Trabajo 1

Luis Suárez Lloréns

Apartado 1: Generación y visualización de datos.

En este apartado, veremos las formas básicas de generar datos de manera aleatoria y representarlos en gráficas. Esto será de gran utilidad en los siguientes apartados y prácticas.

Ejercicio 1.

Generar una matriz con *N* filas, *dim* columnas y con valores obtenidos de una distribución uniforme.

La función de R *runif* nos simula datos de una distribución uniforme. Los datos que obtenemos de la función *runif* están en un único vector. Para transformarlos a una matriz, usamos la función *matrix*, indicándole el número de columnas que queremos con el parámetro *ncol*.

La función queda así:

```
simula_unif <- function(N,dim,rango){
  matrix(runif(N*dim,rango[1],rango[2]),ncol = dim)
}

x = simula_unif(3,4,c(-5,5))
print(x)
```

```
##            [,1]       [,2]       [,3]      [,4]
## [1,] -0.8772695  0.9503272 -4.3485708  4.253946
## [2,] -3.8030706  0.6154775 -0.4984998 -1.454052
## [3,] -1.4750690 -4.0478808 -1.7740366 -1.840243
```

Ejercicio 2.

Generar una matriz con *N* filas, *dim* columnas y con valores obtenidos de una distribución normal.

Hacemos lo mismo que en el ejercicio anterior, pero usando la función de R que genera valores de una normal, *rnorm*. Destacar que para obtener la desviación estandar, que es la información pedida por la función *rnorm*, realizamos la operación *sqrt* sobre todas las varianzas sencillamente con sqrt(sigma). R automáticamente realiza la operación sobre todos los elementos, sin necesidad de bucles.

```
simula_gaus <- function(N,dim,sigma){
  matrix(rnorm(N*dim,sd=sqrt(sigma)),ncol = dim,byrow = TRUE)
}

x = simula_gaus(3,4,c(5,7,1,1))
print(x)
```

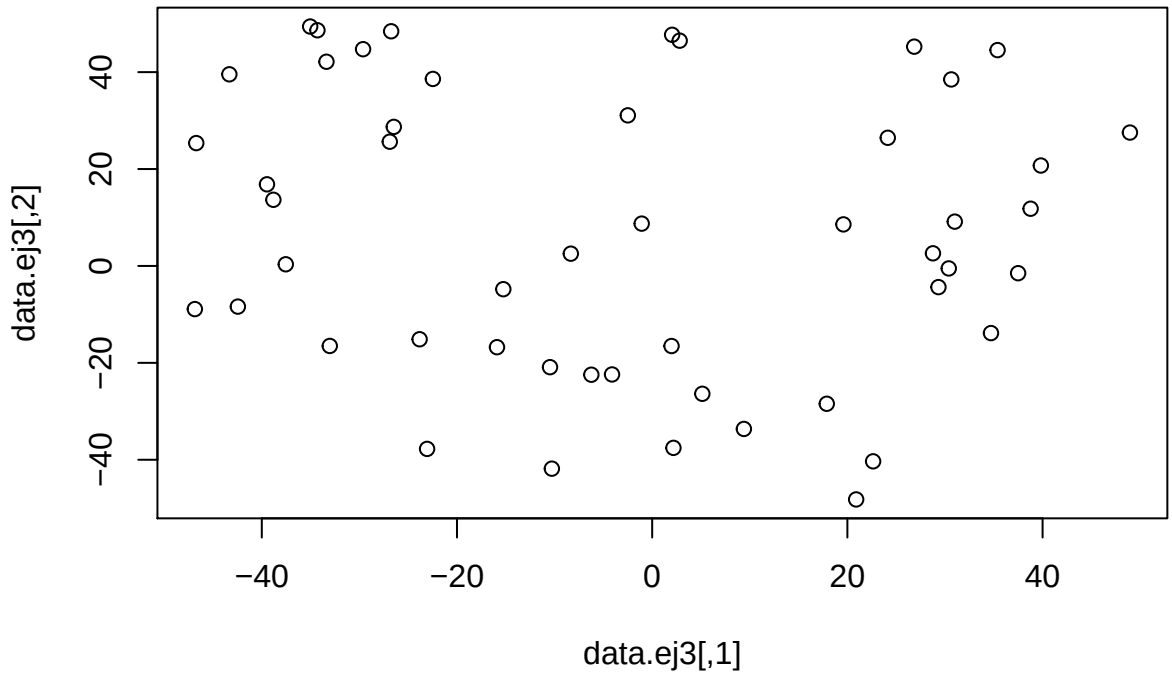
```
##            [,1]         [,2]       [,3]       [,4]
## [1,] -0.9917064 -1.312051766 -0.3886559 -1.9064852
## [2,] -0.8620962  0.001420938  0.6244510 -0.9841314
## [3,]  0.2956310  1.012900537  0.4573872  1.5824320
```

Ejercicio 3.

Mostrar los datos generados por una uniforme.

Para generar los datos, usaremos la función generada en el ejercicio 1. Usaremos $N = 50$, $dim = 2$ y $rango = [-50, 50]$. Tras esto, simplemente tenemos que darle los datos a *plot*.

```
data.ej3 <- simula_unif(50,2,c(-50,50))

plot(data.ej3)
```

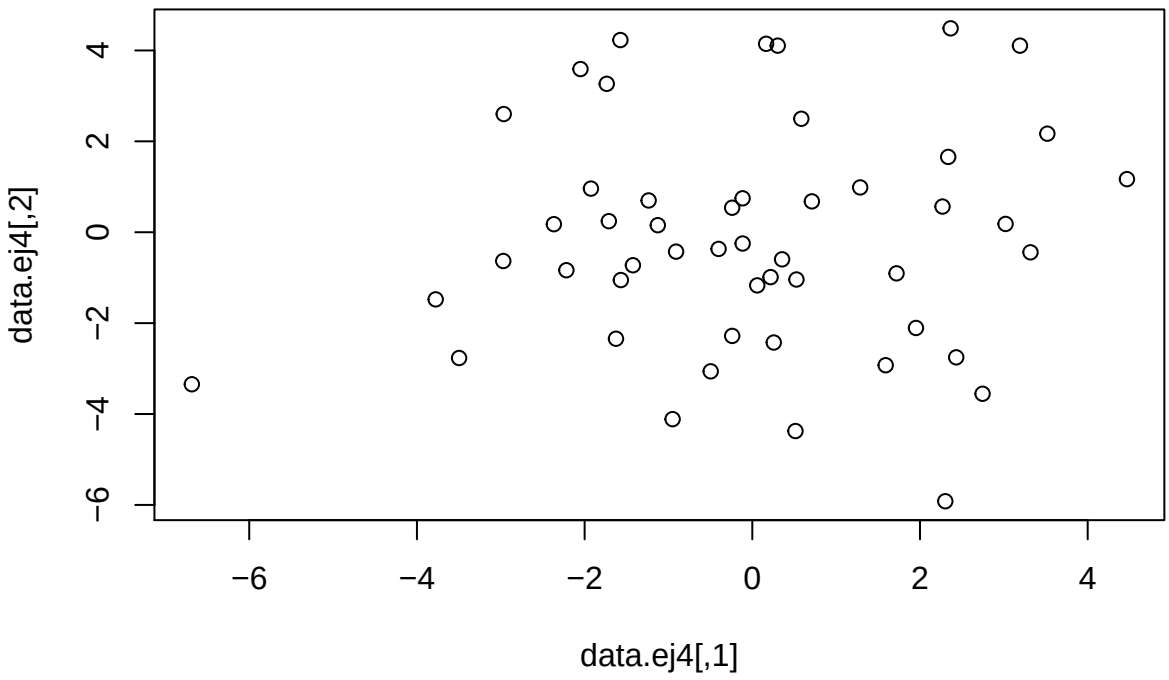


Ejercicio 4.

Mostrar los datos generados por una Gaussiana.

Realizaremos lo mismo que en el ejercicio anterior, pero generando los datos con la función del ejercicio 2. Los parámetros son $N = 50$, $dim = 50$ y $sigma = [5, 7]$.

```
data.ej4 <- simula_gaus(50,2,c(5,7))

plot(data.ej4)
```



Ejercicio 5.

Generar una recta que pase por dos puntos aleatorios.

