Trabajo 1

Francisco Javier Caracuel Beltrán

27 de Marzo de 2017

A continuación, se detalla el resultado del trabajo 1 de Aprendizaje Automático. Este documento incluye todo el código utilizado en el archivo *trabajo1.R*, así como los comentarios añadidos durante su creación.

Se establece el directorio de trabajo

```
setwd("/mnt/A0DADA47DADA18FC/Fran/Universidad/3º/2 Cuatrimestre/AA/Mis prácticas/Práctica 1/datos")
```

Se establece la semilla para la generación de datos aleatorios

```
set.seed(4)
```

1. Ejercicio sobre la complejidad de H y el ruido.

Se deben establecer las funciones que se van a utilizar durante el trabajo.

```
########################################
# Función simula_unif()
# Genera una matriz de "N" puntos y dimensión "dim" con valores comprendidos en
# "rango"
# Por defecto, N = 100, dim = 2, rango = [0,1]
#
simula_unif = function(N=100, dim=2, rango=c(0,1)){

  m = matrix(runif(N*dim, min=rango[1], max=rango[2]),
             nrow=N, ncol=dim, byrow=T)

  m

}

########################################
# Función simula_gaus()
# Genera una matriz de "N" puntos con dimensión "dim". Contiene números
# aleatorios gaussianos de media 0 y desviación típica dada en "sigma".
#
simula_gaus = function(N=2, dim=2, sigma){

  # Si no existe sigma, se termina la ejecución de la función
  if (missing(sigma))
    stop("Debe dar un vector de varianzas")

  # Para la generación se usa sd, y no la varianza
  sigma = sqrt(sigma)

  # Si la dimensión no es igual a la longitud del vector sigma, se termina
  # la ejecución de la función
  if(dim != length(sigma))
    stop ("El numero de varianzas es distinto de la dimensión")
```

```
  # Se crea una función que genera 1 muestra con las desviaciones especificadas
  list_simula_gauss = function(){
                        rnorm(dim, sd = sigma)
                      }

  # Repite N veces, list_simula_gauss y se hace la traspuesta
  m = t(replicate(N, list_simula_gauss()))

  m

}

########################################
# Función simula_recta()
# Devuelve la pendiente y el punto de corte de una recta dando un intervalo "c"
#
simula_recta = function(intervalo=c(-1,1), visible=F){

  # Se generan 2 puntos
  puntos = simula_unif(2, 2, intervalo)

  # Se calcula la pendiente
  a = (puntos[1,2] - puntos[2,2]) / (puntos[1,1] - puntos[2,1])

  # Se calcula el punto de corte
  b = puntos[1,2] - a*puntos[1,1]

  # Si se ha recibido que sea visible, pinta la recta y los 2 puntos
  if (visible) {

    # Si no está abierta la gráfica, se dibuja con plot
    if (dev.cur()==1)
      plot(1, type="n", xlim=intervalo, ylim=intervalo)

    # Pinta de color verde los puntos y la recta
    points(puntos,col=3)
    abline(b,a,col=3)

  }

  # Se devuelve el par "pendiente" y el "punto de corte"
  c(a,b)

}

########################################
# Función pintar_frontera()
# Pinta la frontera de una función. Se le pueden añadir puntos y etiquetas.
# Se ha añadido "add=T" a "contour" para permitir que se dibuje encima de
# la gráfica que ya haya dibujada antes.
#
pintar_frontera = function(f, rango=c(-50,50)) {
```

```
  x = y = seq(rango[1], rango[2], length.out = 100)

  z = outer(x, y, FUN=f)

  # Si no está iniciado el plot, se inicia
  if (dev.cur()==1)
    plot(1, type="n", xlim=rango, ylim=rango)

  # Se pinta la frontera
  contour(x, y, z, levels = 0, drawlabels = FALSE,
          xlim = rango, ylim = rango, xlab = "x", ylab = "y",
          add = T, col=3)

}
```

Apartado 1. Dibujar una gráfica con la nube de puntos de salida correspondiente.

