AA - Practica 2

Guillermo Gómez

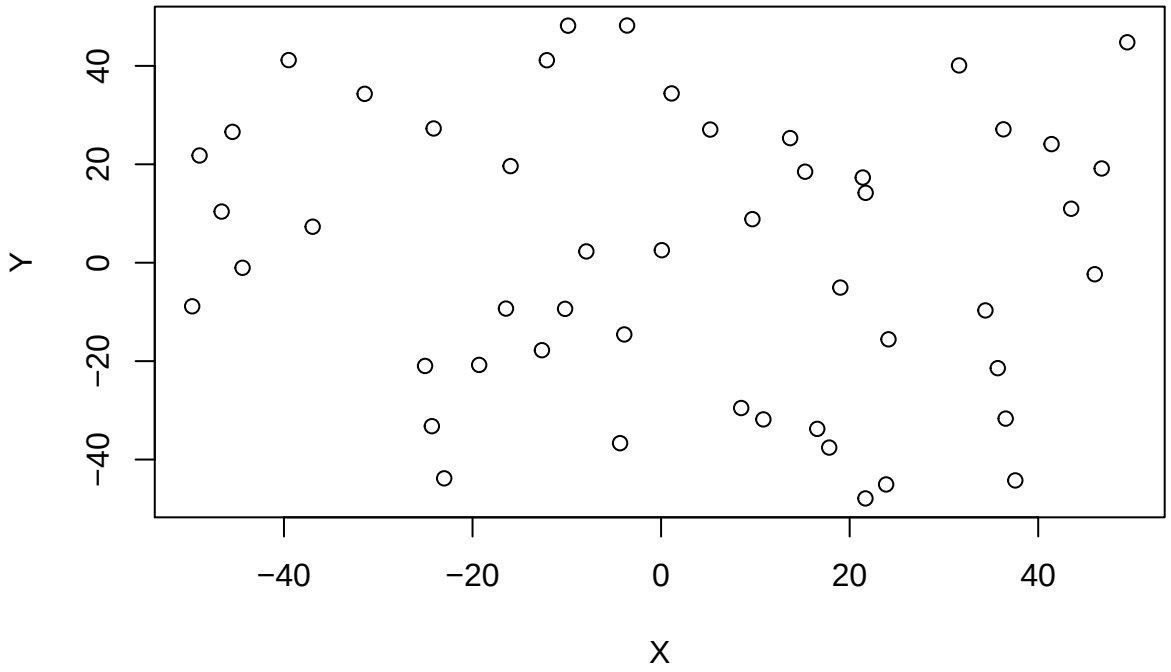
April 17, 2018

1. Ejercicio sobre la complejidad de H y el ruido

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

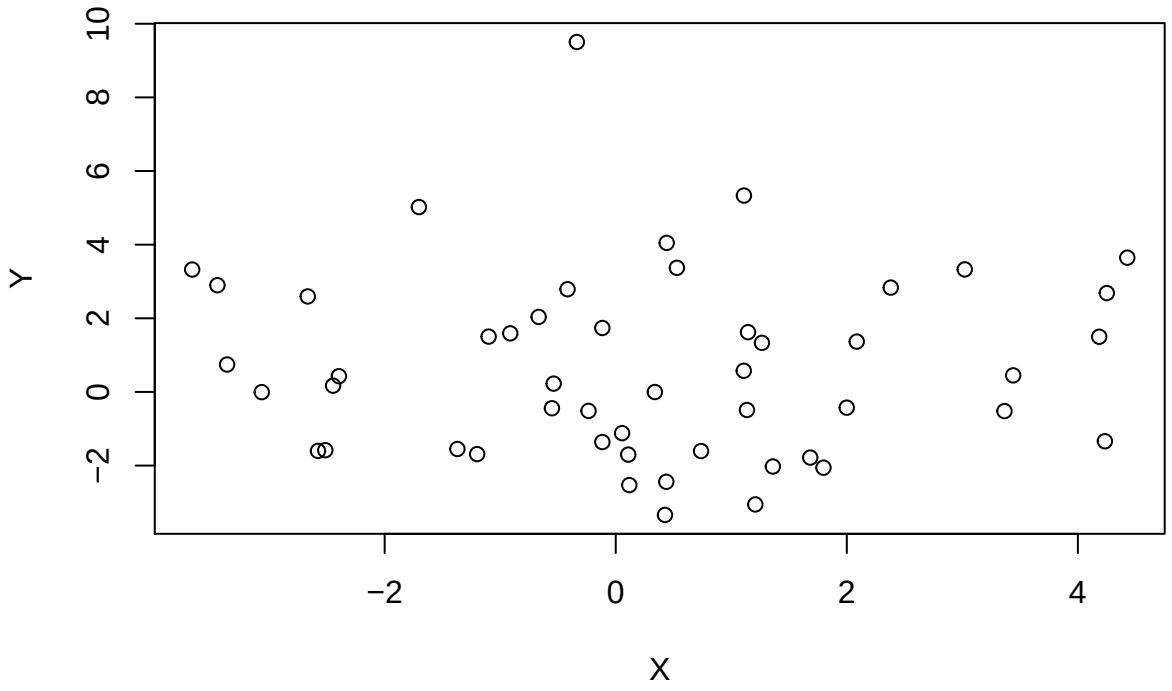
1.1.a. Considere $N = 50$, $dim = 2$, rango $= [-50, 50]$ con simula_unif(N,dim,rango)

```
nubea = simula_unif(N=50,dims = 2,rango = c(-50,50))
colnames(nubea) = c("X","Y")
plot(nubea)
```



1.1.a. Considere $N = 50$, $dim = 2$, sigma $= [5, 7]$ con simula_gaus(N,dim,sigma)

```
nubeb = simula_gaus(N=50,dim=2,sigma = c(5,7))
colnames(nubeb) = c("X","Y")
plot(nubeb)
```



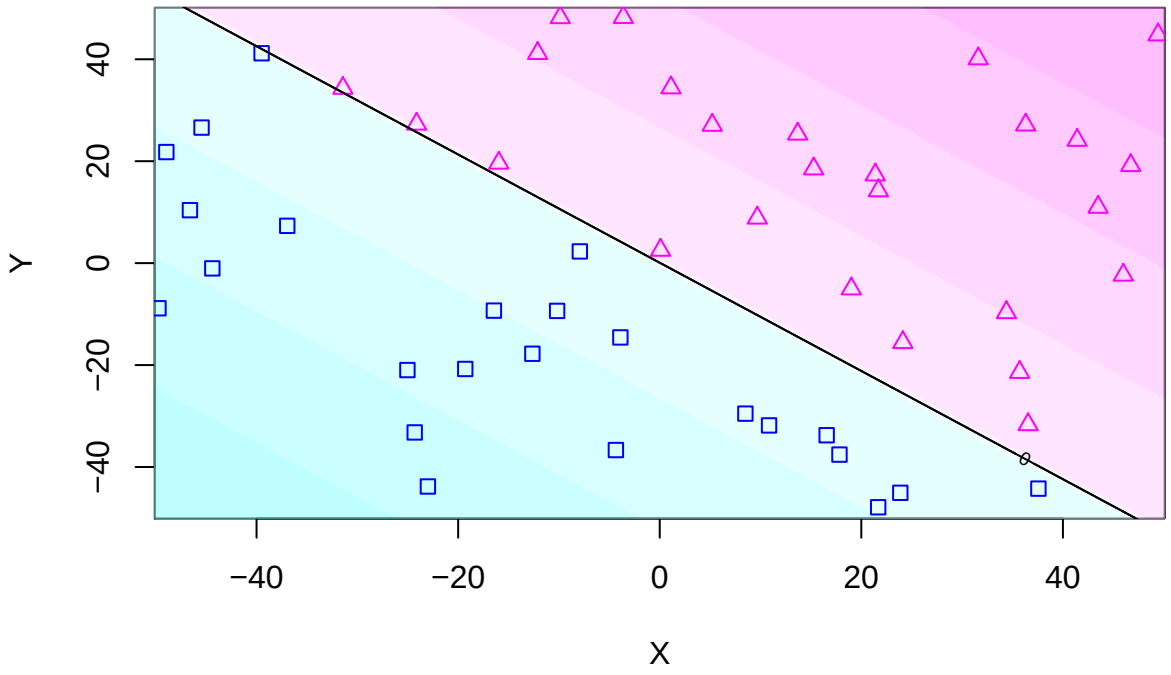
1.2. Con ayuda de la función simula_unif() generar una muestra de puntos 2D a los que vamos a añadir una etiqueta usando el signo de la función $f(x, y) = y - ax - b$, es decir, el signo de la distancia de cada punto a la recta simulada con simula_recta().

1.2.a. Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello.

```
rc = simula_recta()
z = apply(nubea,1,FUN=function(c){sign(c[2]-c[1]*rc[1]-rc[2])})
zori = z
fl = function(x,y){y -rc[1]*x - rc[2]}

#Imprimir error
print(paste("Ein: ", length(which(z != sign(fl(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0"
```

```
pintar_frontera(fl)
points(nubea,col=z+5, pch=z+1) #+5 para azul y rojo
abline(a=rc[2],b=rc[1])
```

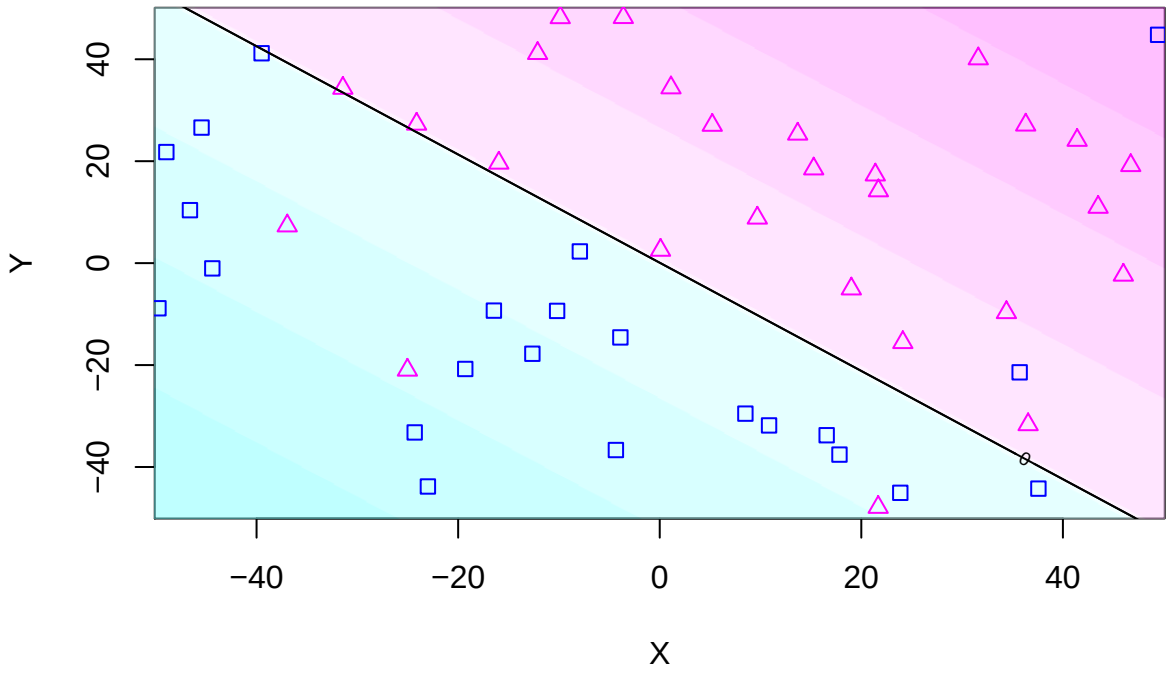


1.2.b. Modifique de forma aleatoria un 10% de las etiquetas positivas y otro 10% de negativas y guarde los puntos con sus nuevas etiquetas. DIbuje de nuevo la gráfica anterior. (Ahora hay puntos mal clasificados respecto de la recta)

```
s=sample(1:length(z),length(z)%/%10,replace = FALSE)
z[s] = z[s]*-1
zrand = z

#Imprimir error
print(paste("Ein: ", length(which(z != sign(fl(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.1"
```

```
pintar_frontera(fl)
points(nubea,col=z+5, pch=z+1) #+5 para azul y rojo
abline(a=rc[2],b=rc[1])
```



Tal y como cabría esperar el error es 0.1, justo los valores que hemos modificado aleatoriamente.