Obtener los valores de a y b es muy sencillo:

$$a = \frac{y_1 - y_2}{x_1 - x_2}$$
$$b = y_1 - a x_1$$

Hay que tener cuidado con el caso $x_1 - x_2 = 0$, es decir, que la linea sea vertical. Ese caso guardaremos $a = INF$ y $b = x_1$.

```
simula_recta <- function(rango){
  puntos <- simula_unif(2,2,rango)
  a <- 0
  b <- 0

  if((puntos[1,1]-puntos[2,1]) != 0){
    a <- (puntos[1,2]-puntos[2,2])/(puntos[1,1]-puntos[2,1])
    b <- puntos[1,2] - a*puntos[1,1]
  }
  else{
    a <- Inf
    b <- puntos[1,1]
  }
```

```
  c(a,b)
}

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

```
## [1]  3.527122 -2.371566
```

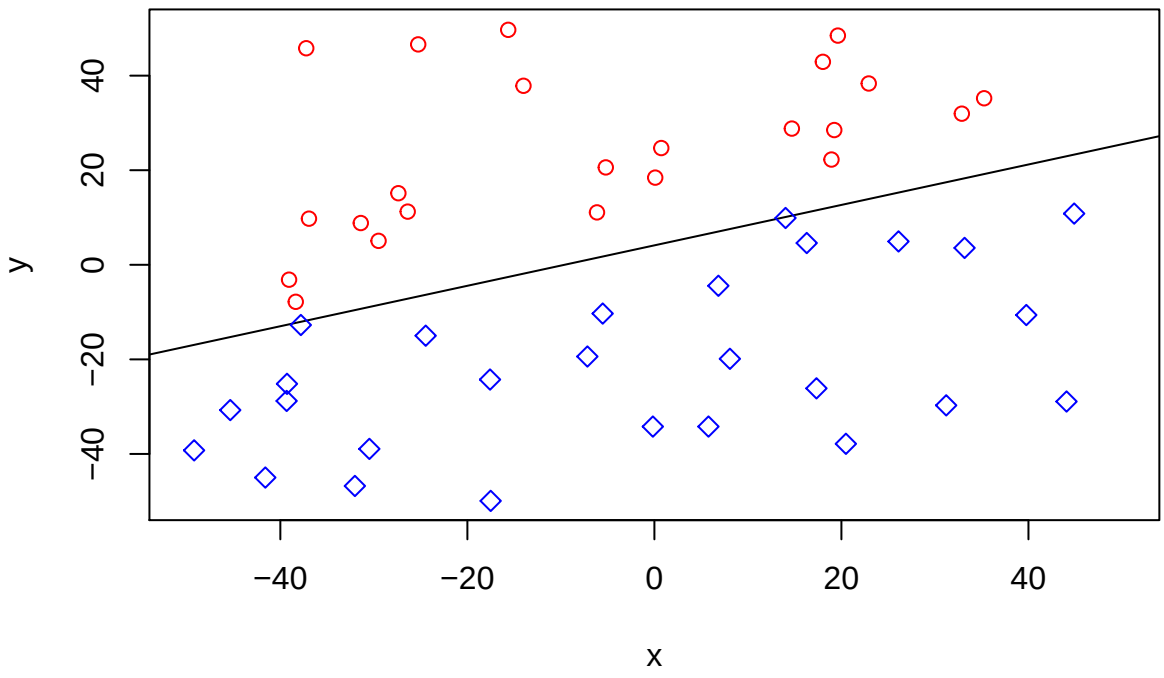
Ejercicio 6.

Etiquetar puntos aleatorios según una recta.

Tenemos que generar una recta y los puntos. Tras esto, generamos un vector con las etiquetas, según el signo de $f(x, y) = y - ax - b$.

Una vez tenemos todos los datos, mostramos los datos con tipos de puntos distintos y la recta con la función *abline*.

```
#Simulacion de recta y datos
recta <- simula_recta(c(-50,50))
data.ej6 <- simula_unif(50,2,c(-50,50))

#Funcion de evaluacion de recta
eval.recta <- function(x,y){
  y-recta[1]*x-recta[2]
}
#Funcion de clasificacion
clasifica <- function(f,x,y){
  sign(f(x,y))
}

clas <- clasifica(eval.recta,data.ej6[,1],data.ej6[,2])

#Plot:
#   Positivos
#   Recta (usando abline)
#   Negativos

plot(data.ej6[clas > 0,], pch = 1, col = 2,xlim =c(-50,50), ylim=c(-50,50), xlab = "x", ylab = "y")
abline(a = recta[2], b = recta[1])
points(data.ej6[clas < 0,], pch = 5, col = 4)
```



Ejercicio 7.

Etiquetar puntos aleatorios según una función.

Vamos a seguir los mismos pasos que el ejercicio anterior. La diferencia que encontramos es que las funciones no son rectas y por tanto, no nos vale la misma forma de dibujar la función. Para poder dibujarlas, usaremos la función de R *contour()*.

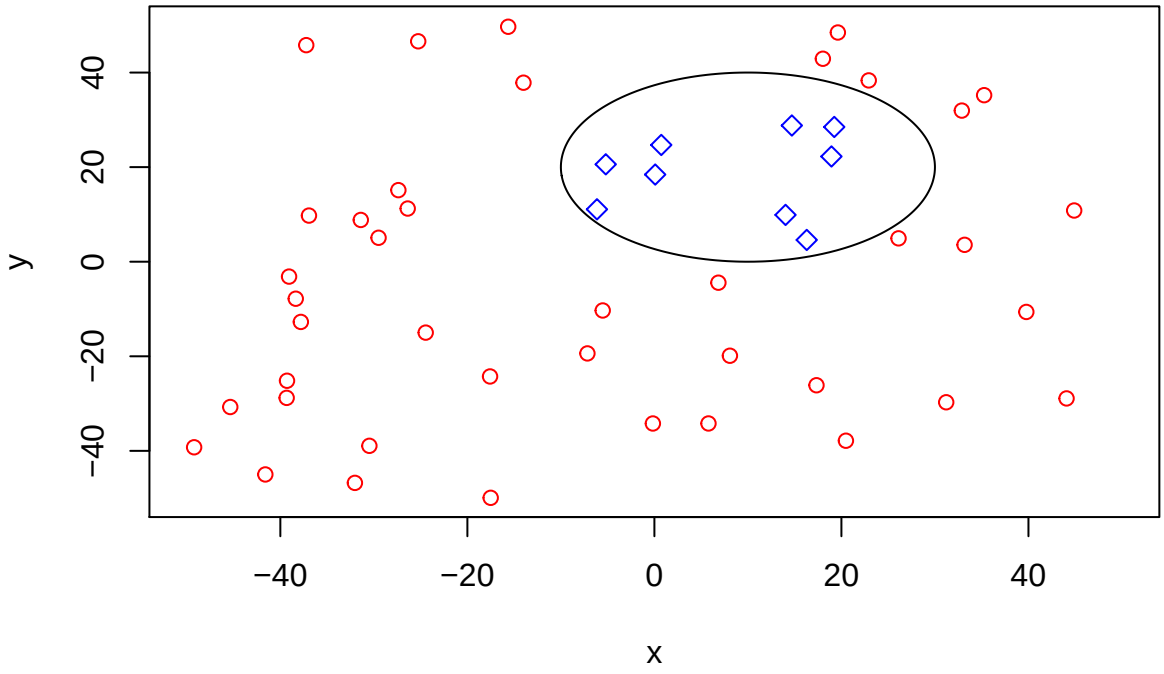
```
#Definicion de funciones
f1 <- function(x,y){
  (x-10)**2+(y-20)**2 - 400
}
f2 <- function(x,y){
  0.5*(x-10)**2+(y-20)**2 - 400
}
f3 <- function(x,y){
  0.5*(x-10)**2-(y+20)**2 - 400
}
f4 <- function(x,y){
  y - 20*x**2 - 5*x + 3
}