En este apartado se pide generar dos muestras utilizando dos de las funciones que han sido implementadas anteriormente y dibujarlas en pantalla. Para dibujarlas se hace uso de la función `plot()`, llamando directamente a las funciones, que devolverán su respectiva muestra.

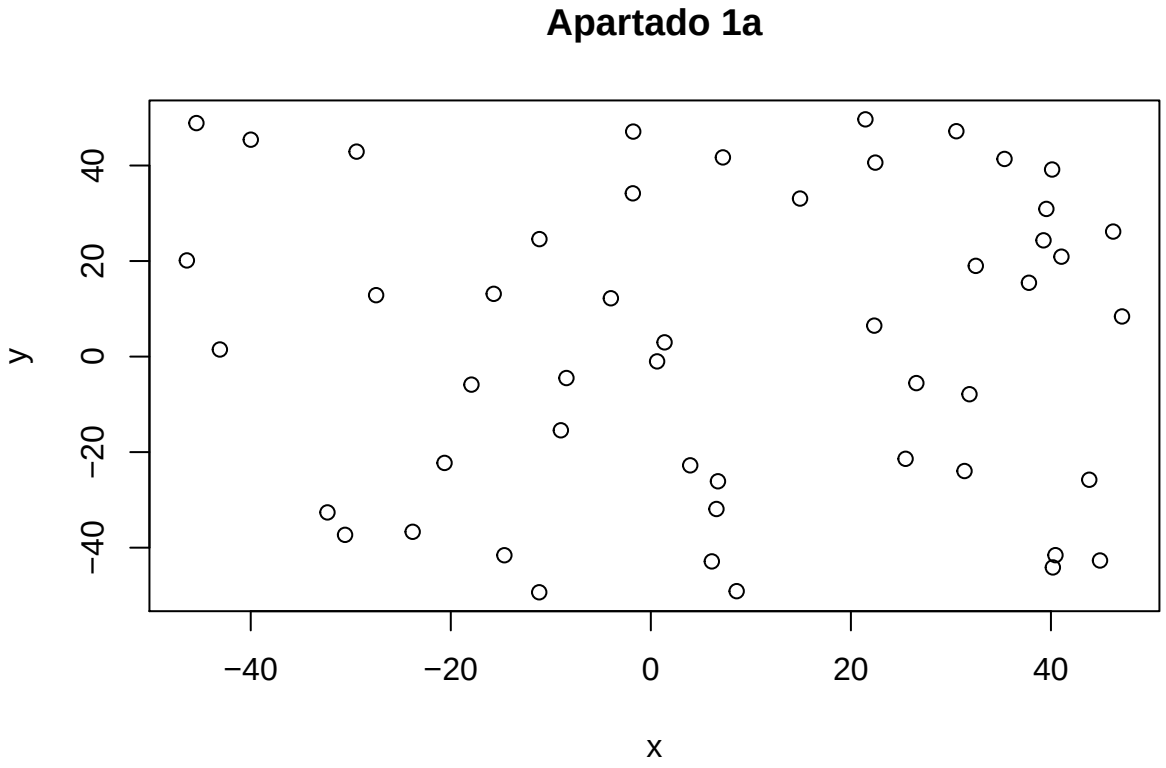
Se van a definir las variables que se van a utilizar en este apartado

```
# Variables que se van a utilizar
N = 50
dim = 2
rango = c(-50, 50)
sigma = c(5,7)
```

a)

Se dibujan los datos que genera directamente la función simula_unif(). Como no pide nada más, se utiliza directamente la función como entrada de datos para la función plot y se le asigna la etiqueta a los ejes de coordenadas y a la gráfica.

```
# a. Dibujar simula_unif
plot(simula_unif(N, dim, rango), main="Apartado 1a", xlab="x",
     ylab="y")
```

