Práctica 1

Anabel Gómez Ríos

1. Ejercicio de Generación y Visualización de datos.

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

Lo primero que hago es fijar una semilla para toda la práctica, puesto que la generación aleatoria se va a utilizar mucho y es conveniente que los resultados se puedan repetir. Fijo la semilla a 237 con la función `set.seed()`.

Como pide una lista, utilizo la función `lapply()`, que itera sobre una lista o un vector (en este caso un vector de la longitud pedida) y devuelve una lista en la que vamos a meter los números aleatorios uniformes generados con la función `runif()`, en la que ya le decimos la dimensión y el rango en el que tiene que muestrear (de forma que si la dimensión es 2 genera los números de dos en dos y los mete de dos en dos en cada elemento de la lista).

```
set.seed(237)
# Para que muestre las gráficas por pantalla
X11()
```

```
simula_unif <- function(N, dim, rango) {
  lapply(1:N, function(x) runif(dim, min = rango[1], max = rango[2]))
}
```

Por ejemplo, vamos a obtener una lista de longitud 4 de vectores de dimensión 3 en el rango (0,1) y la mostramos después:

```
l <- simula_unif(4,3,c(0,1))
cat("Lista de longitud 4 y dimensión 3 en el rango (0,1)\n")
```

```
## Lista de longitud 4 y dimensión 3 en el rango (0,1)
```

```
print(l)
```

```
## [[1]]
## [1] 0.5921064 0.5576325 0.4955501
##
## [[2]]
## [1] 0.3277974 0.5416440 0.8203180
##
## [[3]]
## [1] 0.48786829 0.05446795 0.74735277
##
## [[4]]
## [1] 0.4940525 0.5267060 0.2230311
```

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

Utilizo el mismo procedimiento del apartado anterior, utilizando `lapply()` sobre un vector de la longitud pedida pero utilizando en este caso la función `rnorm()` para que genere los números en la distribución normal (que es la gaussiana) que acepta como parámetros el número de números que tiene que generar, la media y la desviación típica. Como nosotros le estamos pasando el vector de varianzas lo único que tenemos que hacer es pasarle a `rnorm()` la raíz cuadrada de dicho vector.

```
simula_gauss <- function(N, dim, sigma) {
  lapply(1:N, function(x) rnorm(dim, mean = 0, sqrt(sigma)))
}
```

Vamos a obtener por ejemplo una lista de longitud 3 de vectores de dimensión 2 y vector de varianzas (1,7):

```
s <- simula_gauss(3,2,c(1,7))
cat("Lista de longitud 3 y dimensión 2 con varianza (1,7)\n")
```

```
## Lista de longitud 3 y dimensión 2 con varianza (1,7)
```

```
print(s)
```

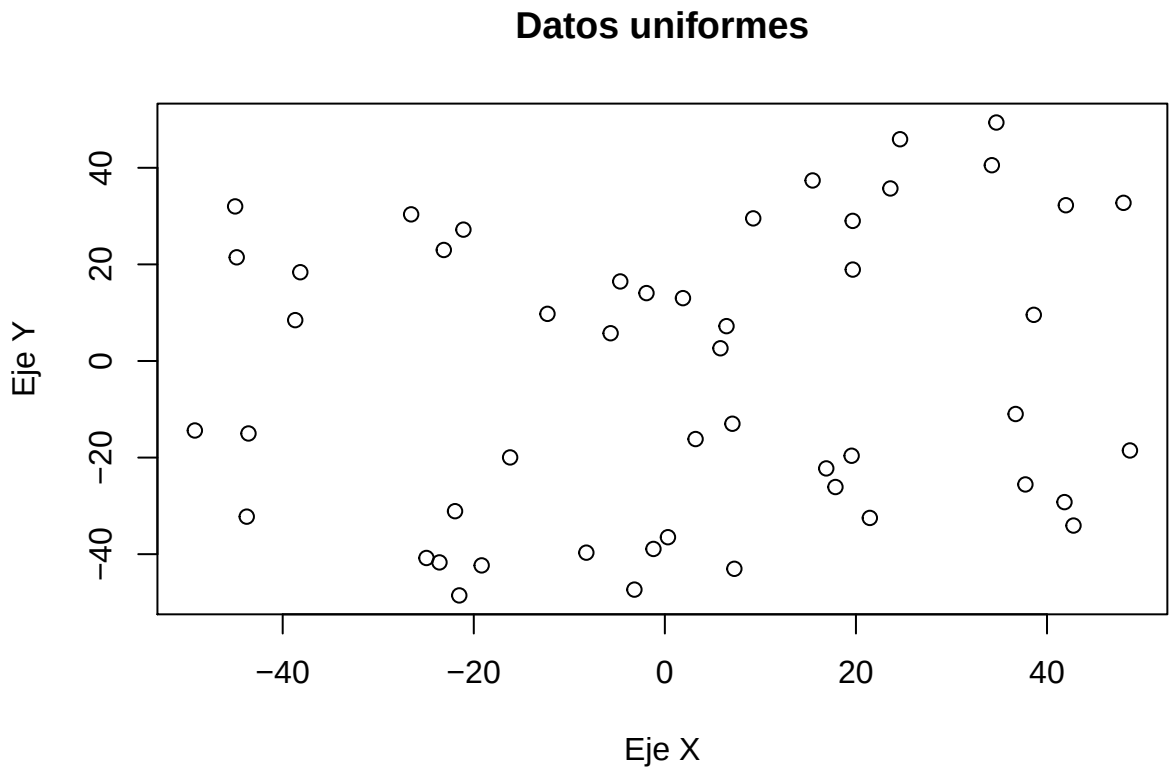
```
## [[1]]
## [1] -0.8273568 -1.2751785
##
## [[2]]
## [1] -0.6827164 -5.8140902
##
## [[3]]
## [1] -1.094434 -1.445302
```

Como vemos, la varianza 1 se coge para la primera coordenada de los vectores y la varianza 7 para la segunda coordenada. Si ponemos por ejemplo c(1,7,3) como vector de varianzas y la dimensión de los vectores es 2, ignora el número 3.

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

Como nos dicen el rango, lo hacemos con la generación uniforme, por lo que le pasamos a la función `simula_unif()` que acabamos de hacer los tres datos que nos dan y obtenemos las coordenadas x e y de los datos (que serán la primera y segunda columna respectivamente) con la función `rapply()`, a la que le pasamos como función que devuelva la primera y la segunda coordenada de cada elemento de la lista. De esta forma obtenemos dos vectores, uno con todas las primeras coordenadas y otro con las segundas. Estos dos vectores es lo que le pasamos a la función `plot()` para que los pinte, junto con el tipo de punto que queremos (en este caso un círculo alrededor del punto) y los títulos de los ejes, con lo que obtenemos una gráfica en la que aparecen los puntos entre -50 y 50 en cada eje, como nos pedían.

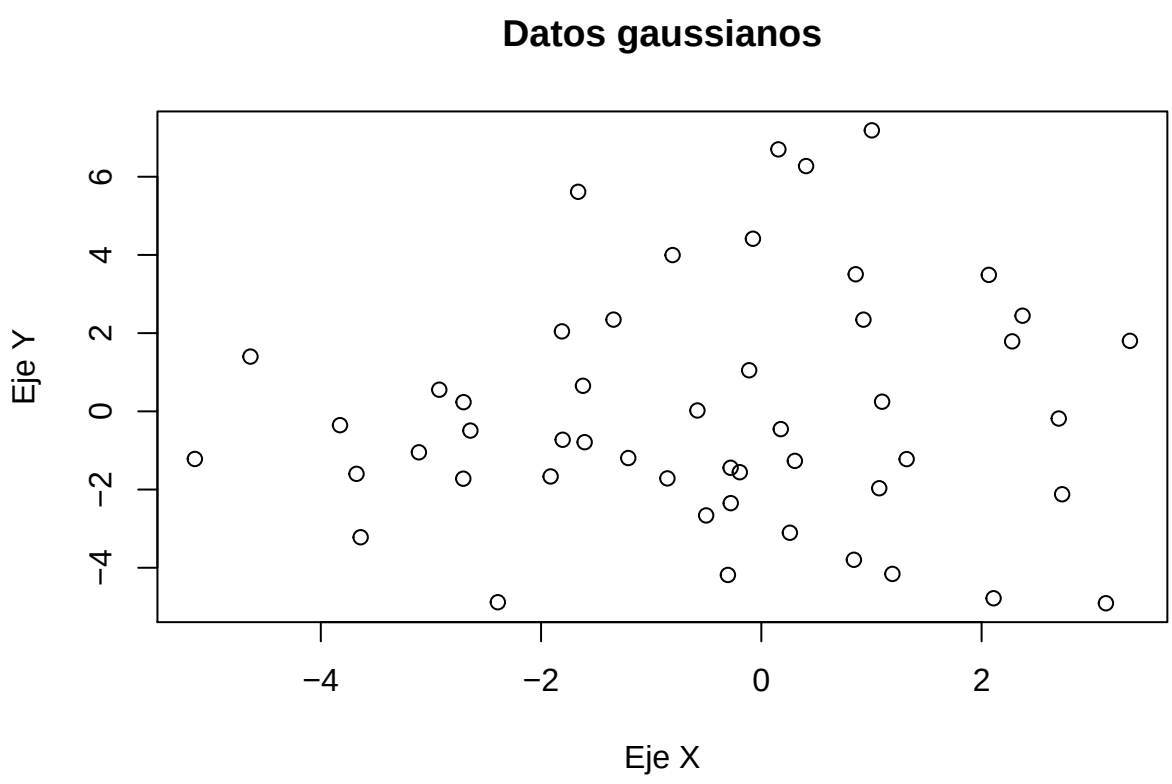
```r
lista <- simula_unif(50, 2, c(-50,50))
# Obtenemos las coordenadas x e y, que son la primera y segunda columna
# de la lista (primera componente y segunda componente de cada vector en la lista)
x <- rapply(lista, function(x) x[1])
y <- rapply(lista, function(x) x[2])
plot(x, y, type = "p", xlab = "Eje X", ylab = "Eje Y", main="Datos uniformes")
```



4. Suponer `N = 50, dim = 2, sigma = [5,7]`. Dibujar una gráfica de la salida de la función correspondiente.

En este caso utilizamos la función `simula_gauss()` que acabamos de crear y utilizamos `rapply()` de nuevo para obtener las coordenadas x e y en vectores, que serán los que le pasemos a `plot()`, obteniendo de nuevo una gráfica, esta vez con una varianza de 5 en el eje X y una varianza de 7 en el eje Y.

```r
lista2 <- simula_gauss(50, 2, c(5,7))
# Obtenemos las coordenadas x e y, que son la primera y segunda columna
# de la lista (primera componente y segunda componente de cada vector en la lista)
x <- rapply(lista2, function(x) x[1])
y <- rapply(lista2, function(x) x[2])
plot(x, y, type = "p", xlab = "Eje X", ylab = "Eje Y", main="Datos gaussianos")
```



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

Le pasamos a la función como parámetro el cuadrado que queremos que la recta corte, que será el intervalo que le pasaremos a la función `simula_unif()`. Una vez hecho esto calculamos a y b como la recta que pasa por los puntos que hemos generado con `simula_unif()`: si (x_1, y_1) y (x_2, y_2) son los puntos por los que pasa la recta, entonces $a = \frac{y_2 - y_1}{x_2 - x_1}$ la pendiente de la recta y con a despejamos b con el primer punto $b = y_1 - ax_1$. Por último devolvemos un vector con a y b.

```
simula_recta <- function(intervalo) {
  l <- simula_unif(2, 2, intervalo)
  a <- (l[[2]][2] - l[[1]][2]) / (l[[2]][1] - l[[1]][1])
  b <- l[[1]][2] - a * l[[1]][1]
  c(a,b)
}
```

Simulamos la recta pedida que corta al cuadrado $[-50, 50] \times [-50, 50]$

```
cat("Coeficientes de la recta que corta al cuadrado (-50,50)x(-50,50)\n")
```

```
## Coeficientes de la recta que corta al cuadrado (-50,50)x(-50,50)
```

```
print(simula_recta(c(-50,50)))
```

```
## [1] -1.427238 -3.351577
```

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

Vamos primero a construir una función que nos devuelva la etiqueta +1 o -1 según el signo de la f dada (suponiendo que la evaluación ya ha sido hecha):

```
etiquetar <- function(valor) {
  if (valor > 0) {
    return(+1)
  }
  else {
    return(-1)
  }
}
```

Ahora generamos la recta con la función indicada y utilizamos la función anterior para etiquetar los puntos haciendo uso también de la función `lapply()`, iterando sobre un vector del tamaño de la lista que serán las posiciones, de forma que en cada iteración nos quedamos con el elemento i de la lista (el punto i-ésimo) y para cada punto evaluamos en la recta dada y etiquetamos con la función anterior:

```
r <- simula_recta(c(-50,50))
etiquetas <- lapply(1:length(lista), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- lista[[i]]
  f <- p[2] - r[1]*p[1] - r[2]
  etiquetar(f)
})
```

Esto nos devuelve en `etiquetas` una lista de 50 vectores, todos con una componente. Lo vamos a pasar a un vector de 50 componentes con la función `unlist()`:

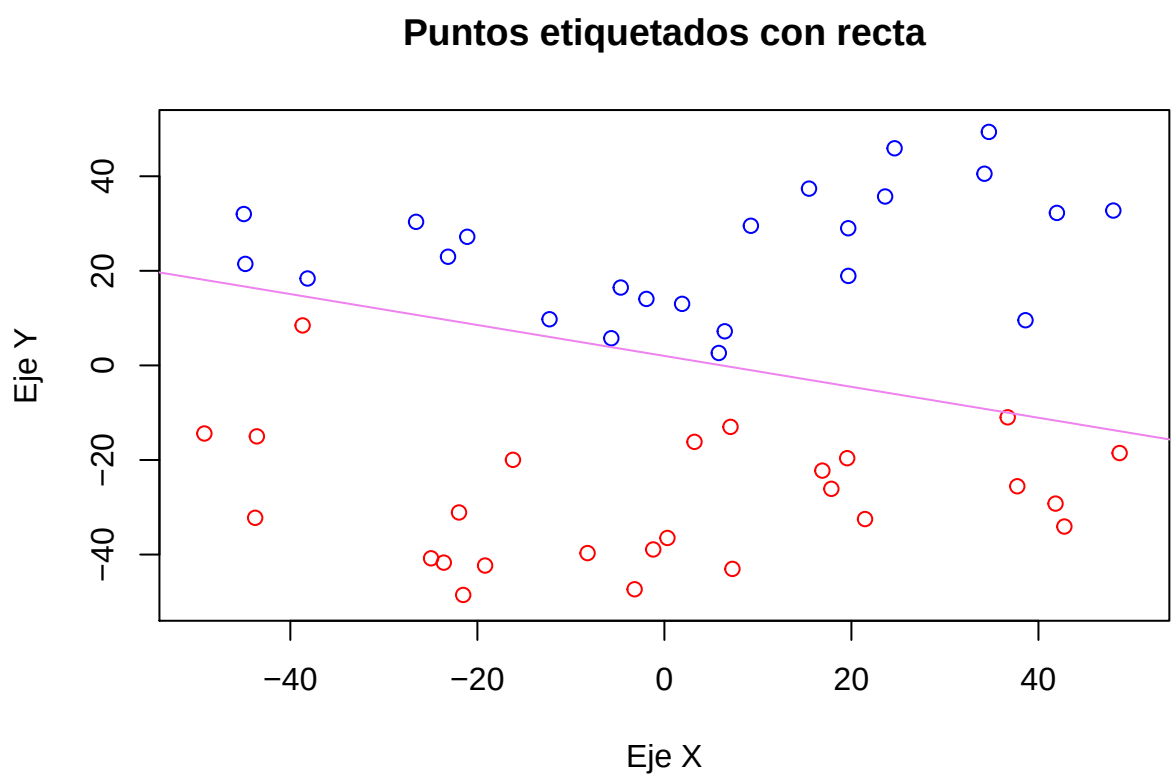
```
etiquetas <- unlist(etiquetas)
```

Vamos a dibujar el resultado. Pintamos en una misma gráfica los puntos etiquetados con +1 (en rojo) y los puntos etiquetados con -1 (en azul), esto lo hacemos utilizando el mismo vector de etiquetas que ya teníamos, sumándole una cantidad positiva (puesto que los colores empiezan en 1 y en nuestro vector tenemos almecenados muchos -1). Pintamos también la recta que hemos utilizado para etiquetar los puntos utilizando la función `contour()`, que junto con `outer()` permite dibujar una gráfica dando la expresión implícita de la misma (lo que nos va a venir muy bien para el apartado 7 de este ejericio). Vamos también a dar la opción de no pintar gráfica junto con los puntos para que sea más general la función y nos sirva para más cosas, utilizando los valores por defecto:

```
pinta_particion <- function(lista_puntos, etiquetas=NULL, visible=FALSE, f=NULL,
                            rango = c(-50,50), main="") {
  x <- rapply(lista_puntos, function(x) x[1])
  y <- rapply(lista_puntos, function(x) x[2])
  if(is.null(etiquetas))
     etiquetas=1
  else etiquetas = etiquetas+3
  plot(x, y, type = "p", col = etiquetas, xlab = "Eje X", ylab = "Eje Y",
       xlim = rango, ylim = rango, main = main)

  # Si queremos pintar función junto con los datos
  if(visible) {
    sec <- seq(-60, 60, length.out = 1500)
    z <- outer(sec, sec, f)
    contour(sec, sec, z, col = "violet", levels = 0, add = TRUE, drawlabels = F)
  }
}
```