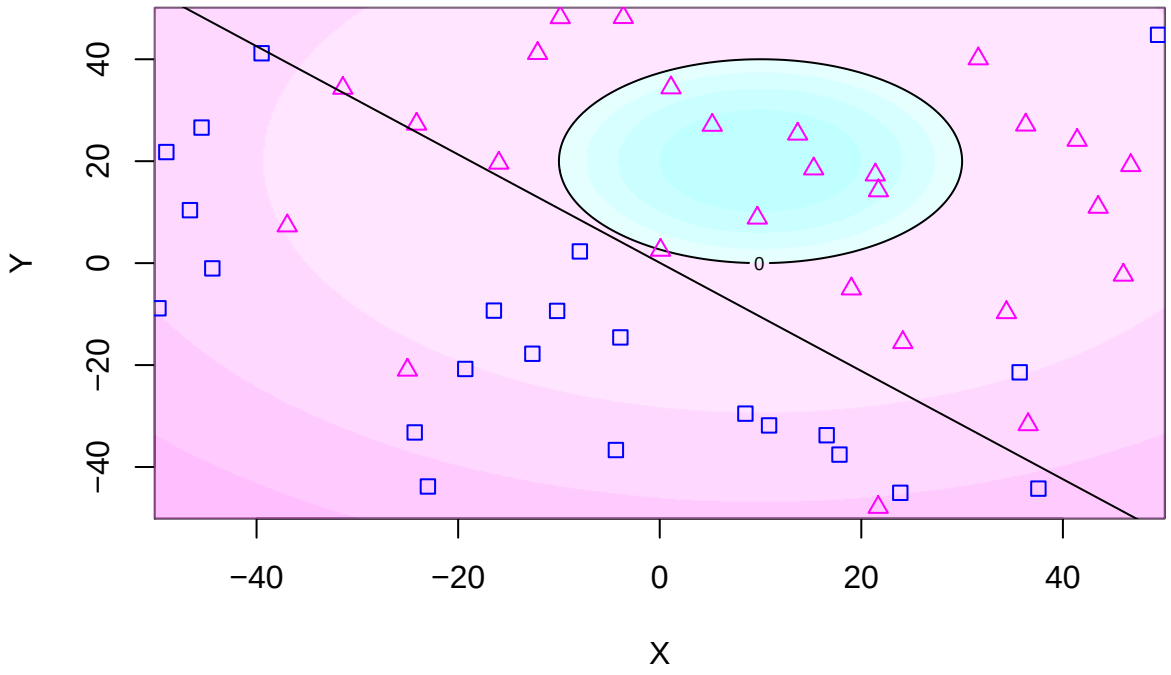
1.3. Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta.

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

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

#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.6"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1)
abline(a=rc[2],b=rc[1])
```

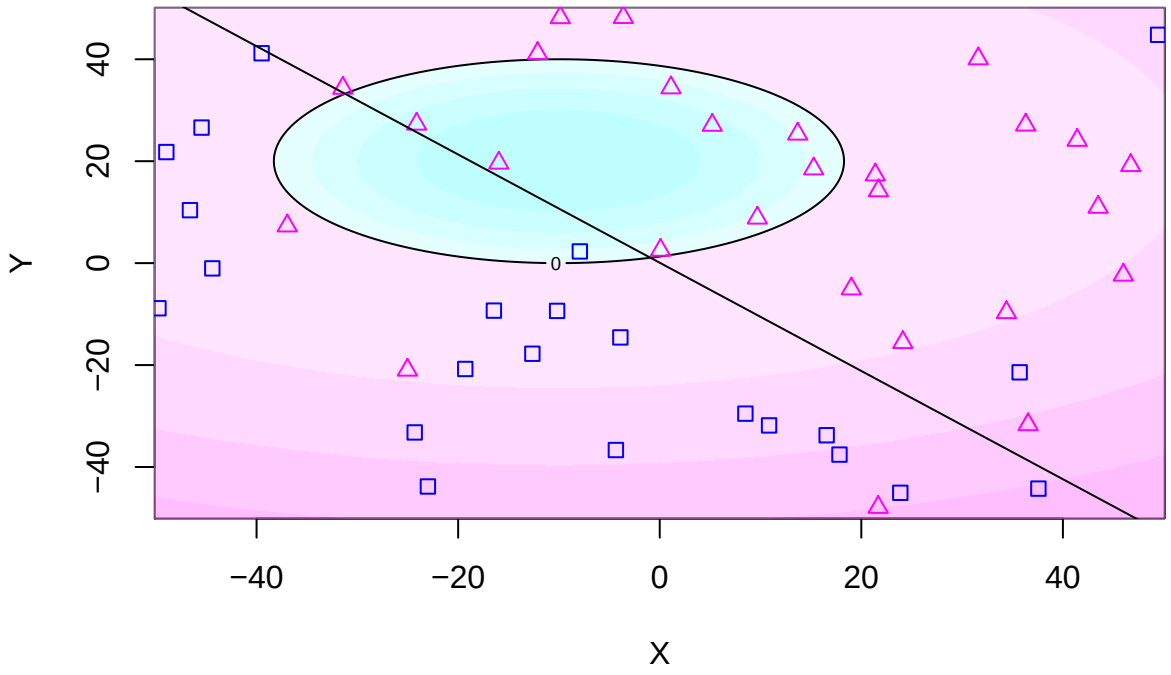


1.3.b. $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$

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

#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.6"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1)
abline(a=rc[2],b=rc[1])
```

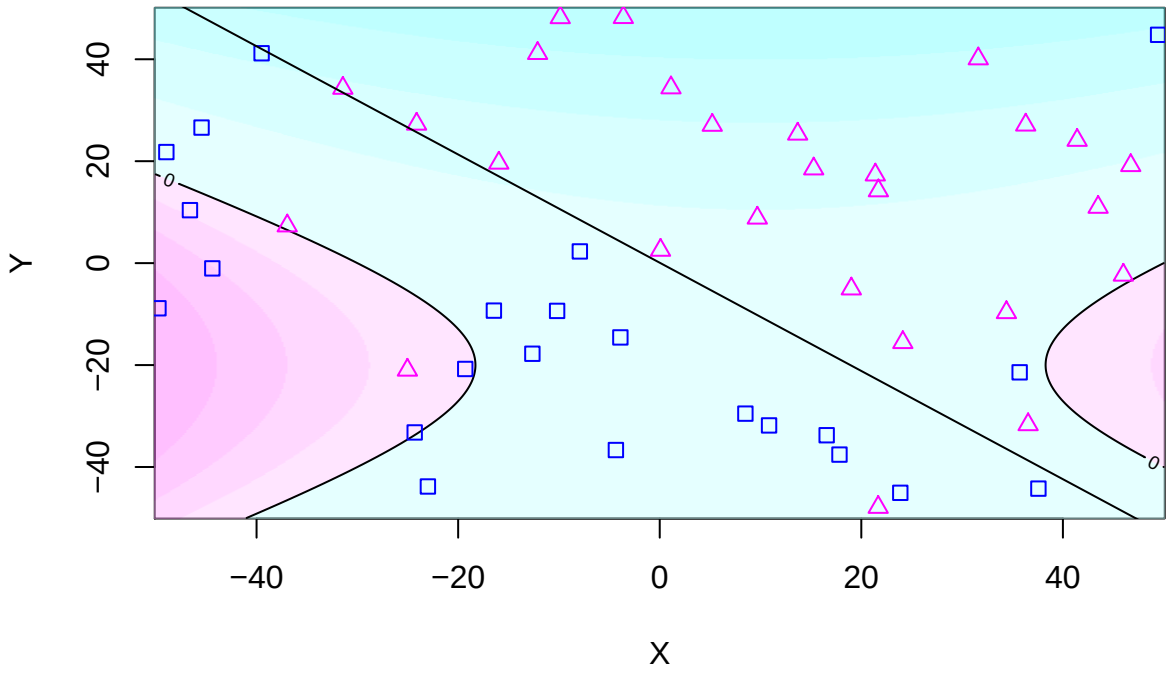


1.3.c. $f(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$

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

#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.62"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1)
abline(a=rc[2],b=rc[1])
```

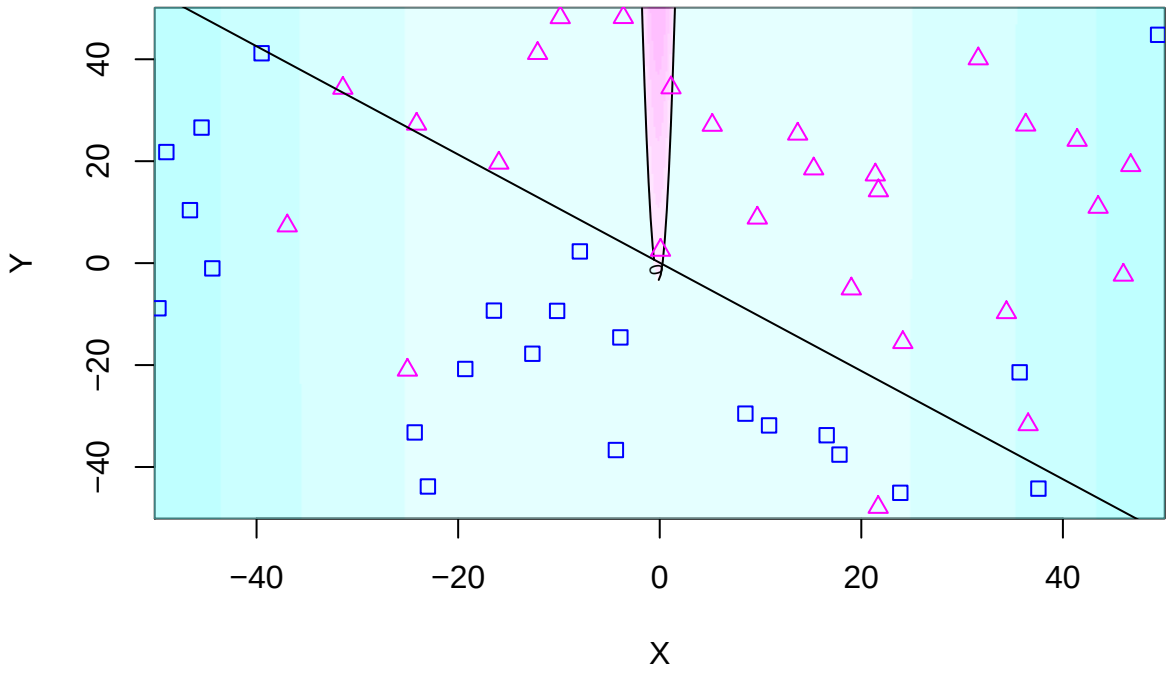


1.3.d $f(x, y) = y - 20x^2 - 5x + 3$

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

#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.5"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1) #+3 para azul y rojo
abline(a=rc[2],b=rc[1])
```



1.3.e Análisis de los resultados

Como cabría esperar, las cuatro funciones que definían sendas fronteras de clasificación arrojan unos resultados mucho peores, entre un 50% y un 62% peor y no podría ser de otra manera por dos motivos.

En primer lugar, **las funciones no están ajustadas a los datos**, tomemos por ejemplo la primera función y desarrollémosla.

$$f(x,y) = (x-10)^2 + (y-20)^2 - 400$$
$$f(x,y) = x^2 + 100 - 10x + y^2 + 400 - 20y - 400$$
$$f(x,y) = x^2 - 10x + y^2 - 20y + 100$$

Ahora la ponemos en un formato que podamos ajustar por regresión lineal, usamos la regresión lineal por simplicidad y rapidez a fin de mostrar un ejemplo.

$$f(x_1,x_2) = x_1^2 - 10x_1 + x_2^2 - 20x_2 + 100$$
$$f(x_1,x_2,x_3,x_4) = x_1 - 10x_2 + x_3 - 20x_4 + 100$$

Y definamos los pesos a ajustar, sustituyendo las constantes por w.

$$f(x_1,x_2,x_3,x_4) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$$

Siendo $x_1 = x$, $x_2 = x^2$, $x_3 = y$, $x_4 = y^2$

```
getPIw = function(X,Y){
  Xt = t(X)
  Xpi=solve(Xt%*%X)%*%Xt
  w=Xpi%*%Y
  w
```

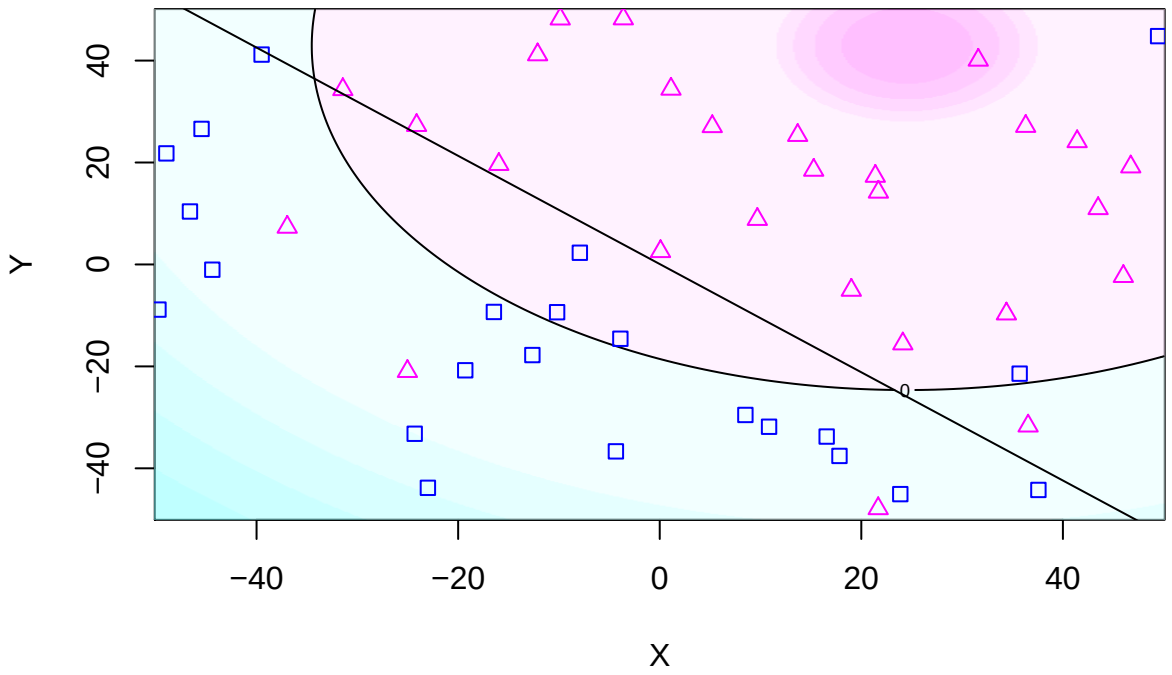
```
}

X = matrix(c(nubea[,1],nubea[,1]^2,nubea[,2],nubea[,2]^2,rep_len(1,50)),ncol = 5)
w = getPIw(X,z)

print(w)
```

```
##               [,1]
## [1,]  0.0147811832
## [2,] -0.0003014422
## [3,]  0.0197666862
## [4,] -0.0002304355
## [5,]  0.4457360632
```

```
f = function(x,y){x*w[1]+x^2*w[2]+y*w[3]+y^2*w[4]+w[5]}

#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0.18"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1) #+3 para azul y rojo
abline(a=rc[2],b=rc[1])
```