#Funcion de dibujado
draw.clasificacion <- function(f,data,rango){
  clas <- clasifica(f,data[,1],data[,2])
  x_graph = seq(-50,50,length.out = 500)
```

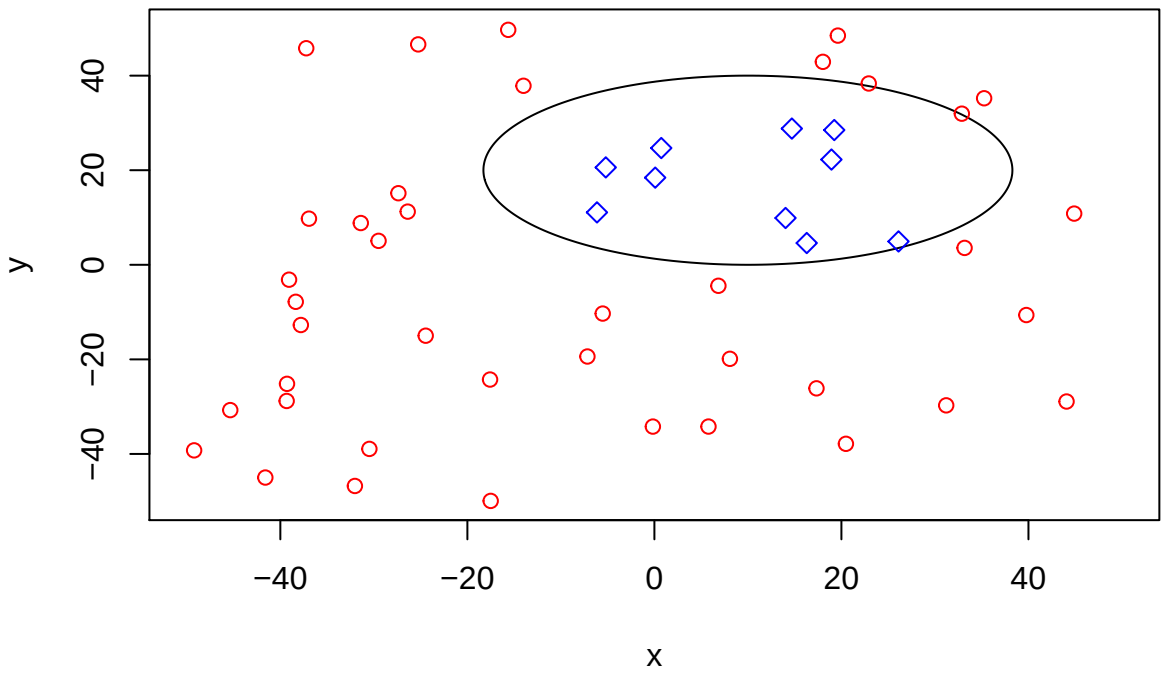
```
  y_graph = seq(-50,50,length.out = 500)
  z = outer(x_graph, y_graph,f)
  contour(x_graph,y_graph,z, levels = 0, drawlabels = FALSE,xlim =rango, ylim=rango, xlab = "x", ylab =
  points(data[clas > 0,], pch = 1, col = 2)
  points(data[clas < 0,], pch = 5, col = 4)
}
```

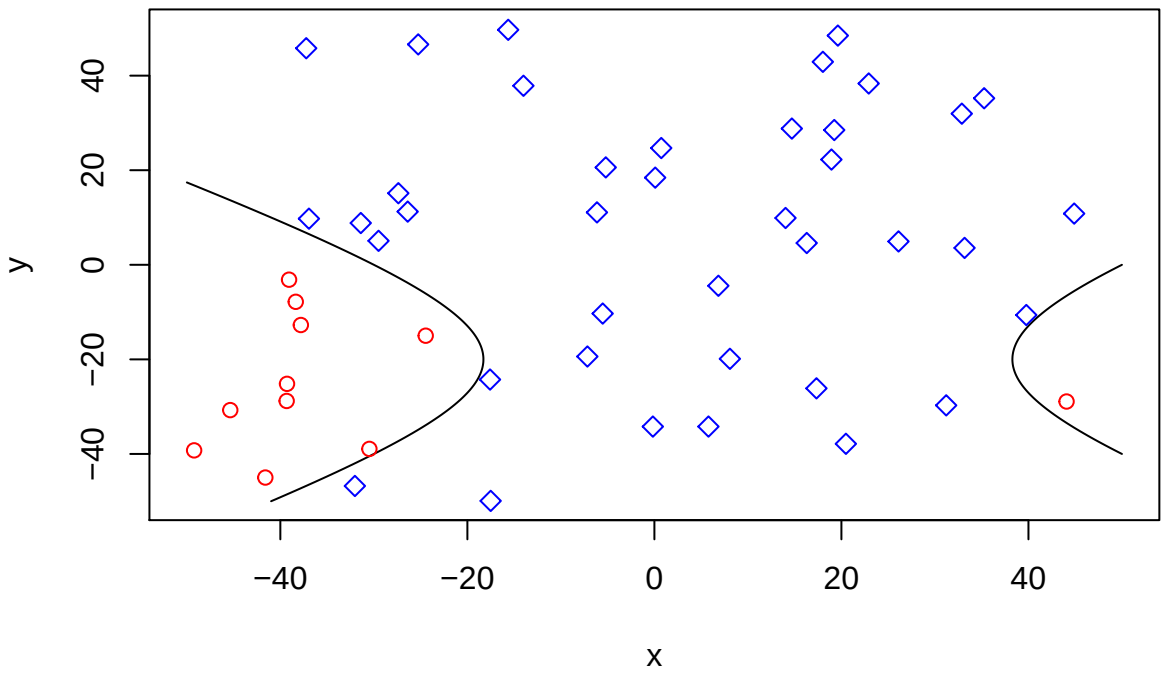
Una vez tenemos hemos creado la función para dibujar la clasificación, solo tenemos que ejecutarlas pasando las distintas funciones. Por ejemplo, para la función 1 sería:

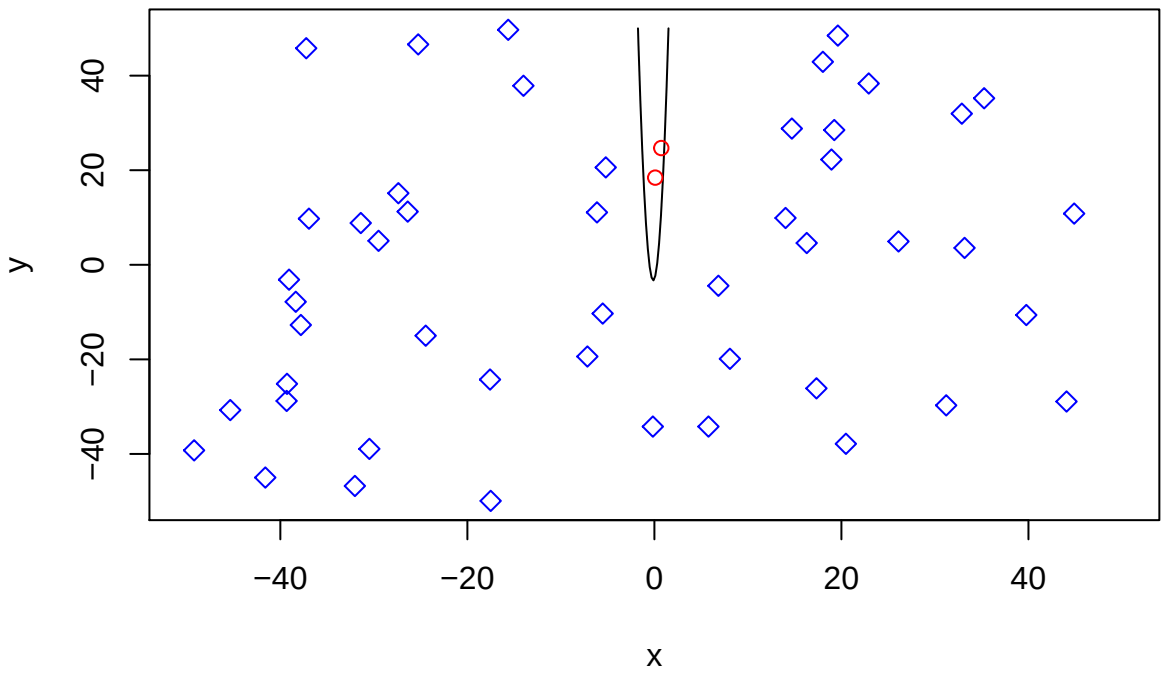
```
draw.clasificacion(f1,data.ej6,c(-50,50))
```

Y aquí están las gráficas.




y
40
20
0
−20
−40
−40
−20
0
20
40
x


y
40
20
0
−20
−40
−40
−20
0
20
40
x



Ejercicio 8.

Modificar un 10% de etiquetas.

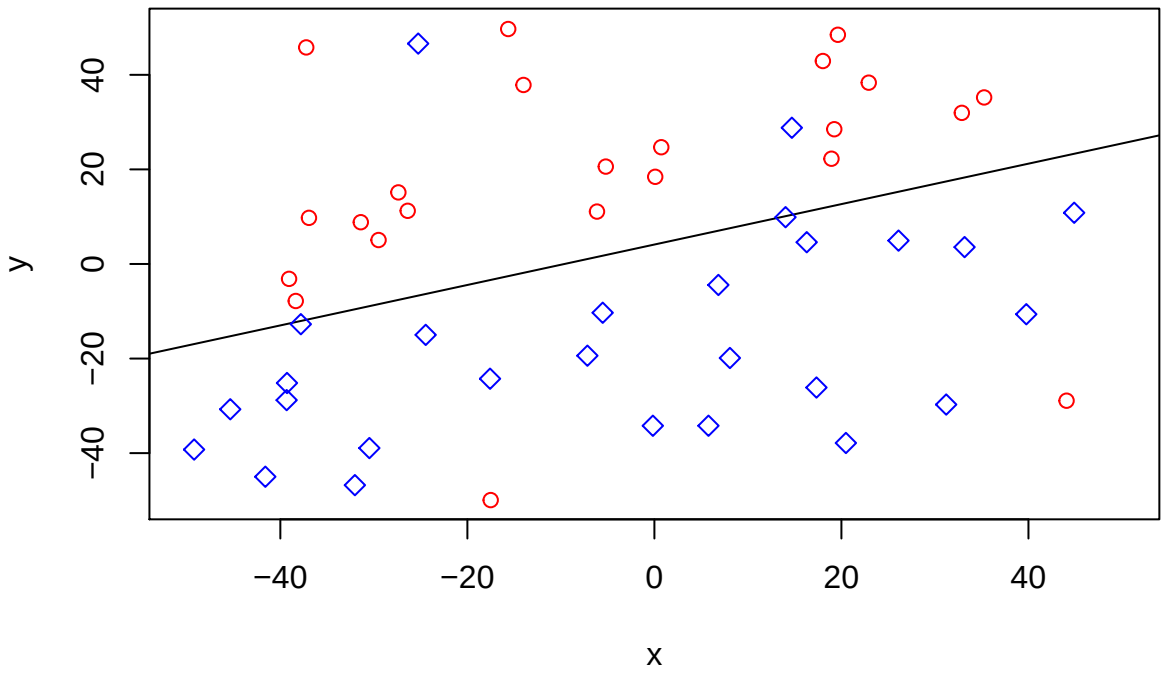
Para cambiar las etiquetas, primero, separamos los puntos en dos grupos, según su clasificación y los volvemos a concatenar. Así, obtendremos los puntos ordenados con respecto a la clasificación.

Después, generamos dos vectores, que representán la clasificación de cada grupo. En vez de generar uno completamente verdadero y otro falso, añadimos el 10% de valores mal clasificado, como dice el ejercicio. Tras esto, solo hace falta reordenarlo con la orden *sample()* y juntarlos.

Tras modificar las etiquetas, solo tenemos que repetir la respresentación del ejercicio 6.