Utilizamos la función para pintar la recta junto con los puntos:

```
pinta_particion(lista, etiquetas, TRUE, function(x,y) r[1]*x-y+r[2],
                main="Puntos etiquetados con recta")
```



7. Usar la muestra generada en el apartado anterior y etiquetarla con +1,-1 usando el signo de cada una de las siguiente funciones y visualizar el resultado del etiquetado de cada función junto con su gráfica y comparar el resultado con el caso lineal. ¿Qué consecuencias extrae sobre la forma de las regiones positiva y negativa?:

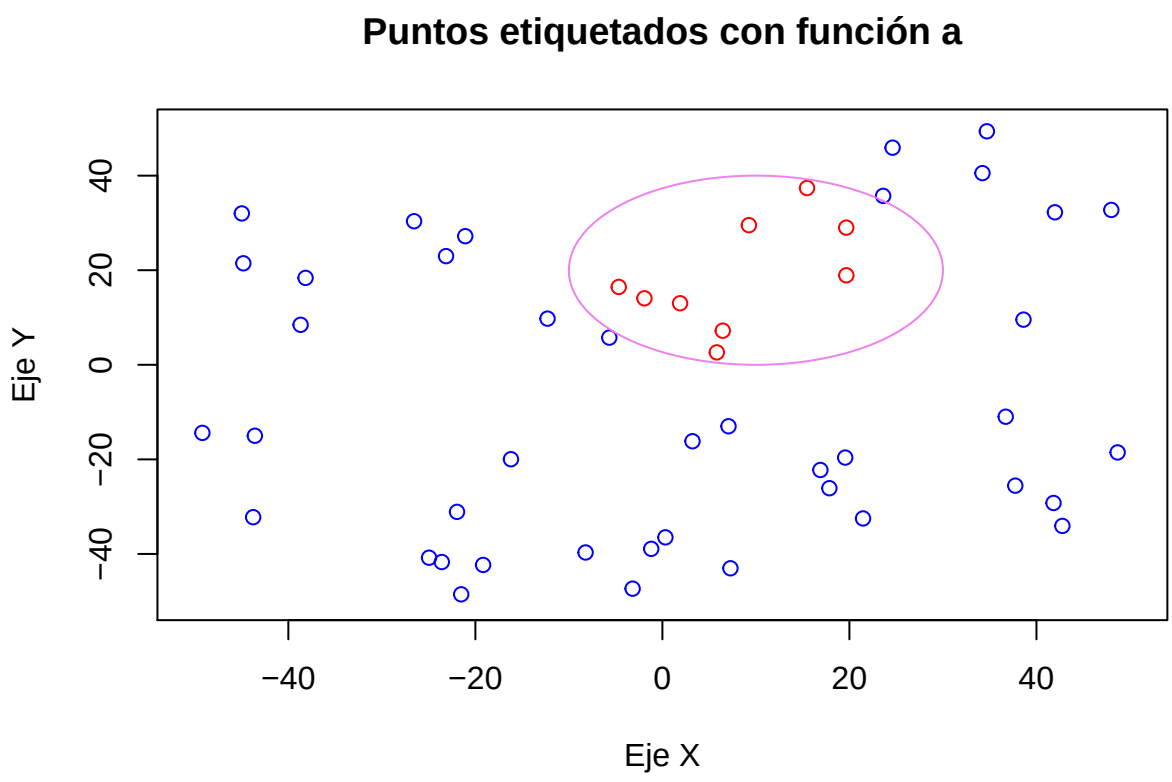
Vamos a utilizar la función para etiquetar que hemos hecho en el apartado anterior y la función para dibujar que acabamos de hacer también. En todos los subapartados hacemos lo mismo: etiquetamos los datos según la función del mismo modo que antes y se los pasamos a la función anterior para que los pinte.

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

```
cat("Apartado 7\n")
```

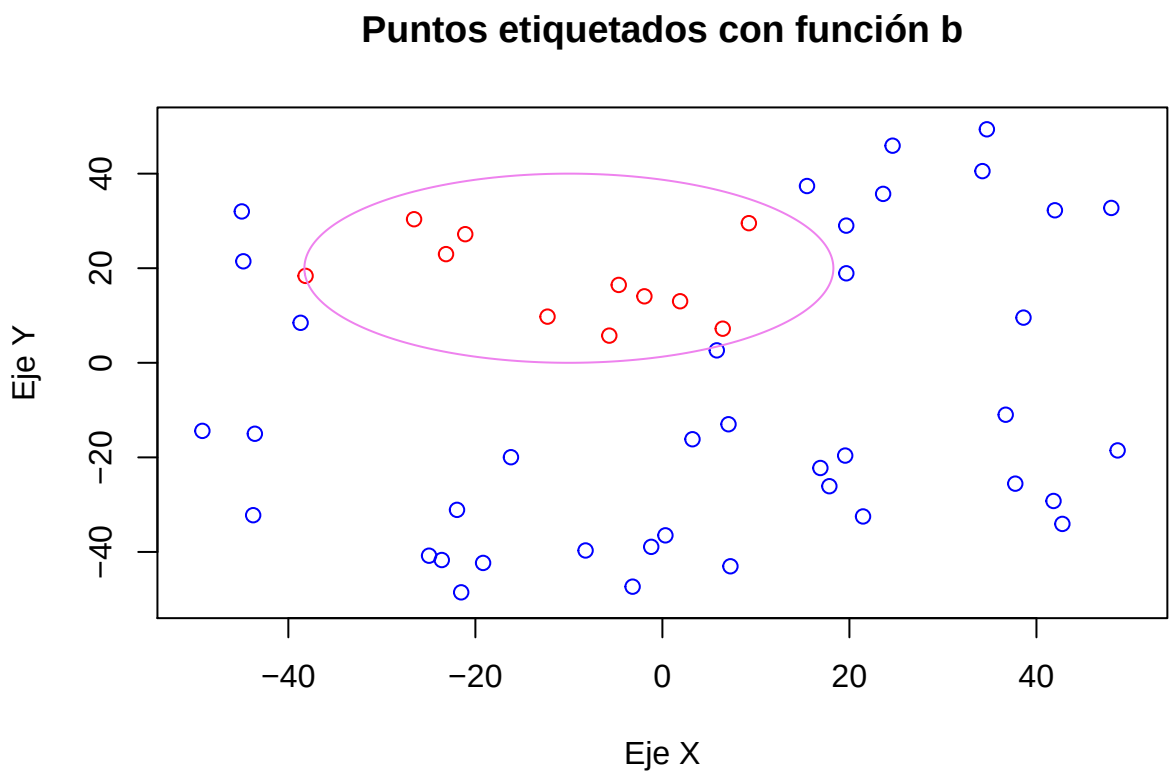
```
## Apartado 7
```

```
etiquetasFA <- lapply(1:length(lista), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- lista[[i]]
  f1 <- (p[1]-10)^2 + (p[2] - 20)^2 - 400
  etiquetar(f1)
})
etiquetasFA <- unlist(etiquetasFA)

pinta_particion(lista, etiquetasFA, TRUE, function(x,y) (x-10)^2 + (y-20)^2 -
                  400, main="Puntos etiquetados con función a")
```



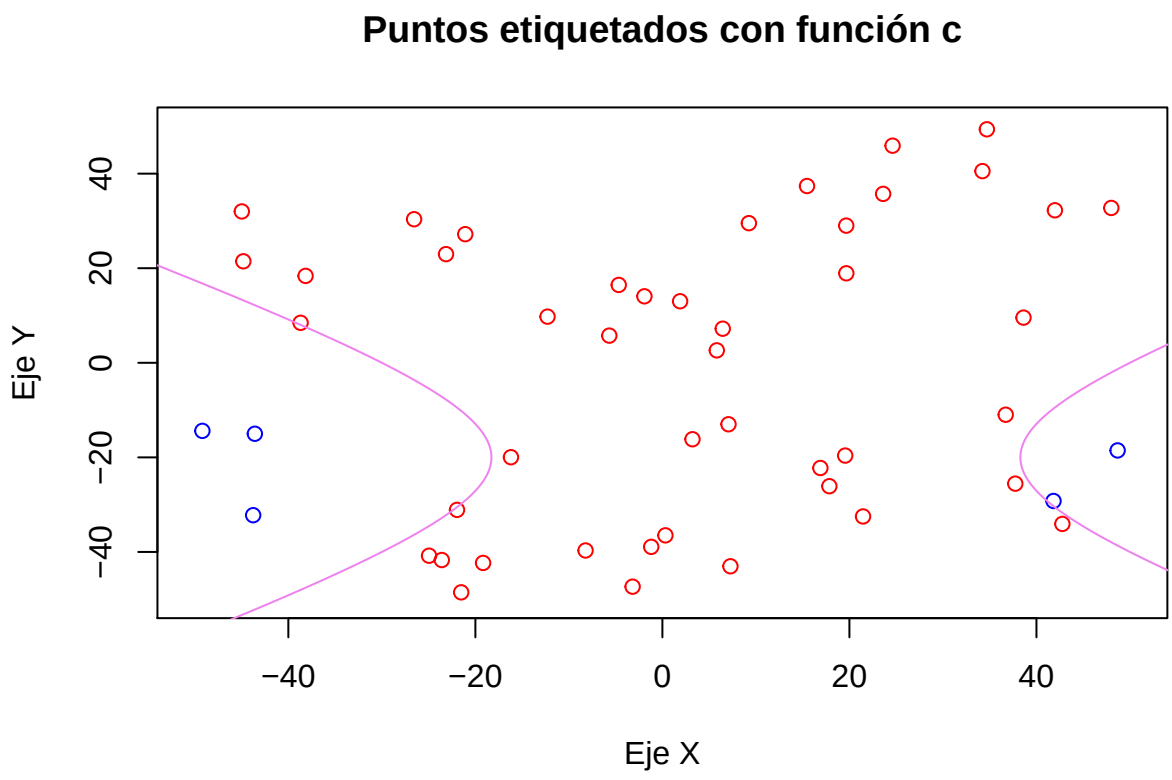
b) $f(x, y) = 0.5 * (x + 10)^2 + (y - 20)^2 - 400$

```
etiquetasFB <- lapply(1:length(lista), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- lista[[i]]
  f2 <- 0.5*(p[1]+10)^2 + (p[2] - 20)^2 - 400
  etiquetar(f2)
})
etiquetasFB <- unlist(etiquetasFB)

pinta_particion(lista, etiquetasFB, TRUE, function(x,y) 0.5*(x+10)^2 + (y-20)^2
                - 400, main="Puntos etiquetados con función b")
```



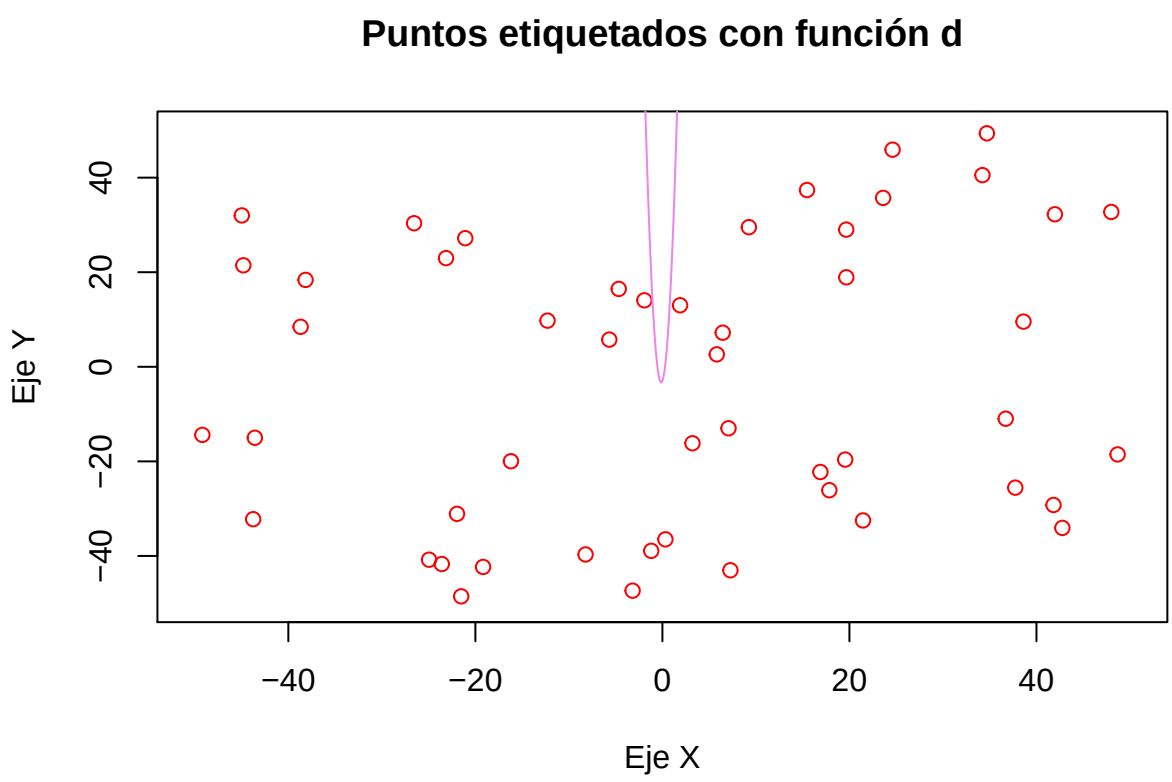
c) $f(x, y) = 0.5 * (x - 10)^2 - (y + 20)^2 - 400$

```
etiquetasFC <- lapply(1:length(lista), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- lista[[i]]
  f3 <- 0.5*(p[1]-10)^2 - (p[2] + 20)^2 - 400
  etiquetar(f3)
})
etiquetasFC <- unlist(etiquetasFC)

pinta_particion(lista, etiquetasFC, TRUE, function(x,y) 0.5*(x-10)^2 - (y+20)^2
                - 400, main="Puntos etiquetados con función c")
```



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

```
etiquetasFD <- lapply(1:length(lista), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- lista[[i]]
  f4 <- p[2] - 20*(p[1]^2) - 5*p[1] + 3
  etiquetar(f4)
})
etiquetasFD <- unlist(etiquetasFD)
pinta_particion(lista, etiquetasFD, TRUE, function(x,y) y - 20*x^2 - 5*x + 3,
                main="Puntos etiquetados con función d")
```



Como vemos, al ser ahora las funciones cuadráticas y no lineales, los datos no son linealmente separables (los que caen dentro de las gráficas se quedan en el centro o a ambos lados) y por tanto el perceptron no será capaz de separar totalmente estas clasificaciones.

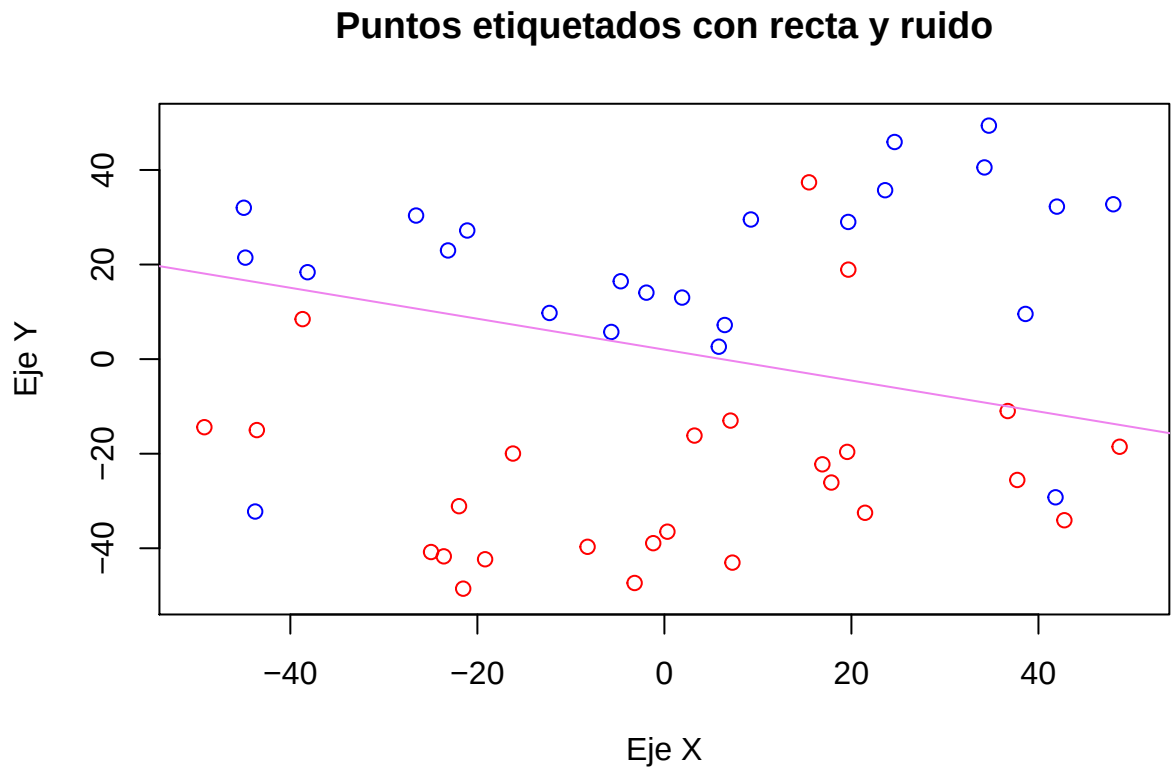
8. Considerar de nuevo la muestra etiquetada en el apartado 6. Modifique las etiquetas de un 10% aleatorio de muestras positivas y otro 10% de muestras negativas.