Ahora sí nos encontramos en una situación más justa para compararlas y lo que observamos es que el error en la predicción ha aumentado aproximadamente un 60% y esto es debido al segundo motivo por el que estos modelos más complicados no son capaz de predecir mejor, y es que **no reflejan mejor el fenómeno que subyace a los datos**. De hecho, si observamos los valores para la w vemos que los que corresponden a los cuadrados son 2 órdenes de magnitud menor que los que corresponden a las variables elevadas a la unidad; es más, cabría esperar que si usáramos por ejemplo SGD, con suficientes datos e iteraciones —ya que los datos son CASI linealmente separables— los pesos de las varibales al cuadrado se aproximaran cada vez más a 0 y la función acabara dibujando una enorme elipse que intentara superponer la sección inferior izquierda de su frontera con la línea que corresponde a la función original que definió los datos.

En definitiva, la conclusión que podemos extraer de aquí es que la clase de funciones elegida para ajustar a unos datos tiene que ser idealmente tan *compleja* como la naturaleza del fenómeno que queremos explicar, mayor complejidad o menor complejidad producirá necesariamente un mayor E_{out} que la clase ideal. Por ejemplo, imaginemos que queremos predecir si un usuario compraría unos zapatos de una marca dada, donde el perfil de usuarios que compran esa marca son aquellos con edades entre 20 y 30 años con un sueldo de entre 1000€ y 2000€, si nos fijamos vemos que ahora el fenómenos es explicado por la intersección de dos horquillas, esto no es linealmente separable y dibujaría como un rectángulo de valores positivos rodeados de valores negativos; en esta situación, la función que utiliza los cuadrados de las variables tendrá mayor capacidad para explicar el fenómenos que una recta, porque se ajusta más a la naturaleza que explica el fenómeno.

2. Modelos lineales

2.1. Algoritmo perceptrón

```
ajusta_PLA = function(datos,label,max_iter,vini,calc_ein=FALSE,alpha=1){
  if(ncol(datos)!=(length(vini)-1)) stop("tamaño incorrecto")
  datos = cbind(datos,ind=1)
  w = vini
  wt = w-1
  iter = 0
  ein = c()
  while(!all(w == wt) && iter < max_iter){
    #print(max_iter)
    wt = w
    iter = iter+1
    for(i in 1:nrow(datos)){
      res = sign(w%*%t(datos[i,,drop=FALSE]))
      #print(res)
      if(res != label[i]){
        #update
        w = w + alpha*(label[i]*datos[i,])

      }
    }
    if(calc_ein){
      cl = sign(w %*% t(datos))
      ein = c(ein,length(which(cl != label))/length(label))
    }
  }
  #print(paste("Iters:",iter))
  list(w=w,iters=iter,ein=ein)
}
```

2.1.a Ejecutar el algoritmo PLA con los datos del ejercicio 1.2.a

Con w igual al vector 0, como el funcionamiento del PLA es determinista sólo lo ejecutamos una vez

```
datos = nubea
z = zori
w = matrix(rep_len(0,ncol(nubea)+1),nrow = 1)
```

```
res = ajusta_PLA(datos,z,100,w)
w = res$w
iters = res$iters
f = function(x,y){x*w[1]+y*w[2]+w[3]}

#Imprimir w
print(paste("Iters:",iters))
```

```
## [1] "Iters: 4"
```

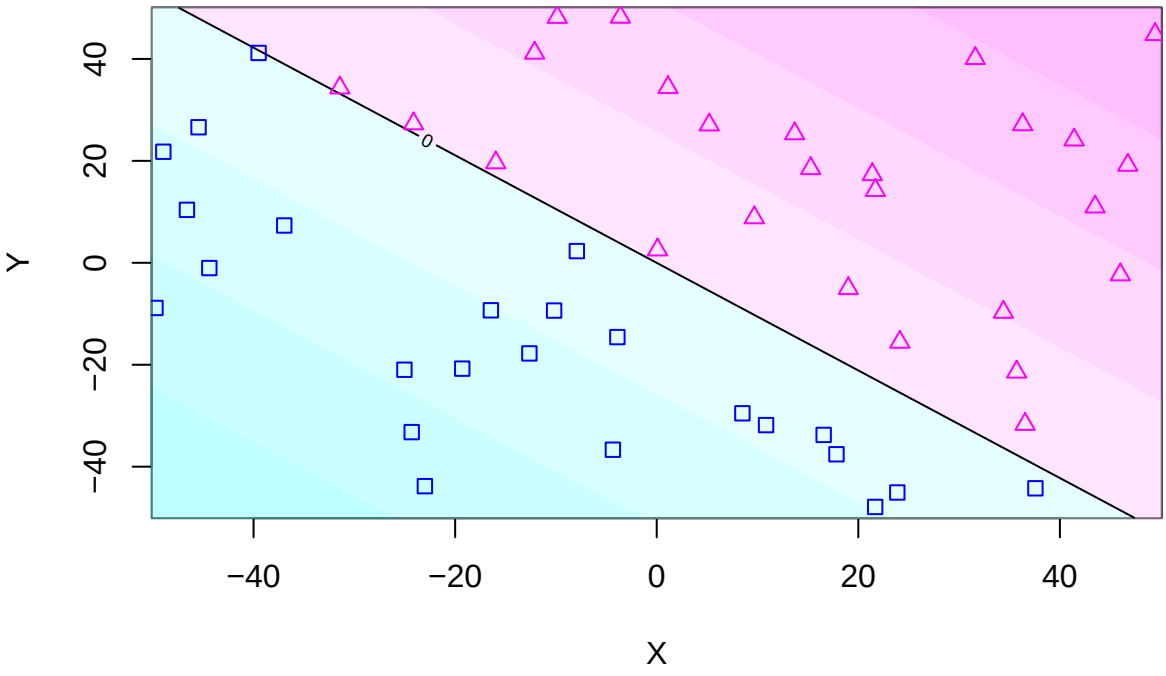
```
cat("w:",w,"\n")
```

```
## w: 84.83451 80.38427 1
```

```
#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0"
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1) #+3 para azul y rojo
```



Realizado con 100 repeticiones en vez de 10

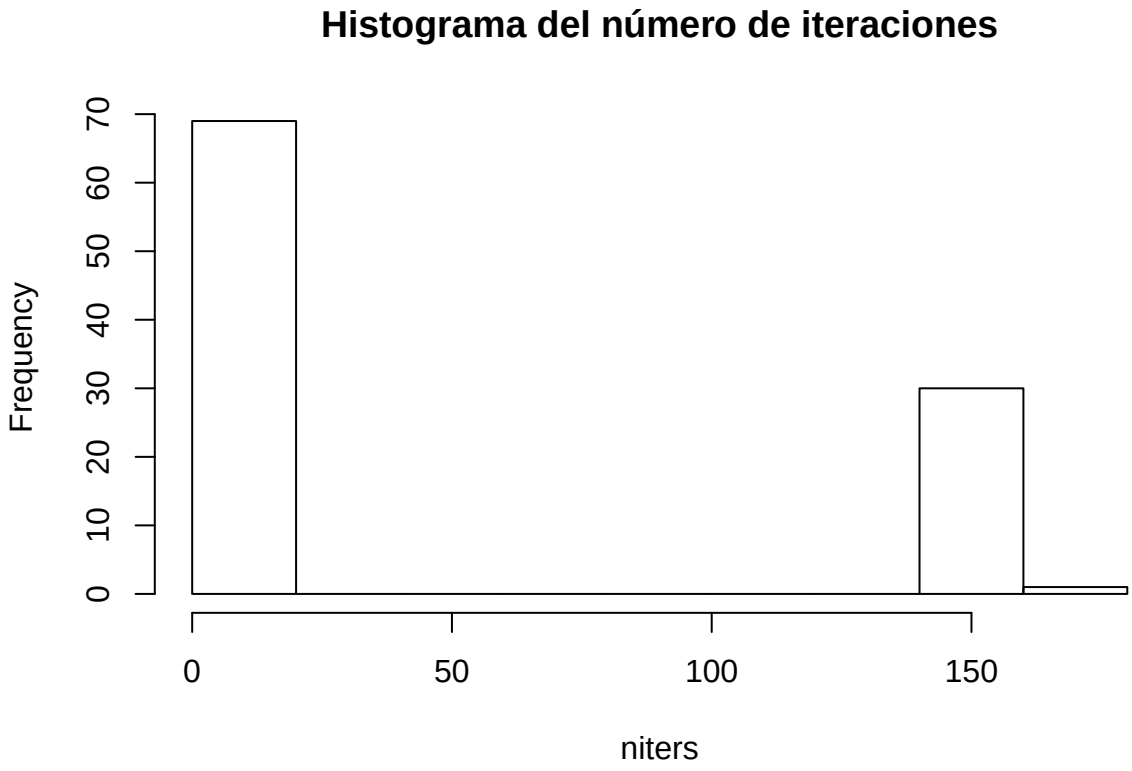
Con w inicializado aleatoriamente, lo probamos 100 veces. Como los datos son linealmente separables y nos aseguraremos de que salga de la función por converger y no por límite de iteraciones, no vamos a comprobar ni E_{in} ni w. Esta última puede variar entre iteraciones pero siempre separando correctamente los datos así que no nos preocupamos en almacenarla para el estudio de las iteraciones.

```
datos = nubea
z = zori
w_ini = lapply(1:100,function(i){runif(3,0,1)})
res = lapply(1:100,function(i){ajusta_PLA(datos,z,1000,matrix(w_ini[[i]],nrow=1))})

ws = lapply(res,function(e){e$w})
ws = matrix(unlist(ws),ncol = 3,byrow = TRUE)
```

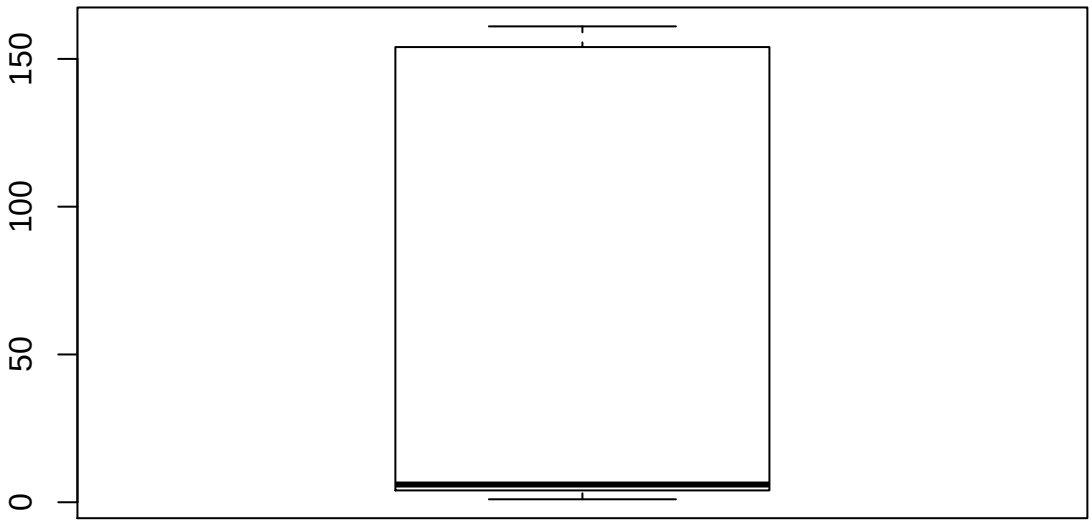
```r
niters = as.integer(lapply(res,function(e){e$iters}))

#Comprobamos si alguno se ha salido del número de iteraciones
paste("Número de ajustes que agotaron el límite: ",length(which(niters == 1000)))
```

```
## [1] "Número de ajustes que agotaron el límite:  0"
```

```r
#Histograma
hist(niters, main="Histograma del número de iteraciones")
```




```r
#Diagrama de caja y bigotes
boxplot(niters)
```



```r
#Resumen
print("Summary:")
```

```
## [1] "Summary:"
```

```r
summary(niters)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    4.00    6.00   51.16  154.00  161.00
```