```
clas <- clasifica(eval.recta,data.ej6[,1],data.ej6[,2])

#Ordenamos los datos, primero negativos y luego positivos
data.ej8 = rbind(data.ej6[clas < 0,], data.ej6[clas > 0,])

#Negativos
long = length(clas[clas<0])
valores_camb = trunc(long/10)

clas_neg = rep(c(1,-1),times=c(valores_camb,long-valores_camb))
clas_neg = sample(clas_neg,size = long)
#Positivos
long = length(clas[clas>0])
valores_camb = trunc(long/10)
```

```r
clas_pos = rep(c(1,-1),times=c(long-valores_camb,valores_camb))
clas_pos = sample(clas_pos,size = long)

#Unimos las etiquetas en un unico vector
label.ej8 = c(clas_neg,clas_pos)

#Mostramos resultados
plot(data.ej8[label.ej8 == 1,], pch = 1, col = 2,xlim =c(-50,50), ylim=c(-50,50), xlab = "x", ylab = "y
abline(a = recta[2], b = recta[1])
points(data.ej8[label.ej8 == -1,], pch = 5, col = 4)
```



Apartado 2: Ajuste del Algoritmo Perceptron

Ejercicio 1:

Implementa el algoritmo perceptron.

Vamos a hacer una pequeña modificación sobre lo que nos pide el ejercicio. Además de devolver los coeficientes calculados, devolveremos el número de iteraciones necesarias para conseguirlo. Este dato nos será de gran utilidad en el resto de la sección.

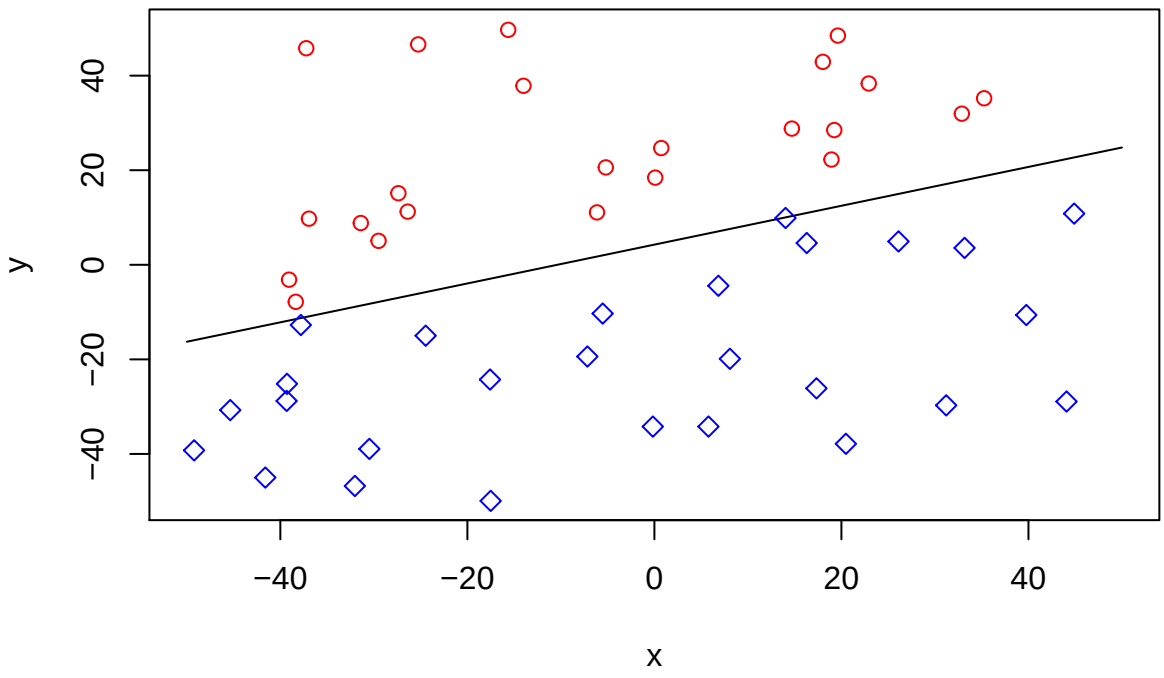
```r
# Calcula el hiperplano para clasificar
# datos: datos que se usan para clasificar
# label: etiqueta de los datos
# max_iter: número máximo de iteraciones
```

```r
# vini: vector inicial
#
# return: coeficientes del hiperplano
ajusta_PLA <- function(datos, label, max_iter, vini){
  cambio = TRUE
  coef = vini
  iteracion = 0
  num_datos = length(label)+1

  while(cambio && iteracion < max_iter){
    i=1
    cambio = FALSE

    while(i<num_datos){

      if(sign(t(c(datos[i,],1)) %*% coef) != label[i]){
        coef <- coef + label[i]*c(datos[i,],1)
        cambio = TRUE
      }
      i <- i+1
    }

    iteracion <- iteracion + 1
  }

  return(list(coef,iteracion))
}
```

Ejercicio 2

Primero, para ver que funciona correctamente, vamos a mostrar los resultados cuando tomamos como coeficientes iniciales el vector *(0,0,0)*.

```
## Se han realizado  220  iteraciones.
```



Como podemos ver, no obtenemos exactamente la misma linea que en el apartado anterior, pero obtenemos una linea que separa también perfectamente los conjuntos de puntos.

Tras comprobar que funciona correctamente, vamos a obtener el número de iteraciones medio, dados coeficientes iniciales *runif(3,0,1)*.

```
## La media de las 10 ejecuciones del PLA es:  205.7
```

Ejercicio 3

Para calcular el porcentaje de fallos, realizamos:

$$\frac{\sum_n |etiq_i - pred_i|}{n} * 0.5$$

Aclarar que multiplicamos por 0.5 ya que, si fallamos al clasificar, en la sumatoria aparecerá un +2. Multiplicando por 0.5 se arregla el error.

Un ejemplo del código que realiza esto es el siguiente:

```
info_PLA <- ajusta_PLA(data.ej8,label.ej8,10,c(0,0,0))
coef <- info_PLA[[1]]
error <- mean(abs(label.ej8-sign(h(data.ej8[,1],data.ej8[,2]))))*0.5
```

Realizamos este ajuste a cada una de las soluciones obtenidas con el PLA, con 10, 100 y 1000 iteraciones respectivamente

```
## PLA con 10 iteraciones

## Porcentaje de fallos:  0.28

## PLA con 100 iteraciones

## Porcentaje de fallos:  0.24

## PLA con 1000 iteraciones

## Porcentaje de fallos:  0.16
```

Para empezar, podemos ver que hay error. Es lógico, al usar los datos del ejercicio 8 del apartado anterior, tenemos datos que no es posible separar por una línea, luego necesariamente aparecerá dicho error.

Por otro lado, el error que obtenemos parece reducirse con el paso de las iteraciones, pero veremos en el próximo ejercicio que no necesariamente es así.

Ejercicio 4

Repetimos el mismo análisis que en el apartado anterior, pero con la primera función del apartado 7.