Visualice los puntos con las nuevas etiquetas y la recta del apartado 6.

Primero hacemos una función que cambia el 10% de etiquetas de cada parte, para ello nos creamos un vector con las posiciones en las que hay etiquetas positivas y otro con las posiciones en las que hay etiquetas negativas y utilizamos la función `sample()` para elegir de cada uno de estos dos vectores el 10%, que serán las posiciones que cambiaremos:

```
cambiar_etiquetas <- function(etiquetas) {
  num <- 1:length(etiquetas)
  etiquetas_cambiadas <- etiquetas
  # Cogemos las posiciones de las etiquetas positivas y negativas
  positivos <- num[etiquetas > 0]
  negativos <- num[etiquetas < 0]
  # Comprobamos que hay algún elemento que cambiar y obtenemos el 10%
  # de posiciones aleatorias
  if(length(positivos)*0.1 > 0) {
    cambiar1 <- sample(positivos, length(positivos)*0.1)
```

```
    # Cambiamos las etiquetas que hemos obtenido antes
    etiquetas_cambiadas[cambiar1] <- -1
  }
  if(length(negativos)*0.1 > 0) {
    cambiar2 <- sample(negativos, length(negativos)*0.1)
    # Cambiamos las etiquetas que hemos obtenido antes
    etiquetas_cambiadas[cambiar2] <- +1
  }
  # Devolvemos las etiquetas cambiadas
  etiquetas_cambiadas
}

# Llamamos a esta función para cambiar las etiquetas y pintamos los datos con
# estas nuevas etiquetas
etiquetas2 <- cambiar_etiquetas(etiquetas)
pinta_particion(lista, etiquetas2, TRUE, function(x,y) r[1]*x-y+r[2],
                main="Puntos etiquetados con recta y ruido")
```

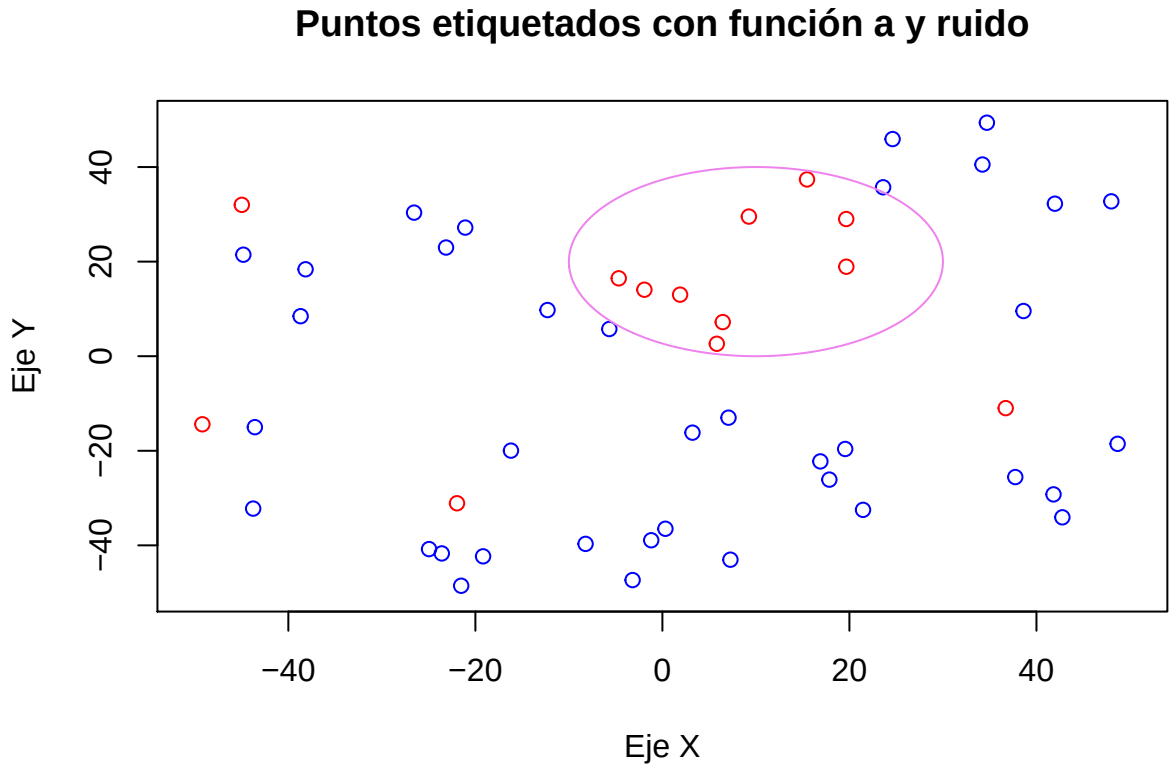


Podemos ver como dentro de la región de puntos azules hay dos rojos y dentro de la región roja hay dos azules, haciendo los datos no linealmente separables.

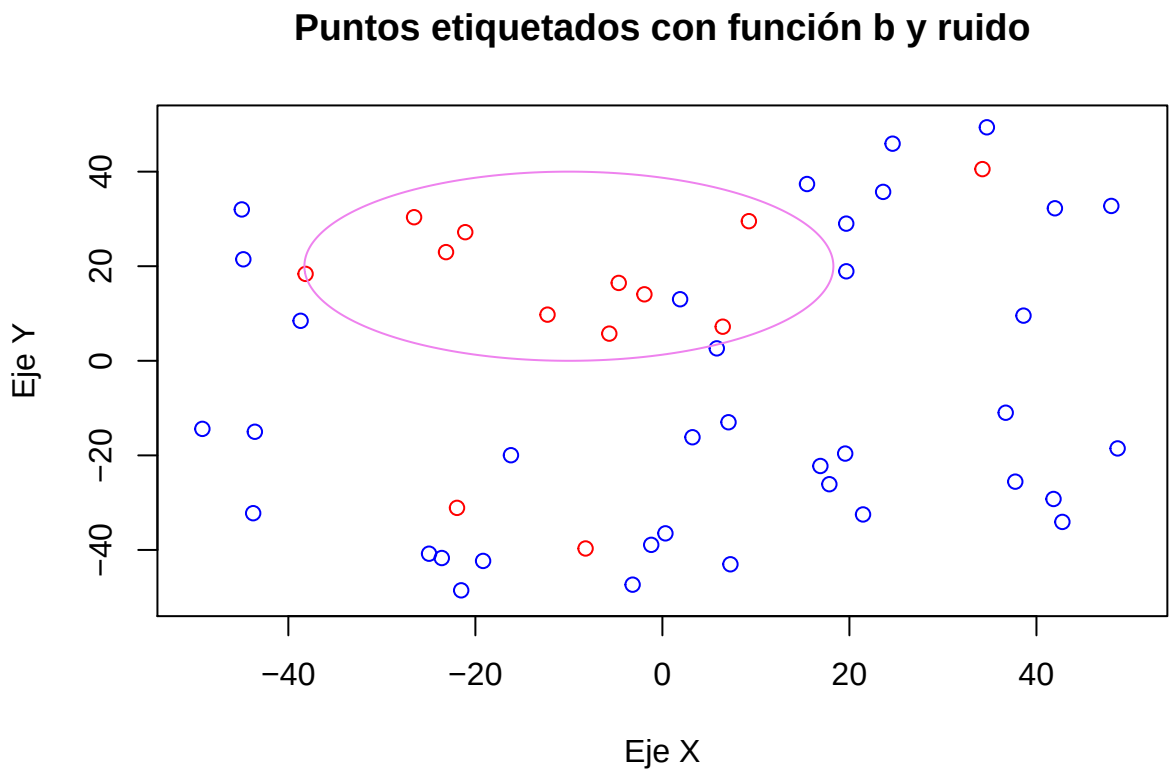
En una gráfica aparte visualice de nuevo los mismos puntos pero junto con las funciones del apartado 7.

Lo único que tenemos que hacer es llamar a la función para pintar cambiando las etiquetas y junto con las funciones del apartado 7.

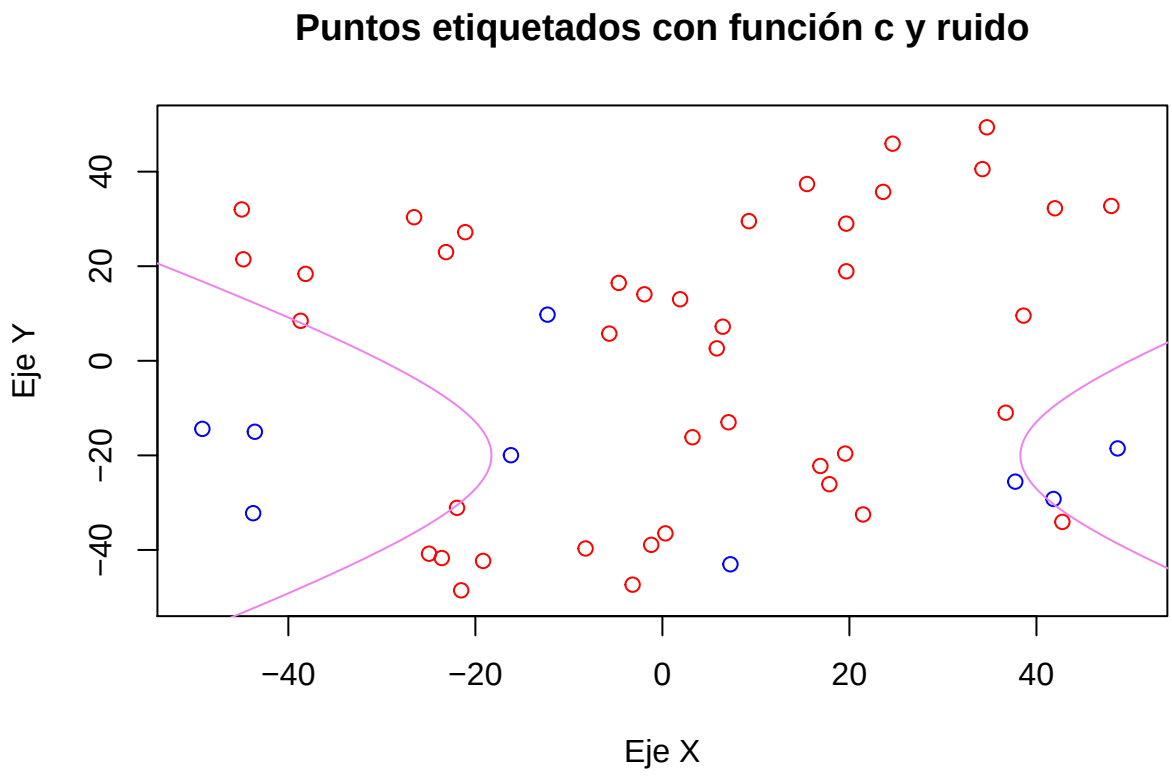
```
pinta_particion(lista, cambiar_etiquetas(etiquetasFA), TRUE,
                function(x,y) (x-10)^2 + (y-20)^2 - 400,
                main="Puntos etiquetados con función a y ruido")
```



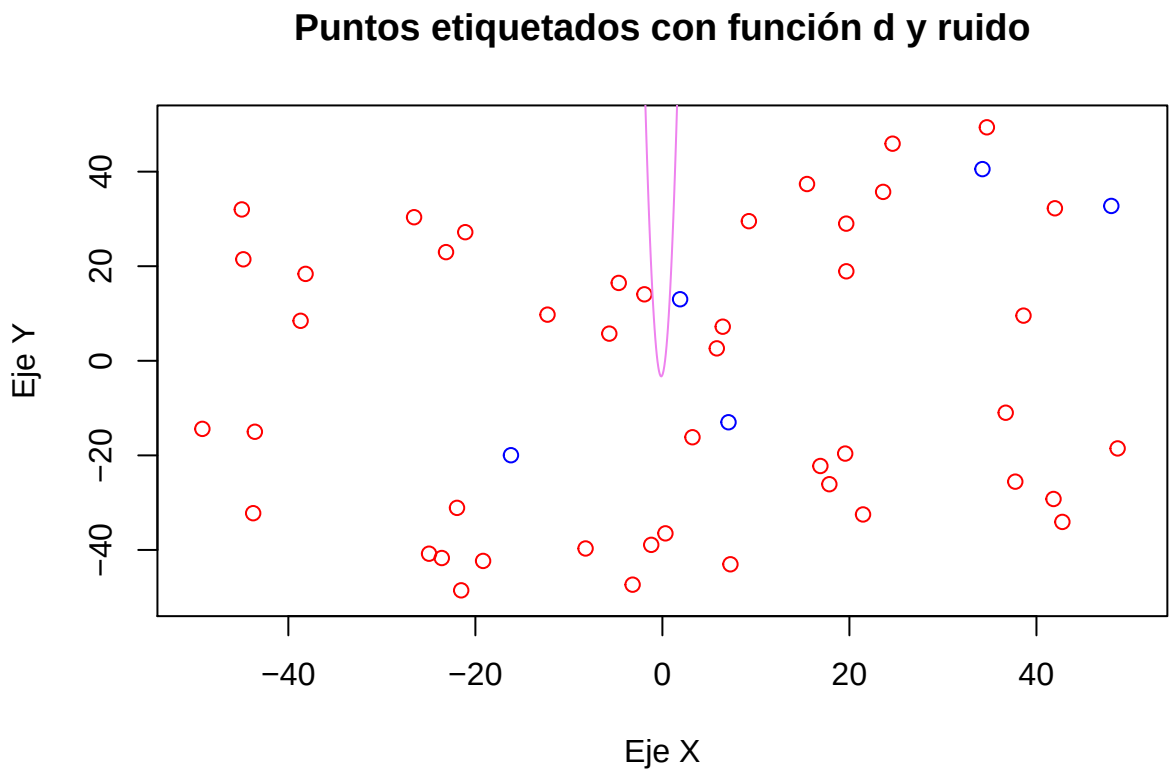

```
pinta_particion(lista, cambiar_etiquetas(etiquetasFB), TRUE,
                function(x,y) 0.5*(x+10)^2 + (y-20)^2 - 400,
                main="Puntos etiquetados con función b y ruido")
```




```
pinta_particion(lista, cambiar_etiquetas(etiquetasFC), TRUE,
                function(x,y) 0.5*(x-10)^2 - (y+20)^2 - 400,
                main="Puntos etiquetados con función c y ruido")
```




```
pinta_particion(lista, cambiar_etiquetas(etiquetasFD), TRUE,
                function(x,y) y - 20*x^2 - 5*x + 3,
                main="Puntos etiquetados con función d y ruido")
```



2. Ejercicio de Ajuste del Algoritmo Perceptron

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

Aunque pide que la salida sean los coeficientes del hiperplano, en el segundo apartado pide también el número de iteraciones que han sido necesarias para converger, por lo que vamos a devolver una lista cuya primera componente tenga ω, los coeficientes del hiperplano, y la segunda componente sea el número de iteraciones necesario para converger, es decir, la iteración en la que se para.

Lo que hacemos es seguir el algoritmo dado en clase de teoría: iteramos sobre todos los datos mientras no se superen el máximo de iteraciones que le pasamos como parámetro o mientras no se haya encontrado una solución ya. Se considera que se ha encontrado una solución cuando se pasa una vez por todos los datos y no ha habido cambios (es decir, todos estaban correctamente clasificados), que es algo que controlamos con dos variables booleanas. Sabemos si los datos están bien clasificados porque nos quedamos con el signo del producto escalar entre el ω actual y cada punto. En caso de que el signo no coincida con la etiqueta para ese punto, movemos la recta sumándole el valor real de la etiqueta en ese punto por el mismo punto.

```r
ajusta_PLA <- function(datos, label, max_iter, vini) {
  parada <- F
  fin <- F
  w <- vini
  iter <- 1
  # Mientras no hayamos superado el máximo de iteraciones o
  # no se haya encontrado solución
  while(!parada) {
    # iteramos sobre los datos
    for (j in 1:nrow(datos)) {
      if (sign(crossprod(w, datos[j,])) != label[j]) {
        w <- w + label[j]*datos[j,]
        # La variable fin controla si se ha entrado en el if
        fin <- F
      }
    }
    # Si no se ha entrado en el if, todos los datos estaban bien
    # clasificados y podemos poner a TRUE la variable parada.
    if(fin == T) {
      parada = T
    }
    else {
      fin = T
    }
    iter <- iter + 1
    if (iter >= max_iter) parada = T
  }
  # Devolvemos el hiperplano y el número máximo de iteraciones al que hemos
  # llegado.
  list(w, iter)
}
```

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

El algoritmo anterior opera con los datos dentro de una matriz. Lo que hacemos es pasar los datos, que teníamos en una lista, a una matriz con la función `matrix()`, a la que le pasamos los datos, el número de filas y columnas y si queremos que meta los datos por filas, que en nuestro caso sí queremos, puesto que los tenemos almacenados así. Además, le añadimos a cada punto una tercera coordenada a 1 para pasarlos a coordenadas homogéneas, lo que se traduce en una tercera columna en la matriz toda puesta a 1's.