Dibujamos uno para comprobar que todo es correcto

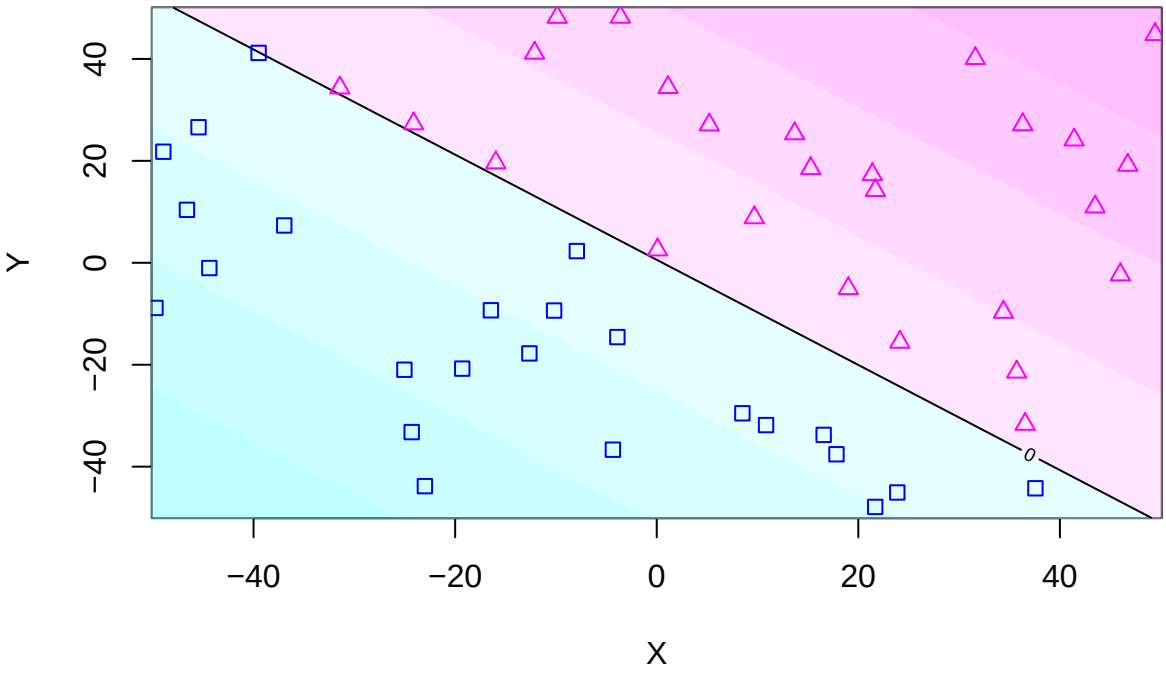
```r
w = matrix(runif(3,0,1),nrow = 1)
res = ajusta_PLA(datos,z,1000,w)
w = res$w
iters = res$iters
f = function(x,y){x*w[1]+y*w[2]+w[3]}

#Imprimir w
cat("w:",w,"\n")
```

```
## w: 458.8857 444.8755 -260.8184
```

```r
#Imprimir error
print(paste("Ein: ", length(which(z != sign(f(nubea[,1],nubea[,2]))))/dim(nubea)[1]))
```

```
## [1] "Ein:  0"
```

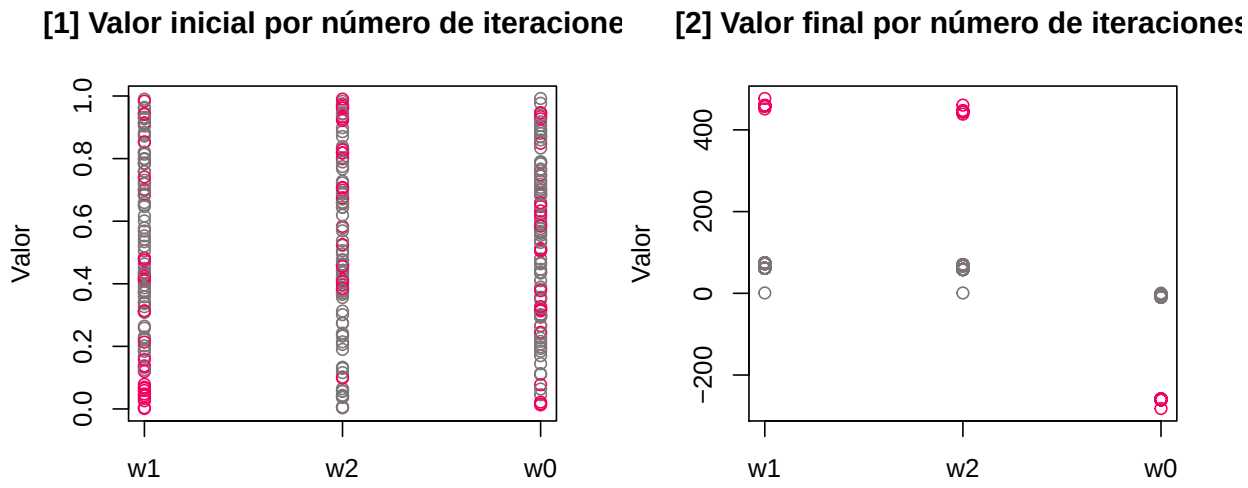
```r
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1) #+3 para azul y rojo
```

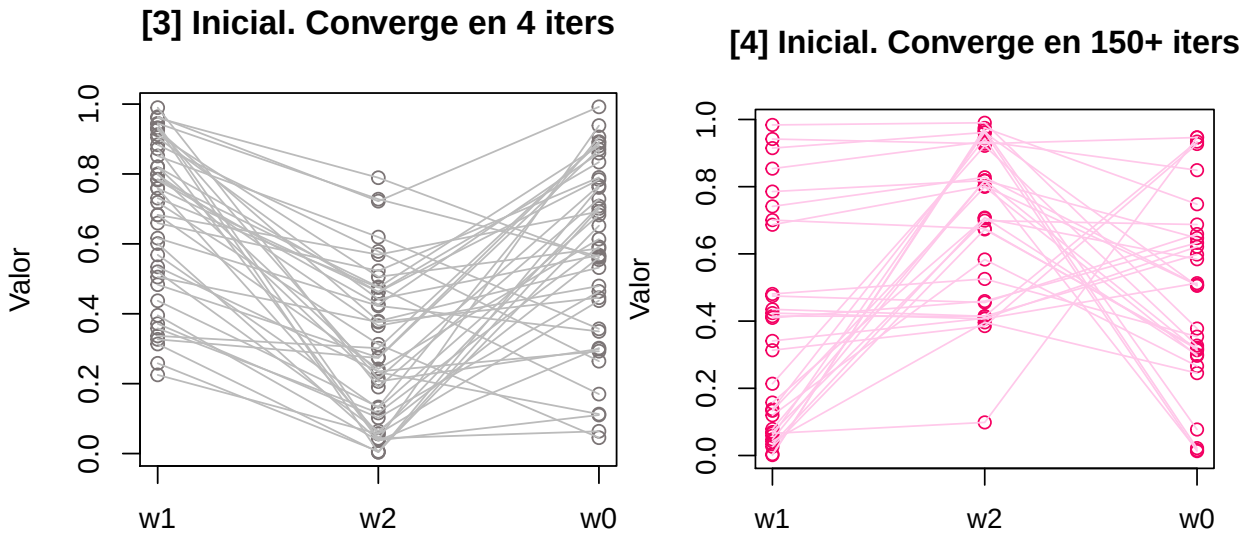


2.1.a. Conclusiones

Aclarar que los grises equivalen a pocas iteraciones y los rosas vivos a muchas iteraciones, sin embargo no he sabido añadir esta leyenda al plot sin usar ggplot

```
plot(y=unlist(w_ini),x=rep_len(c(1,2,3),300),col=hcl(c=rep(niters,each=3),
                  l=rep_len(50,300)),xaxt="n", ylab = "Valor", xlab = "",
                  main="[1] Valor inicial por número de iteraciones")
axis(1,at=c(1,2,3),las=2,labels = c("w1","w2","w0"), las=1)

plot(y=unlist(t(ws)),x=rep_len(c(1,2,3),300),col=hcl(c=rep(niters,each=3),
                  l=rep_len(50,300)),xaxt="n", ylab = "Valor", xlab = "",
                  main="[2] Valor final por número de iteraciones")
axis(1,at=c(1,2,3),las=2,labels = c("w1","w2","w0"), las=1)

cond = niters == 4
len=length(which(cond))
plot(y=unlist(w_ini[cond]),x=rep_len(c(1,2,3),len*3),col=hcl(c=rep(niters[cond],each=3),
                  l=rep_len(50,len*3)),xaxt="n", ylab = "Valor", xlab = "",
                  main="[3] Inicial. Converge en 4 iters")
axis(1,at=c(1,2,3),las=2,labels = c("w1","w2","w0"), las=1)
for(i in 1:len){
  lines(y=unlist(w_ini[cond][i]),x=c(1,2,3),col="#BBBBBB")
}

cond = niters > 150
len=length(which(cond))
plot(y=unlist(w_ini[cond]),x=rep_len(c(1,2,3),len*3),col=hcl(c=rep(niters[cond],each=3),
                  l=rep_len(50,len*3)),xaxt="n", ylab = "Valor", xlab = "",
                  main="[4] Inicial. Converge en 150+ iters")
axis(1,at=c(1,2,3),las=2,labels = c("w1","w2","w0"), las=1)
for(i in 1:len){
  lines(y=unlist(w_ini[cond][i]),x=c(1,2,3),col="#ffc8ea")
}
```





Como se puede observar, parece que no existe relación clara entre el valor inicial de los pesos y el tiempo que tarda en converger, pues como se puede observar en [1],[3],[4] no se aprecia —al menos de forma inmediata— una tendencia en cómo se relacionan los tres valores. Sí podemos esforzarnos ver una tendencia aparente con forma de flecha, cuando $w_1 > w_2 < w_0$ parece que tiende a terminar en pocas iteraciones, y sin embargo, la flecha hacia arriba $w1 < w_2 > w_0$ produce ejecuciones de más iteraciones.

Sin embargo, observando los valores finales[2], si podemos sacar alguna conclusión clara, por algún razón —quizás la antes expuesta—, algunos valores no convergen con pesos próximos a 0 y tienen que ascender o descender mucho más para hallar un punto de convergencia, por lo cual, dada la velocidad de crecimiento limitada por xy, si el peso destino es mucho mayor que éstos, harán falta más iteraciones para llegar.

2.1.b Ejecutar el algoritmo PLA con los datos del ejercicio 1.2.b

Vamos a repetir el experimento anterior para datos que no son linealmente separables, podemos esperar ver, al menos, cómo agota el número de iteraciones al no poder converger.

Inicializado los pesos en 0

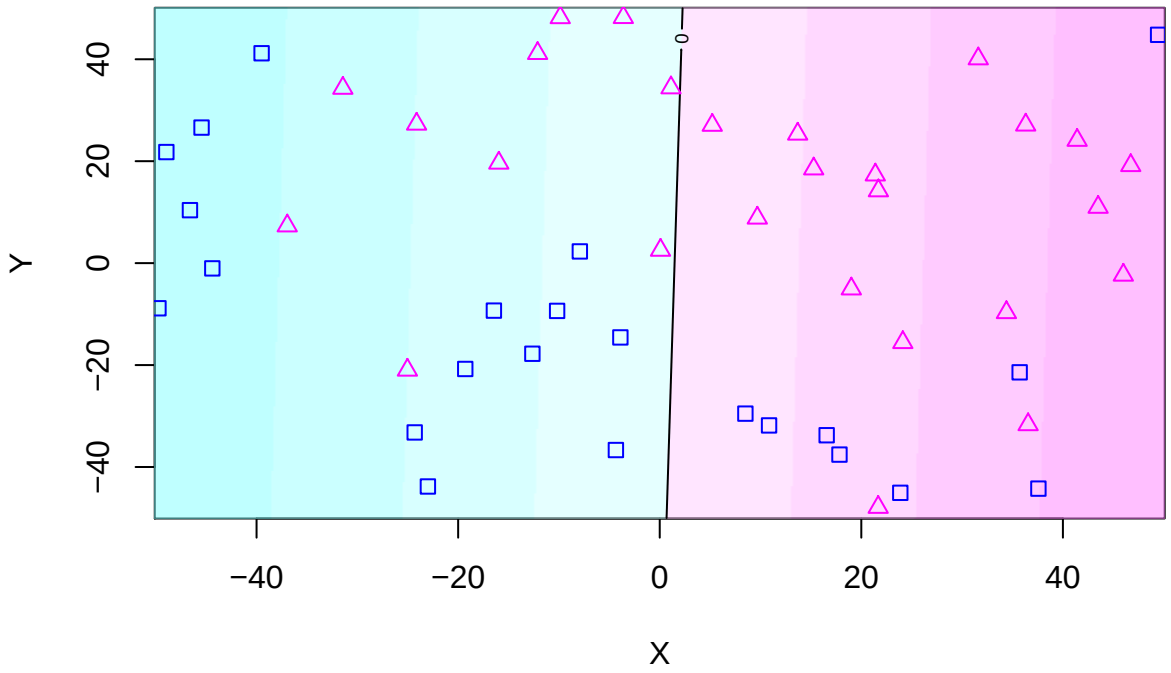
```
datos = nubea
z = zrand
w = matrix(rep_len(0,ncol(nubea)+1),nrow = 1)
res = ajusta_PLA(datos,z,100,w,calc_ein = TRUE)
w = res$w
iters = res$iters
ein = res$ein
f = function(x,y){x*w[1]+y*w[2]+w[3]}

#Imprimir w
print(paste("Iters:",iters))
```

```
## [1] "Iters: 100"
```

