) ){ # Si hay umbral de diferencia
    w_antiguo = w_inicial + 2*umbral_dif

    # Mientras no llegemos al máximo de iteraciones y no se llegue al umbral de diferencia
    while( i < num_iteraciones && (abs(w_antiguo-w_inicial) > umbral_dif) ){
      w_antiguo = w_inicial

      # Se actualiza restándole el producto del learning rate y el gradiente calculado
      w_inicial = w_inicial - nu*df(w_inicial,x,y,porcentaje)
      i = i+1
    }
  }
  else # Si no hay umbral de diferencia
  {
    while( i < num_iteraciones ){
      w_antiguo = w_inicial
      w_inicial = w_inicial - nu*df(w_inicial,x,y,porcentaje)
      i = i+1
    }
  }


  cat("\nSGD ha acabado. Pesos: ", w_inicial)
  cat("\nIteraciones: ", i, '\n')

  w_inicial
}
```

El gradiente que usaremos en el SGD es el siguiente:

$$g_t = \frac{1}{N}\sum_{i=1}^{N} \frac{-y_n x_n}{1 + e^{y_n w(t)^T x_n}}$$

Implementado, y teniendo en cuenta que en el SGD no evaluamos todos los puntos, si no un porcentaje aleatorio, quedaría así:

```
gradiente <- function(w,x,y,porcentaje){
  suma = c(0,0,0)
  N = length(y)*porcentaje
```

```
  for( i in 1:N ){
    irand = sample(length(y),1)

    a = y[irand]*x[irand,]
    b = exp(y[irand]*t(w)%*%x[irand,])

    suma = suma + as.vector(a)/(as.vector(b)+1)


  }

  suma = suma/length(y)

  -suma


}
```

Ya tenemos las funciones necesarias para calcular los pesos.

Para calcular los pesos, debemos meter una columna de unos a los datos, para obtener la variable independiente.

```
datos = cbind(1,datos)
```

Ya está todo listo. Vamos a calcular los pesos. PD: No he usado el umbral de diferencia de 0.01, ya que llega instantáneamente y al algoritmo no le da tiempo de ajustar bien los pesos.

```
pesos=c(0,0,0)
pesos = SGD(gradiente,datos,datos_etiquetados[,3],num_iteraciones = 100000, pesos, porcentaje=0.2,nu=0.
```

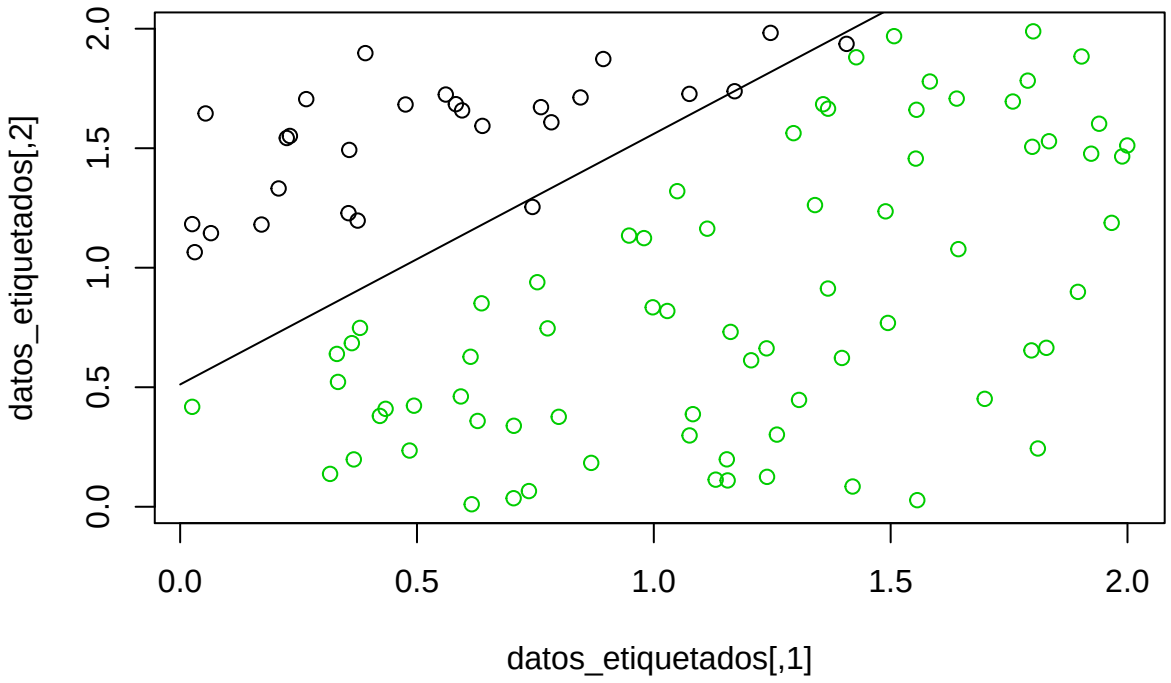
```
##
## SGD ha acabado. Pesos:  2.53668 5.198937 -4.958186
## Iteraciones:  1e+05
```

Ya tenemos los pesos. Para ver esta función sobre los puntos, usaremos la siguiente función, despejando la y:

```
funcion_rl_para_dibujar <- function(x)
{
  -(pesos[1]+x*pesos[2])/pesos[3]
}
```

Veamos esta función sobre los puntos:

```
plot(datos_etiquetados,col=datos_etiquetados[,3]+2)
lines(c(0,2),c(funcion_rl_para_dibujar(0),funcion_rl_para_dibujar(2)))
```



Como vemos, acierta con casi todos los puntos.

Vamos a estimar las etiquetas con la nueva función a partir de nuestros datos. Para ello, necesitamos la sigmoide:

```
sigmoide <- function(x)
{
  1/(1+exp(-x))
}
```

Consideraremos que un punto tiene etiqueta +1 si su probabilidad es mayor o igual a 0.5, y -1 si su probabilidad es menor a 0.5.

```
datosSGD = datos_etiquetados

for( i in 1:100 )
{
  if( sigmoide(pesos%*%datos[i,]) >= 0.5 ) datosSGD[i,3] = 1
  else datosSGD[i,3] = -1
}
```

Vamos a calcular el error que hemos obtenido.

```
fallos = 0
for( i in 1:100 )
{
  if( datosSGD[i,3] != datos_etiquetados[i,3] ) fallos = fallos +1
}

print(fallos)
```

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

Hemos obtenido 3 fallos, por lo que parece que nuestra función clasifica bien, como ya intuíamos en el gráfico.

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

Vamos a generar 1000 nuevos puntos, y los clasificaremos de acuerdo a su probabilidad estimada. Le añadiremos una columna de unos para poder usar a la hora de etiquetar la variable independiente

```
datosEout = simula_unif(1000,2,c(0,2))
datosEout_para_estimar = cbind(1,datosEout)
```

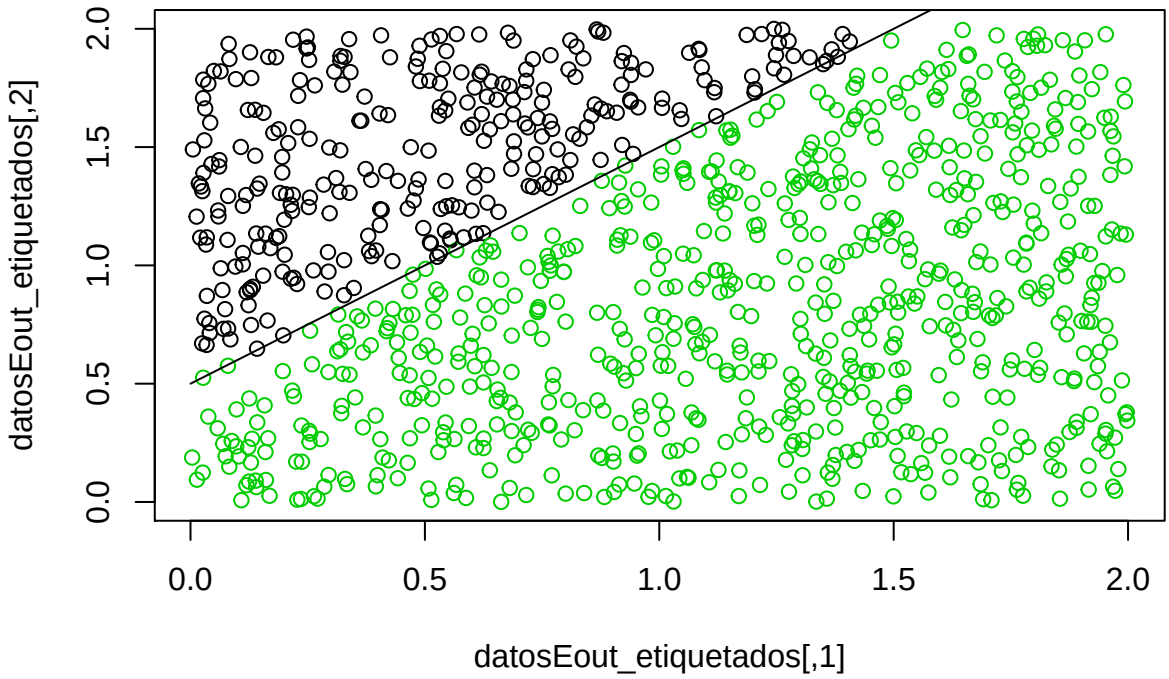
Primero, vamos a etiquetar los puntos de acuerdo a la función original que elegimos aleatoriamente:

$$y = x + 0.5$$