A esta matriz es a la que le hacemos el PLA y nos quedamos con la solución ω y con el número de iteraciones que necesita para converjer, y pintamos los puntos junto con la solución dada por el algoritmo. El PLA devuelve en ω en la última componente el término independiente (en este caso b). Como lo que tenemos es una recta, vamos a dejar con coeficiente a 1 la y, que será la segunda componente de ω y además la vamos a despejar, por lo que tenemos que dividir la solución del PLA entre menos la segunda componente de la solución.

```r
# Metemos los datos, que teníamos en una lista llamada "lista" en
# una matriz.
m <- matrix(unlist(lista), 50, 2, byrow=TRUE)
```

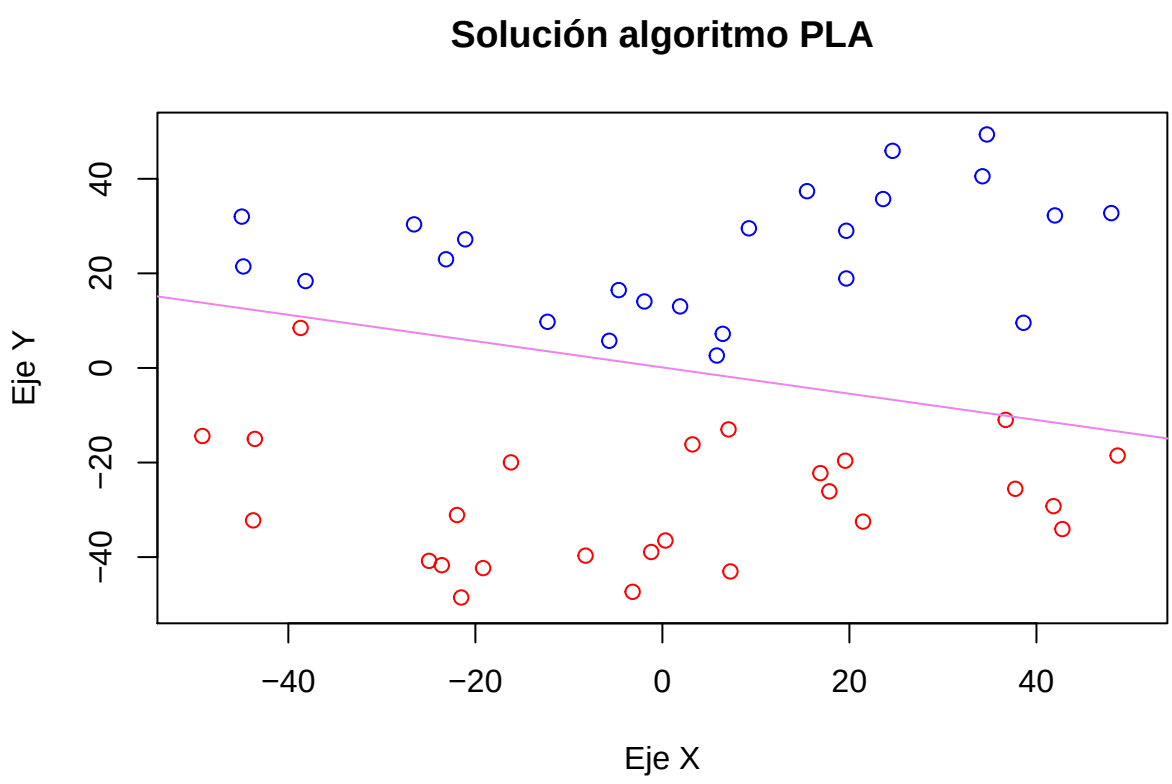
```r
datos <- matrix(1, 50, 3)
datos[1:50, 1:2] <- m
sol <- ajusta_PLA(datos, etiquetas, 20, c(0,0,0))
iter1 <- sol[[2]]
cat("El número de iteraciones necesarias para converger en PLA ha sido:\n")
```

```
## El número de iteraciones necesarias para converger en PLA ha sido:
```

```r
print(iter1)
```

```
## [1] 11
```

```r
w <- sol[[1]]
w <- -w / w[2]
pinta_particion(lista, etiquetas, TRUE, function(x,y) y-w[3]-w[1]*x,
                main="Solución algoritmo PLA")
```



En este caso el número de iteraciones necesario para que el algoritmo converja es 11. Vamos a hacerlo ahora generando números aleatorios entre 0 y 1, 10 veces. Lo que he hecho ha sido meter en una lista gracias a la función `lapply` las iteraciones que hacen falta en cada una de las 10 veces que vamos a llamar el algoritmo y hacer la media de todos ellos con la función `mean()`.

```r
# Metemos los 10 vectores iniciales aleatorios en una lista
waleatorios <- simula_unif(10, 3, c(0,1))
iteraciones <- lapply(1:10, function(i) {
  wi <- waleatorios[[i]]
  sol <- ajusta_PLA(datos, etiquetas, 200, wi)
  sol[[2]]
})

iteraciones <- unlist(iteraciones)
cat("La media de las iteraciones para que el PLA converja ha sido: \n")
```

```
## La media de las iteraciones para que el PLA converja ha sido:
```

```r
print(mean(iteraciones))
```

```
## [1] 10.9
```

Generando números aleatorios entre 0 y 1 para el w inicial, el número medio de iteraciones que son necesarias para que el algoritmo converja es 10.9. Hay que tener en cuenta que las iteraciones que necesita para converjer dependen de los datos (en mi caso, de que he fijado la semilla a 237). Con otros datos podría ser mucho más o incluso menos. En mi caso se justifica que tarde tan poco porque el vector inicial son números aleatorios entre 0 y 1 (o en el caso de antes directamente el (0,0,0)) y los datos son separables por una recta que está próxima al (0,0,0), lo que hace que sea rápido para el PLA separarlos.

3. Ejecutar el algoritmo PLA con los datos generados en el apartado 8 del ejercicio 1, usando valores de 10, 100 y 1000 para `max_iter`. Etiquetar los datos de la muestra usando la función solución encontrada y contar el número de errores respecto de las etiquetas originales. Valorar el resultado.

Vamos a hacer primero una función que cuente las diferencias entre dos vectores de etiquetas, es decir, la cantidad de posiciones en los que dos vectores de etiquetas tienen valores distintos.

```r
cuenta_diferencias <- function(etiquetas1, etiquetas2) {
  vf <- etiquetas1 == etiquetas2
  length(vf[vf == FALSE])
}
```

Vamos a hacer ahora una función que nos devuelva el número de errores respecto de las etiquetas originales, utilizando la función anterior. Esta función recibe como parámetro la lista que devuelve como solución el PLA y el vector de etiquetas originales:

```r
cuenta_errores <- function(sol_PLA, etiquetas_originales) {
  w <- sol_PLA[[1]]
  w <- -w / w[2]
  # Recordemos que los datos que hay en la matriz "datos" son los mismos
  # puntos que hay en la matriz "lista"
  etiquetas_cambiadas <- lapply(1:length(lista), function(i) {
    # Obtenemos los puntos uno a uno y los etiquetamos
    p <- lista[[i]]
    f <- -w[1]*p[1] + p[2] - w[3]
```

```
    etiquetar(f)
  })
  # Devolvemos el número de errores que da la solución
  cuenta_diferencias(etiquetas_originales, etiquetas_cambiadas)
}
```

Vamos a ejecutarlo ahora con 10, 100 y 1000 iteraciones y vamos a pintar también la solución que da.

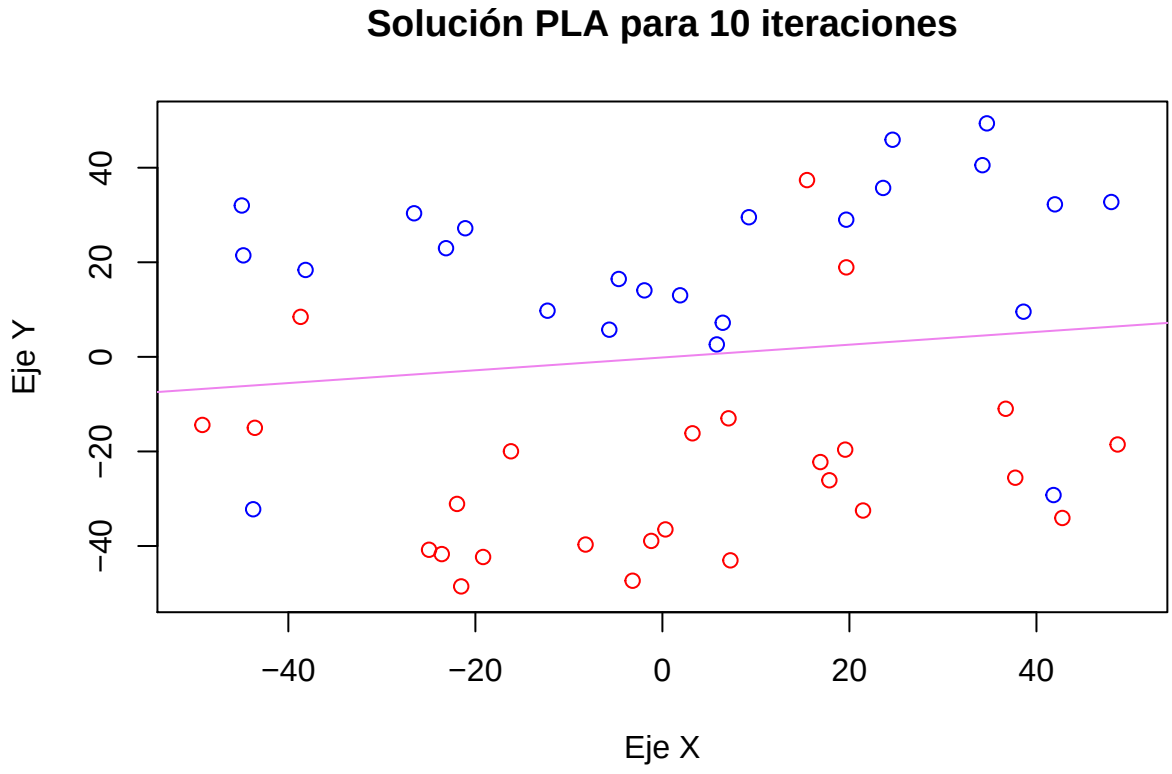
```
sol1 <- ajusta_PLA(datos, etiquetas2, 10, c(0,0,0))
cat("El número de fallos para 10 iteraciones ha sido:\n")
```

```
## El número de fallos para 10 iteraciones ha sido:
```

```
print(cuenta_errores(sol1, etiquetas2))
```

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

```
w1 <- sol1[[1]]
w1 <- -w1/w1[2]
pinta_particion(lista, etiquetas2, TRUE, function(x,y) w1[1]*x-y+w1[3],
                main="Solución PLA para 10 iteraciones")
```



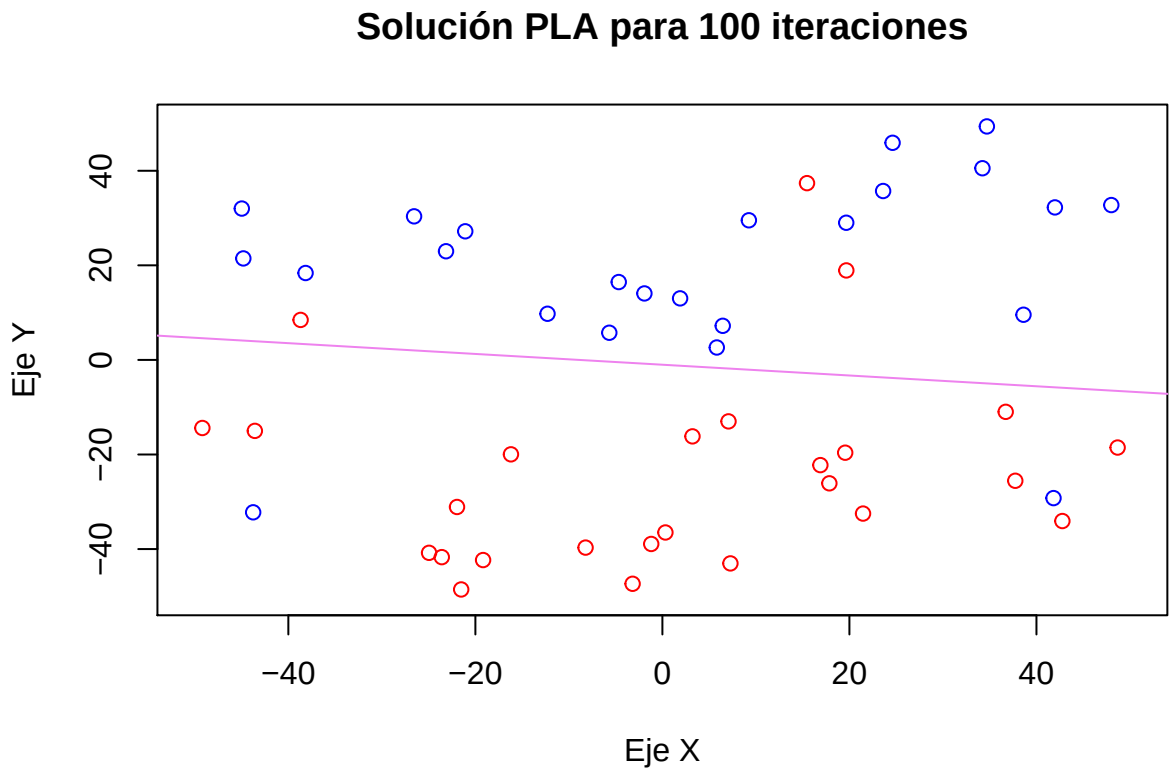
```
sol2 <- ajusta_PLA(datos, etiquetas2, 100, c(0,0,0))
cat("El número de fallos para 100 iteraciones ha sido:\n")
```

```
## El número de fallos para 100 iteraciones ha sido:
```

```
print(cuenta_errores(sol2, etiquetas2))
```

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

```
w2 <- sol2[[1]]
w2 <- -w2/w2[2]
pinta_particion(lista, etiquetas2, TRUE, function(x,y) w2[1]*x-y+w2[3],
                main="Solución PLA para 100 iteraciones")
```



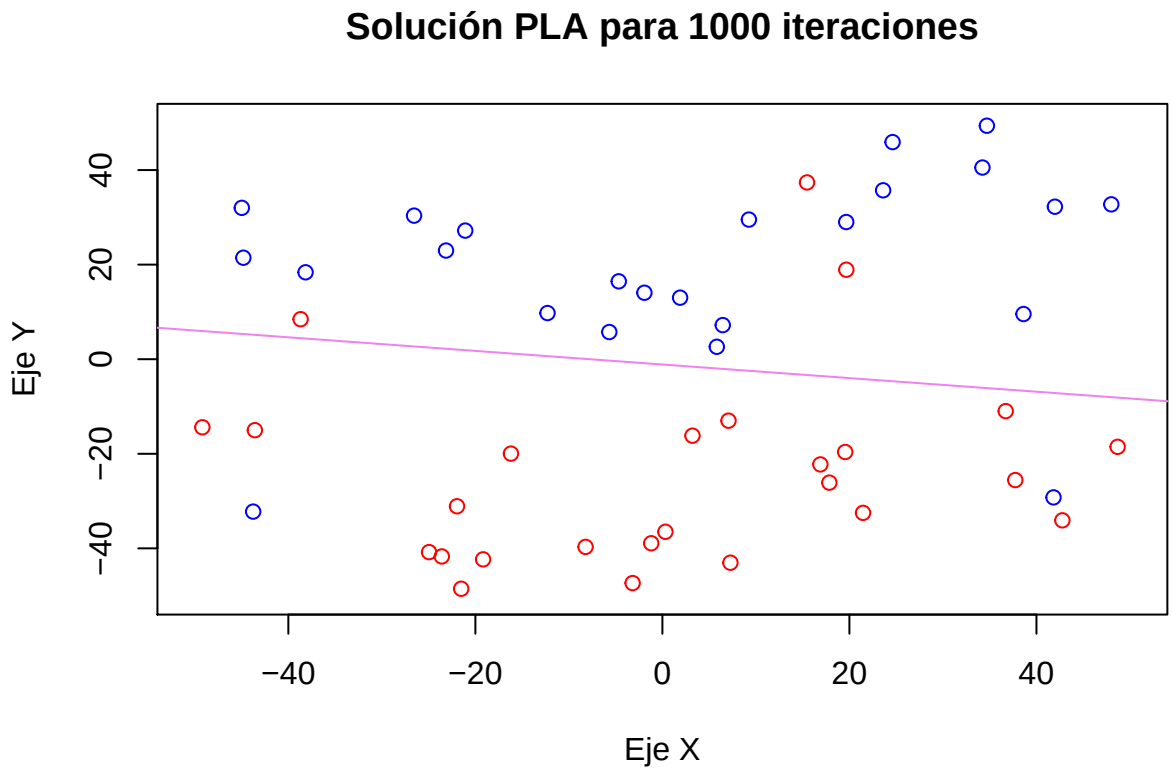

```
sol3 <- ajusta_PLA(datos, etiquetas2, 1000, c(0,0,0))
cat("El número de fallos para 1000 iteraciones ha sido:\n")
```

```
## El número de fallos para 1000 iteraciones ha sido:
```

```
print(cuenta_errores(sol3, etiquetas2))
```

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

```
w3 <- sol3[[1]]
w3 <- -w3/w3[2]
pinta_particion(lista, etiquetas2, TRUE, function(x,y) w3[1]*x-y+w3[3],
                main = "Solución PLA para 1000 iteraciones")
```



Como vemos, en los tres casos el número de puntos mal clasificados es 5. Esto es porque al cambiar etiquetas al azar, los datos han dejado de ser linealmente separables, con lo que el perceptron no puede llegar a una solución en la que todos los puntos estén bien clasificados, da igual el número de iteraciones que le pongamos.