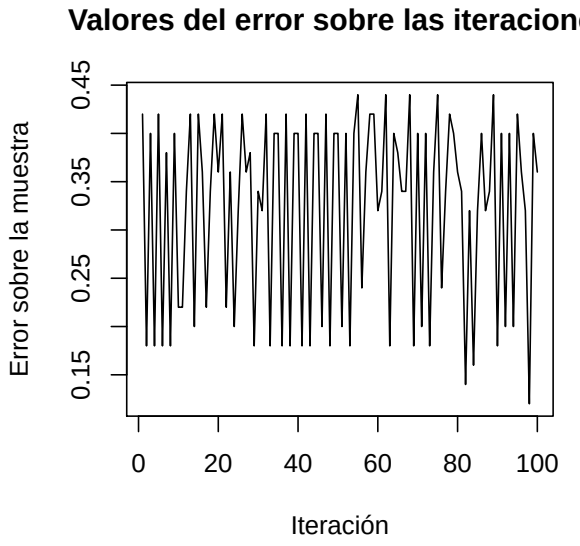
```
cat("w:",w,"\n")
```

```
## w: 46.83199 -0.7434567 -69
```

```
pintar_frontera(f)
points(nubea,col=z+5, pch=z+1) #+3 para azul y rojo
```



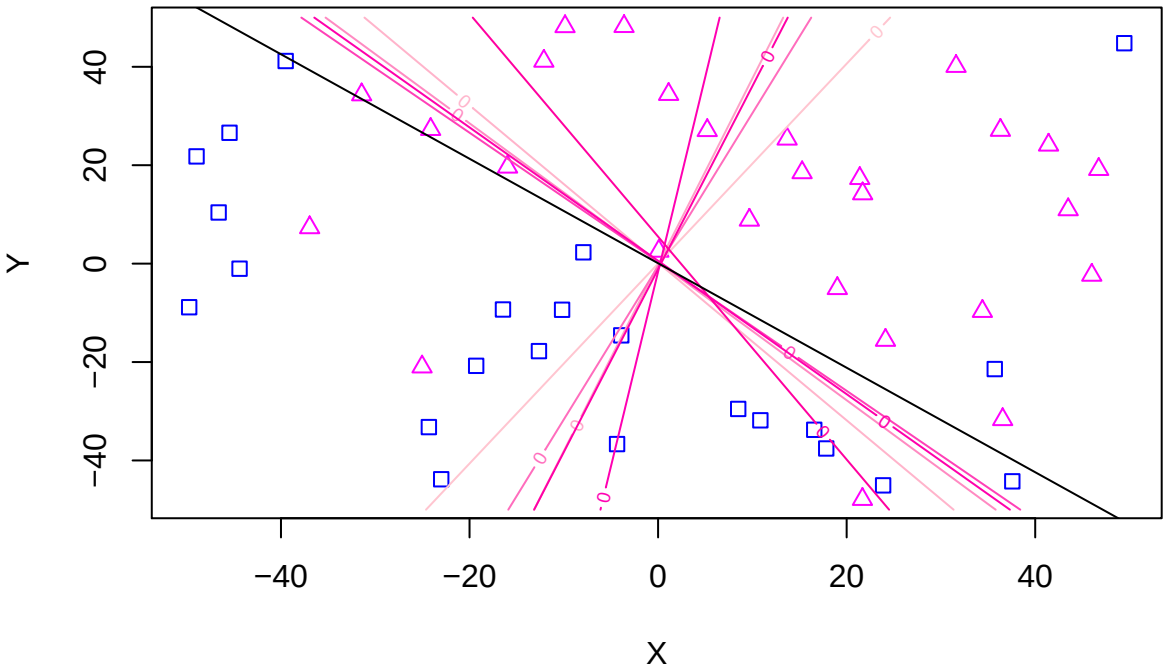

```
plot(y=ein, x=1:iters,type = 'l', xlab = "Iteración", ylab="Error sobre la muestra", main="Valores del e
hist(ein, main="Histograma del error", xlab = "Error")
```



Lo primero que apreciamos es cómo efectivamente agota todas las iteraciones, además que si bien la clasificación parece bastante pobre el E_{in} obedece como a una especie de periodo y no es capaz de acercarse a ningún punto de equilibrio donde se estabilice. Habiendo errores, tanto sobre 0.42 así como cercanos a 0.10, el mínimo error posible. Esto nos dice mucho sobre cómo funciona PLA, en especial dos cosas, la primera es que aumentar el número de iteraciones permitidas no garantiza una mejor solución, como se puede observar en la gráfica; y segundo, cuando parte de una w con poca capacidad expresiva la mayoría de los casos que corrigen la w están bien clasificados —0.9 frente a 0.1 en nuestro caso—, dando como resultado una actualización de la w mejor que la anterior, sin embargo, cuando la w está muy ajustada a la recta original con la que clasificamos los valores, los únicos casos que actualizan la w son aquellos mal clasificados, produciendo un empeoramiento dramático de la capacidad de clasificación correcta.

Veamos cómo se modifica la frontera que dibuja la w a lo largo de las iteraciones, los colores más pálidos son iteraciones más bajas y los más vivos iteraciones más altas, únicamente ejecutado para 10 iteraciones, con la recta objetivo como referencia.

```
w = matrix(rep_len(0,ncol(nubea)+1),nrow = 1)
plot(nubea,col=z+5, pch=z+1)
for(i in 1:10){
  res = ajusta_PLA(datos,z,1,w,calc_ein = TRUE)
  w = res$w
  iters = res$iters
  ein = res$ein
  f = function(x,y){x*w[1]+y*w[2]+w[3]}
  pintar_frontera(f,color = FALSE, add=TRUE, line_col = hcl(c=i/10*360))
}
abline(a=rc[2],b=rc[1])
```



Efectivamente, como esperábamos va alternando fronteras próximas a la frontera óptima con fronteras alejadas de esta. Veamos sin embargo, cómo se comporta con w inicializadas aleatoriamente.

Inicializdos los pesos de forma aleatoria

```
MAX_ITER = 100
datos = nubea
z = zrand
w_ini = lapply(1:10,function(i){runif(3,0,1)})
res = lapply(1:10,function(i){ajusta_PLA(datos,z,MAX_ITER,matrix(w_ini[[i]],nrow=1))})

ws = lapply(res,function(e){e$w})
ws = matrix(unlist(ws),ncol = 3,byrow = TRUE)

niters = as.integer(lapply(res,function(e){e$iters}))

#Comprobamos si alguno se ha salido del número de iteraciones
```

```r
paste("Número de ajustes que agotaron el límite: ",length(which(niters == MAX_ITER)))
```

```
## [1] "Número de ajustes que agotaron el límite:  10"
```

```r
#Resumen
print("Summary:")
```

```
## [1] "Summary:"
```

```r
summary(niters)
```

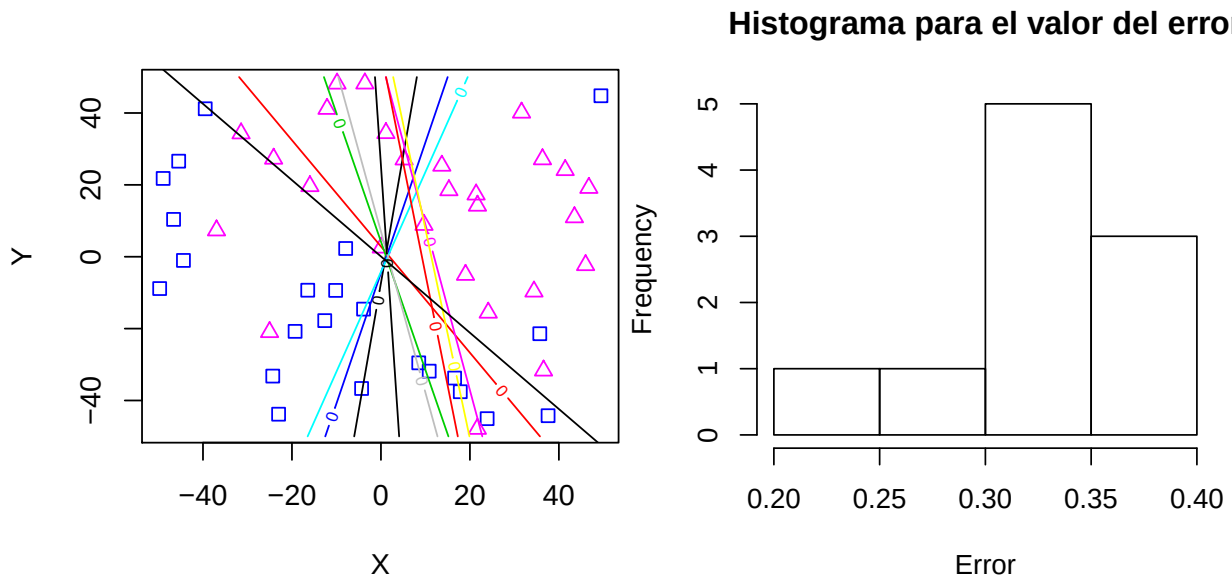
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     100     100     100     100     100     100
```

Efectivamente, todos los valores agotan el número de iteraciones disponibles, tal y como esperábamos, veamos qué tal los resultados.

```r
plot(nubea,col=z+5, pch=z+1)
eins = c()
for(i in 1:nrow(ws)){
  w = ws[i,]
  f = function(x,y){x*w[1]+y*w[2]+w[3]}
  eins = c(eins,length(which(z != sign(f(nubea[,1],nubea[,2]))))/nrow(nubea))
  pintar_frontera(f,color = FALSE, add=TRUE, line_col = i)
}
abline(a=rc[2],b=rc[1])

summary(eins)
```

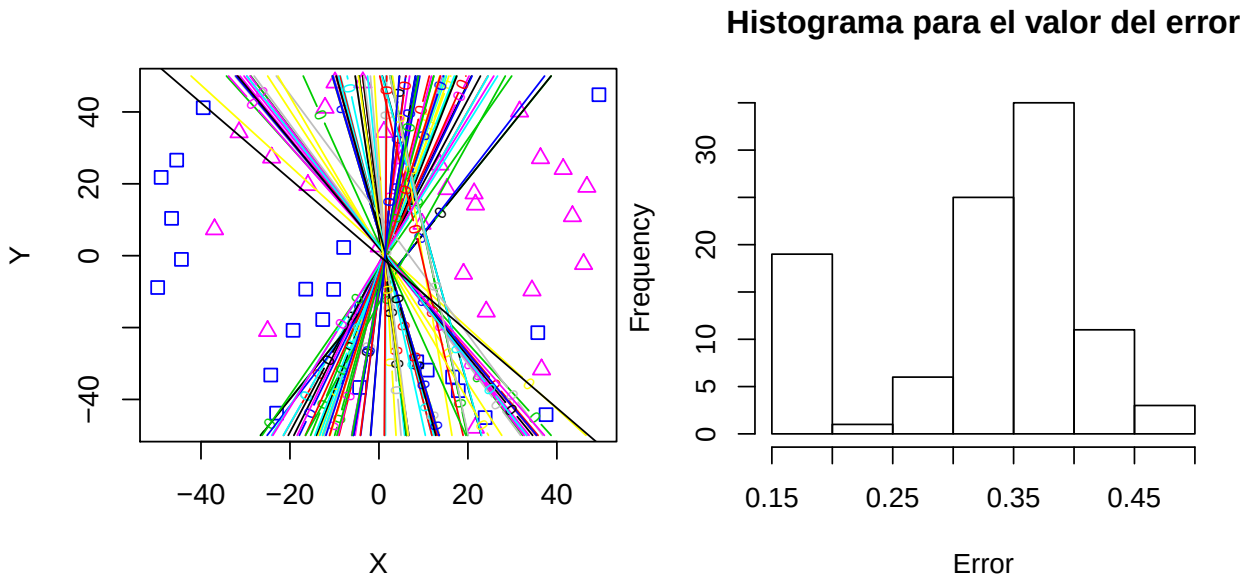
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.200   0.320   0.340   0.330   0.355   0.400
```

```r
hist(eins, main="Histograma para el valor del error",xlab = "Error")
```



Probemos ahora para 100 w iniciales.