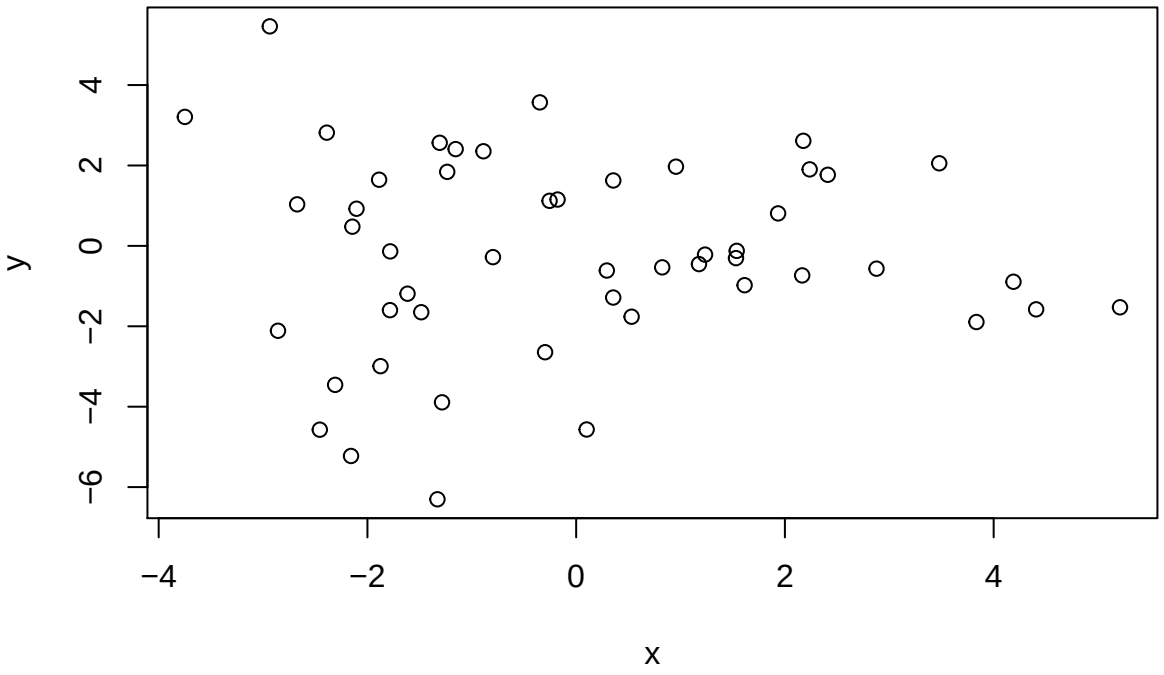


b)

Se dibujan los datos que genera directamente la función simula_gauss(). Como no pide nada más, se utiliza directamente la función como entrada de datos para la función plot y se le asigna la etiqueta a los ejes de coordenadas y a la gráfica.

```
# b. Dibujar simula_gaus
plot(simula_gaus(N, dim, sigma), main="Apartado 1b", xlab="x",
     ylab="y")
```

Apartado 1b



Apartado 2. Con ayuda de la función simula_unif() generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo...

a)