4. Repetir el análisis del punto anterior usando la primera función del apartado 7 del ejercicio 1.

La función es $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$ y tenemos las etiquetas originales guardadas en un vector llamado `etiquetasFA`.

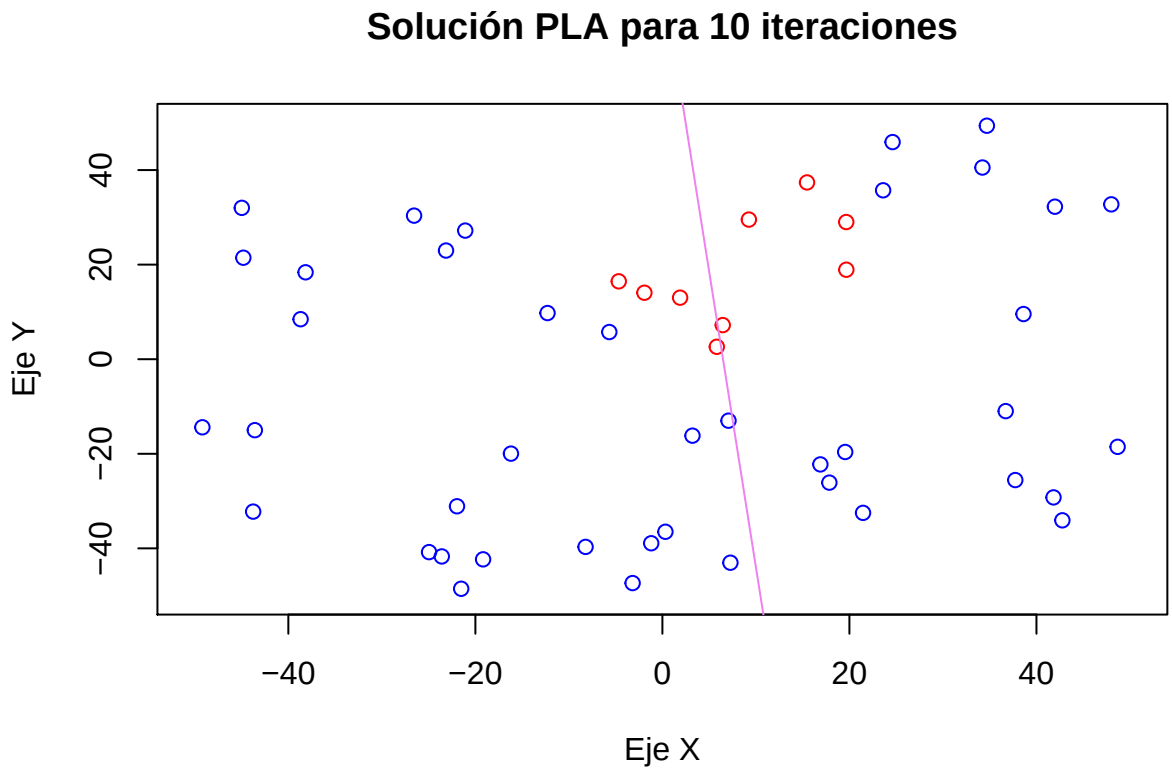
```
sol1 <- ajusta_PLA(datos, etiquetasFA, 10, c(0,0,0))
cat("El número de fallos con la primera función del apartado 7 y 10 iteraciones ha sido:\n")
```

```
## El número de fallos con la primera función del apartado 7 y 10 iteraciones ha sido:
```

```
print(cuenta_errores(sol1, etiquetasFA))
```

```
## [1] 30
```

```
w1 <- sol1[[1]]
w1 <- -w1 / w1[2]
pinta_particion(lista, etiquetasFA, TRUE, function(x,y) w1[1]*x-y+w1[3],
                main="Solución PLA para 10 iteraciones")
```



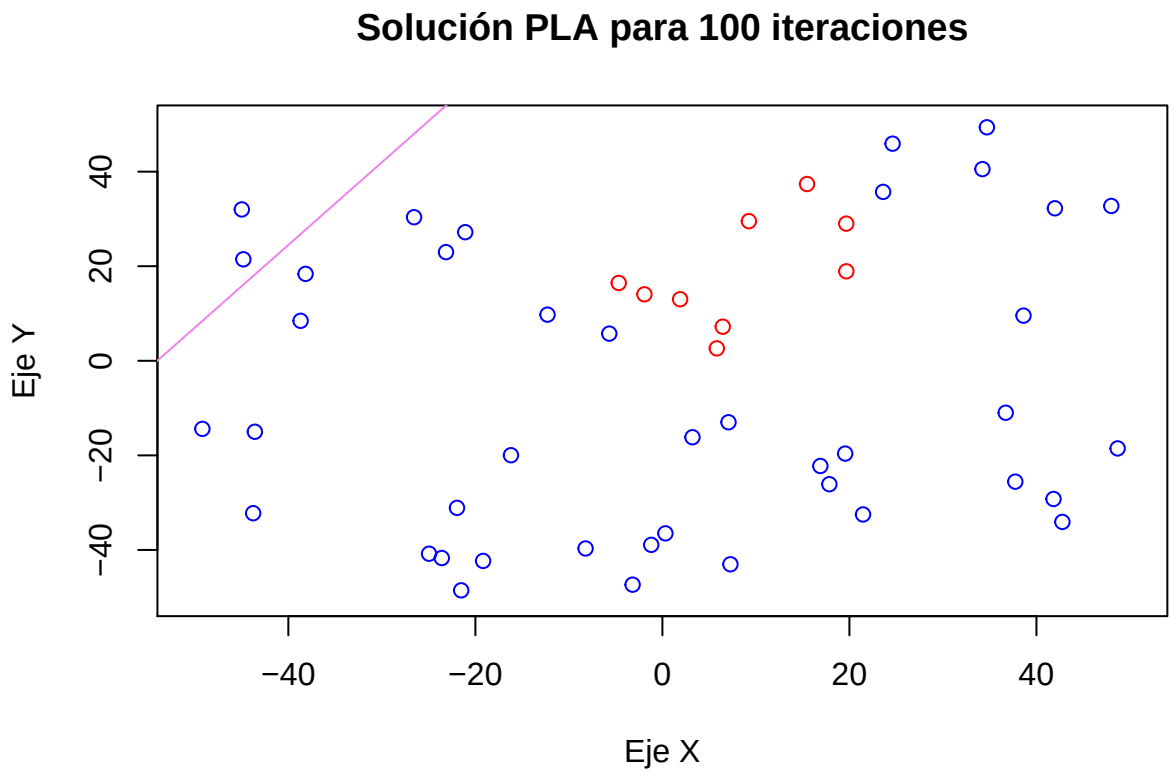

```
sol2 <- ajusta_PLA(datos, etiquetasFA, 100, c(0,0,0))
cat("El número de fallos con la primera función del apartado 7 y 100 iteraciones ha sido:\n")
```

```
## El número de fallos con la primera función del apartado 7 y 100 iteraciones ha sido:
```

```
print(cuenta_errores(sol2, etiquetasFA))
```

```
## [1] 39
```

```
w2 <- sol2[[1]]
w2 <- -w2 / w2[2]
pinta_particion(lista, etiquetasFA, TRUE, function(x,y) w2[1]*x-y+w2[3],
                main="Solución PLA para 100 iteraciones")
```



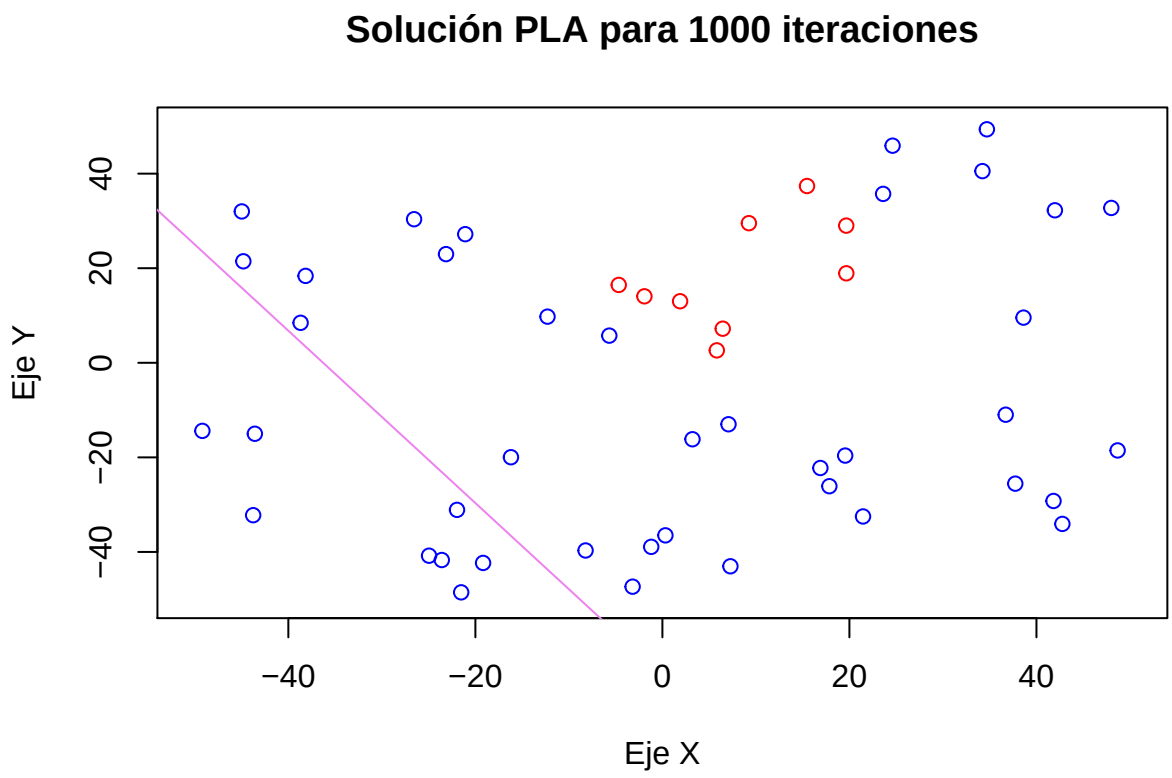

```
sol3 <- ajusta_PLA(datos, etiquetasFA, 1000, c(0,0,0))
cat("El número de fallos con la primera función del apartado 7 y 1000 iteraciones ha sido:\n")
```

```
## El número de fallos con la primera función del apartado 7 y 1000 iteraciones ha sido:
```

```
print(cuenta_errores(sol3, etiquetasFA))
```

```
## [1] 17
```

```
w3 <- sol3[[1]]
w3 <- -w3 / w3[2]
pinta_particion(lista, etiquetasFA, TRUE, function(x,y) w3[1]*x-y+w3[3],
                main="Solución PLA para 1000 iteraciones")
```



En este caso los datos no eran linealmente separables de entrada, ya que se habían clasificado en base a una función cuadrática, con lo que el perceptron no va a llegar a una solución en la que todos los datos estén bien clasificados, de nuevo, da igual el número de iteraciones que le pongamos, el número de errores depende de en qué momento estuviera la recta cuando paramos el algoritmo, pero es "aleatorio" en tanto que la recta no pararía de moverse nunca.

5. Modifique la función `ajusta_PLA` para que le permita visualizar los datos y soluciones que va encontrando a lo largo de las iteraciones. Ejecute con la nueva versión el apartado 3 del ejericio 2.

Lo que hacemos es utilizar la función `Sys.sleep(x)`, que para la ejecución durante x segundos. Añadimos esta función cuando el perceptrón ha dado una vuelta entera a los datos.

```
ajusta_PLA_sol <- function(datos, label, max_iter, vini) {
  parada <- F
  fin <- F
  w <- vini
  iter <- 1
  # Mientras no hayamos superado el máximo de iteraciones o
  # no se haya encontrado solución
  while(!parada) {
    # iteramos sobre los datos
    for (j in 1:nrow(datos)) {
      if (sign(crossprod(w, datos[j,])) != label[j]) {
        w <- w + label[j]*datos[j,]
```

```
        # La variable fin controla si se ha entrado en el if
        fin <- F
      }
    }

    # Pintamos la gráfica
    w2 <- -w / w[2]
    pinta_particion(lista, label, TRUE, function(x,y) -w2[1]*x + y - w2[3])

    # Paramos la ejecución para que se pueda ver la gráfica durante medio segundo
    Sys.sleep(1)

    # Si no se ha entrado en el if, todos los datos estaban bien
    # clasificados y podemos poner a TRUE la variable parada.
    if(fin == T) {
      parada = T
    }
    else {
      fin = T
    }
    iter <- iter + 1
    if (iter >= max_iter) {parada = T}
  }
  # Devolvemos el hiperplano y el número máximo de iteraciones al que hemos
  # llegado.
  list(w, iter)
}
```

Llamamos ahora a esta función con 10, 100 y 1000 como número de iteraciones

```
# Llamamos a la función
ajusta_PLA_sol(datos, etiquetas2, 10, c(0,0,0))
ajusta_PLA_sol(datos, etiquetas2, 100, c(0,0,0))
cat("Nota: la ejecución con 1000 iteraciones está comentada.
    Descomentar si se quiere ver.")
#ajusta_PLA_sol(datos, etiquetas2, 1000, c(0,0,0))
```

6. A la vista de la conducta de las soluciones observadas en el apartado anterior, proponga e implemente una modificación de la función original `sol = ajusta_PLA_MOD(...)` que permita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando los datos del apartado 7 del ejercicio 1.

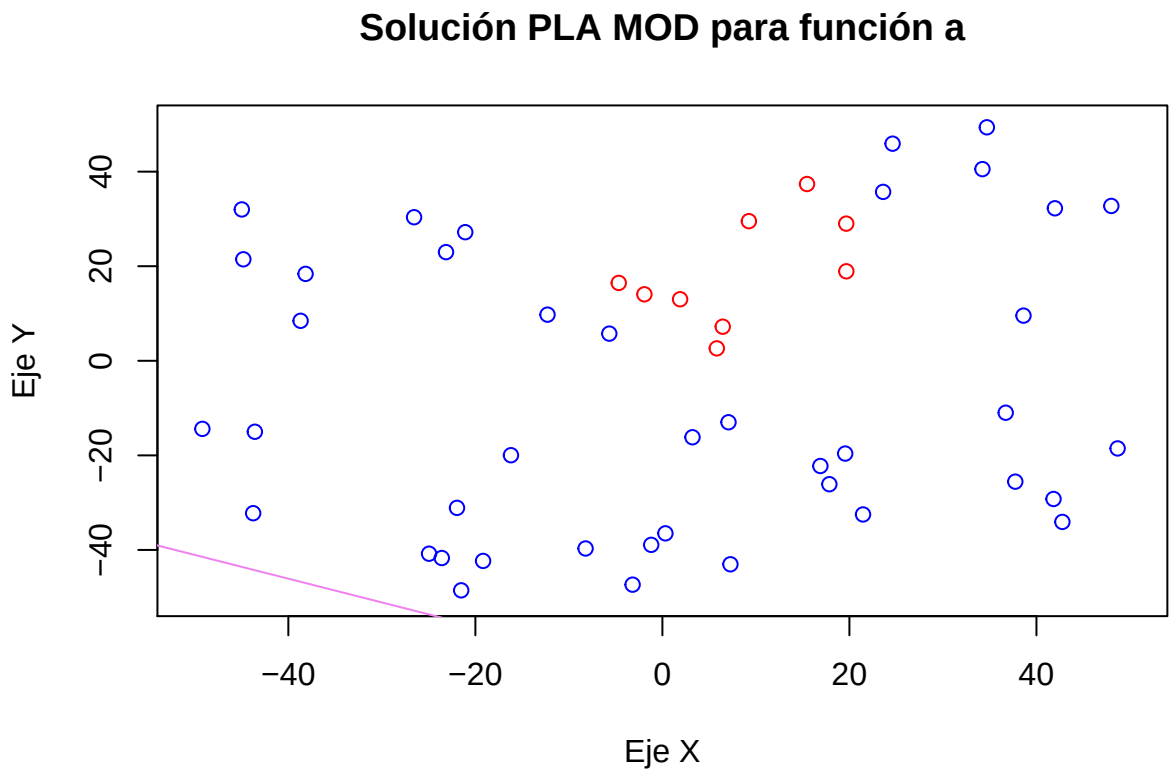
Lo que vamos a hacer es ir contando los errores (con la función que hemos hecho en apartados anteriores) que hay al etiquetar con la solución que va devolviendo el algoritmo en cada iteración (una pasada a los datos) y quedarnos con aquella que tenga menos errores. Además devolvemos como tercer elemento de la lista el número de errores de la mejor solución, (la mejor solución es además la que estamos devolviendo como primer elemento de la lista). El resto del algoritmo continúa igual:

```
ajusta_PLA_MOD <- function(datos, label, max_iter, vini) {
  parada <- F
```

```
  fin <- F
  w <- vini
  wmejor <- w
  iter <- 1
  errores_mejor <- nrow(datos)

  # Mientras no hayamos superado el máximo de iteraciones o
  # no se haya encontrado solución
  while(!parada) {
    # iteramos sobre los datos
    for (j in 1:nrow(datos)) {
      if (sign(crossprod(w, datos[j,])) != label[j]) {
        w <- w + label[j]*datos[j,]
        # La variable fin controla si se ha entrado en el if
        fin <- F
      }
    }

    # Contamos el número de errores que hay en la solución actual y si
    # es menor que el número de errores en la mejor solución de las que
    # llevamos, nos quedamos con la actual
    l <- list(w, 1)
    errores_actual <- cuenta_errores(l, label)
    if(errores_actual < errores_mejor) {
      wmejor <- w
      errores_mejor <- errores_actual
    }

    # Si no se ha entrado en el if, todos los datos estaban bien
    # clasificados y podemos poner a TRUE la variable parada.
    if(fin == T) {
      parada = T
    }
    else {
      fin = T
    }
    iter <- iter + 1
    if (iter >= max_iter) parada = T
  }
  # Devolvemos el hiperplano, el número máximo de iteraciones al que hemos
  # llegado y el número de errores de la mejor solución que hemos encontrado
  list(wmejor, iter, errores_mejor)
}
```