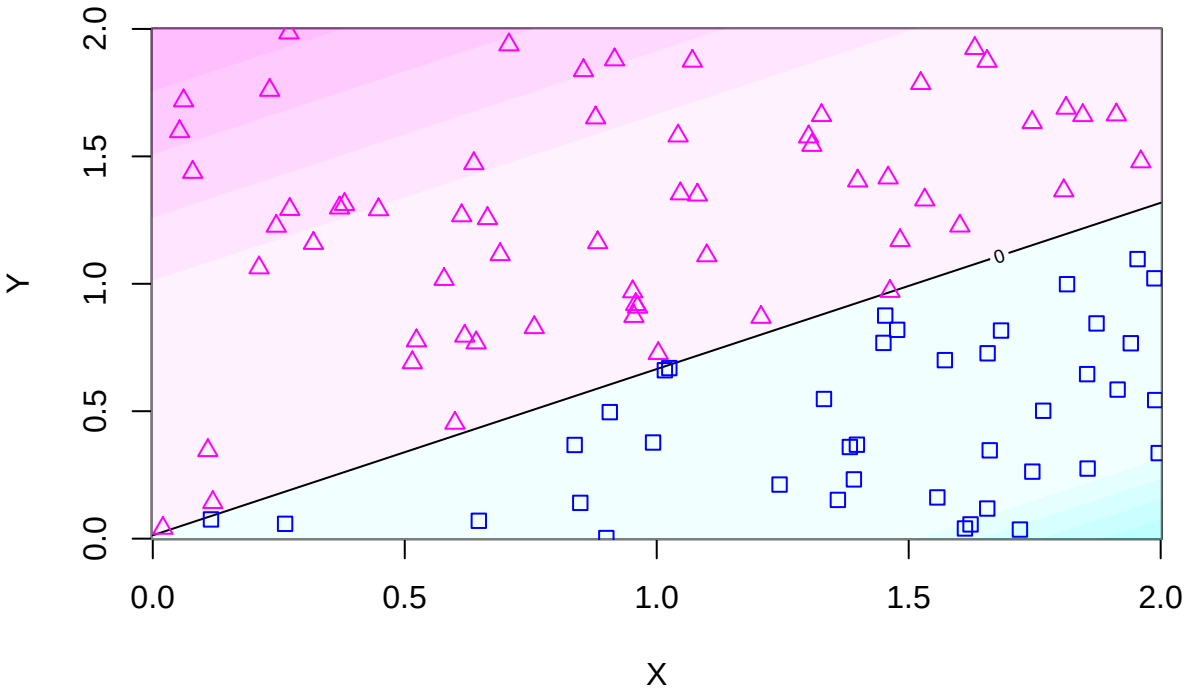
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1600  0.2950  0.3400  0.3334  0.4000  0.4800
```



Tal y como esperabamos acaba generando valores aproximadamente entre los dos límites que veíamos para el apartado anterior con w inicializada con todos los pesos a 0, y de la misma forma que ahí donde se podía ver a simple vista que aproximadamente cada tres o cuatro iteraciones se solía dar 1 valor razonablemente bueno, aquí igual, con unos 20 valores por debajo de 0.2 y 80 por encima.

2.2. Regresión logística

```
datos = simula_unif(N=100,dims=2,c(0,2))
p1 = c(0,runif(1,max=2))
p2 = runif(2,max=2)
a = p1[2]
b = (p2[2]-a)/p2[1]
f = function(x,y){y - (x*b + a)}
f_real = f
z = apply(datos,1,function(r){sign(f(r[1],r[2]))})

pintar_frontera(f,rango=c(0,2))
points(datos, col=z+5, pch=z+1)
```



2.2.a. Implementar regresión logística con SGD

Como la condición de salida es igual al valor de η asumo que se nos pide SGD con tamaño del minibatch igual a 1, pues si no la velocidad de actualización de w con un minibatch mayor de 50 sería demasiado lenta y saldría del SGD en la primera iteración.

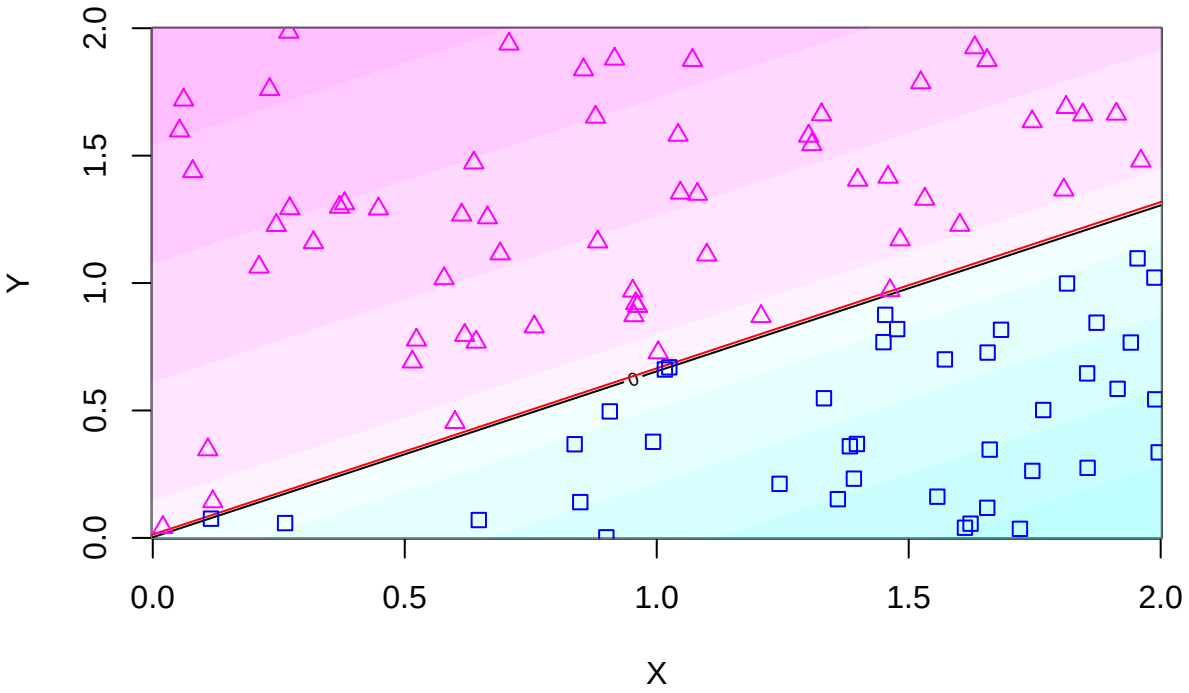
```
ein = function(datos,label,w){
  datos = cbind(datos,rep_len(1,nrow(datos)))
  l = lapply(1:length(label),function(i){
    log(1+exp(-label[i]*w%*%t(datos[i,,drop=FALSE])))
  })
  mean(unlist(l))
}
getSGDw = function(datos,label,w0,eta,max_iter){
  w = w0
  N = length(label)
  datos = cbind(datos,rep_len(1,nrow(datos)))
  for(i in 1:max_iter){
    wold = w
    s = sample(N) # Reordena los índices de la matriz de datos
    sX = datos[s,]
    sY = label[s]
    for (j in 1:nrow(sX)){
      ex = as.double(exp(-sY[j]*w%*%t(sX[j,,drop=FALSE])))
      gein = -sY[j]*sX[j,,drop=FALSE]*(ex/(1+ex))
      w = w - eta*gein
    }
    if(norm(wold-w)<0.01) break
  }
  w
}
```

2.2.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 (> 900)

```
w0 = c(0,0,0)
eta = 0.01
w = getSGDw(datos,z,w0,eta,100000)
cat("w:",unlist(w),"\n")
```

```
## w: -4.476012 6.873378 -0.01699554
```

```
f = function(x,y){x*w[1]+y*w[2]+w[3]}
#La f calculada en negro
pintar_frontera(f,rango=c(0,2))
#La f original en rojo
pintar_frontera(f_real,color=FALSE,add=TRUE,line_col = "red")
points(datos, col=z+5, pch=z+1)
```

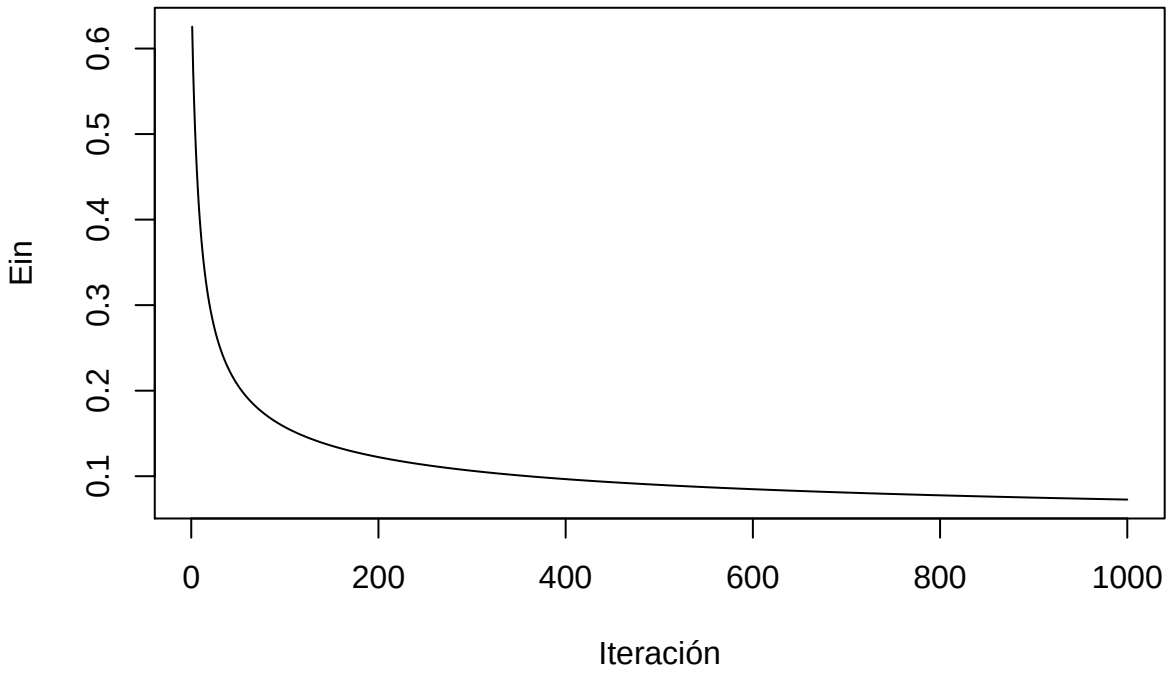



```
sample_ein = ein(datos,z,w)
print(paste("Ein:",sample_ein))
```

```
## [1] "Ein: 0.11460451518555"
```

Veamos cómo mejora con cada iteración

```
ITERS = 1000
w = c(0,0,0)
eta = 0.01
eins = rep_len(Inf,ITERS)
for(i in 1:ITERS){
  w = getSGDw(datos,z,w,eta,1)
  eins[i] = ein(datos,z,w)
}
plot(1:ITERS,eins,type = 'l',ylab = "Ein", xlab="Iteración")
```



Y ahora vamos a generar una nueva muestra de 10000 elementos para estimar E_{out}

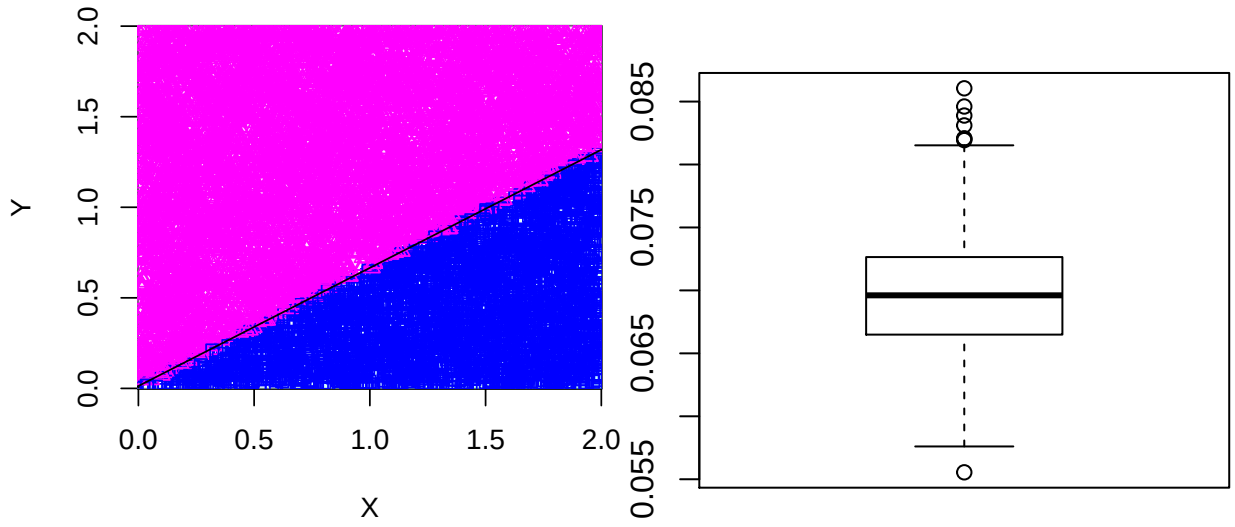
```
#Eout para una muestra de 10000
dout = simula_unif(N=10000,dims=2,c(0,2))
zout = apply(dout,1,function(r){sign(f(r[1],r[2]))})

#Pintamos los datos y la línea original
pintar_frontera(f,rango=c(0,2))
points(dout, col=zout+5, pch=zout+1)
pintar_frontera(f_real,color=FALSE,add=TRUE)

#Imprimimos Eout
print(paste("Eout:",ein(dout,zout,w)))
```

```
## [1] "Eout: 0.06807960186872"
```

```
#Estudio del valor de Eout
eouts = c()
#Durante 1000 iteraciones
for(t in 1:1000){
  dout = simula_unif(N=1000,dims=2,c(0,2))
  zout = apply(dout,1,function(r){sign(f(r[1],r[2]))})
  eouts = c(eouts,ein(dout,zout,w))
}
summary(eouts)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.05555 0.06649 0.06961 0.06964 0.07264 0.08605
```

```
boxplot(eouts)
```



Como se puede observar, el error fuera de la muestra ronda el 0.7, menor incluso que el error interno, aunque esto se puede deber a que por azar los puntos de entrenamiento hayan salido más concentrados próximos al horizonte, sin embargo ya sabemos que el modelo lineal tiene una extraordinaria generalización del error y muy buena convergencia, más aún para datos linealmente separables.

2.2.c. Etiquetas con ruido

Sin embargo, y a modo de experimento, vamos a ver cómo el verdadero potencial de la regresión logística sobre SGD —como pudimos intuir en la práctica anterior— es cuando los datos no son linealmente separables.