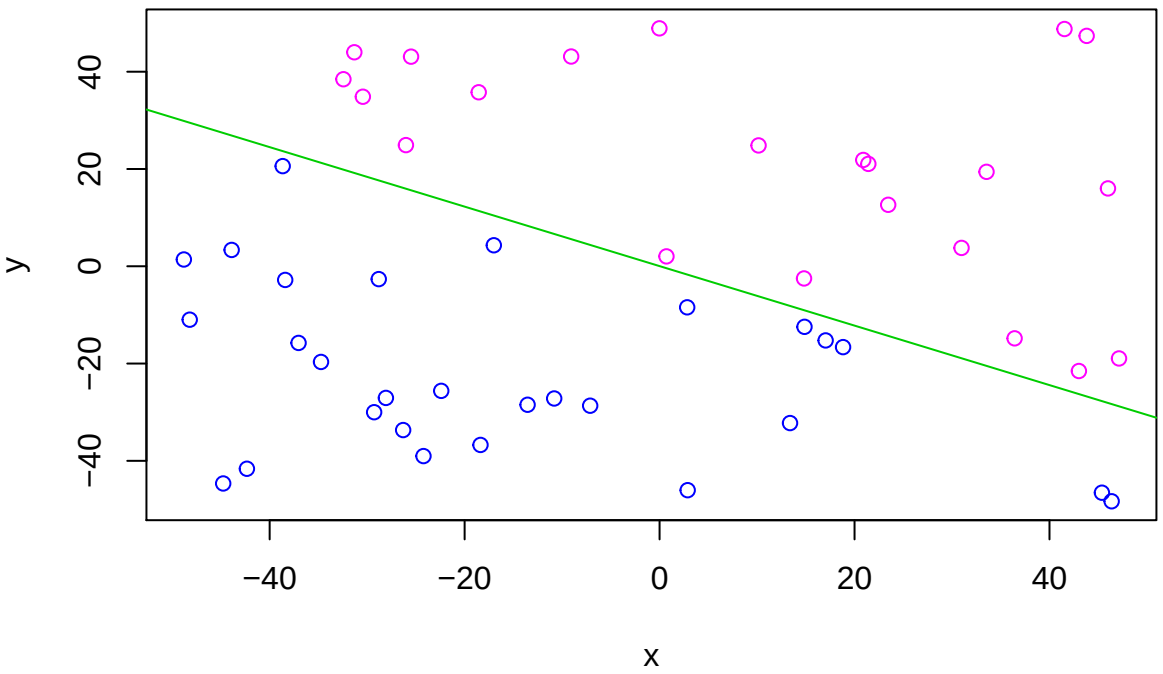
Se crea la función f(x, y) = y - ax - b, para que nos indique la etiqueta que va a tener un determinado punto de la muestra

```
########################################
# getValueF2a() -> "get" el "value" de la "F" definida en el apartado 2a
# Función que devuelve el resultado de f(x,y), dependiendo de la entrada
# "punto" con respecto a la recta "recta".
# f(x,y) = y - ax - b
getValueF2a = function(punto, recta){

  # Si falta algún parámetro se termina la función
  if(missing(punto) | missing(recta))
    stop("Faltan algunos parámetros")

  # f(x,y) = y - ax - b
  punto[2] - recta[1]*punto[1] - recta[2]

}
```

- Lo que se hace en este punto es:
  - Generar la muestra que se va a utilizar para el ejercicio con `simula_unif()`.
  - Generar la recta que se va a utilizar para separar los datos con `simula_recta()`.
  - Para poder separar la muestra, se le dará color a los puntos que estén a cada lado de la recta. Se aprovecha el signo que devuelve la función `getValueF2a()` y como `plot()`utiliza enteros para colorear los puntos, se le envía al atributo *col* el vector con los colores, sumándole 5 a cada elemento

del vector (ya que tenemos etiquetas con -1 y 1 y los colores en `plot()` comienzan en 1).

```
# Se generan los puntos que se van a utilizar para pintarlos
puntos2a = simula_unif(N, dim, rango)

# Se genera la recta que va a separar los puntos
datosRecta = simula_recta()

# Hay que crear un vector con los colores que va a tener cada punto,
# dependiendo de la recta que se tiene en "datosRecta".
# La función getValueF2a devuelve el resultado de aplicar la
# función f(x,y) = y - ax - b a cada par de puntos.
# Se utiliza apply para llamar a la función por cada tupla de la matriz
# y se obtiene un vector con el resultado.
# 1 en el segundo parámetro indica que se ejecute por filas.
valuesF2a = apply(puntos2a, 1, getValueF2a, datosRecta)

# Lo que interesa es tener etiquetado cada punto, para eso se crea
# un vector con -1, 1, teniendo en cuenta el signo de cada punto.
labels2a = sign(valuesF2a)

# Se dibujan los puntos creados.
# Se envía como atributo color el vector con las etiquetas de cada punto,
# sumándole 5 para que codifique el color.
# Al tener etiquetas -1 y 1, al menos, habría que sumar 2 para que detecte
# correctamente la codificación de colores que empieza en 1.

plot(puntos2a, main="Apartado 2a", xlab="x", ylab="y", col=labels2a+5)

# Se añade la recta creada.
# abline necesita recibir como primer parámetro el punto de corte y como
# segundo la pendiente.
abline(datosRecta[2], datosRecta[1], col=3)
```



b)