Vamos a probarlo ahora con las funciones del ejercicio 7 (es decir, con las etiquetas que tenemos almacenadas en `etiquetasFA`, `etiquetasFB`, `etiquetasFC` y `etiquetasFD`)

```
sol1_MOD <- ajusta_PLA_MOD(datos, etiquetasFA, 1000, c(0,0,0))
w1 <- sol1_MOD[[1]]
errores <- sol1_MOD[[3]]
w1 <- -w1 / w1[2]
pinta_particion(lista, etiquetasFA, TRUE, function(x,y) -w1[1]*x + y - w1[3],
                main="Solución PLA MOD para función a")
```



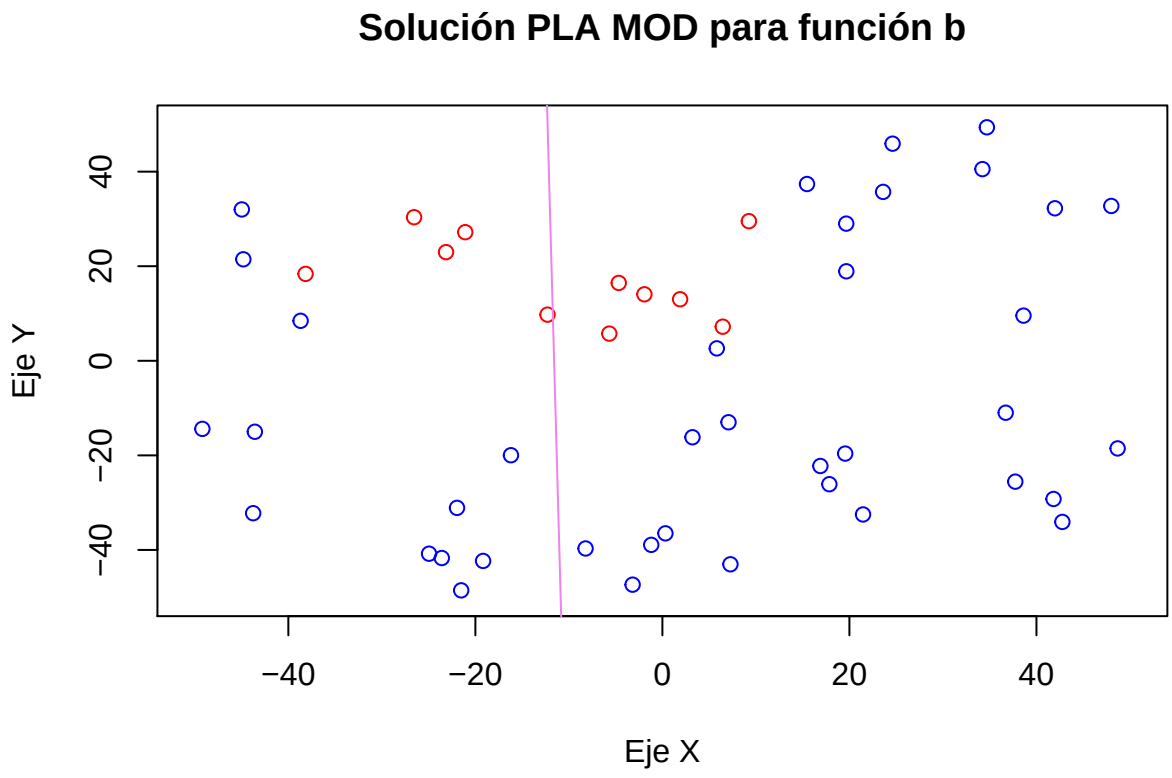

```
cat("El número de errores para el PLA MOD y función a ha sido:\n")
```

```
## El número de errores para el PLA MOD y función a ha sido:
```

```
print(errores)
```

```
## [1] 9
```

```
sol2_MOD <- ajusta_PLA_MOD(datos, etiquetasFB, 1000, c(0,0,0))
w2 <- sol2_MOD[[1]]
errores <- sol2_MOD[[3]]
w2 <- -w2 / w2[2]
pinta_particion(lista, etiquetasFB, TRUE, function(x,y) -w2[1]*x + y - w2[3],
                main="Solución PLA MOD para función b")
```



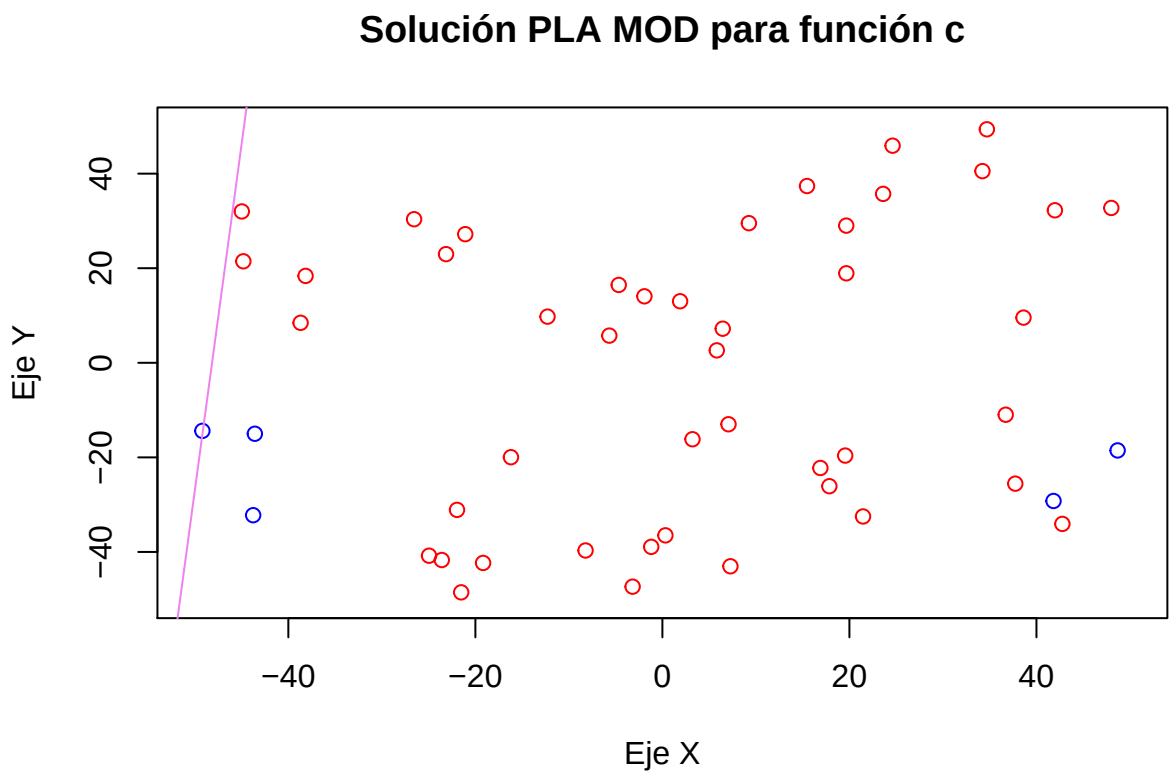

```
cat("El número de errores para el PLA MOD y función b ha sido:\n")
```

```
## El número de errores para el PLA MOD y función b ha sido:
```

```
print(errores)
```

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

```
sol3_MOD <- ajusta_PLA_MOD(datos, etiquetasFC, 1000, c(0,0,0))
w3 <- sol3_MOD[[1]]
errores <- sol3_MOD[[3]]
w3 <- -w3 / w3[2]
pinta_particion(lista, etiquetasFC, TRUE, function(x,y) -w3[1]*x + y - w3[3],
                main="Solución PLA MOD para función c")
```



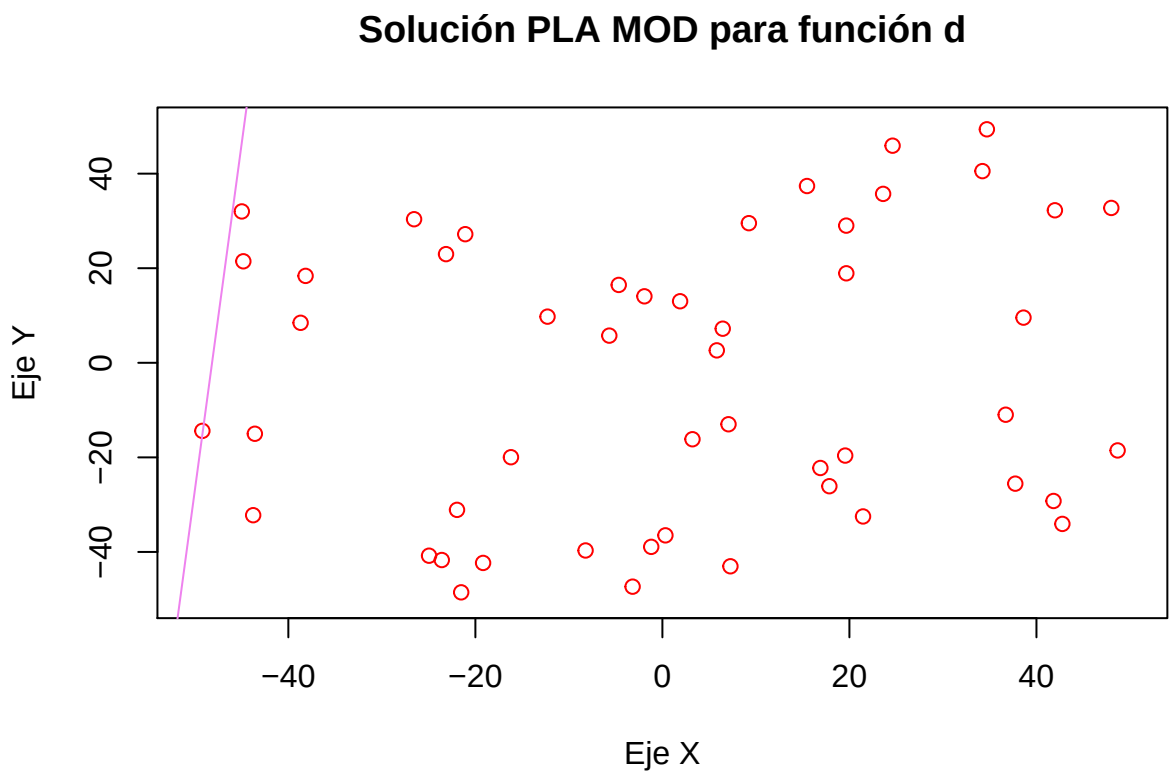

```
cat("El número de errores para el PLA MOD y función c ha sido:\n")
```

```
## El número de errores para el PLA MOD y función c ha sido:
```

```
print(errores)
```

```
## [1] 4
```

```
sol4_MOD <- ajusta_PLA_MOD(datos, etiquetasFD, 1000, c(0,0,0))
w4 <- sol4_MOD[[1]]
errores <- sol4_MOD[[3]]
w4 <- -w4 / w4[2]
pinta_particion(lista, etiquetasFD, TRUE, function(x,y) -w3[1]*x + y - w3[3],
                main="Solución PLA MOD para función d")
```




```
cat("El número de errores para el PLA MOD y función d ha sido:\n")
```

```
## El número de errores para el PLA MOD y función d ha sido:
```

```
print(errores)
```

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

Como vemos, en la mayoría de los casos la mejor solución es la recta que se queda a un lado porque sale mejor (en tanto a que hay menos errores) no clasificar bien únicamente los que se quedaban dentro de las regiones de las funciones cuadráticas que clasificar bien algunos de ellos pero a la vez clasificar mal algunos de los que se quedaban fuera.

Comparando con el apartado anterior, el número de errores es bastante más bajo que cuando paramos la ejecución a las 10, 100 o 1000 iteraciones, sin tener en cuenta el quedarnos con la mejor de las soluciones por las que hemos pasado, lo que tiene sentido.

3. Ejercicio sobre Regresión Lineal

1. Abra el fichero zip.info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscrito que hay en el fichero zip.train

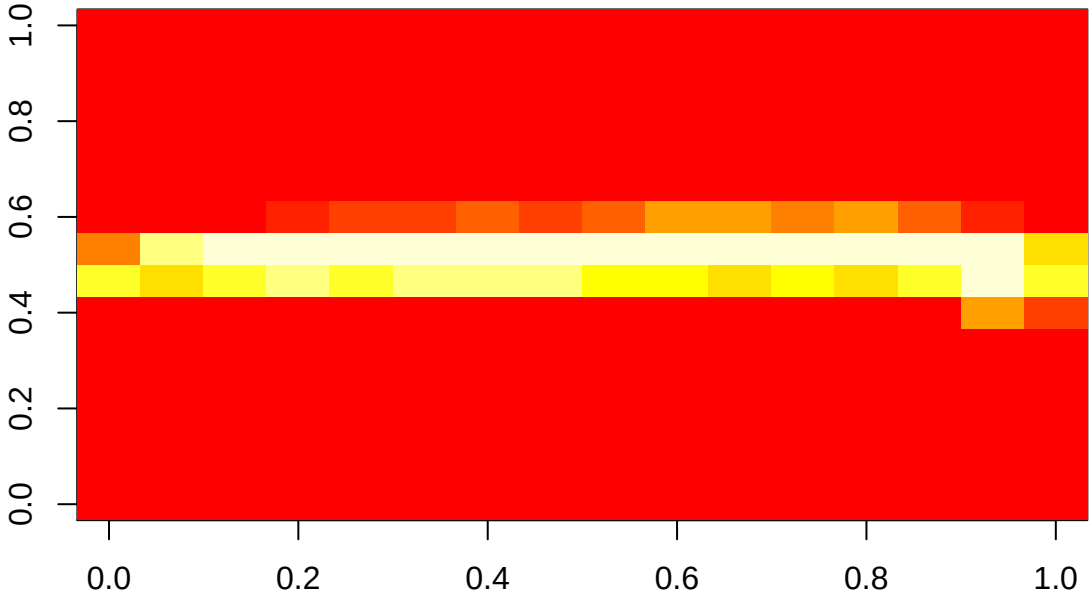
2. Lea el fichero zip.train dentro de su código y visualice las imágenes. Seleccione sólo las instancias de los números 1 y 5. Guárdelas como matrices de tamaño 16×16

Utilizamos la función `read_table()` indicándole que la separación es un espacio. Esto nos carga en `data` todas las instancias en un data_frame, una por fila, donde la primera columna es el número que se está representando y el resto de columnas son los datos. Sin embargo nos introduce una última columna en la que todos los datos son NA que vamos a eliminar. Nos quedamos con aquellas filas en las que la primera columa sea un 1 o un 5 y después a estas les eliminamos también la primera columna, para quedarnos sólo con los datos, pero metiendo en dos data_frame distintos las instancias del 1 y las del 5, para poder diferenciarlas. Después pasamos estos dos data_frame a matrices con `data.matrix` (sigue dejando cada instancia en una fila) y una vez tenemos la matriz podemos utilizar `lapply()` para extraer cada fila y de cada fila hacer una matriz de 16×16 y meterlas todas en una lista, con lo que nos queda una lista de matrices con las instancias del 1 y otra lista de matrices con las instancias del 5. A continuación mostramos con la función `image()` la primera instancia del número 1 y la primera del número 5.