```
z_ruido = z
s = sample(length(z),length(z)%/%10)
z_ruido[s] = z_ruido[s]*-1

w0 = c(0,0,0)
eta = 0.01
w = getSGDw(datos,z_ruido,w0,eta,100000)
cat("w:",unlist(w),"\n")
```

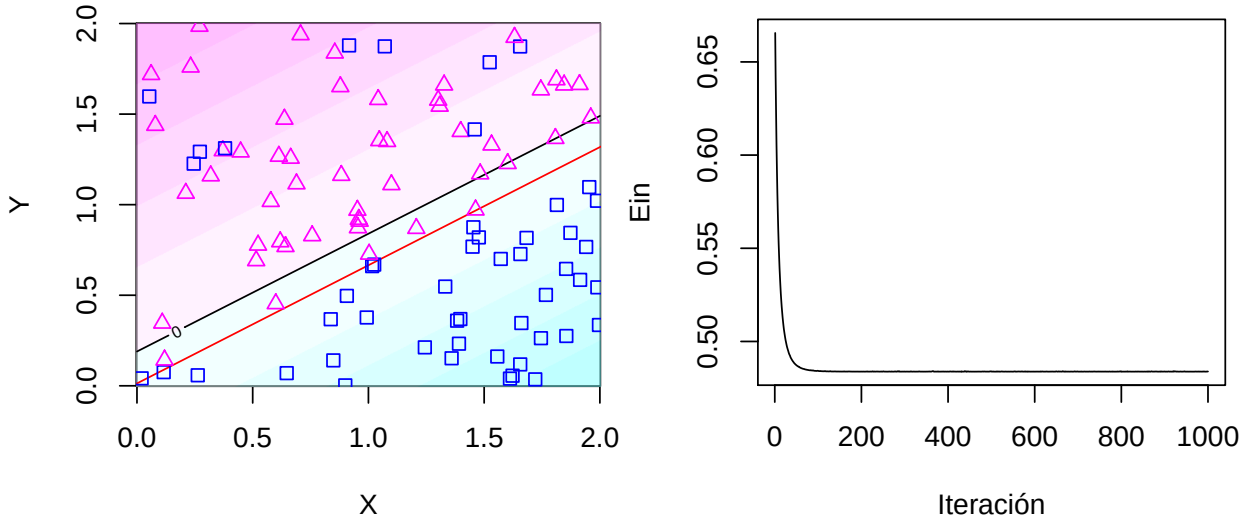
```
## w: -1.39219 2.140978 -0.4053715
```

```
f = function(x,y){x*w[1]+y*w[2]+w[3]}
#La f calculada en negro
pintar_frontera(f,rango=c(0,2))
#La f original en rojo
pintar_frontera(f_real,color=FALSE,add=TRUE,line_col = "red")
points(datos, col=z_ruido+5, pch=z_ruido+1)

print(paste("Ein:",ein(datos,z_ruido,w)))
```

```
## [1] "Ein: 0.486559456823166"
```

```
ITERS = 1000
w = c(0,0,0)
eta = 0.01
eins = rep_len(Inf,ITERS)
for(i in 1:ITERS){
  w = getSGDw(datos,z_ruido,w,eta,1)
  eins[i] = ein(datos,z_ruido,w)
```

```
}
plot(1:ITERS,eins,type = 'l',ylab = "Ein", xlab="Iteración")
```



Vemos cómo al contrario que el Perceptrón, que por diseño oscilaba sobre los valores que minimizaban el error —esto lo solucionaremos en el bonus con el *Pocket*—, el SGD se acaba estabilizando en un punto que minimiza el error sin embargo, debido al ruido, en este caso, eleva el horizonte pues por azar todos los valores con la etiqueta cambiada se encuentran en la zona positiva, y para intentar mejorar el acierto sobre la probabilidad de catalogar correctamente esos puntos sacrifica los más próximos a la frontera de decisión que sí estaban correctamente clasificados. Este es el principal riesgo de las muestras pequeñas con ruido, y podríamos garantizar convergencias de E_{out} y E_{in} con mayor seguridad si aumentáramos el tamaño de la muestra de entrenamiento, donde el error, uniformemente distribuido atenuaría el riesgo de sobreajuste.

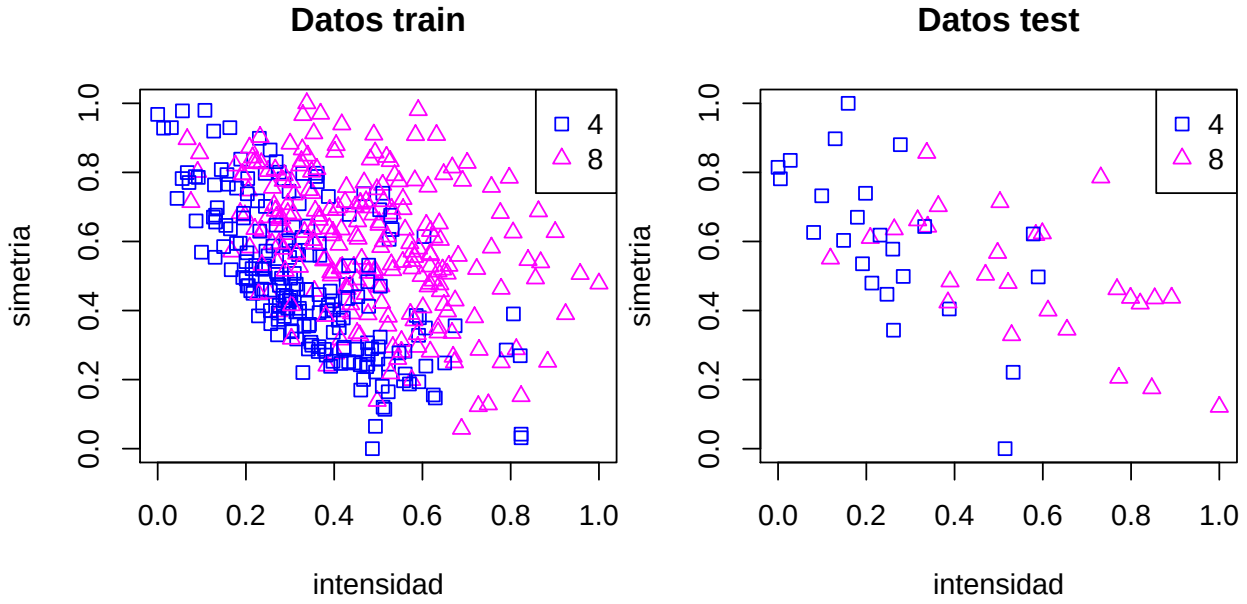
3. Clasificación de dígitos

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

Nuestra clasificación binaria se realizará sobre los dígitos 4 y 8 correspondiéndole al 4 la etiqueta -1 y al 8 la etiqueta +1. Una vez obtenida la intensidad promedio y la simetría normalizamos los valores para ahorrarle al clasificador el esfuerzo de compensar la desigualdad en las horquillas de valores de las etiquetas.

```
###Datos train
dat = readData15("datos/zip.train")
grises = dat$grises
digitos = dat$digitos

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

datosTr = normalize(as.matrix(cbind(intensidad,simetria)))

etiquetasTr = digitos
etiquetasTr[etiquetasTr==4]=-1
etiquetasTr[etiquetasTr==8]=1

plot(datosTr,col=etiquetasTr+5, pch=etiquetasTr+1, main="Datos train")
```

```
legend("topright",legend = c("4","8"),pch=c(0,2), col=c(4,6))

###Datos test
dat = readData15("datos/zip.test")
grises = dat$grises
digitos = dat$digitos

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

datosTst = normalize(as.matrix(cbind(intensidad,simetria)))

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

plot(datosTst,col=etiquetasTst+5, pch=etiquetasTst+1, main="Datos test")
legend("topright",legend = c("4","8"),pch=c(0,2), col=c(4,6))
```



Vemos cómo los datos no son en absoluto linealmente separables y podemos esperar un E_{in} alto, quizás estas características son insuficientes para clasificar correctamente el problema, aún así sigamos con el experimento

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

```
getPIw = function(X,Y){
  N = length(Y)
  X = cbind(X,rep_len(1,N))
  Xt = t(X)
  Xpi=solve(Xt%*%X)%*%Xt
  w=Xpi%*%Y
  as.double(w)
}
```

```r
getSGDw = function(X,Y,w0,eta,max_iter){
  w = w0
  N = length(Y)
  X = cbind(X,rep_len(1,N))

  for(i in 1:max_iter){
    wold = w
    s = sample(N,size=N/4)
    sX = X[s,]
    sY = Y[s]
    w = w - eta*colMeans((-sY*sX)/as.vector(1+exp(sY*sX%*%w)))
    if(norm(as.matrix(wold-w))<0.0001) break
  }
  w
}

getFW = function(w){
  function(x,y){x*w[1]+y*w[2]+w[3]}
}

getE = function(X,w,Y,indX=cbind(X,rep_len(1,length(Y)))){
  sum((sign(indX %*% w)-Y)!=0)/length(Y)
}

getPLAPocketw = function(datos,label,max_iter,vini){
  if(ncol(datos)!=(length(vini)-1)) stop("tamaño incorrecto")
  datos = cbind(datos,ind=1)
  w = vini
  bestw = w
  bestein = getE(w=w,Y=label,indX=datos)
  wt = w-1
  iter = 0
  while(!all(w == wt) && iter < max_iter){
    #print(max_iter)
    wt = w
    iter = iter+1
    for(i in 1:nrow(datos)){
      res = sign(w%*%t(datos[i,,drop=FALSE]))
      if(res != label[i]){
        w = w + (label[i]*datos[i,])
      }
      nein = getE(w=w,Y=label,indX=datos)
      if(nein < bestein){
        bestw = w
        bestein = nein
      }
    }
  }
  list(w=bestw,iters=iter)
}
```

3.2.(a,b) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada y calcular E_{in} y E_{test} (error sobre los datos del test)

Vamos a ver cómo se comportan los datos con SGD y pseudoinversa sin mejora para posteriormente comprobar cómo mejora la solución el PLA-Pocket

```
w1 = getSGDw(datosTr,etiquetasTr,c(0,0,0),0.01,100000)
w2 = getPIw(datosTr,etiquetasTr)

einSGD = getE(datosTr,w1,etiquetasTr)
einPI = getE(datosTr,w2,etiquetasTr)
etestSGD = getE(datosTst,w1,etiquetasTst)
etestPI = getE(datosTst,w2,etiquetasTst)

print(paste("Ein SGD:",einSGD))
```

```
## [1] "Ein SGD: 0.224537037037037"
```

```
print(paste("Ein PI:",einPI))
```

```
## [1] "Ein PI: 0.247685185185185"
```

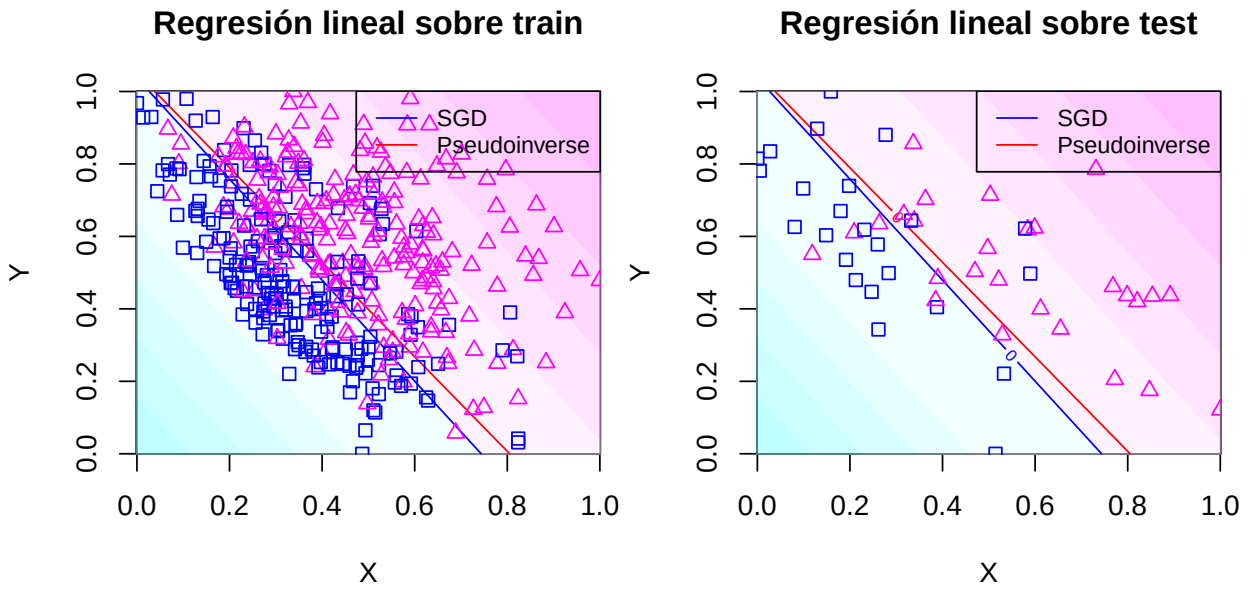
```
print(paste("Etest SGD:",etestSGD))
```