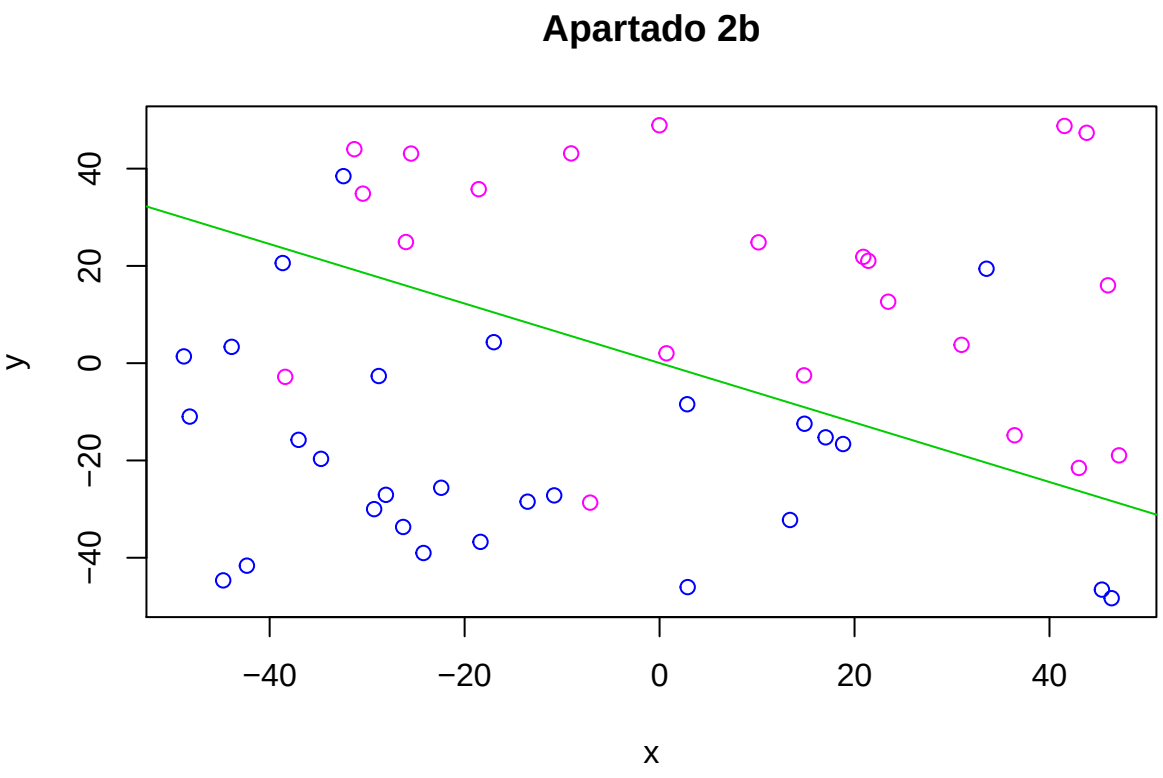
En este apartado se va a añadir ruido a la muestra.

Para hacerlo se van a crear dos funciones. `posChange()` devuelve las posiciones a las que hay que cambiar el signo de la etiqueta aplicándole un porcentaje y `changeLabel()` hace uso de la primera función aprovechando que devuelve un vector, para aplicarselo a la muestra.