```
## PLA con 10 iteraciones

## Porcentaje de fallos:  0.64

## PLA con 100 iteraciones

## Porcentaje de fallos:  0.36

## PLA con 1000 iteraciones

## Porcentaje de fallos:  0.38
```

Volvemos a estar en las mismas, los datos no son separables linealmente, luego hay errores. En este caso, podríamos intentar ajustar con nuestro PLA esta función, cambiando los datos de la entrada del algoritmo para que tenga en cuenta términos cuadráticos. Pero de manera lineal, estos son los valores que cabría esperar.

En cuanto a los errores, en este caso vemos como no es monótonamente decreciente. Esto se debe a que en realidad, la recta generada por el algoritmo PLA va modificandose, pero sin controlar de ninguna manera si va a mejor o no. Por tanto, puede que con más iteraciones, no mejore el resultado si no que empeore, como podemos ver aquí.

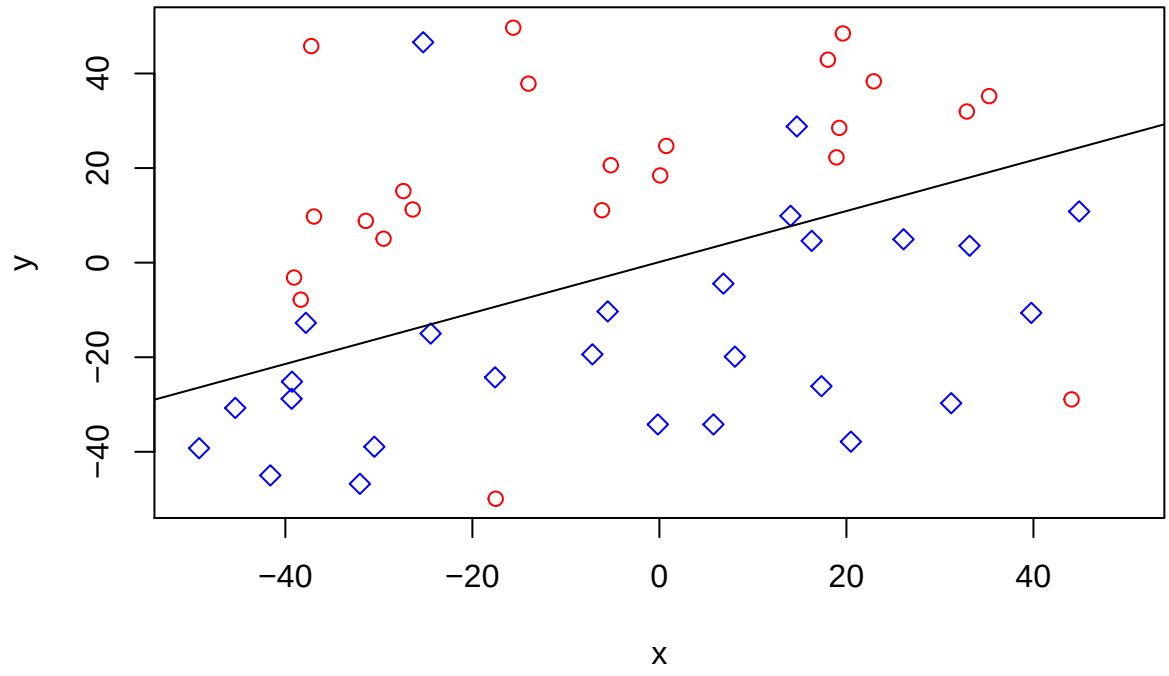
Ejercicio 5

Vamos a generar una función, *dibuja_PLA*, que nos va dibujando las diferentes rectas por las que va pasando la ejecución del PLA.

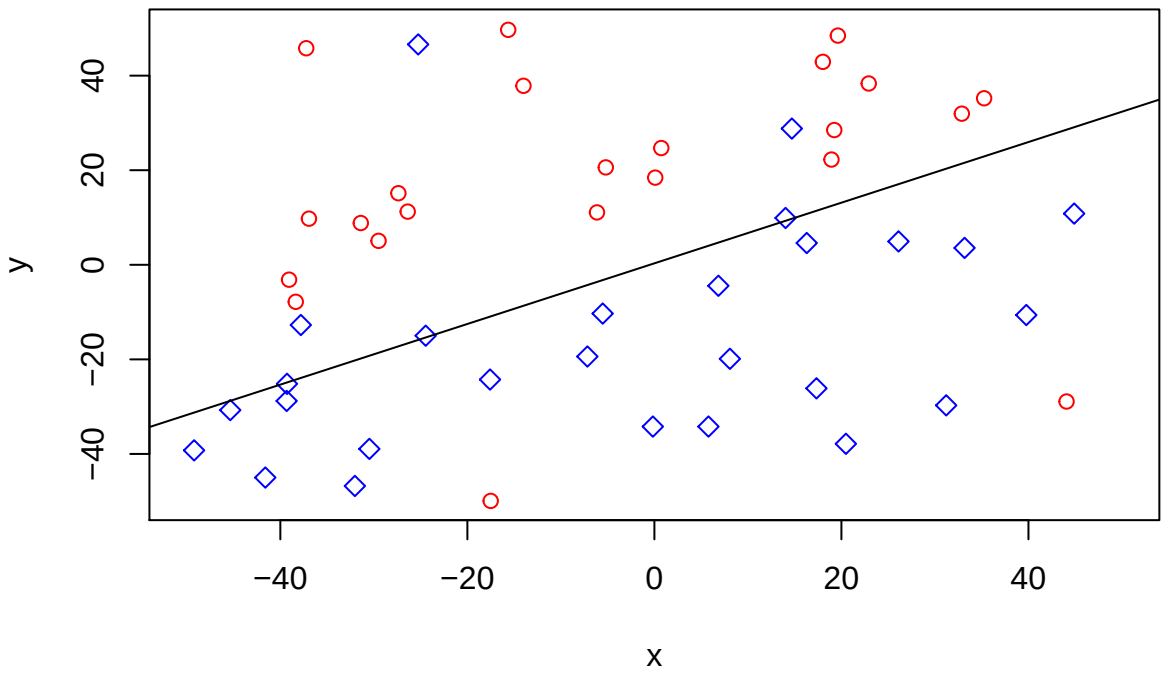
```
dibuja_PLA <- function(datos, label, max_iter, vini){
  coef <- vini

  for( i in 1:max_iter){
    info_PLA <- ajusta_PLA(datos,label,1,coef)
    coef <- info_PLA[[1]]
    error <- mean(abs(label-sign(datos[,1]*coef[1]+ datos[,2]*coef[2] + coef[3])))*0.5

    print(paste("Iteracion: ",i))
    print(paste("Porcentaje de fallos: ",error))

    plot(datos[label > 0,], pch = 1, col = 2,xlim =c(-50,50), ylim=c(-50,50), xlab = "x", ylab = "y")
    points(datos[label <= 0,], pch = 5, col = 4)
    abline(-coef[3]/coef[2], -coef[1]/coef[2])
  }
}
```

Y usamos esa función para mostrar, por ejemplo, los primeros 5 pasos.

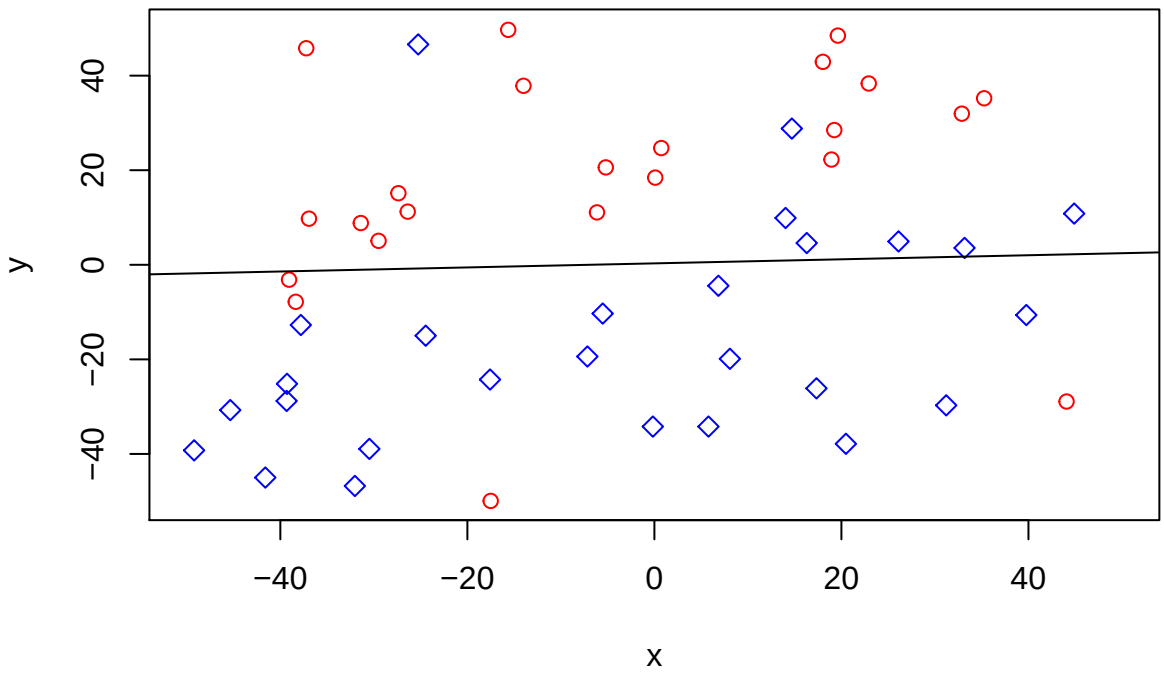
```
## [1] "Iteracion:  1"
## [1] "Porcentaje de fallos:  0.12"
```



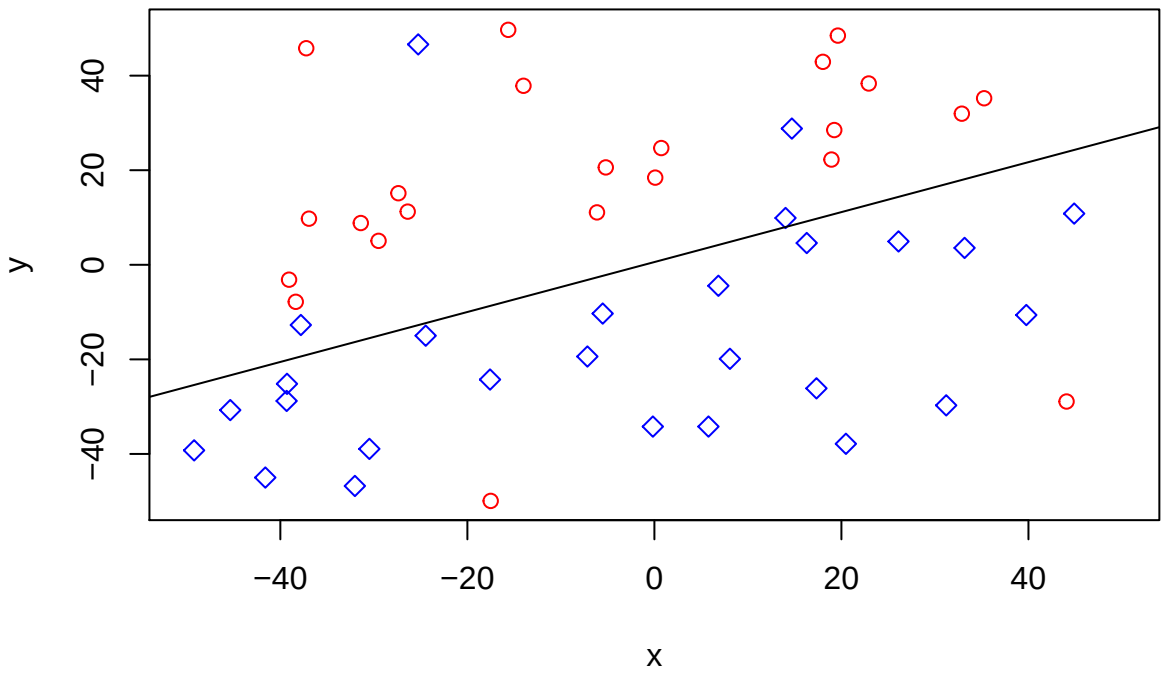