```
## [1] "Etest SGD: 0.215686274509804"
```

```
print(paste("Etest PI:",etestPI))
```

```
## [1] "Etest PI: 0.235294117647059"
```

```
f1 = getFW(w1)
f2 = getFW(w2)

#Train
pintar_frontera(f1,rango=c(0,1),line_col = "blue", main="Regresión lineal sobre train")
pintar_frontera(f2,rango=c(0,1),add=TRUE,col=FALSE,line_col = "red")
points(datosTr, col=etiquetasTr+5, pch=etiquetasTr+1)
legend("topright", legend=c("SGD", "Pseudoinverse"),
       col=c("blue", "red"), lty=c(1,1), cex=0.8)

#Test
pintar_frontera(f1,rango=c(0,1),line_col = "blue", main="Regresión lineal sobre test")
pintar_frontera(f2,rango=c(0,1),add=TRUE,col=FALSE,line_col = "red")
points(datosTst, col=etiquetasTst+5, pch=etiquetasTst+1)
legend("topright", legend=c("SGD", "Pseudoinverse"),
       col=c("blue", "red"), lty=c(1,1), cex=0.8)
```




```
w1 = getPLAPocketw(datosTr,etiquetasTr,1000,w1)
w1 = w1$w
w2 = getPLAPocketw(datosTr,etiquetasTr,1000,w2)
w2 = w2$w

einSGDPocket = getE(datosTr,w1,etiquetasTr)
einPIPocket = getE(datosTr,w2,etiquetasTr)
etestSGDPocket = getE(datosTst,w1,etiquetasTst)
etestPIPocket = getE(datosTst,w2,etiquetasTst)

print(paste("Ein SGD+Pocket:",einSGDPocket))
```

```
## [1] "Ein SGD+Pocket: 0.222222222222222"
```

```
print(paste("Ein PI+Pocket:",einPIPocket))
```

```
## [1] "Ein PI+Pocket: 0.226851851851852"
```

```
print(paste("Etest SGD+Pocket:",etestSGDPocket))
```

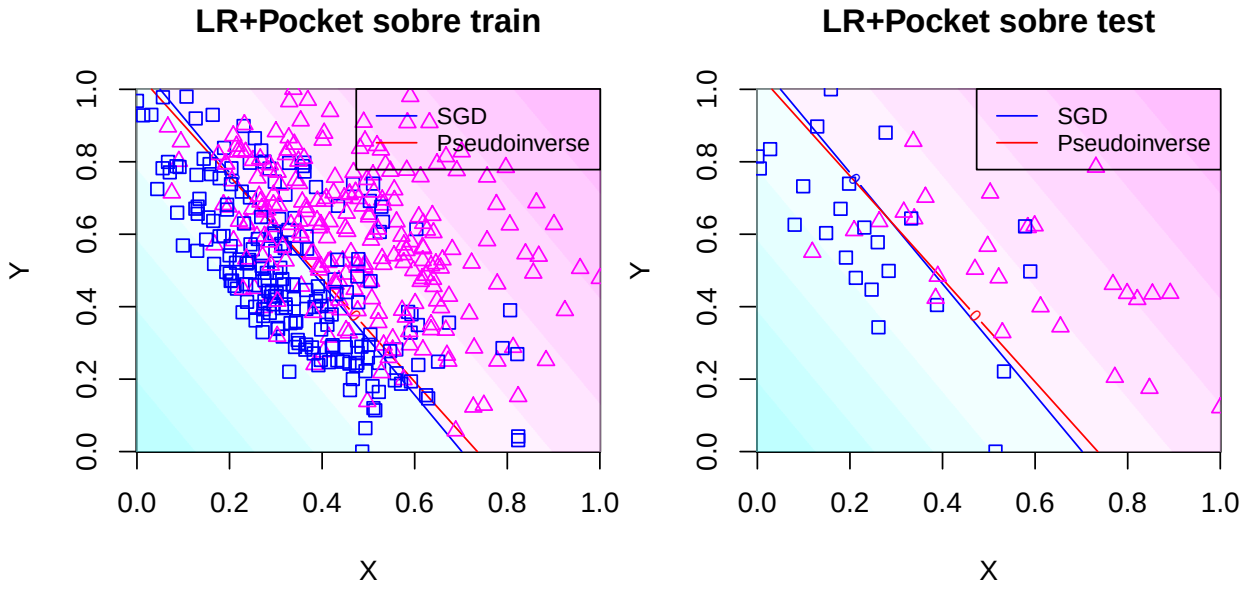
```
## [1] "Etest SGD+Pocket: 0.196078431372549"
```

```
print(paste("Etest PI+Pocket:",etestPIPocket))
```

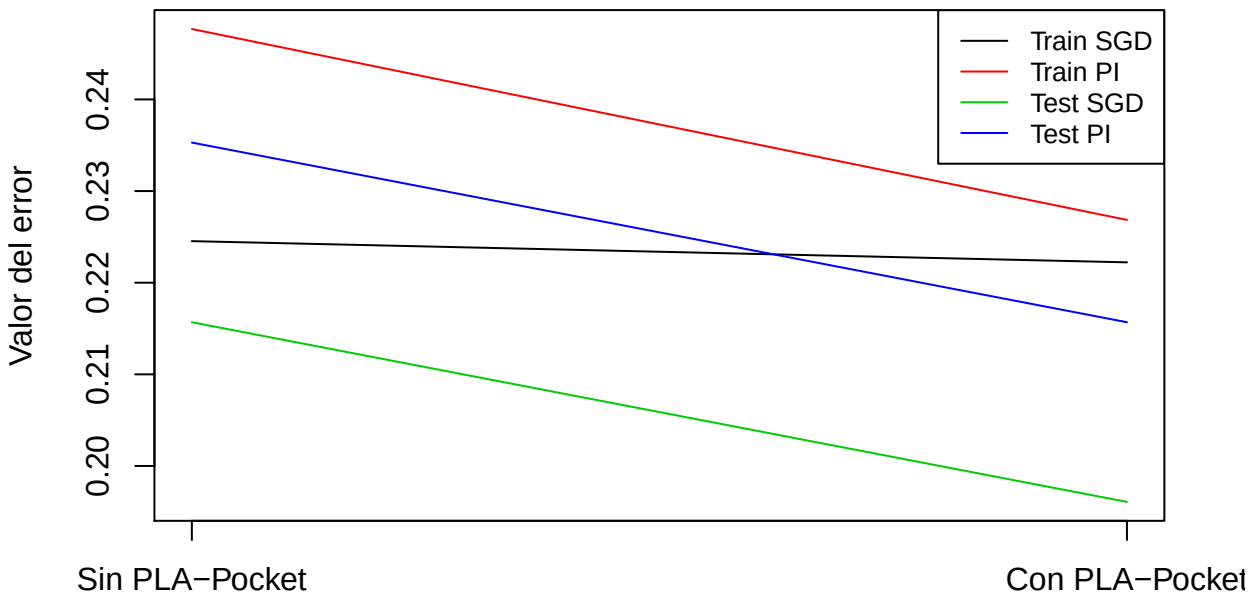
```
## [1] "Etest PI+Pocket: 0.215686274509804"
```

```
f1 = getFW(w1)
f2 = getFW(w2)

#Train
pintar_frontera(f1,rango=c(0,1),line_col = "blue", main="LR+Pocket sobre train")
pintar_frontera(f2,rango=c(0,1),add=TRUE,col=FALSE,line_col = "red")
points(datosTr, col=etiquetasTr+5, pch=etiquetasTr+1)
legend("topright", legend=c("SGD", "Pseudoinverse"),
       col=c("blue", "red"), lty=c(1,1), cex=0.8)

#Test
pintar_frontera(f1,rango=c(0,1),line_col = "blue", main="LR+Pocket sobre test")
```

```
pintar_frontera(f2,rango=c(0,1),add=TRUE,col=FALSE,line_col = "red")
points(datosTst, col=etiquetasTst+5, pch=etiquetasTst+1)
legend("topright", legend=c("SGD", "Pseudoinverse"),
       col=c("blue", "red"), lty=c(1,1), cex=0.8)
```




```
#Mejora
min_val = min(einSGD,einPI,etestSGD,etestPI,einSGDPocket,einPIPocket,etestSGDPocket,etestPIPocket)
max_val = max(einSGD,einPI,etestSGD,etestPI,einSGDPocket,einPIPocket,etestSGDPocket,etestPIPocket)

plot(1:2,c(einSGD,einSGDPocket),ylim = c(min_val,max_val), xlim=c(1,2),type="l",col=1,
     main="Evolución del error", xaxt='n', ylab = "Valor del error", xlab="")
lines(1:2,c(einPI,einPIPocket),ylim = c(min_val,max_val), xlim=c(1,2),type="l",col=2)
lines(1:2,c(etestSGD,etestSGDPocket),ylim = c(min_val,max_val), xlim=c(1,2),type="l",col=3)
lines(1:2,c(etestPI,etestPIPocket),ylim = c(min_val,max_val), xlim=c(1,2),type="l",col=4)
legend("topright",legend=c("Train SGD", "Train PI", "Test SGD", "Test PI"),
       col=1:4, lty=rep_len(1,4), cex=0.8)
axis(1,at=c(1,2),las=2,labels = c("Sin PLA-Pocket","Con PLA-Pocket"), las=1)
```



Parece que SGD, al menos en esta ocasión, consigue mejores resultados que la pseudoinversa, también era de esperar teniendo en cuenta que computacionalmente es la opción más costosa. Asimismo, podemos observar cómo el PLA-Pocket es capaz casi siempre de darle un empujoncito a nuestro resultado consiguiendo un error menor, incluso en el SGD que no mejora significativamente con el train pero sí supone una mejora sobre el test.

3.2.c Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas, una basada en E_{in} y otra basada en E_{test}. Usar una tolerania $\delta = 0.05$

Vamos a utilizar el resultado del SGD+Pocket por no hacer demsiado complicada la salida, aunque podríamos haber usado PI+Pocket. Asumimos que la clase $|H|$ es infinita, entonces

$$\delta = 2 * e^{-2\epsilon^2 N}$$
$$\epsilon = \sqrt{\frac{1}{2N} ln \frac{2}{\delta}}$$

```
getEps = function(delta,N){
  sqrt(log(2/delta)/(2*N))
}

trE = getEps(0.05,length(etiquetasTr))
tstE = getEps(0.05,length(etiquetasTst))

#Para el train
print("Train")
```

```
## [1] "Train"
```

```
paste("    Cota inferior:",einSGDPocket-trE)
```

```
## [1] "    Cota inferior: 0.156880532580582"
```

```r
paste("    Cota superior:",einSGDPocket+trE)
```

```
## [1] "    Cota superior: 0.287563911863863"
```

```r
#Para el test
print("Test")
```

```
## [1] "Test"
```

```r
paste("    Cota inferior:",etestSGDPocket-tstE)
```

```
## [1] "    Cota inferior: 0.0059061809145648"
```

```r
paste("    Cota superior:",etestSGDPocket+tstE)
```

```
## [1] "    Cota superior: 0.386250681830533"
```

Es extraño preguntarse qué cota es mejor o peor, lo que sí podemos afirmar y que ya conocíamos de la teoría es que el tamaño de la cota es inversamente proporcional al tamaño de la muestra, así para el train, donde había muchos más datos tenemos una cota de unos 0.13 de diferencia entre los dos puntos, mientras que para el test, con muchísimos menos datos, la horquilla llega hasta casi los 0.4. Sí podemos decir que para realizar una estimación de E_{out} nos interesa una muestra cuanto mayor mejor, pues reflejará con más fidelidad la verdadera distrubición de los datos de los que extraímos la muestra y nos dará una horquilla menor con la misma confianza como aquí sucede en el caso del train.