Se hace uso de la función `which()`que devuelve un vector con las posiciones de las etiquetas que cumplen la condición que se especifica.

```
########################################
# Función que devuelve el "p"% de posiciones aleatorias de un vector dado
posChange = function(values, p=10){

  # Se genera un p% de posiciones que se van a cambiar sin repeteción
  change = sample(values, length(values)*0.01*p)

  change

}

########################################
# Función changeLabel
changeLabel = function(labels, p=10){

  # Se obtienen las posiciones de las etiquetas que son positivas y
  # negativas
  posPositiveLabel = which(labels>0)
  posNegativeLabel = which(labels<0)

  # Se obtienen las posiciones de las etiquetas que se van a cambiar
```

```r
  posPositiveChangeLabel = posChange(posPositiveLabel)
  posNegativeChangeLabel = posChange(posNegativeLabel)

  # Se hace una copia del vector de etiquetas
  changedLabels2b = labels2a

  # Se cambia el signo de aquellas etiquetas que se han modificado
  labels[posPositiveChangeLabel] <- -labels[posPositiveChangeLabel]
  labels[posNegativeChangeLabel] <- -labels[posNegativeChangeLabel]

  labels

}

changedLabels2b = changeLabel(labels2a)

# Se dibujan los puntos con las nuevas etiquetas con ruido
plot(puntos2a, main="Apartado 2b", xlab="x", ylab="y", col=changedLabels2b+5)

# Se añade la recta creada
abline(datosRecta[2], datosRecta[1], col=3)
```



Apartado 3. Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta...

Como se quiere pintar la muestra de distinta manera dependiendo de una serie de funciones, se definen primero todas las funciones.

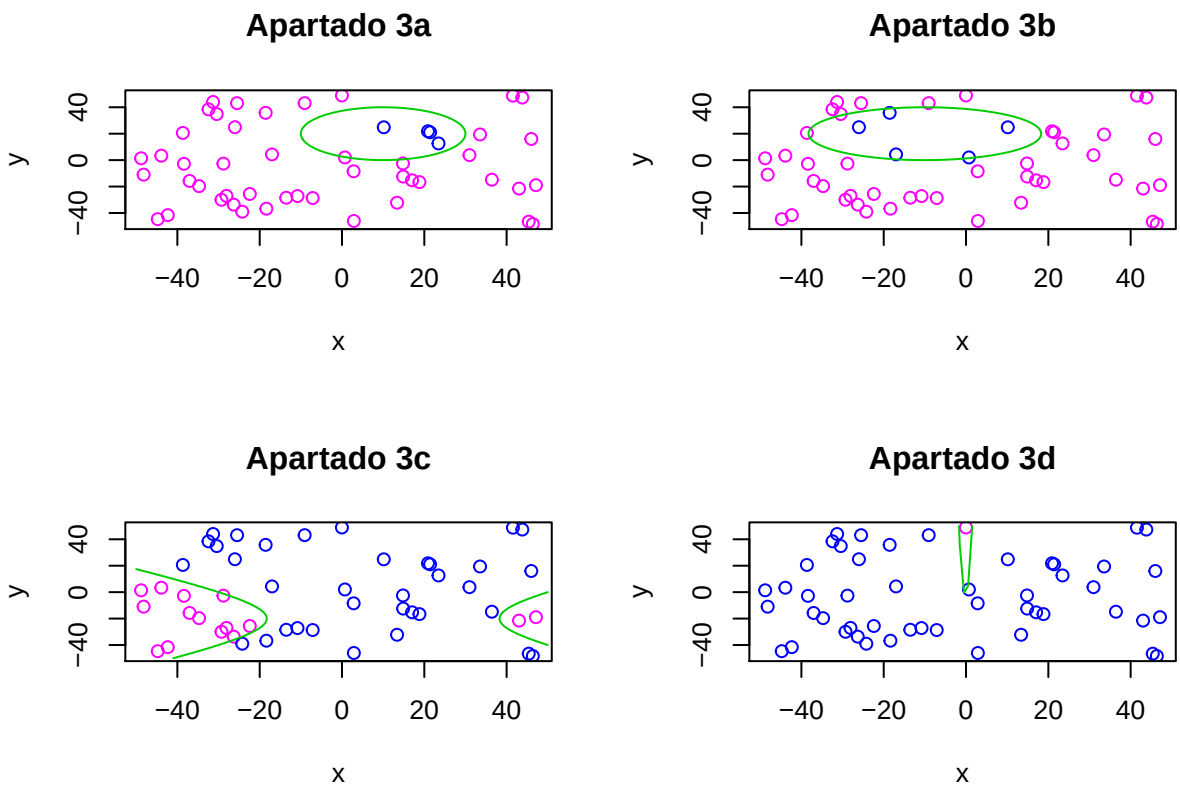
Cuando ya están definidas las funciones y partiendo de la muestra generada en el apartado anterior, se hace

uso de éstas y se crean las etiquetas que se van a utilizar para darle color, para enviárselas a `plot()` y que se dibuje gráficamente.