```
datosEout_etiquetados = matrix(c(datosEout,funcion_clasificadora(datosEout[,1],datosEout[,2])),nrow=1000
datosEout_bien_etiquetados = datosEout_etiquetados
```

Gráficamente los puntos son estos:

```
plot(datosEout_etiquetados,col=datosEout_etiquetados[,3]+2)
lines(c(0,2),c(funcion_para_dibujar(0),funcion_para_dibujar(2)))
```

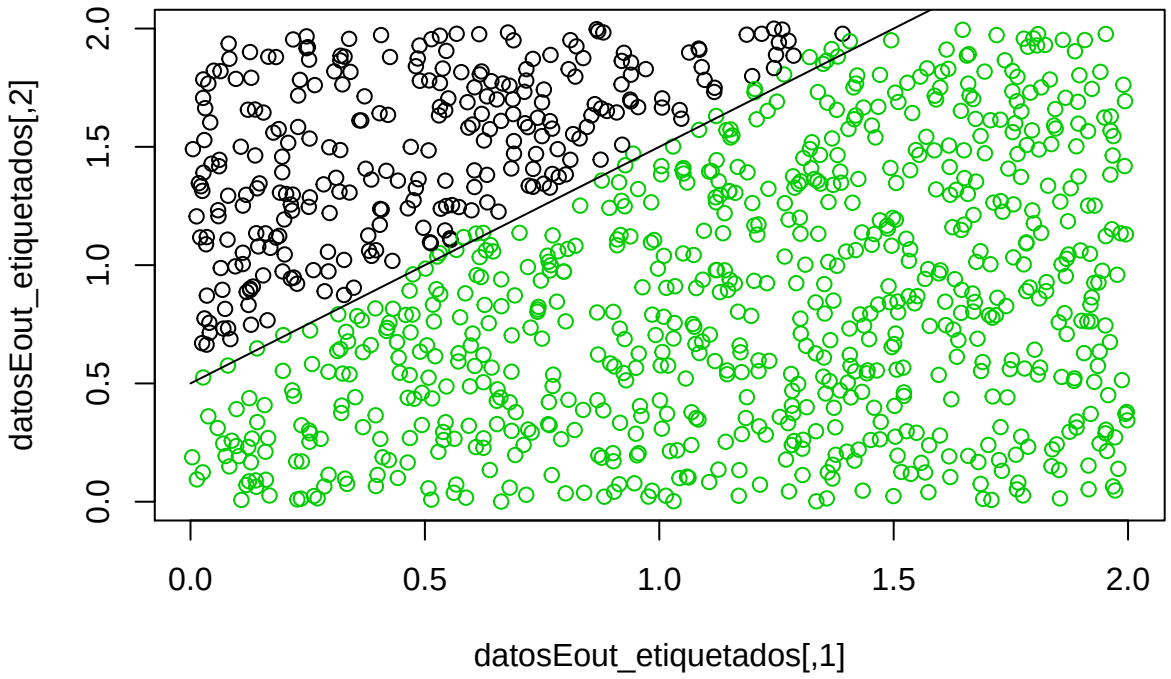


Vamos a etiquetar los puntos de acuerdo a nuestra estimación de probabilidad. Si la probabilidad es >= 0.5, etiquetaremos el punto como +1. Si la probabilidad es <0.5, etiquetaremos el punto como -1.

```
for( i in 1:1000 )
{
  if( sigmoide(pesos%*%datosEout_para_estimar[i,]) >= 0.5 ) datosEout_etiquetados[i,3] = 1
  else datosEout_etiquetados[i,3] = -1
}
```

Gráficamente los puntos, etiquetados con nuestra función, son estos:

```
plot(datosEout_etiquetados,col=datosEout_etiquetados[,3]+2)
lines(c(0,2),c(funcion_para_dibujar(0),funcion_para_dibujar(2)))
```



Como vemos, algunos puntos están mal clasificados. Vamos a calcular este error exactamente.

```
fallos = 0
for( i in 1:1000 )
{
  if( datosEout_etiquetados[i,3] != datosEout_bien_etiquetados[i,3] ) fallos = fallos +1
}

print(fallos)
```

```
## [1] 18
```

18 errores de 1000. Un 0.018% de error. Como vemos, hemos estimado una función de probabilidad que estima con muy buena fiabilidad.