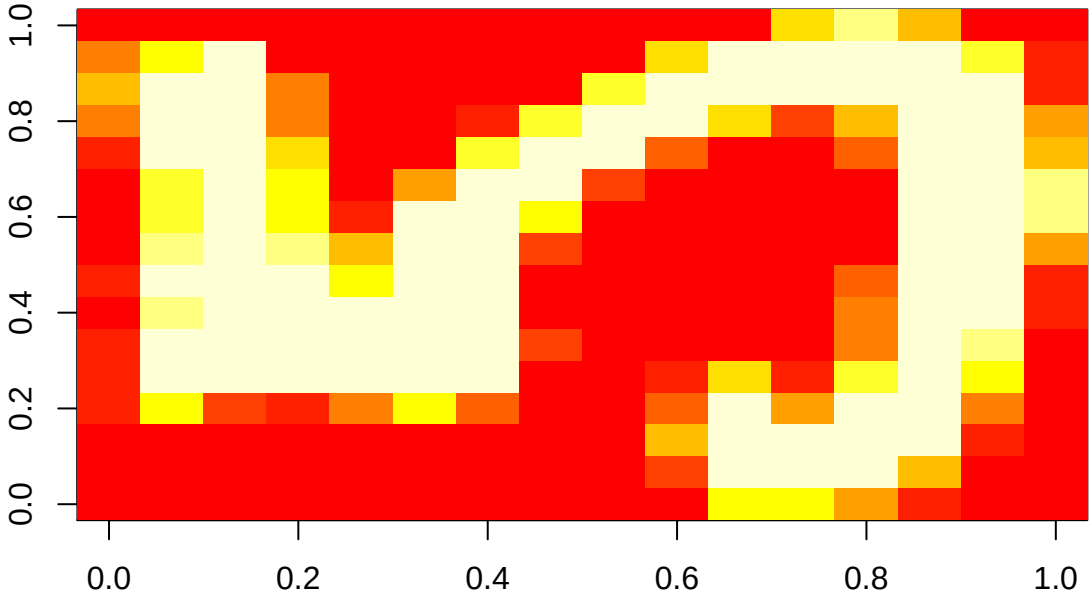
```
data <- read.table("datos/zip.train", sep=" ")
```

```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

```
numero <- data$V1
frame_num1 <- data[numero==1,]
frame_num5 <- data[numero==5,]
# Eliminamos de cada uno la primera columna, que guarda el número del que
# son los datos, y la última, que tiene NA
frame_num5 <- frame_num5[,-258]
frame_num1 <- frame_num1[,-258]
frame_num5 <- frame_num5[,-1]
frame_num1 <- frame_num1[,-1]

# Las dos siguientes líneas pasan estos dos data.frame a matrices
# y los siguen dejando por filas (para poder utilizar apply sobre una matriz)
datos_num1 <- data.matrix(frame_num1)
datos_num5 <- data.matrix(frame_num5)

# Lo siguiente hace una lista de matrices, una por cada instancia de número 1 o 5
lista_num5 <- lapply(split(datos_num5, seq(nrow(datos_num5))), function(x) {
    matrix(x, 16, 16, T)
})
lista_num1 <- lapply(split(datos_num1, seq(nrow(datos_num1))), function(x) {
    matrix(x, 16, 16, T)
})

# Mostramos una imagen de un 1 y una imagen de un 5
image(lista_num1[[1]])
```




```
image(lista_num5[[1]])
```



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

Hacemos una función para calcular la simetría vertical presente en las imágenes, que lo que hace es invertir la matriz por columnas con la función `rev()` y calcular la diferencia en valor absoluto entre la matriz original y la invertida. A continuación sumamos los elementos de la matriz y devolvemos el cambio de signo de la suma.

```
calcular_simetria <- function(mat) {
  # Invertimos la matriz por columnas
  mat_invertida = apply(mat, 2, function(x) rev(x))
  # Calulamos el valor absoluto de la diferencia de cada elemento entre las dos
  # matrices
  dif = abs(mat - mat_invertida)
  # Sumamos los elementos de la matriz
  suma <- sum(dif)
  # Devolvemos el signo cambiado de la suma
  -suma
}
```

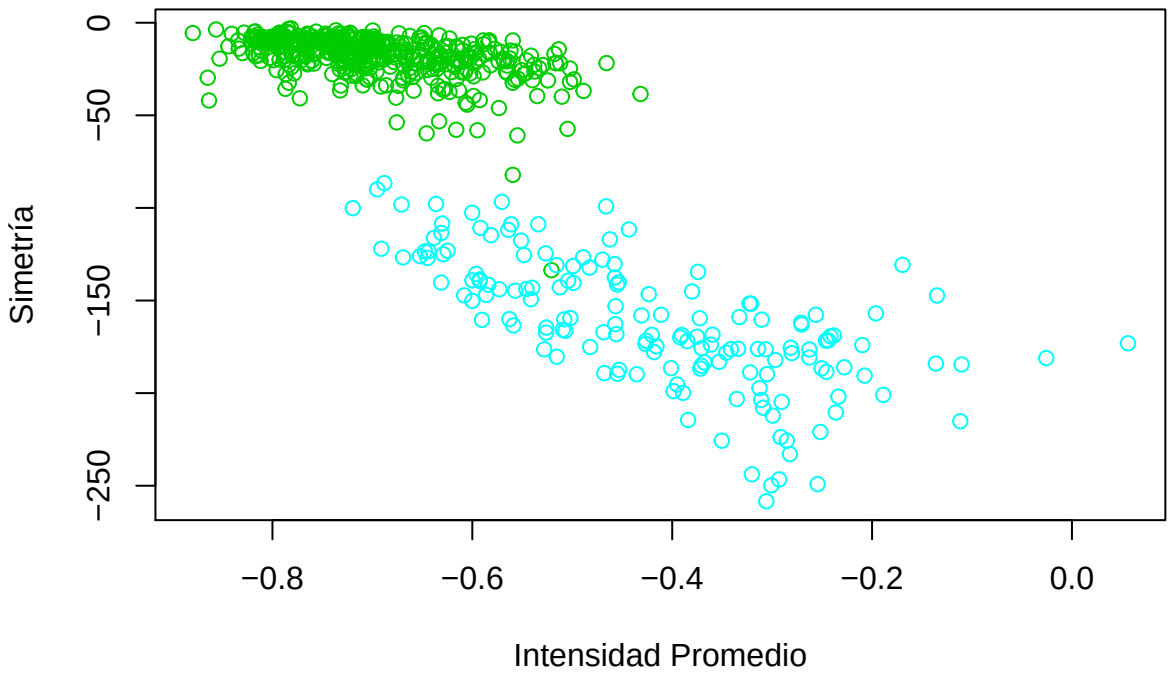
4. Representar en los ejes {X=Intensidad Promedio, Y=Simetría} las instancias seleccionadas de 1's y 5's.

Vamos a hacer primero una variante de la función `pinta_particion` de los dos ejercicios anteriores, que ahora recibirá los datos de los ejes X e Y en vectores

```
pinta_grafica <- function(coordX, coordY, etiquetas=NULL, visible=FALSE, a=NULL, b=NULL,
                          xlab="Intensidad Promedio", ylab="Simetría", main="") {
  if(is.null(etiquetas))
     etiquetas=1
  else etiquetas = etiquetas+3

  plot(coordX, coordY, type = "p", col = etiquetas, xlab = xlab, ylab = ylab,
       main = main)

  if(visible) {
    abline(b,a)
  }
}
```

La intensidad promedio es la media de los valores de la matriz, que se puede calcular simplemente con `mean(m)` siendo `m` una matriz. Vamos a representar entonces en una gráfica las instancias, utilizando la función `mean()` y la función para calcular la simetría que hemos hecho en el apartado anterior.

```
# Calculamos primero las intensidades y las simetrías de todas las matrices,
# es decir, de todas las instancias de 1's y 5's
simetria_1 <- lapply(lista_num1, function(m) calcular_simetria(m))
simetria_5 <- lapply(lista_num5, function(m) calcular_simetria(m))
intensidad_1 <- lapply(lista_num1, function(m) mean(m))
intensidad_5 <- lapply(lista_num5, function(m) mean(m))

# Ponemos las listas anteriores como vectores
simetria_1 <- unlist(simetria_1)
simetria_5 <- unlist(simetria_5)
intensidad_1 <- unlist(intensidad_1)
intensidad_5 <- unlist(intensidad_5)

# Pintamos en una gráfica la intensidad y simetría de las instancias de ambos
# números
intensidad <- c(intensidad_1, intensidad_5)
simetria <- c(simetria_1, simetria_5)
color <- c(rep.int(0, length(intensidad_1)), rep.int(2, length(intensidad_5)))
pinta_grafica(intensidad, simetria, color)
```



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

Formamos primero una matriz con los datos y le calculamos la pseudoinversa haciendo uso de la descomposición en valores singulares y devolvemos después $w = X^{-1}y$ donde X^{-1} es la pseudoinversa de X e y son las etiquetas. Utilizamos para ello la función `La.svd()` que devuelve la descomposición UDV^T.
Hay que tener en cuenta que la matriz diagonal D la devuelve como un vector, sólo con la diagonal principal, por lo que tenemos que transformarla a una matriz diagonal, puesto que la vamos a multiplicar con otras matrices. Utilizamos el razonamiento seguido en clase y si le calculamos la descomposición en valores singulares una matriz X entonces su pseudoinversa es $V(D^2)^{-1}V^TX^T$, con lo que en el método tenemos que calcular también la inversa del cuadrado de la matriz diagonal, lo que es fácil al ser diagonal (simplemente tenemos que cambiar cada valor por su inverso, o dejarlo a 0 si era 0). Con todo esto, el método es el siguiente:

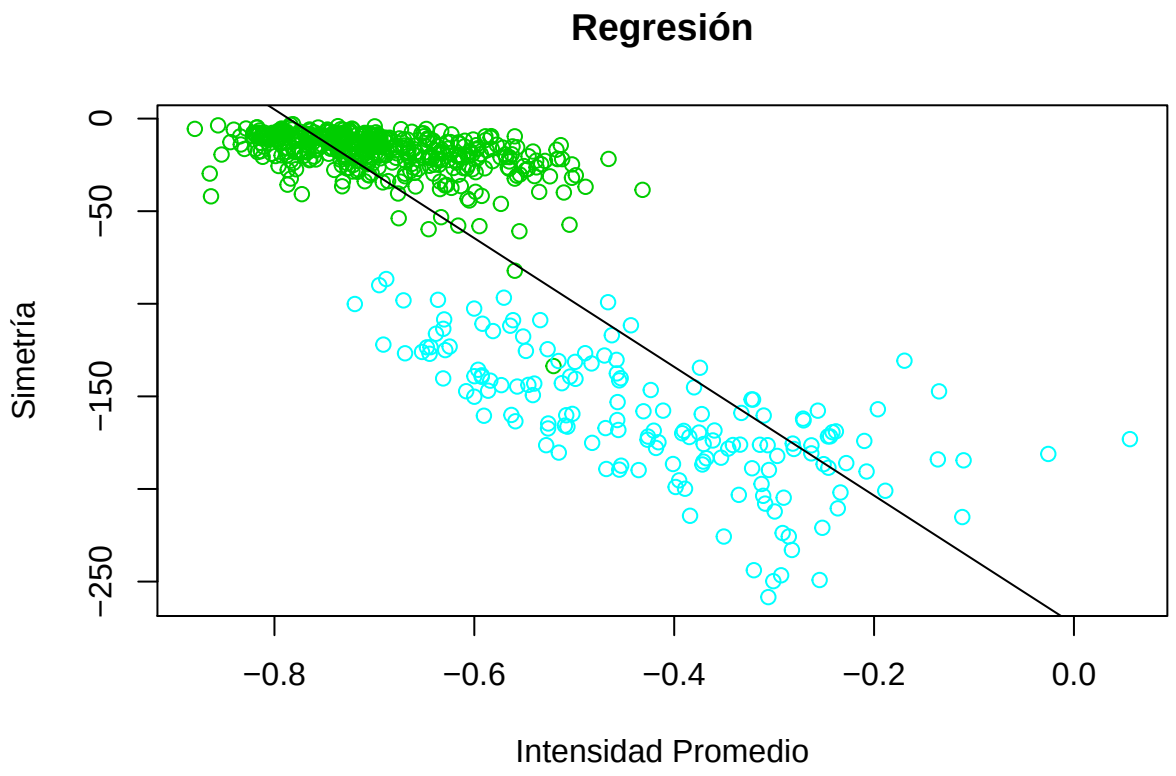
```
Regress_Lin <- function(datos, label) {
  descomp <- La.svd(datos)
  vt <- descomp[[3]]
  # Creamos la inversa de la matriz diagonal al cuadrado
  diag <- matrix(0, length(descomp[[1]]), length(descomp[[1]]))
  for (i in 1:length(descomp[[1]])) {
    diag[i,i] = descomp[[1]][i]
    if (diag[i,i] != 0) {
      diag[i,i] = 1/(diag[i,i]^2)
    }
```

```
  }
  prod_inv <- t(vt) %*% diag %*% vt
  pseud_inv <- prod_inv %*% t(datos)

  w <- pseud_inv %*% label
  w
}
```

6. Ajustar un modelo de regresión lineal a los datos de (Intensidad Promedio, Simetría) y pintar la solución junto con los datos. Valorar el resultado.

Como queremos ajustar la intensidad a la simetría lo que hacemos es pasar como datos la intensidad (poniéndo los puntos en coordenadas homogéneas, por lo que añadimos una columna de 1 al final) y pasar como etiquetas la simetría:

```
w <- Regress_Lin(cbind(intensidad, 1), simetria)
pinta_grafica(intensidad, simetria, color, TRUE, w[1], w[2],
              main="Regresión")
```



Como vemos el resultado tiene sentido puesto que es la recta que mejor se ajusta a los datos (es decir, que menos distancia tiene a ellos en valor absoluto), que es justo lo que queríamos en el método de regresión lineal.

7. En este ejercicio exploramos cómo funciona la regresión lineal en problemas de clasificación. Para ello generamos datos usando el mismo procedimiento que en ejercicios anteriores. Suponemos $X = [-10, 10] \times [-10, 10]$ y elegimos muestras aleatorias uniformes dentro de X. La función f en cada caso será una recta aleatoria que corta a X y que asigna etiqueta a cada punto con el valor de su signo. En cada ejecución generamos una nueva función f.

Como ahora queremos utilizar la regresión para clasificación, tenemos que pasarle como datos lo que obtengamos más una columna de 1 al final para pasarlos a coordenadas homogéneas y como etiquetas las que nos dé la recta aleatoria que generemos.

a) Fijar el tamaño de muestra $N = 100$. Usar regresión lineal para encontrar g y evaluar E_{in}, (el porcentaje de puntos incorrectamente clasificados). Repetir el experimento 1000 veces y promediar los resultados. ¿Qué valor obtiene para E_{in}?

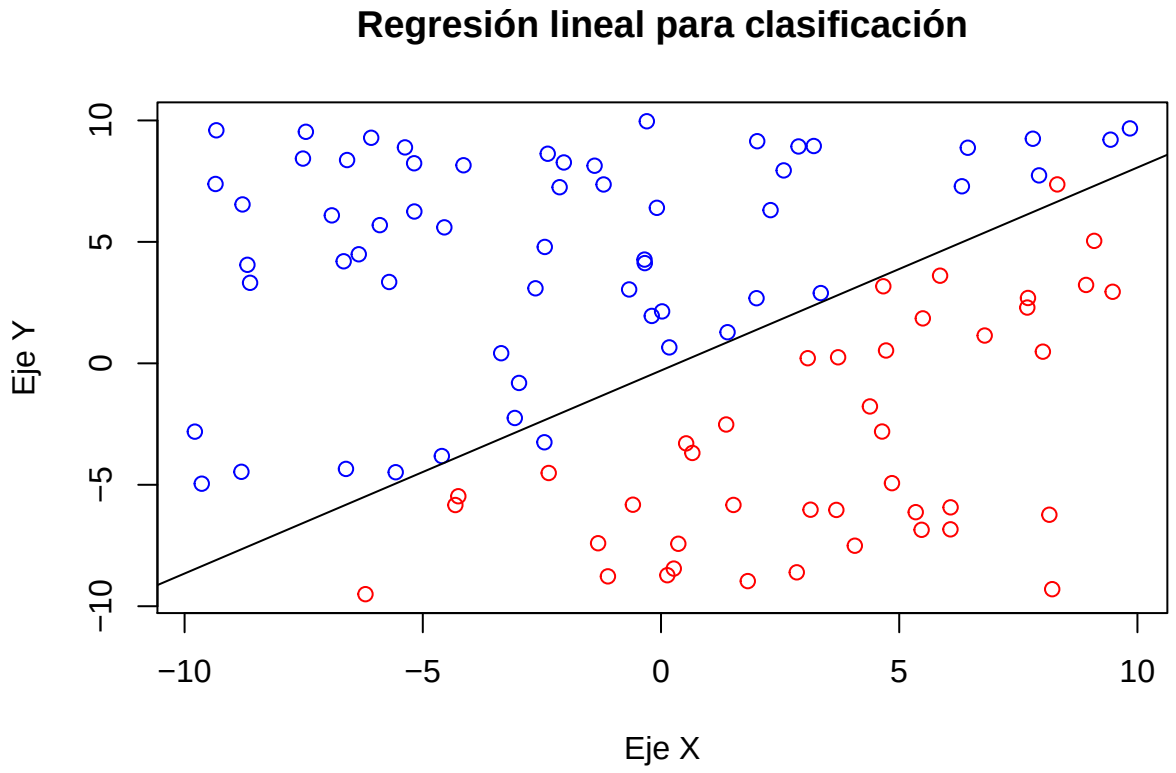
Generamos los datos en el intervalo pedido y creamos una función que cada vez que la llamemos nos cree una recta aleatoria con la función `simula_recta()` que ya tenemos hecha de un ejercicio anterior y nos devuelva las etiquetas de los datos que hemos generado en base a esa recta.