```
## [1] "Iteracion:  2"
## [1] "Porcentaje de fallos:  0.14"
```



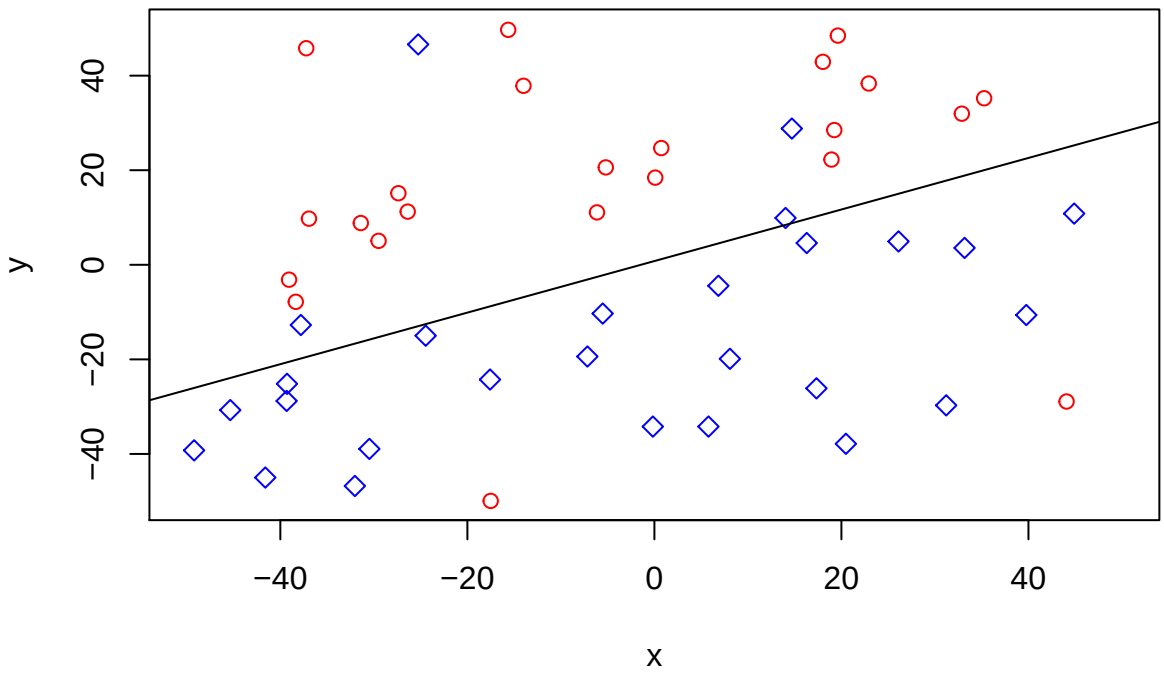

```
## [1] "Iteracion:  3"
## [1] "Porcentaje de fallos:  0.22"
```




```
## [1] "Iteracion:  4"
## [1] "Porcentaje de fallos:  0.12"
```




```
## [1] "Iteracion:  5"
## [1] "Porcentaje de fallos:  0.12"
```



Ejercicio 6

Para mejorar el algoritmo PLA, lo que vamos a almacenar siempre la mejor solución por la que hayamos pasado. Teniendo esto, nuestro PLA funcionará como una función de busqueda, que explora el espacio de soluciones(que es infinito), y nos devuelve el mejor valor que ha encontrado en su exploración.

```
ajusta_PLA_MOD <- function(datos, label, max_iter, vini){
  cambio = TRUE
  coef = vini
  iteracion = 0
  num_datos = length(label)+1
  mejor_error = 1
  mejor_coef = c(0,0,0)

  while(cambio && iteracion < max_iter){
    i=1
    cambio = FALSE

    while(i<num_datos){

      if(sign(t(c(datos[i,],1)) %*% coef) != label[i]){
        coef <- coef + label[i]*c(datos[i,],1)
        cambio = TRUE

        error <- mean(abs(label-sign(datos[,1]*coef[1]+ datos[,2]*coef[2] + coef[3])))*0.5
        if(mejor_error > error){
```

```
          mejor_error <- error
          mejor_coef <- coef
        }
      }
      i <- i+1
    }

    iteracion <- iteracion + 1
  }

  return(list(mejor_coef,iteracion))
}
```

Y ahora, lo usamos para las 4 funciones indicadas.

```
## [1] "Funcion 1"
```

```
## [1] "Porcentaje de fallos:  0.18"
```

```
## [1] "Funcion 2"
```

```
## [1] "Porcentaje de fallos:  0.2"
```

```
## [1] "Funcion 3"
```

```
## [1] "Porcentaje de fallos:  0.06"
```

```
## [1] "Funcion 4"
```

```
## [1] "Porcentaje de fallos:  0.04"
```

Destacar que, para la función 1, usada en el ejercicio 4 de esta sección, se ha conseguido una gran mejora, con el mismo número de iteraciones, gracias a la nueva versión del algoritmo PLA.

Apartado 3: Regresión Lineal

Ejercicio 1

Carga de datos.

