Practica2

Adrián Jesús Peña Rodríguez

April 20, 2018

Ejercicio 1:

En este ejercicio se nos pide experimentar con datos generados por nosotros mismos, esto lo haremos con la función simula_unif y simula_gaus. Para la división de nuestro dataset generado usaremos la función simula_recta.

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

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

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

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

simula_recta = 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
}
```

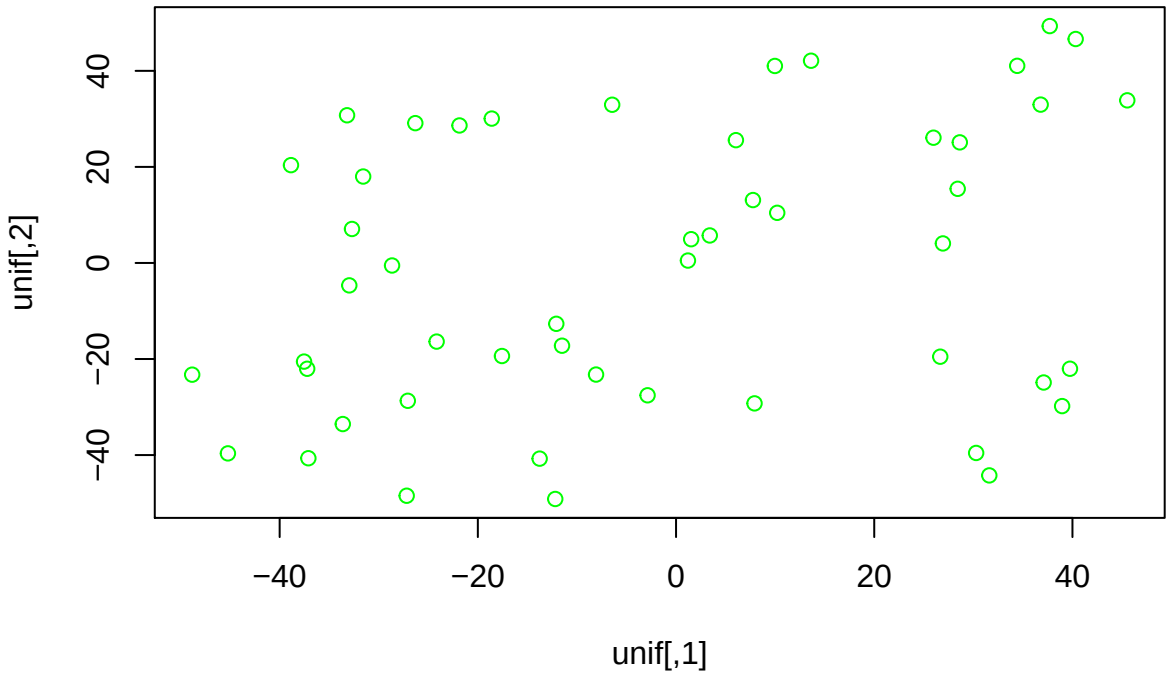
Apartado 1.1

a)

Nos piden dibujar unos datos que debemos generar con en este caso simula_unif, es decir, generaremos un dataset con valores aleatorios en una distribución uniforme.

```
# Ejercicio 1
# a)
unif = simula_unif(N=50,dim=2,rango=c(-50,50))
```

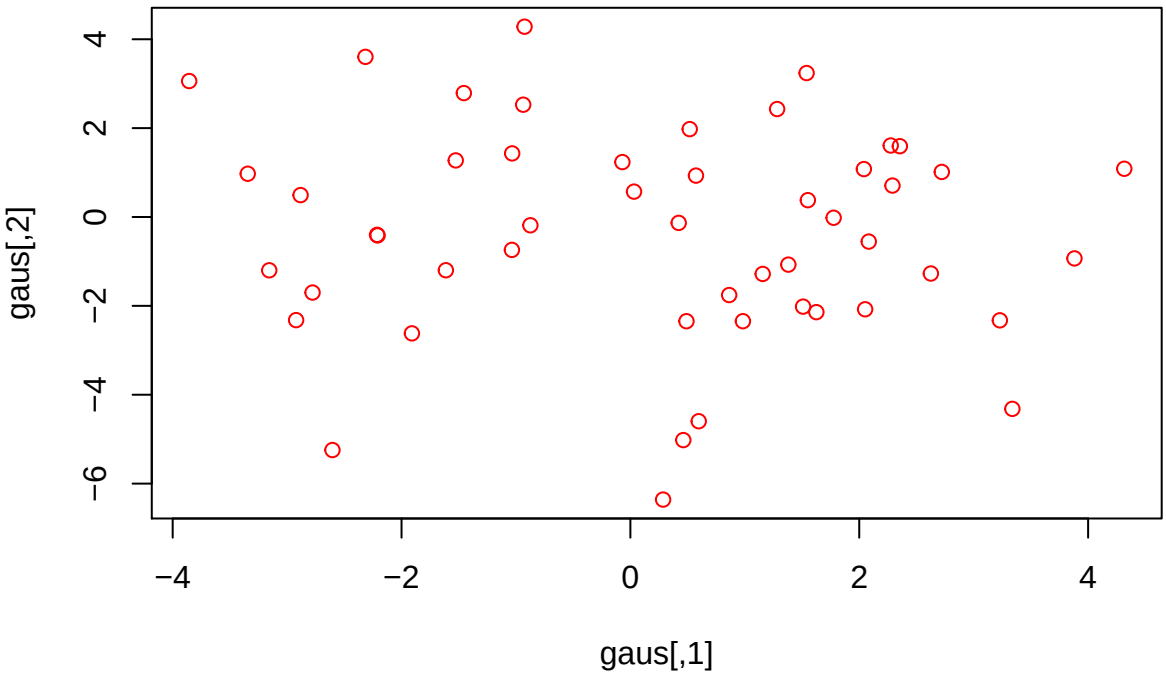
```r
unif = cbind(unif, 1)
plot(unif, col="green")
```



b)

Ahora se nos pide que generemos un dataset con la función simula_gaus, es decir, una distribución normal o de Gauss, con los valores de sigma = [5,7]

```r
# b)
gaus = simula_gaus(N=50,dim=2,sigma=c(5,7))
plot(gaus,col="red")
```



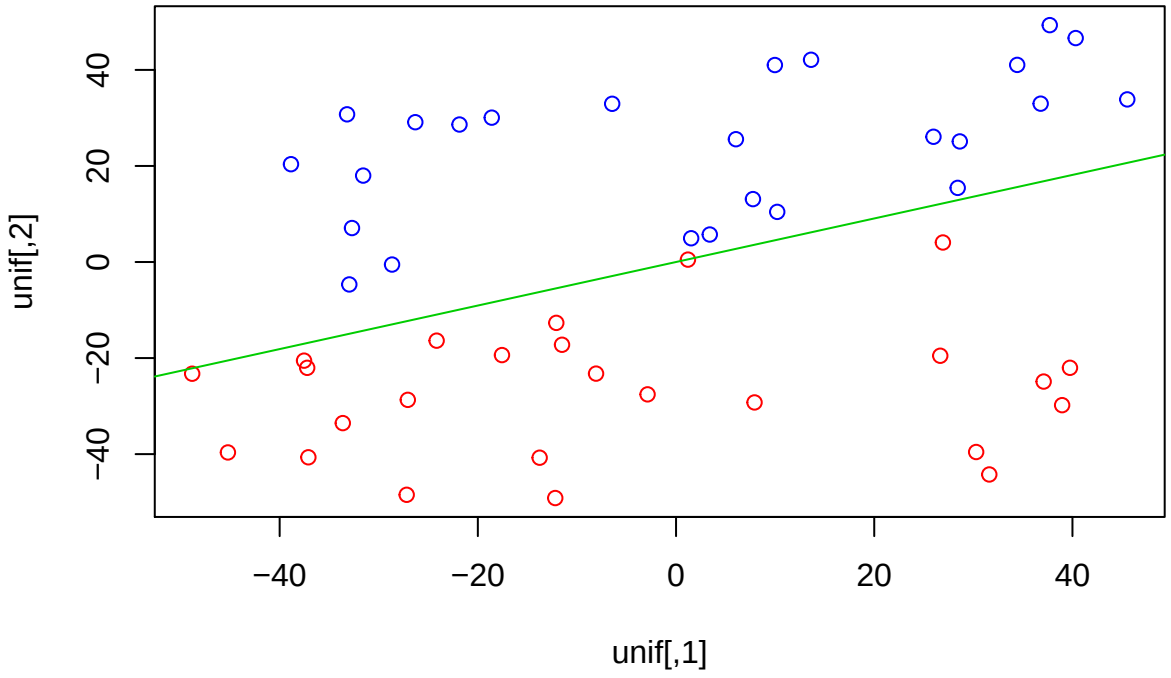
Apartado 1.2

a)

A continuación, debemos generar una muestra 2D de puntos con la función simula_unif y posteriormente vamos a añadirle etiquetas en función de f(x,y) = y - a*x - b. Tenemos que dentro de esta función encontramos un a y un b, a es la pendiente de nuestra recta y b el punto de corte, por lo que tenemos que generar una recta con nuestra función simula_recta y usar la a y b que esta nos devuelve para generar las etiquetas en función de dicha recta. Una vez tenemos esto lo sacamos por pantalla:

```
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
}

f = function(x,y,a,b){
  if (y - a * x - b >= 0)
    1
  else
    -1
}
```

```r
# Ejercicio 2
signos = c()

recta = simula_recta(visible=F)

for(i in 1:dim(unif)[1]){
  signos[i] = f(unif[i,1], unif[i,2], recta[1], recta[2])
}

# a)
plot(unif, col = signos + 3)
abline(recta[2],recta[1],col=3)
```

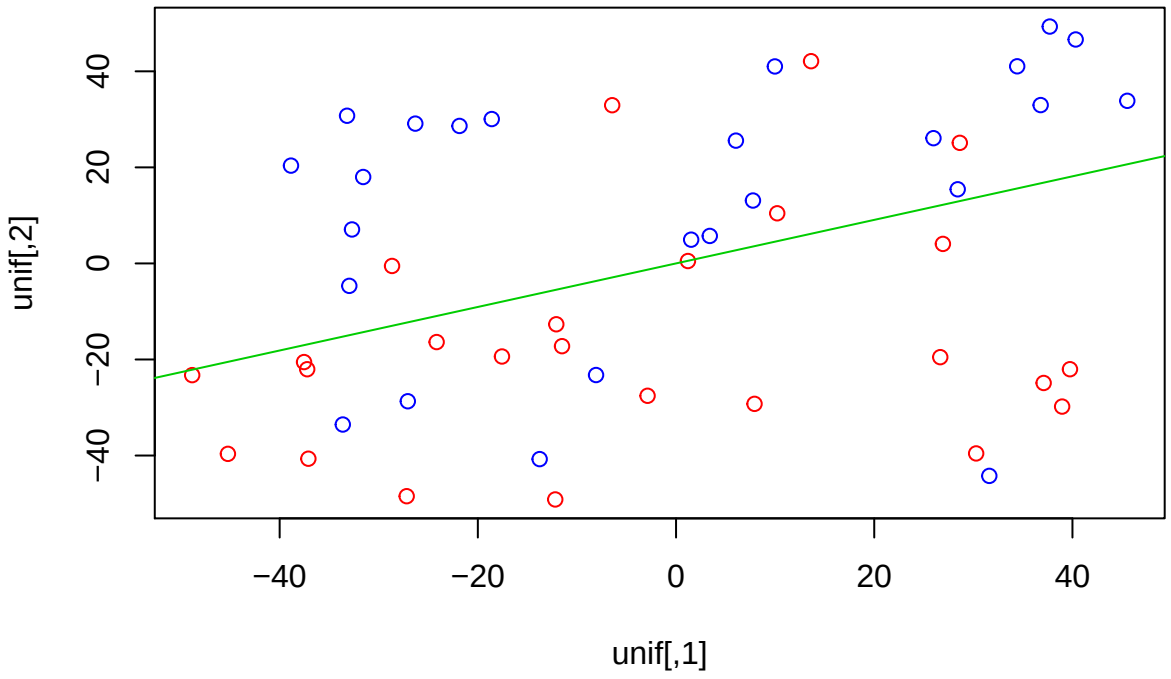


b)

Ahora debemos de introducir ruido, para ello debemos de cambiar aleatoriamente el 10% de la etiquetas positivas y otro 10% de las negativas, esta tarea la realizamos con la siguiente función la cual lleva dos contadores para saber cuantas positivas y negativas a cambiado y no cambiar ninguna más:

```r
genera_etiquetas_y_ruido = function(muestra, porcentaje) {
  N = length(muestra)
  posiciones = sample(1:length(muestra),length(muestra),replace=F)
  positivas = 0
  negativas = 0
  tope = length(muestra) * porcentaje

  for(i in 1:N) {
    if(muestra[posiciones[i]] == 1 & positivas < tope) {
      muestra[posiciones[i]] = muestra[posiciones[i]] * -1
      positivas = positivas + 1
    }
    else if(muestra[posiciones[i]] == -1 & negativas < tope) {
```

```
      muestra[posiciones[i]] = muestra[posiciones[i]] * -1
      negativas = negativas + 1
    }
  }

  muestra
}

# b)
signos_ruido = genera_etiquetas_y_ruido(signos, 0.1)

plot(unif, col = signos_ruido + 3)
abline(recta[2],recta[1],col=3)
```

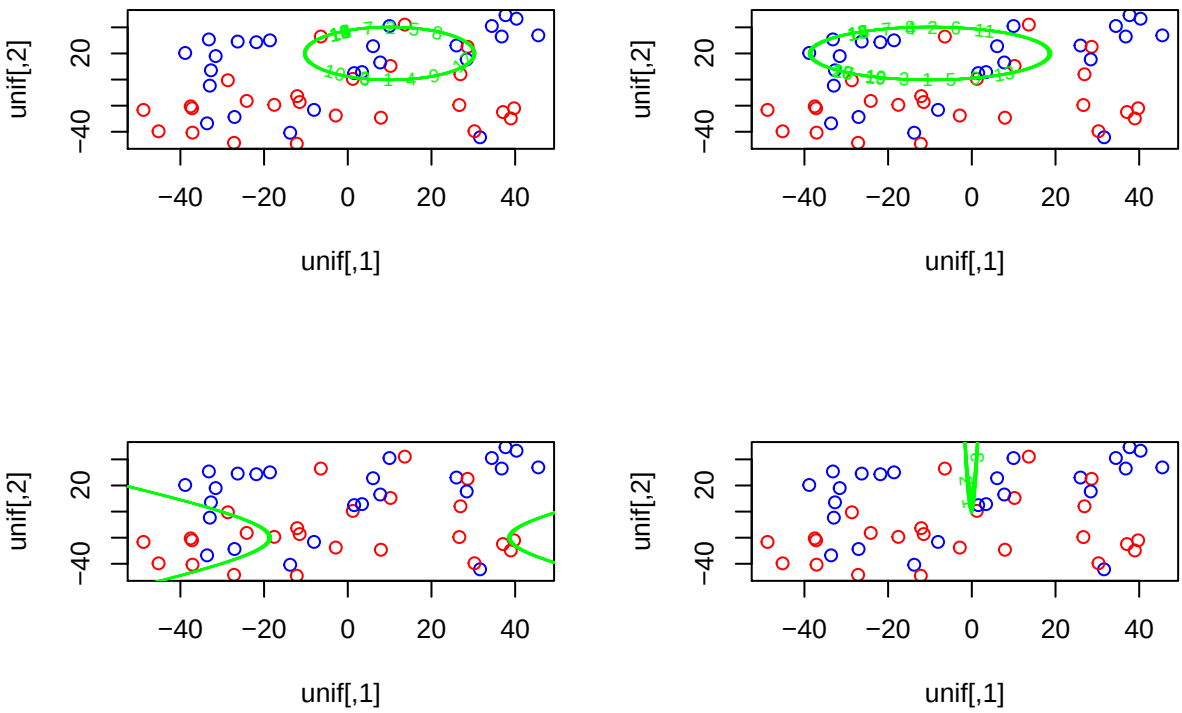


Apartado 1.3

Vamos a declarar las funciones que se nos piden en el ejercicio, no necesitamos despejar y dejar las funciones en función de una de las variables ya que tenemos la función pintar_frontera que se nos paso:

```
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, xlim =rango, ylim=rango, xlab = "x", ylab = "y", add=TRUE, col="green")
}

f1 = function(x, y){
  (x-10)^2 + (y-20)^2 - 400
}
```

```r
f2 = function(x,y){
  0.5 * (x+10)^2 + (y-20)^2 - 400
}

f3 = function(x,y){
  0.5 * (x-10)^2 - (y+20)^2 - 400
}

f4 = function(x,y){
  y - 20*x^2 - 5*x + 3
}

old.par = par(mfrow=c(2,2))
intervalo = c(-100,100)

plot(unif, col = signos_ruido + 3)
pintar_frontera(f1, intervalo)

plot(unif, col = signos_ruido + 3)
pintar_frontera(f2, intervalo)

plot(unif, col = signos_ruido + 3)
pintar_frontera(f3, intervalo)