```
N <- 100
datos <- simula_unif(N, 2, c(-10,10))
#Pasamos la lista de datos una matriz con la tercera columna a 1
datos <- unlist(datos)
datos <- matrix(datos, N, 2, T)
datos <- cbind(datos, 1)

generaEtiquetas <- function() {
  r <- simula_recta(c(-10,10))
  etiquetas <- lapply(1:nrow(datos), function(i) {
    #Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos[i,]
    f <- p[2] - r[1]*p[1] - r[2]
    etiquetar(f)
  })

  etiquetas <- unlist(etiquetas)
  etiquetas
}
```

Usamos ahora regresión lineal para encontrar g 1000 veces y calculamos E_{in} como el porcentaje de puntos mal clasificados.

```
errores <- 0
for (i in 1:1000) {
  etiquetas <- generaEtiquetas()
  # Obtenemos g por regresión lineal
  w <- Regress_Lin(datos, etiquetas)
  # Y calculamos las etiquetas que da esta g
  etiquetas_cambiadas <- unlist(lapply(1:nrow(datos), function(i) {
    p <- datos[i, ]
    sign(crossprod(w,p))
  }))
```

```
  # Contamos las diferencias entre los dos vectores de etiquetas
  errores <- errores + cuenta_diferencias(etiquetas, etiquetas_cambiadas)
}

# Hacemos la media de los errores, teniendo en cuenta que hemos hecho 1000
# iteraciones
errores <- errores / 1000

w <- -w / w[2]
# Pintamos la última de las rectas, por ver cómo lo está haciendo
pinta_grafica(datos[,1], datos[,2], etiquetas, TRUE, w[1], w[3], "Eje X",
              "Eje Y", main="Regresión lineal para clasificación")
```



Como vemos la clasificación es bastante buena.
Calculamos ahora el porcentaje de puntos mal clasificados E_{in}

```
Ein <- 100*errores/N
cat("El porcentaje de puntos mal clasificados dentro de la muestra, Ein:\n")
```

```
## El porcentaje de puntos mal clasificados dentro de la muestra, Ein:
```

```
print(Ein)
```

```
## [1] 3.884
```

En efecto, tenemos un porcentaje de error dentro de la muestra de 3.88%, lo que confirma que la clasificación es bastante buena.

b) Fijar el tamaño de muestra $N = 100$. Usar regresión lineal para encontrar g y evaluar E_{out}. Para ello generar 1000 puntos nuevos y usarlos para estimar el error fuera de la muestra, E_{out} (porcentaje de puntos mal clasificados). De nuevo, ejecutar el experimento 1000 veces y tomar el promedio. ¿Qué valor obtiene de E_{out}? Valore los resultados.

Generamos un conjunto de datos nuevo, como en el apartado anterior. Vamos a generar una nueva recta aleatoria, etiquetar los datos en base a esta recta y obtener g por regresión lineal. Después generamos 1000 puntos nuevos en `datos_out` y vamos a calcular E_{out} repitiéndolo 1000 veces quedándonos con la media de los errores, como hemos hecho en el apartado anterior.

```
N <- 100
datos <- simula_unif(N, 2, c(-10,10))
# Pasamos la lista de datos una matriz con la tercera columna a 1
datos <- unlist(datos)
datos <- matrix(datos, N, 2, T)
datos <- cbind(datos, 1)

r <- simula_recta(c(-10,10))
etiquetas_f <- unlist(lapply(1:nrow(datos), function(i) {
  # Obtenemos los puntos uno a uno y los etiquetamos
  p <- datos[i,]
  f <- p[2] - r[1]*p[1] - r[2]
  etiquetar(f)
}))

# Obtenemos g por regresión
g <- Regress_Lin(datos, etiquetas_f)

# Contamos los errores
errores <- 0
for (i in 1:1000) {
  # Generamos los nuevos datos (los de fuera de la muestra) y los pasamos a
  # una matriz
  datos_out <- simula_unif(1000, 2, c(-10,10))
  datos_out <- unlist(datos_out)
  datos_out <- matrix(datos_out, 100, 2, T)
  datos_out <- cbind(datos_out, 1)

  etiquetas_originales <- unlist(lapply(1:nrow(datos_out), function(i) {
    # Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos_out[i,]
    f <- p[2] - r[1]*p[1] - r[2]
    etiquetar(f)
  }))
  etiquetas_g <- unlist(lapply(1:nrow(datos_out), function(i) {
    # Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos_out[i,]
    sign(crossprod(g,p))
  }))
```

```
  errores <- errores + cuenta_diferencias(etiquetas_originales, etiquetas_g)
}
# Hacemos la media de los errores
errores <- errores / 1000

Eout = 100*errores/N
cat("El porcentaje de puntos mal clasificados fuera de la muestra, Eout:\n")
```

```
## El porcentaje de puntos mal clasificados fuera de la muestra, Eout:
```

```
print(Eout)
```

```
## [1] 9.327
```

En este caso vemos que E_{out} ha aumentado pero el porcentaje de error sigue siendo bajo, 9.327%, por lo que la clasificación sigue siendo muy buena.

c) Ahora fijamos $N = 10$, ajustamos regresión lineal y usamos el vector de pesos encontrado como un vector inicial de pesos para PLA. Ejecutar PLA hasta que converja a un vector de pesos final que separe completamente la muestra de entrenamiento. Anote el número de iteraciones y repita el experimento 1000 veces. ¿Cuál es el valor promedio de iteraciones que tarda PLA en converger? (En cada iteración de PLA elija un punto aleatorio del conjunto de mal clasificados). Valore los resultados.

Ponemos N a 10 y generamos los datos. En cada iteración obtenemos una recta aleatoria, etiquetamos los datos en base a esa recta y obtenemos una g por regresión lineal, que es la que le pasamos al PLA como vector inicial. Sumamos todas las iteraciones y calculamos la media después del bucle.

```
N <- 10
datos <- simula_unif(N, 2, c(-10,10))
#Pasamos la lista de datos una matriz con la tercera columna a 1
datos <- unlist(datos)
datos <- matrix(datos, N, 2, T)
datos <- cbind(datos, 1)

iteraciones <- 0

for (i in 1:1000) {
  #Obtenemos una recta aleatoria, etiquetamos los datos en base a esa recta y
  #obtenemos g, un vector de pesos inicial para el PLA
  r <- simula_recta(c(-10,10))
  etiquetas_f <- unlist(lapply(1:nrow(datos), function(i) {
    #Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos[i,]
    f <- p[2] - r[1]*p[1] - r[2]
    etiquetar(f)
  }))

  g <- Regress_Lin(datos, etiquetas_f)

  #Le pasamos al PLA g como vector inicial y nos quedamos con el número de
```

```
  #iteraciones que tarda en converger
  sol <- ajusta_PLA(datos, etiquetas_f, 1000, g)
  iteraciones <- iteraciones + sol[[2]]
}

iteraciones <- iteraciones / 1000
cat("Número medio de iteraciones para PLA converja:\n")
```

```
## Número medio de iteraciones para PLA converja:
```

```
print(iteraciones)
```

```
## [1] 9.398
```

El número medio de iteraciones es alrededor de 9, lo que está bastante bien, como vemos al empezar ya desde una buena solución se acaba antes. Además coincide con el porcentaje de puntos mal clasificados, lo que tiene bastante sentido.

8. En este ejercicio exploramos el uso de transformaciones no lineales. Consideremos la función objetivo $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 25)$. Generar una muestra de entrenamiento de $N = 1000$ puntos a partir de $X = [-10, 10] \times [-10, 10]$ muestreando cada punto $x \in X$ uniformemente. Generar las salidas usando el signo de la función en los puntos muestreados. Generar ruido sobre las etiquetas cambiando el signo de las salidas a un 10% de puntos del conjunto aleatorio generado.

a) Ajustar regresión lineal, para estimar los pesos ω. Ejecutar el experimento 1000 veces y calcular el valor promedio del error de entrenamiento E_{in}. Valorar el resultado.

En cada iteración creamos un conjunto de 1000 datos que etiquetamos en base a la función que nos han dado, generamos ruido a esas etiquetas (10% de cada parte, con la función que ya tenemos para esto del primer ejercicio) y calculamos el error según las etiquetas con ruido. Hacemos la media de errores y calculamos E_{in} como el porcentaje:

```
errores <- 0
for (i in 1:1000) {
  N <- 1000
  #Generamos la muestra de puntos
  datos <- simula_unif(N, 2, c(-10,10))
  #Pasamos la lista de datos una matriz con la tercera columna a 1
  datos <- unlist(datos)
  datos <- matrix(datos, N, 2, T)
  datos <- cbind(datos, 1)

  #Etiquetamos los puntos en base a la f dada
  etiquetas_f <- unlist(lapply(1:nrow(datos), function(i) {
    #Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos[i,]
    f <- p[1]^2 + p[2]^2 - 25
    etiquetar(f)
  }))
```

```
  #Generamos ruido sobre estas etiquetas cambiando el signo a un 10% de estas
  #etiquetas con la función que ya tenemos
  etiquetas_f <- cambiar_etiquetas(etiquetas_f)

  #Generamos w por regresión
  w <- Regress_Lin(datos, etiquetas_f)
  etiquetas_cambiadas <- unlist(lapply(1:nrow(datos), function(i) {
    p <- datos[i, ]
    sign(crossprod(w,p))
  }))
  errores <- errores + cuenta_diferencias(etiquetas_f, etiquetas_cambiadas)
}
errores <- errores/1000
E_in = 100*errores/N
cat("Porcentaje de números mal clasificados dentro de la muestra, Ein\n")
```

```
## Porcentaje de números mal clasificados dentro de la muestra, Ein
```

```
print(E_in)
```

```
## [1] 25.7007
```

Ahora el error dentro de la muestra es más del 25%. Tiene sentido puesto que le estamos metiendo un ruido a las etiquetas del 10% en cada parte, con lo que la regresión no puede ser tan buena como en el apartado anterior.

b) Ahora, consideremos $N = 1000$ datos de entrenamiento y el siguiente vector de variables: $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Ajustar de nuevo regresión lineal y calcular el nuevo vector de pesos $\widehat{\omega}$. Mostrar el resultado.

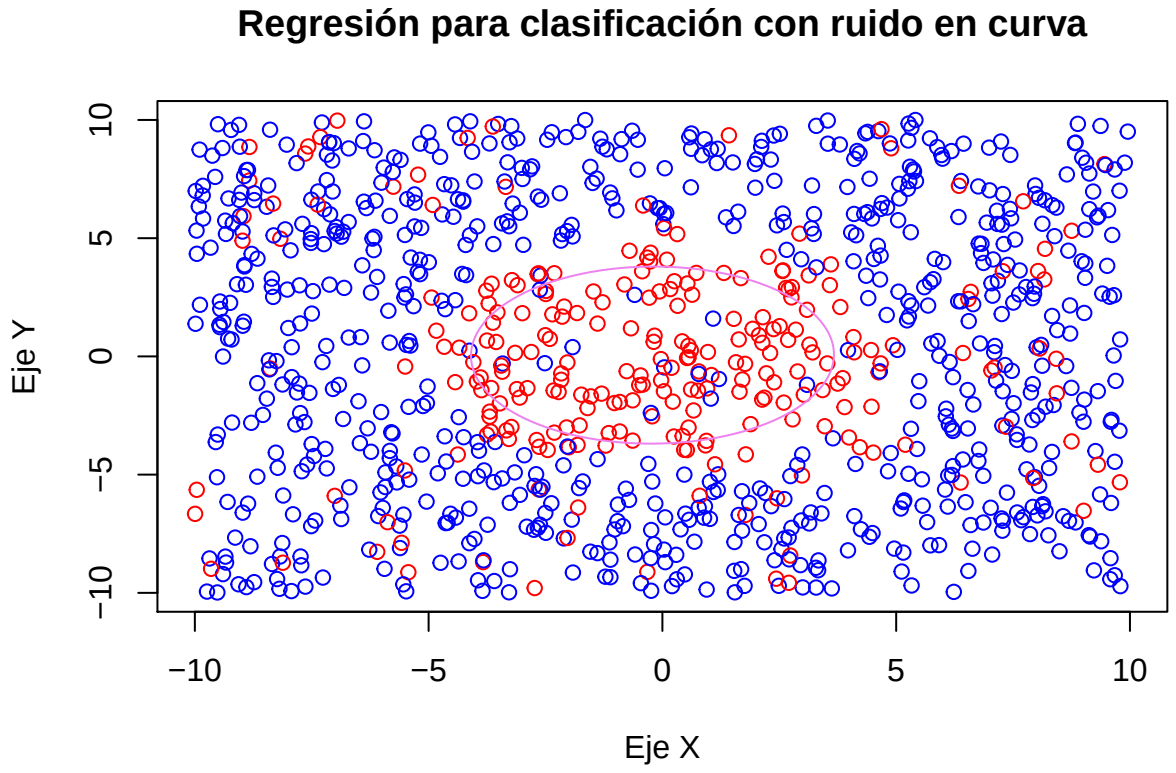
Generamos la muestra de 1000 puntos, la etiquetamos en base a la f dada y hacemos ruido sobre ella. Le hacemos la transformación a los datos en base al vector pedido y estos son los que le pasamos a la regresión.

```
N <- 1000
#Generamos la muestra de puntos
lista_puntos <- simula_unif(N, 2, c(-10,10))
x1 <- rapply(lista_puntos, function(x) x[1])
x2 <- rapply(lista_puntos, function(x) x[2])
datos <- cbind(1, x1, x2, x1*x2, x1^2, x2^2)

#Etiquetamos los puntos en base a la f dada
coord <- cbind(x1, x2)
etiquetas_f <- unlist(lapply(1:nrow(coord), function(i) {
  #Obtenemos los puntos uno a uno y los etiquetamos
  p <- coord[i,]
  f <- p[1]^2 + p[2]^2 - 25
  etiquetar(f)
}))

#Generamos ruido sobre las etiquetas
etiquetas_f = cambiar_etiquetas(etiquetas_f)
```

```r
#Hacemos regresión
w <- Regress_Lin(datos, etiquetas_f)
w
```

```
##               [,1]
## [1,] -0.1362404740
## [2,]  0.0038641883
## [3,] -0.0011166897
## [4,] -0.0001433249
## [5,]  0.0090736473
## [6,]  0.0097470197
```

```r
pinta_particion(lista_puntos, etiquetas_f, TRUE, function(x,y) w[1] + w[2]*x +
                  w[3]*y + w[4]*x*y + w[5]*x^2 + w[6]*y^2, c(-10,10),
                main = "Regresión para clasificación con ruido en curva")
```



Como vemos en este caso la regresión nos devuelve una curva para ajustar los datos.

c) Repetir el experimento anterior 1000 veces calculando en cada ocasión el error fuera de la muestra. Para ello generar en cada ejecución 1000 puntos nuevos y valorar sobre ellos la función ajustada. Promediar los valores obtenidos. ¿Qué valor obtiene? Valorar el resultado.

Utilizamos la ω que tenemos del apartado anteriory vamos a generar 1000 datos nuevos en cada iteración y a calcular el error fuera de la muestra, haciendo la media de los errores primero y después haciendo el porcentaje para E_{out}.

```r
N <- 1000
errores <- 0
for (i in 1:1000) {
  #Generamos los nuevos 1000 puntos con los que calcular el error fuera
  lista_puntos <- simula_unif(N, 2, c(-10,10))
  x1 <- rapply(lista_puntos, function(x) x[1])
  x2 <- rapply(lista_puntos, function(x) x[2])
  datos_out <- cbind(1, x1, x2, x1*x2, x1^2, x2^2)

  etiquetas_originales <- unlist(lapply(1:nrow(datos_out), function(i) {
    #Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos_out[i,]
    f <- p[1]^2 + p[2]^2 - 25
    etiquetar(f)
  }))

  #Generamos ruido sobre las etiquetas originales
  etiquetas_originales <- cambiar_etiquetas(etiquetas_originales)

  etiquetas_w <- unlist(lapply(1:nrow(datos_out), function(i) {
    #Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos_out[i,]
    sign(crossprod(w,p))
  }))

  #Contamos los errores
  errores <- errores + cuenta_diferencias(etiquetas_originales, etiquetas_w)

}

errores = errores/1000
Eout = 100*errores/N
cat("Porcentaje de puntos mal clasificado fuera de la muestra, Eout:\n")
```

```
## Porcentaje de puntos mal clasificado fuera de la muestra, Eout:
```

```r
print(Eout)
```

```
## [1] 40.0332
```

En este caso vemos como el porcentaje de error es considerablemente más alto debido tanto a que la regresión no era ya de por sí tan buena al estar ajustando con ruido, si no que además ahora estamos volviendo a meter ruido en cada iteración, por lo que pone E_{out} a más del 40%.