```
rawdata.num <- read.table("~/AA/data/zip.train", quote="\"", comment.char="", stringsAsFactors=FALSE)
```

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

Ejercicio 2

Guardar solo los 1 y 5.

Si vemos los datos que hemos guardado anteriormente, vemos como la primera columna contiene las etiquetas, y las demás 256 columnas, los datos.

Por tanto, para tomar dichos datos, iteramos por la primera columna, buscando por los valores 1 y 5, y luego almacenando las demás columnas en forma de matriz.

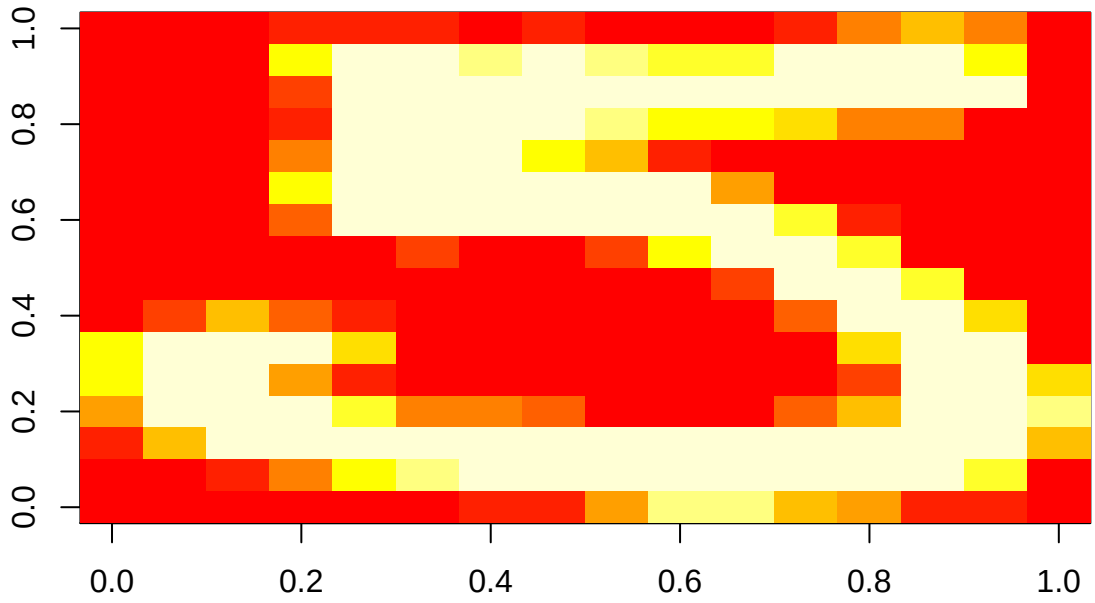
```
rawdata.5 <- rawdata.num[rawdata.num[,1] == 5,2:257]
rawdata.1 <- rawdata.num[rawdata.num[,1] == 1,2:257]

data.5 = data.matrix(rawdata.5)
data.1 = data.matrix(rawdata.1)
```

Vamos a guardar los datos como dos grandes matrices, pero nos va a ser necesario más adelante transformar sus filas en matrices. Para ello, utilizamos la siguiente función, que de paso normaliza los datos entre 0 y 1.

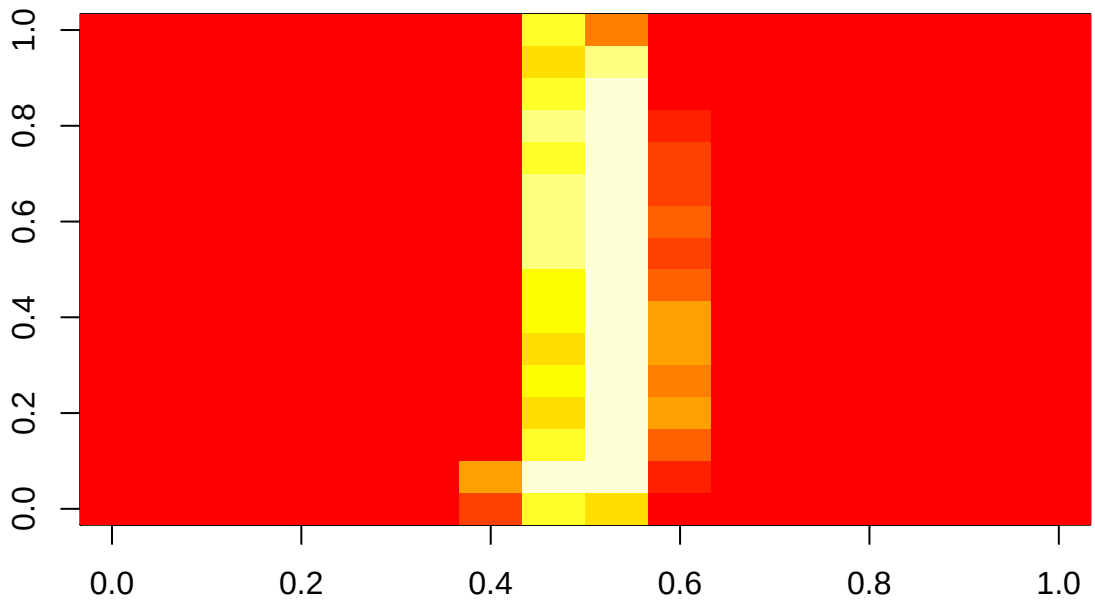
```
trans.matriz <- function(data){
  return(matrix((data*0.5)+0.5,16))
}
```

Finalmente, para mostrar las imágenes usamos la función *image* y nos ayudamos de nuestra función *trans.matriz*. Por la manera de usar los datos de *image* nos vemos obligados a realizar una inversión del orden de las columnas, para poder ver las imágenes en una orientación correcta.

```
image(trans.matriz(data.5[1,])[,16:1])
```



```
image(trans.matriz(data.1[1,])[,16:1])
```



Ejercicio 3

La función que nos devuelve la intensidad media no necesita ser realizada, pues ya disponemos de la función *mean* de R. La función que nos da la simetría sí tiene que realizarse. En la función, transformamos los datos a una matriz, para después iterar por las columnas en ambas direcciones, y restándolas. Luego las sumamos en valor absoluto y le cambiamos el signo.

El código que implementa dicha operación es:

```
simetria <- function(data){
  mat <- trans.matriz(data)
  return(-sum(abs(mat[,1:16] - mat[,16:1])))
}
```

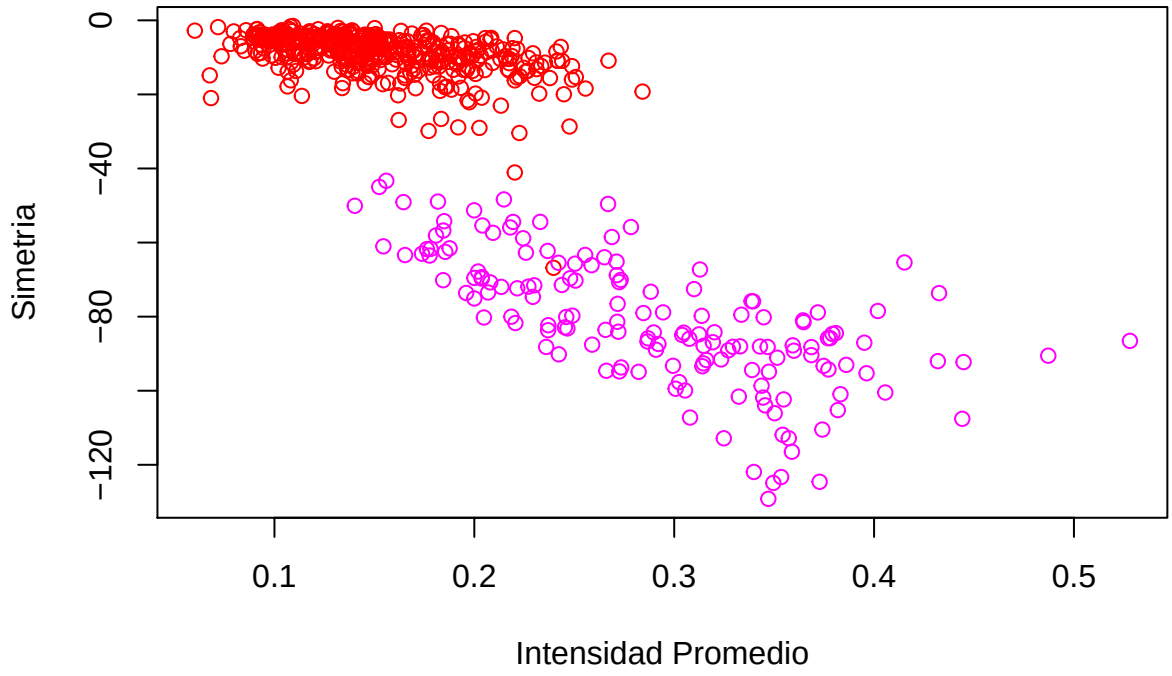
Ahora, aplicaremos a cada fila de la matriz ambas funciones, usando para ello la función *apply* de R. Tras calcular los descriptores de cada clase, los juntamos usando *rbind* y creamos un vector con las etiquetas.

```
descriptores.1 <- t(apply(data.1,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores.5 <- t(apply(data.5,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores <- rbind(descriptores.1,descriptores.5)
etiquetas <- rep(c(1,5), c(nrow(descriptores.1),nrow(descriptores.5)))
```

Ejercicio 4

Simplemente, tomamos los descriptores como datos de entrada para *plot* y usamos etiquetas para los colores (se añade un +1 para obtener colores más diferenciables).