plot(unif, col = signos_ruido + 3)
pintar_frontera(f4, intervalo)
```



Una vez pintadas las gráficas sobre los datos generados en el apartado 2b, se nos pide comparar las formas de las nuevas regiones positivas y negativas de las nuevas funciones con las obtenidas en el caso de la recta. Cuando comparamos las gráficas podemos apreciar que las nuevas regiones positivas y negativas son curvas pues nuestras funciones son ahora cuadráticas en lugar de la lineal del apartado 2. Por lo que debido a esta

nueva forma encierran valores lo cual puede hacer que la función sea mejor clasificador que la función lineal pero esto depende de los datos. Por tanto a la pregunta de si estas funciones más complejas son mejores clasificadores que la función lineal, la respuesta es que depende de los datos ya que por ejemplo en los datos generados en el ejercicio 2b al meterles ruido tenemos que son peores clasificadores que la función lineal, sin embargo, en caso de que nuestros datos tengan digamos unos datos contenidos en otros, estas funciones pueden dar unos resultados más favorables que la función lineal. En conclusión, la respuesta es que no son ni peores ni mejores, depende de los datos que sean mejores o peores, cabe mencionar que me estoy refiriendo a las funciones del ejercicio 3 sin modificar, ya que si por ejemplo tenemos la funcion $0.00001*x^2$ tenemos que es muy parecida a la función x y pueden valer para el mismo tipo de ejercicio, es decir, me refiero a las funciones de este ejercicio en concreto. La siguiente pregunta es en que ganan a la función lineal, me voy a referir generalmente ya que en concreto en este ejercicio no ganan a la lineal en nada. Ganan claramente en versatilidad, ya que permiten que los datos estén distribuidos de más formas que con la función lineal. En el mundo real las funciones cuadráticas como estas son generalmente más utiles que la lineal ya que los datos no suelen estar tan bien definidos como en estos ejercicios. Por ejemplo en la práctica 1 la clasificación de 1 y 5 se podrían conseguir mejores resultados con una función cuadrática que con una lineal, esto lo sabemos por la forma de los datos que aunque sea algo que no debemos mirar al hacer la práctica lo vimos.

Ejercicio 2:

Apartado 2.1 - Algoritmo Perceptron

Se nos pide implementar el algoritmo perceptron, este algoritmo es muy simple y digamos poco eficaz en la mayoría de los casos ya que como los datos no están perfectamente divididos por al menos una recta, el algoritmo perceptron no será capaz de devolver una solución de calidad ya que llegaría al máximo de iteraciones y devolvería el último ajuste. Esto ocurre porque el algoritmo perceptron no tiene memoria y por tanto al no estar perfectamente divididos los datos, devovlerá el último ajuste, mejor o peor que los anteriores. A continuación vemos la implementación, la esencia de esta es la parte de mult = sum(weight * datos[i,]) ya que es lo que hace que se ajuste o no el perceptron, después de esta instrucción truncamos el valor de mult en función de si es mayor o igual a 0 o no. Cuando se procduzca una iteración completa sin ajustar el perceptron, querrá decir que ha convergido y por tanto nos salimos del bucle. Después comprobamos los errores con el vector de pesos obtenido con el perceptron, esto es para el próximo ejercicio ya que esta claro que si converge tendrá cero fallos y en el apartado a converge. La función devuelve el número de fallos, las iteraciones y el vector de pesos obtenido. Además tenemos la función de predicción que como su nombre indica nos valdrá para comprobar la eficacia de los pesos obtenidos por los distintos algoritmos

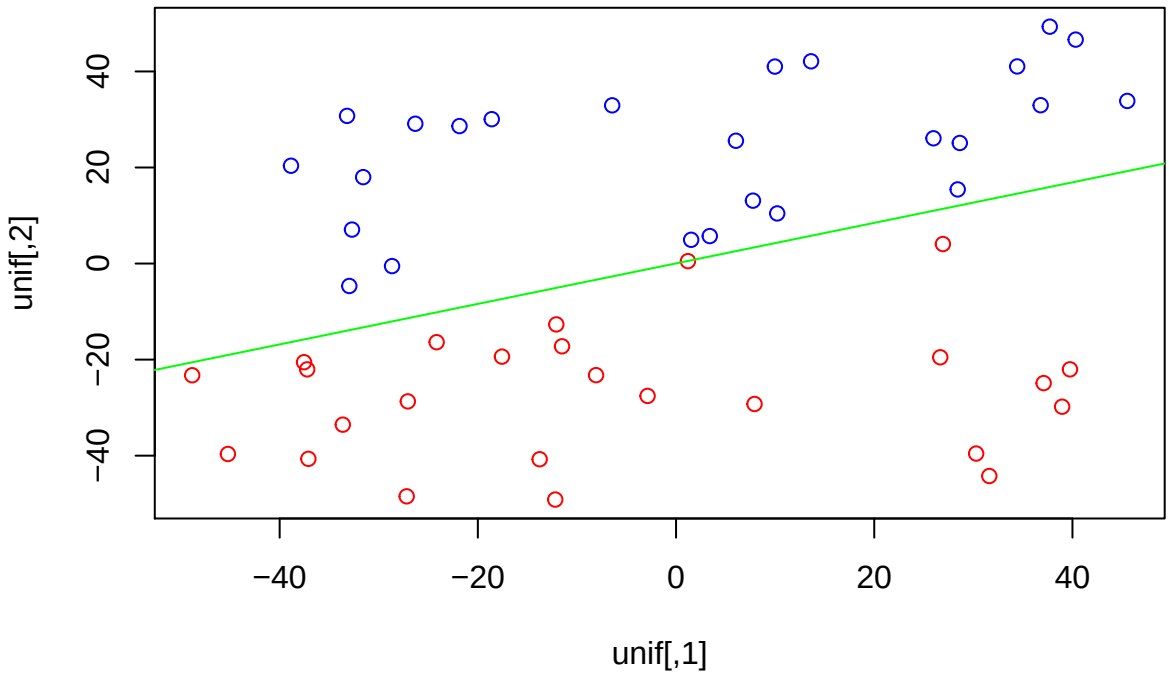
```
pasoARecta= function(w){
  if(length(w)!= 3)
    stop("Solo tiene sentido con 3 pesos")
  a = -w[1]/w[2]
  b = -w[3]/w[2]
  c(a,b)
}

prediccion = function(w, x) {
  sign(t(w) %*% t(t(x)))
}

ajusta_PLA = function(datos, label, max_iter = 10000, vini = c(0,0,0)) {
  weight = vini
  iter = 0
  converge = F

  while(converge == F & iter < max_iter) {
```

```
    converge = T
    for(i in 1:dim(datos)[1]) {
      mult = prediccion(weight, datos[i, ])

      if(mult != label[i]) {
        weight = weight + label[i] * datos[i, ]
        converge = F
      }
    }
    iter = iter + 1
  }

  fallos = 0

  if(converge == F) {
    for(i in 1:dim(datos)[1]) {
      mult = prediccion(weight, datos[i, ])

      if(mult != label[i]) {
        fallos = fallos + 1
      }
    }
  }
  c(fallos, iter, weight)
}
```

a)

Aplicamos la implementación anterior de perceptron a los datos obtenidos en el apartado 2a y podremos apreciar que converge y en cuantas iteraciones lo hace:

```
old.par = par(mfrow=c(1,1))

# a)
vini = rep(0, dim(unif)[2])
max_iter = 100
pla = ajusta_PLA(unif, signos, max_iter, vini)

plot(unif, col = signos + 3)
abline(pasoARecta(pla[3:5])[2], pasoARecta(pla[3:5])[1], col = "green")
```




```
print("PLA con pesos iniciados a 0:")
```

```
## [1] "PLA con pesos iniciados a 0:"
```

```
print(sprintf("Error = %s", pla[1]/dim(unif)[1]))
```

```
## [1] "Error = 0"
```

```
print(sprintf("Iteraciones hasta converger = %s", pla[2]))
```

```
## [1] "Iteraciones hasta converger = 5"
```

```
###############

media = 0
media_f = 0
iteraciones = 10

for(i in 1:iteraciones) {
  pesos_iniciales = runif(n=dim(unif)[2], min=0, max=1)

  pla = ajusta_PLA(unif, signos, max_iter, pesos_iniciales)
  media = media + pla[2]
  media_f = media_f + pla[1]
}

media = media / iteraciones
media_f = media_f / iteraciones

print("PLA con pesos iniciados aleatoriamente, media de las 10 ejecuciones:")
```

```
## [1] "PLA con pesos iniciados aleatoriamente, media de las 10 ejecuciones:"
```