```
plot(descriptores, col = etiquetas+1, xlab = "Intensidad Promedio", ylab = "Simetria")
```



Ejercicio 5

Para crear la función que realiza la regresión lineal, vamos a realizar los siguientes pasos:

1. Calcular SVD de los datos.
2. Calcular la inversa de la matriz diagonal obtenida con el SVD.
3. Calcular la pseudoinversa de los datos.
4. Multiplicar la pseudoinversa por las etiquetas.

Aquí tenemos el código que lo implementa:

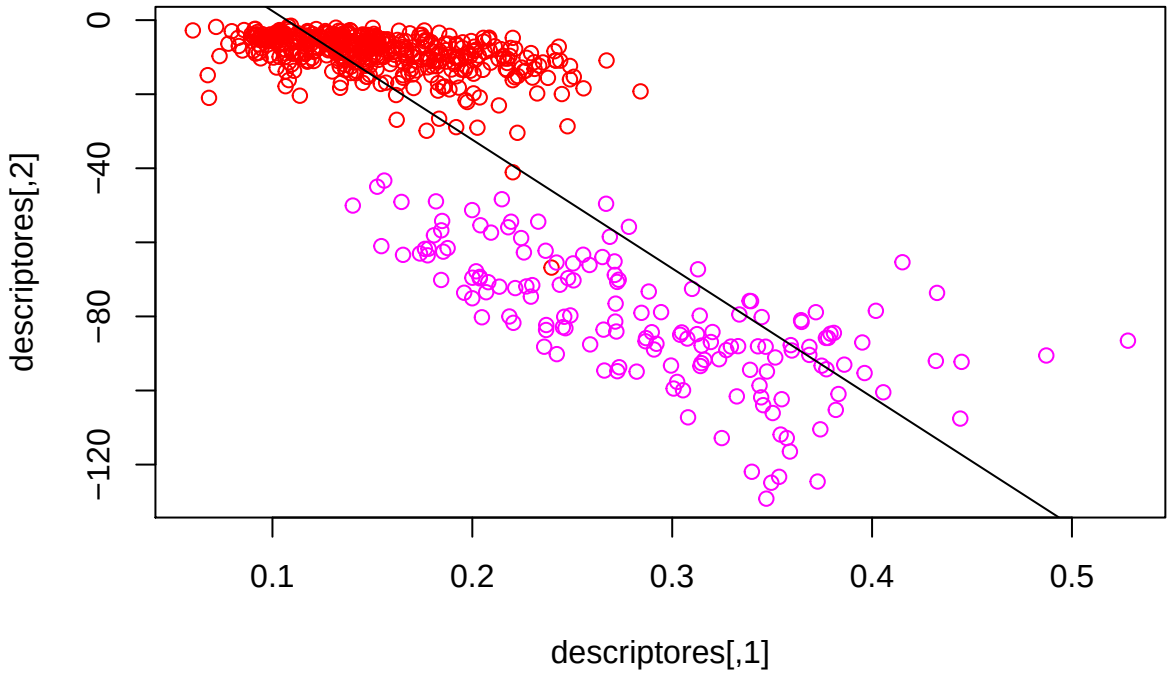
```
Regress_lin <- function(datos,label){
  s <- svd(datos)
  D <- diag(s$d)
  Dinv <- solve(D)

  p.inversa <- (s$v %*% Dinv %*% Dinv %*% t(s$v)) %*% t(datos)
```

```
  return(p.inversa%*%label)
}
```

Ejercicio 6

Para realizar la regresión lineal de los datos, tomamos la intensidad como datos, y la simetría como etiquetas. Entonces obtendremos una línea que, para una intensidad dada, nos tratará de estimar la simetría.

El resultado es el siguiente:



Como podemos ver, funciona correctamente, dandonos una línea en la dirección que parece correcta, pero algo elevada con respecto a la nube de puntos rosa, atraída por los puntos de la parte superior (puntos rojos).

Ejercicio 7

En este apartado, se pide clasificar unos datos usando la regresión lineal. Para ello, los datos serán la entrada y su clasificación, en forma de 1 y -1, sus etiquetas en el algoritmo.

La función que vamos a estudiar en este caso es una recta.

Veamos como se comportan en los siguientes casos.

a

Error en la muestra, con 100 datos. Repetido 1000 veces.

```
## [1] "Error medio:  0.0401100000000001"
```

b

Error fuera de la muestra. 100 datos para realizar la regresión, 1000 para calcular el error. Repetido 1000 veces.

```
## [1] 0.047761
```

Para entender este resultado, hay que compararlo con el de **a**. Como podríamos esperar, este error es mayor, por calcular el error fuera de la muestra y no dentro. Además, el número de datos de los que aprendemos es bastante pequeño comparado con los que tenemos que clasificar después, lo que complica más aún la tarea a nuestra regresión lineal.

Además, en estos casos el PLA nos daría siempre aciertos, pues son linealmente separables.

Pero hay que destacar un detalle, los dos valores se parecen mucho y son bastante bajos, además que su cálculo es directo, y por tanto bastante más rápido que el PLA. Esto nos dice que, pese a ser más preciso el PLA en este caso conceto, nos puede interesar en alguna ocasión la regresión, por ser más rápida y suficientemente buena.

c

Iteraciones necesarias para ajustar PLA, usando la regresión como punto de partida.

Para poder comparar los resultados, vamos a calcular la media de iteraciones del PLA que nos pide el ejercicio y además, la media cuando el PLA parte sin ninguna información, es decir, con coeficientes iniciales (0,0,0).

```
## [1] "Iteraciones con regresión como partida"
```

```
## [1] 7.162
```

```
## [1] "Iteraciones sin información de partida"
```

```
## [1] 11.231
```

Como podemos ver, reducimos el número de iteraciones utilizando los datos de la regresión. Esto podría usarse para acelerar el algoritmo del PLA en situaciones donde sea necesario.

Ejercicio 8

Vamos a realizar algo parecido al ejercicio anterior, pero en este caso la función no será una línea, si no una función cuadrática:

$$f(x, y) = dign(x^2 + y^2 - 25)$$

a

Error dentro de la muestra, pero clasificando con una función lineal. Añadimos un 10% de ruido.

```
## [1] "Error medio: "
```

```
## [1] 0.256954
```

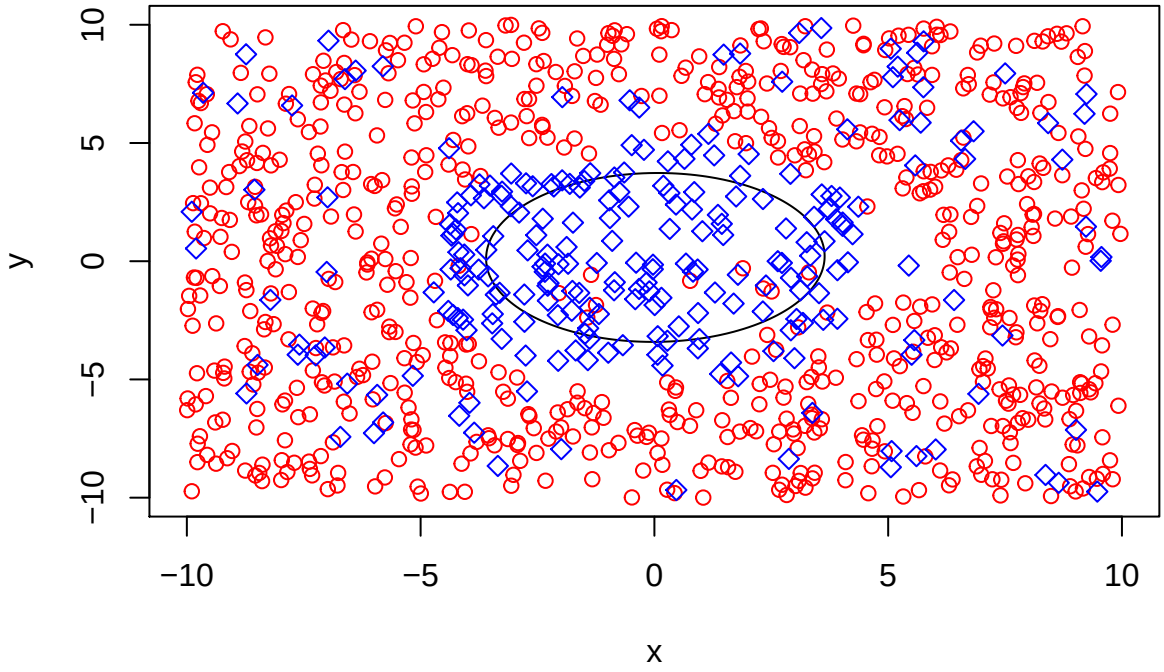
El error obtenido es muy grande. Esto es lógico, pues estamos tratando clasificar una función cuadrática mediante una línea.

b

Ahora, consideramos también los datos $x * y$, x^2 y y^2. Realizamos una unica iteración y mostramos los resultados

```
## [1] "Error medio: "
```

```
## [1] 0.177
```



El error se ha reducido mucho con respecto al intento anterior. Al tener más parámetros que ajustar, y ser estos de orden cuadrático, podemos clasificar mucho mejor. Además, si tenemos en cuenta que hay un 10% de error añadido, el ajuste obtenido es bastante bueno.

c

Ahora repetimos el experimento anterior, pero calculamos el error con puntos de fuera de la muestra.

```
## [1] "Error medio: "
```

```
## [1] 0.068994
```

Al no tener errores los datos que usamos para calcular el error, este baja mucho.

Gracias al añadido de los términos cuadráticos, hemos mejorado el error, a un 7% de media, lo que es un buen resultado, más teniendo en cuenta que estamos viendo el error cometido fuera de la muestra.