```
print(sprintf("Media de errores = %s", media_f/dim(unif)[1]))
```

```
## [1] "Media de errores = 0"
```

```
print(sprintf("Media de iteraciones hasta converger = %s", media))
```

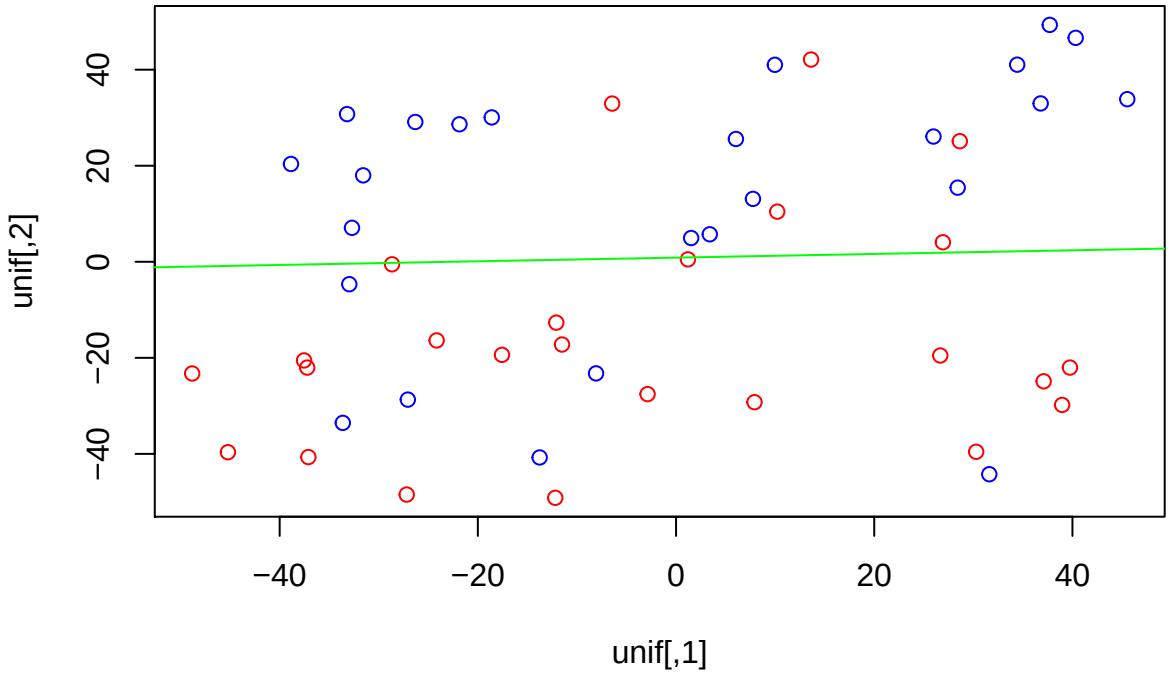
```
## [1] "Media de iteraciones hasta converger = 5.1"
```

```
###############
```

b)

Ahora vamos a aplicar el perceptron a los datos del apartado 2b, es decir, a los que le hemos introducido ruido. Veremos que no converge da igual el número de iteraciones que le pongamos, esto ocurre porque los datos no están perfectamente divididos por una o más rectas como he explicado antes:

```
# b)

pesos_iniciales = rep(0, dim(unif)[2])
pla = ajusta_PLA(unif, signos_ruido, max_iter, pesos_iniciales)

plot(unif, col = signos_ruido + 3)
abline(pasoARecta(pla[3:5])[2], pasoARecta(pla[3:5])[1], col = "green")
```




```
print("PLA con pesos a cero y ruido")
```

```
## [1] "PLA con pesos a cero y ruido"
```

```
print(sprintf("Error = %s", pla[1]/dim(unif)[1]))
```

```
## [1] "Error = 0.22"
```

```
print(sprintf("Iteraciones hasta converger = %s", pla[2]))
```

```
## [1] "Iteraciones hasta converger = 100"
```

```
###############

media = 0
```

```
media_f = 0

for(i in 1:iteraciones) {
  pesos_iniciales = runif(n=dim(unif)[2], min=0, max=1)

  pla = ajusta_PLA(unif, signos_ruido, max_iter, pesos_iniciales)
  media = media + pla[2]
  media_f = media_f + pla[1]
}

media = media / iteraciones
media_f = media_f / iteraciones

print("PLA con pesos iniciados aleatoriamente, media de las 10 ejecuciones:")
```

```
## [1] "PLA con pesos iniciados aleatoriamente, media de las 10 ejecuciones:"
```

```
print(sprintf("Media de errores = %s", media_f/dim(unif)[1]))
```

```
## [1] "Media de errores = 0.26"
```

```
print(sprintf("Media de iteraciones hasta converger = %s", media))
```

```
## [1] "Media de iteraciones hasta converger = 100"
```

```
###############
```

Apartado 2.2 - Regresión logística

En este ejercicio se nos pide implementar regresión logísitca (RL) con gradiente descendente estocástico (SGD). La regresión logística es básicamente un modelo lineal en el que en vez de darnos un valor como tal o -1 o 1 por ejemplo, nos daría una probabilidad como salida. En este modelo la salida es real como en regresión pero acotada como en clasificación. Para poder darnos una probabilidad se usa la función sigmoide ($\theta(x) = \frac{1}{1+e^{-x}}$) la cual podemos ver implementada a continuación, la x de nuestra función se compone de yw^Tx. También podemos ver nuestra función de error para la regresión logística, el cual no usamos ya que para nuestro modelo según el profesor de prácticas lo que debemos de usar es la función de error de clasificación pero con un umbral 0.5 como veremos más adelante:

```
h = function(etiqueta, w, x) {   # Funcion sigmoide, esta es la h de la regresión logística
  #(exp(etiqueta*w*x) / (exp(etiqueta*w*x)+1))
  1 / (1 + exp(-(etiqueta*t(w)%*%x)))
}

e = function(etiqueta, w, x) {
  log(1+exp(-etiqueta*t(w)%*%x))
}
```

Esta función sigmoide de manera suavizada restringe la salida a una probabilidad entre 0 y 1. La usaremos para ajustar nuestro SGD el cual se puede ver a continuación, podemos ver que en este caso lo resaltable del algoritmo y lo que requiere explicación es la parte de la sumatoria. También vemos la función de predicción ya que al usar la sigmoide lo que tenemos es que RL nos devuelve una probabilidad y a partir de 0.5 lo tomamos como 1 y lo demás como -1:

```
prediccion2 = function(w, x) {
  if(t(w) %*% t(t(x)) >= 0.5) {
    1
```

```
  }
  else {
    -1
  }
}

SGD = function(datos, resultados, wini = c(0, 0, 0), nitr = 100000, nu = 0.01, tminibatches, tope = 0.0
  numero_de_iteraciones = 0
  sumatoria = c(0,0,0)
  N = as.integer(length(resultados) * tminibatches)
  wini_anterior = c(999,999,999)

  while(abs(sum(wini_anterior) - sum(wini)) >= tope) {
    sumatoria = c(0,0,0)
    wini_anterior = wini
    for(j in 1:N) {
      aleatorio = sample(length(resultados),1)
      sumatoria = sumatoria + (-as.vector(resultados[aleatorio] * datos[aleatorio,])) * as.vector(h(-res
    }

    sumatoria = -sumatoria / N
    wini = wini + nu * sumatoria
    numero_de_iteraciones = numero_de_iteraciones + 1
  }

  wini
}
```

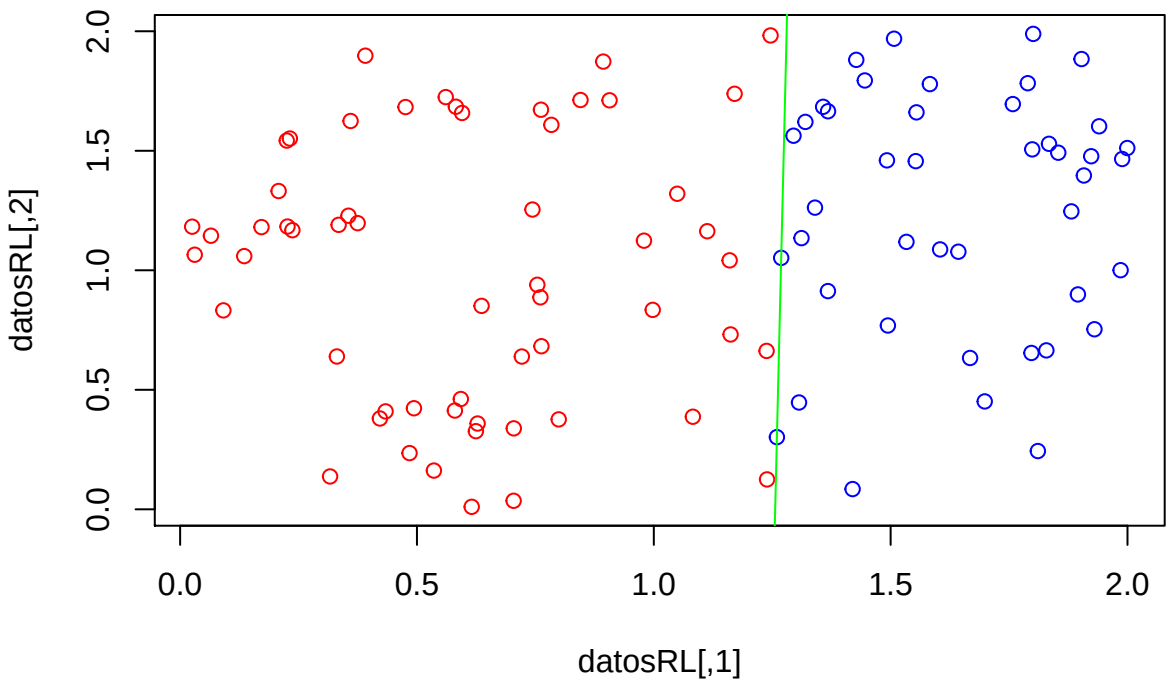
Básicamente para ajustar nuestro gradiente descendente estocástico necesitamos usar la fórmula $\frac{1}{N}\sum_{n=1}^{N} -y_n x_n \theta(-y_n w^T x_n)$ (fórmula sacada del libro pasado en prácticas "LearningFromDataChap3") que usa nuestra función sigmoide descrita anteriormente. Esto es lo que hará que nuestro SGD se ajuste. La sumatoria es la parte $-y_n x_n \theta(-y_n w^T x_n)$ y posteriormente se divide entre N que es el número de elementos que tiene nuestro minibatch. Una vez dividimos entre N y le cambiamos el signo tan solo debemos de actualizar el peso actual sumandole la tasa de aprendizaje por la sumatoria obtenida anteriormente. Debido a como es el algoritmo de gradiente descendente, sabemos que es posible que se quede en un óptimo local y es por eso que nos da peores resultados que PLA para los datos generados (ya que están clasificados según una recta creada por nosotros como se ve en el código de abajo), esto ocurre en este ejercicio pero en la vida real dificilmente PLA dará mejores resultados que gradiente descendente con regresión logísitca para este tipo de ejercicios. Luego comprobamos los fallos que hemos tenido y devolvemos los pesos para poder dibujar la recta y ver mejor esto. En cuanto a la condición de parada descrita en el guión de la práctica, la hemos tenido que cambiar puesto que con un tope de 0.01 el algoritmo paraba demasiado pronto y el ajuste era prácticamente nulo de ahí que cuando llamamos a esta función la llamaremos con un tope de 0.0000001 con el cual el ajuste es factible.

NOTA: Se que este algoritmo esta pensado para dar un porcentaje y no un SI o un NO pero para poder ver los fallos más claramente lo he considerado como o 1 o -1 a la hora de predecir, se puede ver en la función de más arriba llamada predecir2.

Primero preparamos los datos y luego ejecutamos el SGD con RL para obtener nuestro ajuste:

```
genera_recta = function (x, y){
  a = (x[2] - y[2]) / (x[1]-y[1]) # calculo de la pendiente
  b = x[2]-a*x[1]  # calculo del punto de corte
  c(a,b) # devuelve el par pendiente y punto de corte
}
```

```
fSGD = function(x,y,a,b){
  if (-y + a * x + b >= 0)
    1
  else
    -1
}

# Preparando datos
datosRL = simula_unif(N=100,dim=2,rango=c(0,2))
datosRL = cbind(datosRL, 1)
puntos = sample(1:dim(datosRL)[1], 2, replace = FALSE)
rectaE = genera_recta(datosRL[puntos[1],], datosRL[puntos[2],])

etiquetas = c()

for(i in 1:dim(datosRL)[1]) {
  etiquetas[i] = fSGD(datosRL[i, 1], datosRL[i, 2], rectaE[1], rectaE[2])
}

plot(datosRL, col = etiquetas + 3)
abline(rectaE[2], rectaE[1], col = "green")
```

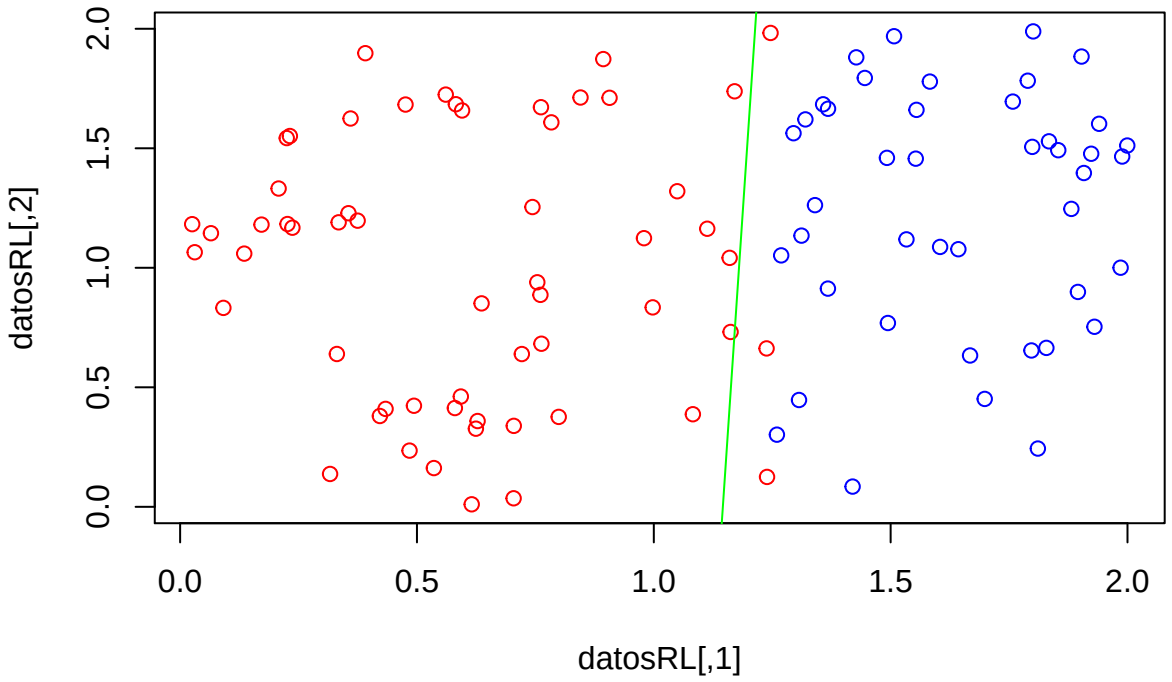



```
pesosSGD = rep(0, dim(datosRL)[2])  # Pesos inicializados a 0
pesosSGD = SGD(datos = datosRL, resultados = etiquetas, nitr = 10000, tminibatches = 0.2, tope = 0.00000

fallos = 0
for(i in 1:dim(datosRL)[1]) {
  mult = prediccion2(pesosSGD, datosRL[i, ])

  if(mult != etiquetas[i]) {
    fallos = fallos + 1
  }
```

```
}

print(sprintf("Ein RL = %s", fallos/dim(datosRL)[1]))
```

```
## [1] "Ein RL = 0.01"
```

```
plot(datosRL, col = etiquetas + 3)
abline(pasoARecta(pesosSGD)[2], pasoARecta(pesosSGD)[1], col = "green")
```



Cabe destacar el problema con el tope comentado anteriormente de ahí que se le pase como tope 0.0000001. Podemos apreciar lo dicho anteriormente, a pesar de estar perfectamente divididos (los hemos clasificado según una recta creada por nosotros como se ve arriba), no da una solución tan buena como la daría PLA en este dataset, sin embargo en el mundo real la inmensa mayoría de veces si no todas las veces para estos tipos de ejercicios SGD con RL daría mejores resultados, en parte porque PLA lo más seguro es que nunca sea capaz de converger y no tiene memoria. **El error que nos da es un valor que no concuerda con la gráfica, esto ocurre porque como usamos como umbral 0.5 en la prediccion2 los valores cercanos a la propia recta no están bien clasificados del todo y puede ser que por la forma de la sigmoide clasifique un número bien aunque en la gráfica aparentemente no.**

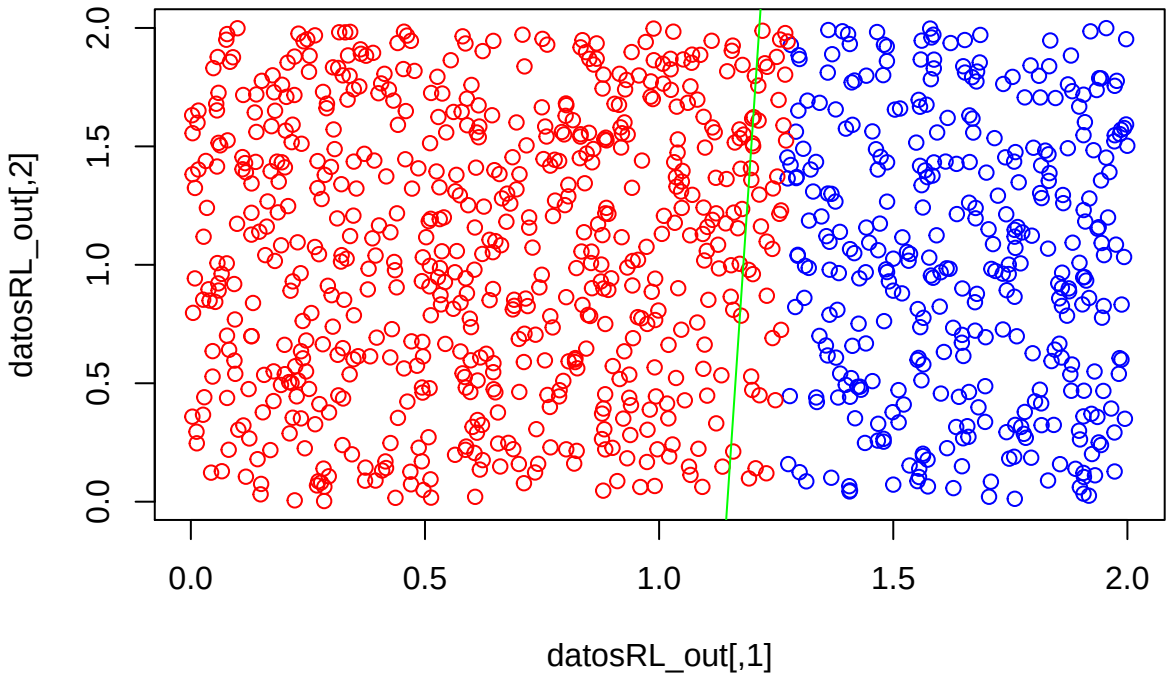
Ahora vamos a ver lo que ocurre si creamos un dataset nuevo con más de 999 muestras y lo etiquetamos según la misma recta que el anterior y probamos nuestro vector de pesos:

```
# b)
datosRL_out = simula_unif(N=1000,dim=2,rango=c(0,2))
datosRL_out = cbind(datosRL_out, 1)

etiquetas = c()

for(i in 1:dim(datosRL_out)[1]) {
  etiquetas[i] = fSGD(datosRL_out[i, 1], datosRL_out[i, 2], rectaE[1], rectaE[2])
}

fallos = 0
for(i in 1:dim(datosRL_out)[1]) {
```

```
  mult = prediccion2(pesosSGD, datosRL_out[i, ])

  if(mult != etiquetas[i]) {
    fallos = fallos + 1
  }
}

print(sprintf("Eout RL = %s", fallos/dim(datosRL_out)[1]))
```

```
## [1] "Eout RL = 0.006"
```

```
plot(datosRL_out, col = etiquetas + 3)
abline(pasoARecta(pesosSGD)[2], pasoARecta(pesosSGD)[1], col = "green")
```



Vemos que existen fallos pero la tasa de estos no se ha disparado, esto ocurre porque hemos usado la misma recta para etiquetar.

Puesto que la diferencia entre un peso y el anterior nunca llega a ser cero, este algoritmo no nos garantiza una seguridad plena en su aprendizaje.

BONUS

Vamos a preparar los datos de la misma manera que en la práctica 1 solo que esta vez cogeremos los 4 y los 8:

```
fSimetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -sum(A)
}

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

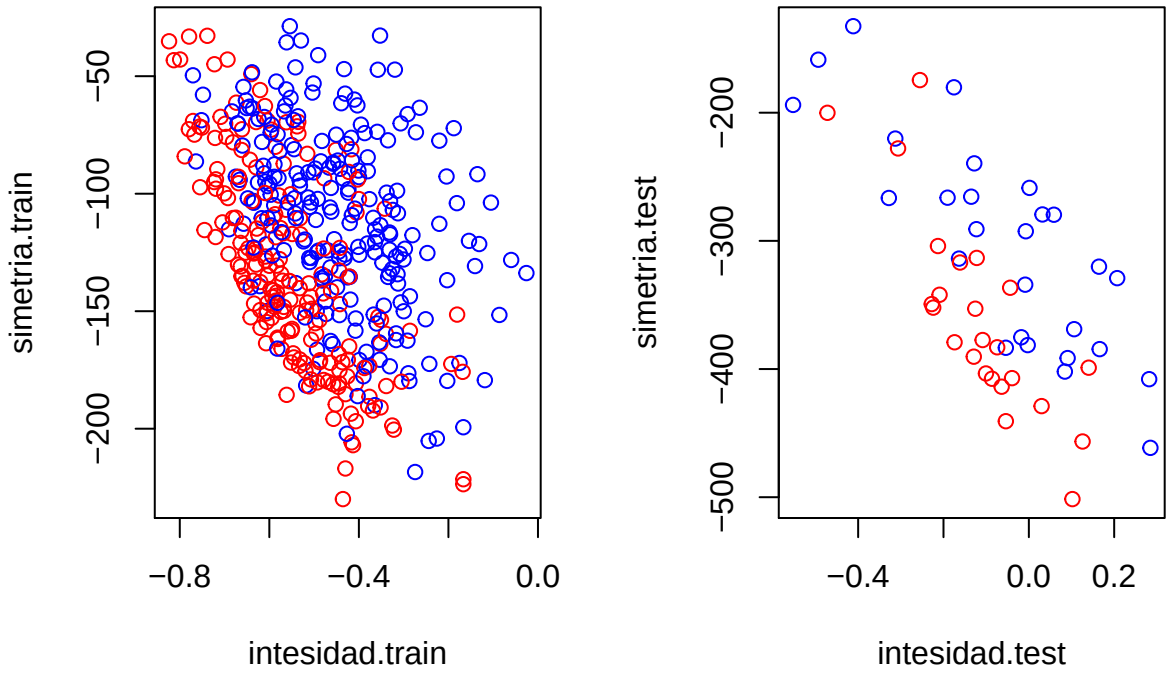
```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : number of items read is not a multiple of the number of columns
```

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

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : number of items read is not a multiple of the number of columns
```

```r
# Ahora nos quedamos solamente con los 1 y los 5
digitos48.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]
digitos48.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]

digitos.train = digitos48.train[,1]    # vector de etiquetas(clases) del train, en este caso columna 1
ndigitos.train = nrow(digitos48.train)  # numero de muestras del train
digitos.test = digitos48.test[,1]    # vector de etiquetas(clases) del test, en este caso columna 1
ndigitos.test = nrow(digitos48.test)  # numero de muestras del test

# se retira la clase y se monta una matriz 3D: 599*16*16
grises.train = array(unlist(subset(digitos48.train,select=-V1)),c(ndigitos.train,16,16)) # Cada numero
rm(digit.train)
rm(digitos48.train)
# Ahora con el test
grises.test = array(unlist(subset(digitos48.test,select=-V1)),c(ndigitos.test,16,16))
rm(digit.test)
rm(digitos48.test)

# 2º. Vamos a obtener la intensidad media de nuestro dataset:

intesidad.train = apply(X = grises.train, MARGIN = 1, FUN = mean) # Intensidad de cada uno de los numer
intesidad.test = apply(X = grises.test, MARGIN = 1, FUN = mean)

# 3º. Vamos a obtener la simetria respecto al eje vertical de nuestro dataset:

simetria.train = apply(X = grises.train, MARGIN = 1, FUN = fSimetria) # Simetria de cada uno de los num
simetria.test = apply(X = grises.test, MARGIN = 1, FUN = fSimetria) # Simetria de cada uno de los numer

rm(grises.train) # Liberamos memoria pues ya no necesitamos a grises
rm(grises.test)

# 4º. Vamos a recodificar nuestras etiquetas, cambiando el 5 por -1

digitos.train[digitos.train == 4] = -1
digitos.test[digitos.test == 4] = -1
digitos.train[digitos.train == 8] = 1
digitos.test[digitos.test == 8] = 1

# 5º. Componemos datosTr, que es la unión por columnas de la intensidad con la simetria, columna 1 = in

datosTr = as.matrix(cbind(intesidad.train, simetria.train))
datosTst = as.matrix(cbind(intesidad.test, simetria.test))

# Introducimos la variable independiente (1) en nuestros datos de intesidad y simetria
datosTr = cbind(datosTr, 1)
datosTst = cbind(datosTst, 1)

old.par = par(mfrow=c(1,2))
plot(intesidad.train, simetria.train, col = digitos.train + 3)
```

```r
plot(intesidad.test, simetria.test, col = digitos.test + 3)
```



Podemos apreciar en el plot de arriba que los datos en este caso están muy mezclados y es que al parecer en el dataset los 4 y los 8 son bastante parecidos según nuestro criterio de simetría e intensidad. Esto puede ocasionar que sea bastante dificil ajustar un modelo lineal. Para hacer más fácil y mejor el ajuste una de las posibles técnicas sería aplicarle una transformación para ver si se posicionan mejor, pero esto no se pide en el bonus.

A continuación tenemos el algoritmo de SGD de la práctica 1 un poco retocado pero en esencia es lo mismo y nos servirá para comparar este con el algoritmo PLA-Pocket

```r
# Gradiente descendiente estocástico

SGD_noRL = function(datos, resultados, wini = c(0, 0, 0), nitr = 100000, nu, tminibatches) {
  numero_de_iteraciones = 0
  sumatoria = c(1,1,1)
  N = as.integer(length(resultados) * tminibatches)

  for(i in 1:nitr) {
    for(i in 1:N) {
      aleatorio = sample(length(resultados),1)
      sumatoria = sumatoria - (as.vector(resultados[aleatorio] * datos[aleatorio,])) * (as.vector((exp(
    }

    sumatoria = sumatoria / N
    wini = wini - nu * sumatoria
    numero_de_iteraciones = numero_de_iteraciones + 1
  }

  wini
}
```

```r
# REGRESION CON GRADIENTE DESCENDENTE ESTOCÁSTICO
max_iter = 100
pesos_2a_sgd = SGD_noRL(datosTr, digitos.train, c(0, 0, 0), max_iter, 0.01, 0.5)  # Calculo de los peso
fallos_tr = 0
for(i in 1:dim(datosTr)[1]) {   # Medimos Ein con el vector de pesos actual
  mult = prediccion(pesos_2a_sgd, datosTr[i, ])

  if(mult != digitos.train[i]) {
    fallos_tr = fallos_tr + 1
  }
}

cuatros = length(digitos.train[digitos.train == -1])
ochos = length(digitos.train[digitos.train == 1])

# Imprimimos por pantalla las dos gráficas y la recta de regresión donde podemos ver que ocurre
old.par = par(mfrow=c(1,2))
plot(datosTr, col = digitos.train + 3, type = "p", ylim=c(-250,50))
abline(pasoARecta(pesos_2a_sgd)[2], pasoARecta(pesos_2a_sgd)[1], col = "green")
print(sprintf("Ein = %s", fallos_tr/dim(datosTr)[1]))
```

```
## [1] "Ein = 0.534722222222222"
```

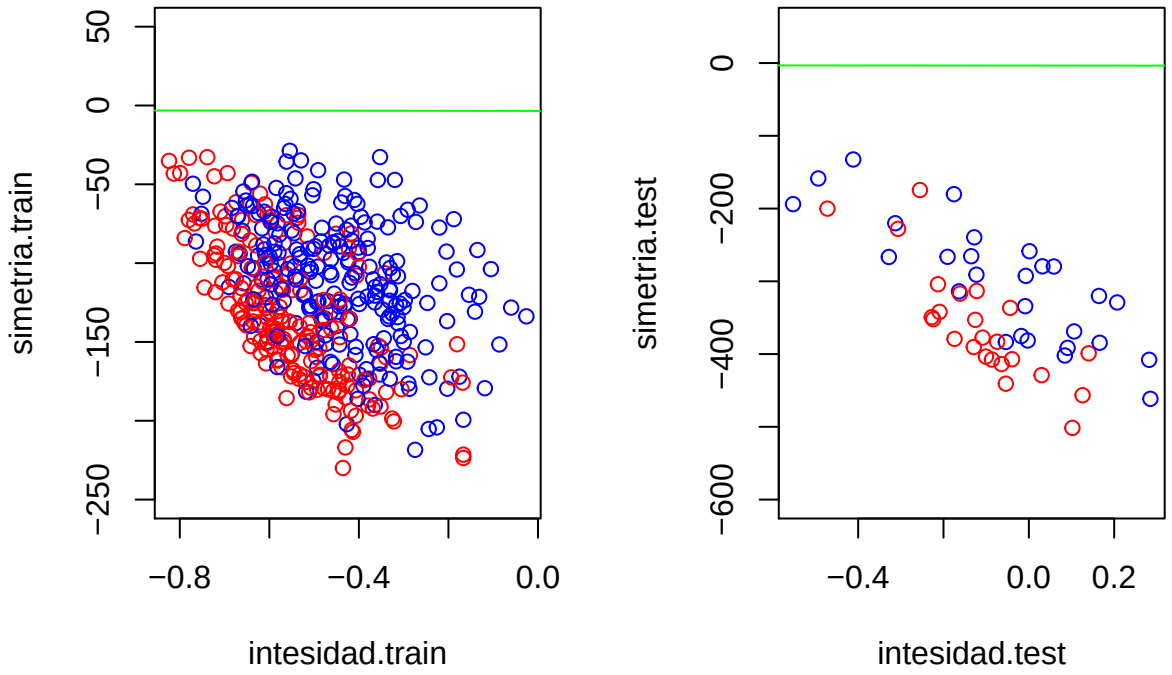
```r
print(sprintf("Nº de cuatros = %s", cuatros))
```

```
## [1] "Nº de cuatros = 201"
```

```r
print(sprintf("Nº de ochos = %s", ochos))
```

```
## [1] "Nº de ochos = 231"
```

```r
cuatros = length(digitos.test[digitos.test == -1])
ochos = length(digitos.test[digitos.test == 1])
fallos_tr = 0
for(i in 1:dim(datosTst)[1]) {   # Medimos Ein con el vector de pesos actual
  mult = prediccion(pesos_2a_sgd, datosTst[i, ])

  if(mult != digitos.test[i]) {
    fallos_tr = fallos_tr + 1
  }
}

plot(datosTst, col = digitos.test + 3, type = "p", ylim=c(-600,50))
abline(pasoARecta(pesos_2a_sgd)[2], pasoARecta(pesos_2a_sgd)[1], col = "green")
```




```
print(sprintf("Etest = %s", fallos_tr/dim(datosTst)[1]))
```

```
## [1] "Etest = 0.529411764705882"
```

```
print(sprintf("Nº de cuatros = %s", cuatros))
```

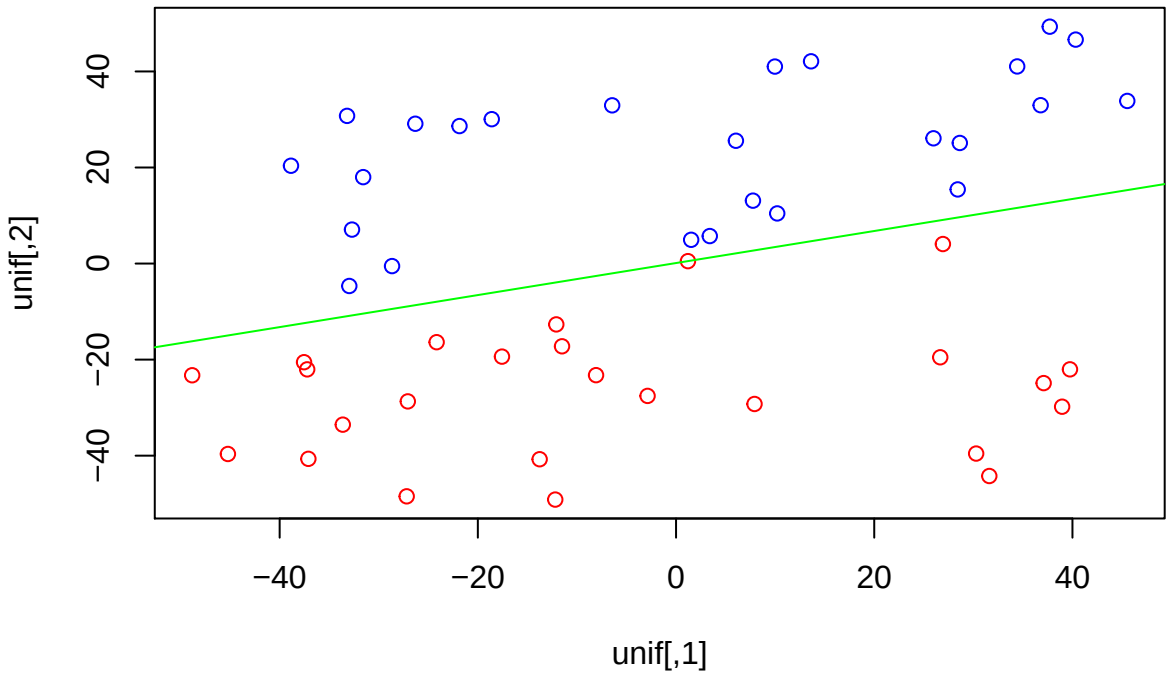
```
## [1] "Nº de cuatros = 24"
```

```
print(sprintf("Nº de ochos = %s", ochos))
```

```
## [1] "Nº de ochos = 27"
```

En vista del ajuste obtenido arriba podemos sospechar que algo no está funcionando bien por lo que probamos con un dataset que sabemos que está dividido perfectamente:

```
## Comprobacion funcionamiento
pesos_2a_sgd = SGD_noRL(unif, signos, c(0, 0, 0), 1000, 0.01, 0.5)  # Calculo de los pesos mediante SGD
plot(unif, col = signos + 3)
abline(pasoARecta(pesos_2a_sgd)[2], pasoARecta(pesos_2a_sgd)[1], col = "green")
```



Vemos que funciona correctamente ya que con el dataset unif sabemos que recta divide los datos y la encuentra correctamente, por lo que llegamos a la conclusión de que lo que ocurre es que puesto que los datos están tan mezclados, el algoritmo encuentra como mejor ajuste colocar la recta arriba de estos para de este modo conseguir acertar al menos todos los datos de una de las etiquetas. Esta etiqueta es el 8 que como esta impreso arriba son 231 contra 201 de los cuatros por lo que nuestro algoritmo clasificará bien los ochos ya que esto proporciona las mejores predicciones en cuanto a porcentaje de acierto total aunque esto signifique fallar todos los cuatros.

Con las características de los datos es imposible separarlos linealmente, debido a esto los algortimos(ambos) tienen un comportamiento muy similar, si los datos a clasificar fueran por ejemplo 1 y 8 sería más probable que se pudiesen separar debido a su grado de simetría e intesidad.

Ahora vemos con el algoritmo PLA-Pocket el cual debería de llegar a la misma conclusión ya que consiste en básicamente tirar t PLAs y quedarse con la mejor iteración del PLA:

```
pocket = function(datos, label, max_iter, vini) {
  pesos_pla = vini
  iter = 0
  fallos = 0
  fallos_anterior = length(datos)

  for(j in 1:max_iter) {
    for(i in 1:dim(datos)[1]) {
      mult = prediccion(pesos_pla, datos[i, ])  # Vemos si nuestro vector de pesos clasifica bien la mu

      if(mult != label[i]) {  # Si no la clasifica bien ajustamos
        pesos_pla = pesos_pla + label[i] * datos[i, ]

        for(i in 1:dim(datos)[1]) {   # Medimos Ein con el vector de pesos actual
          mult = prediccion(pesos_pla, datos[i, ])

          if(mult != label[i]) {
            fallos = fallos + 1
```

```
          }
        }

        if(fallos < fallos_anterior) {
          weight = pesos_pla
        }
        fallos_anterior = fallos
        fallos = 0
      }
    }
  }

  weight
}

vini = rep(0, dim(datosTr)[2])
old.par = par(mfrow=c(1,2))
pock = pocket(datosTr, digitos.train, max_iter, vini)
plot(datosTr, col = signos + 3, ylim=c(-250,50))
abline(pasoARecta(pock)[2], pasoARecta(pock)[1], col = "green")

fallos_tr = 0
for(i in 1:dim(datosTr)[1]) {   # Medimos Ein con el vector de pesos actual
  mult = prediccion(pock, datosTr[i, ])

  if(mult != digitos.train[i]) {
    fallos_tr = fallos_tr + 1
  }
}

cuatros = length(digitos.train[digitos.train == -1])
ochos = length(digitos.train[digitos.train == 1])

print(sprintf("Ein = %s", fallos_tr/dim(datosTr)[1]))
```

```
## [1] "Ein = 0.465277777777778"
```

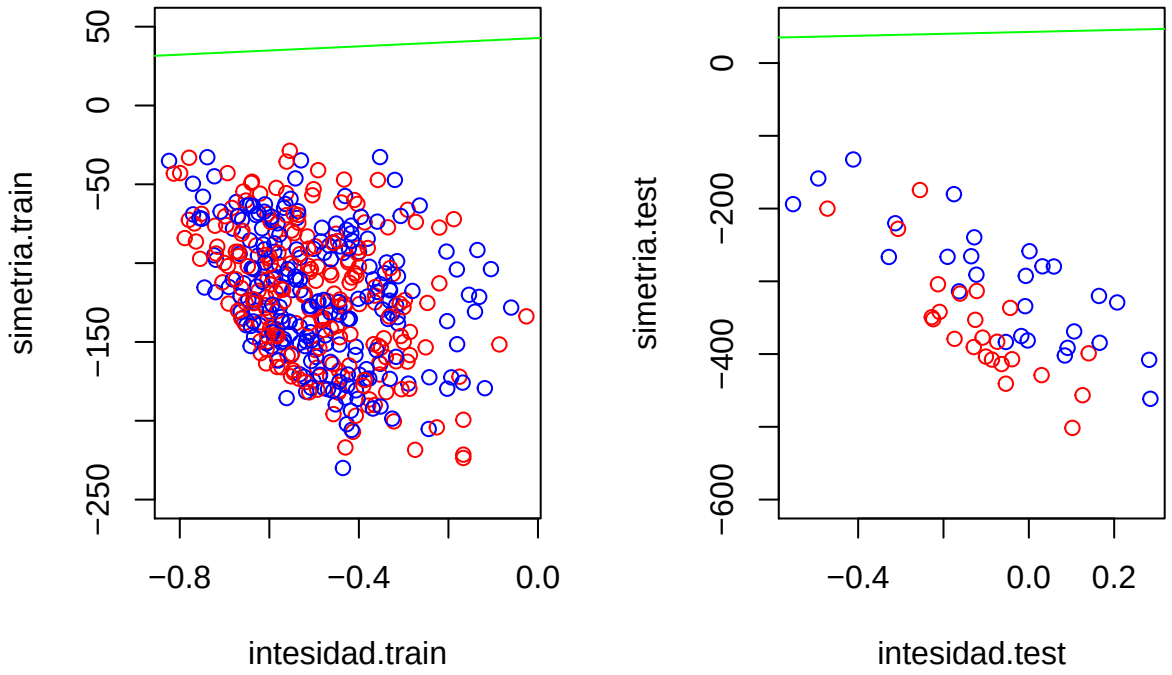
```
print(sprintf("Nº de cuatros = %s", cuatros))
```

```
## [1] "Nº de cuatros = 201"
```

```
print(sprintf("Nº de ochos = %s", ochos))
```

```
## [1] "Nº de ochos = 231"
```

```
plot(datosTst, col = digitos.test + 3, ylim=c(-600,50))
abline(pasoARecta(pock)[2], pasoARecta(pock)[1], col = "green")
```




```
print(sprintf("Iteraciones de ambos algortimos = %s", max_iter))
```

```
## [1] "Iteraciones de ambos algortimos = 100"
```

```
cuatros = length(digitos.test[digitos.test == -1])
ochos = length(digitos.test[digitos.test == 1])
fallos_tr = 0
for(i in 1:dim(datosTst)[1]) {   # Medimos Ein con el vector de pesos actual
  mult = prediccion(pock, datosTst[i, ])

  if(mult != digitos.test[i]) {
    fallos_tr = fallos_tr + 1
  }
}

print(sprintf("Etest = %s", fallos_tr/dim(datosTst)[1]))
```

```
## [1] "Etest = 0.470588235294118"
```

```
print(sprintf("Nº de cuatros = %s", cuatros))
```

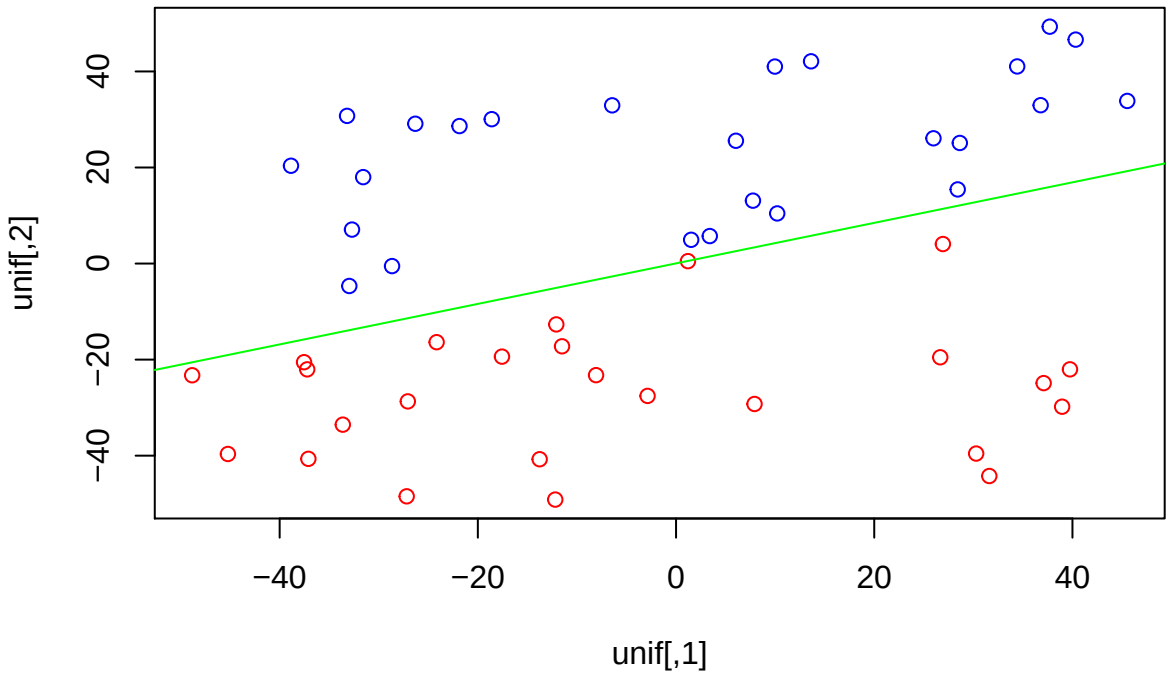
```
## [1] "Nº de cuatros = 24"
```

```
print(sprintf("Nº de ochos = %s", ochos))
```

```
## [1] "Nº de ochos = 27"
```

Vemos que llega a la misma conclusión como debe de ocurrir pero por seguridad y por ver si realmente funciona bien lo volveremos a probar con el dataset de comprobación:

```
## Comprobacion funcionamiento
pock = pocket(unif, signos, max_iter, vini)
plot(unif, col = signos + 3)
abline(pasoARecta(pock)[2], pasoARecta(pock)[1], col = "green")
```



Y vemos que el funcionamiento es el correcto.

Lo que se nos pide en el último apartado del bonus es básicamente aplicar la teoría de **Vapnik–Chervonenkis** la cual nos sirve para saber la medida de la capacidad(complejidad, poder de expresión, flexibilidad) del espacio de funciones que pueden ser aprendidas mediante el algoritmo de clasificación. La dimensión VC tiene utilidad en teoría de aprendizaje estadístico, porque puede predecir el límite superior probabilístico sobre el error de test del modelo de clasificación.