También se hace uso de la función `pintar_frontera()`, enviándole las funciones y nos añadirá a la gráfica la sección que separa unos datos de otros de la muestra.

```
########################################
# getValueF3a() -> "get" el "value" de la "F" definida en el apartado 3a
# Función que devuelve el resultado de f(x,y), dependiendo de x, y
# f(x,y) = (x-10)^2 + (y-20)^2 - 400
getValueF3a = function(x, y){

  # Si falta algún parámetro se termina la función
  if(missing(x) | missing(y))
    stop("Faltan los puntos")

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

}

########################################
# getValueF3b() -> "get" el "value" de la "F" definida en el apartado 3b
# Función que devuelve el resultado de f(x,y), dependiendo de x, y
# f(x,y) = 0.5*((x+10)^2) + (y-20)^2 - 400
getValueF3b = function(x, y){

  # Si falta algún parámetro se termina la función
  if(missing(x) | missing(y))
    stop("Faltan los puntos")

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

}

########################################
# getValueF3c() -> "get" el "value" de la "F" definida en el apartado 3c
# Función que devuelve el resultado de f(x,y), dependiendo de x, y
# f(x,y) = 0.5*((x-10)^2) - (y+20)^2 - 400
getValueF3c = function(x, y){

  # Si falta algún parámetro se termina la función
  if(missing(x) | missing(y))
    stop("Faltan los puntos")

  # 0.5*((x-10)^2) - (y+20)^2 - 400
  0.5*((x-10)^2) - (y+20)^2 - 400

}

########################################
# getValueF3d() -> "get" el "value" de la "F" definida en el apartado 3d
# Función que devuelve el resultado de f(x,y), dependiendo de x, y
# f(x,y) = y - 20*(x^2) - 5*x + 3
```

```r
getValueF3d = function(x, y){

  # Si falta algún parámetro se termina la función
  if(missing(x) | missing(y))
    stop("Faltan los puntos")

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

}

#######################################
# Se generan los valores de las distintas funciones definidas
valuesF3a = mapply(getValueF3a, puntos2a[,1], puntos2a[,2])
valuesF3b = mapply(getValueF3b, puntos2a[,1], puntos2a[,2])
valuesF3c = mapply(getValueF3c, puntos2a[,1], puntos2a[,2])
valuesF3d = mapply(getValueF3d, puntos2a[,1], puntos2a[,2])

# Lo que interesa es tener etiquetado cada punto, para eso se crea
# un vector con -1, 1, teniendo en cuenta el signo de cada punto y
# para cada función
labels3a = sign(valuesF3a)
labels3b = sign(valuesF3b)
labels3c = sign(valuesF3c)
labels3d = sign(valuesF3d)

# Se dibujan los puntos creados.
# Se envía como atributo color el vector con las etiquetas de cada punto,
# sumándole 4 para que codifique el color.
# Al tener etiquetas -1 y 1, al menos, habría que sumar 2 para que detecte
# correctamente la codificación de colores que empieza en 1.
# Se dibuja la gráfica y la frontera para cada función

# Se van a comparar las 4 funciones
par(mfrow=c(2,2))

plot(puntos2a, main="Apartado 3a", xlab="x", ylab="y", col=labels3a+5)
pintar_frontera(getValueF3a)

plot(puntos2a, main="Apartado 3b", xlab="x", ylab="y", col=labels3b+5)
pintar_frontera(getValueF3b)

plot(puntos2a, main="Apartado 3c", xlab="x", ylab="y", col=labels3c+5)
pintar_frontera(getValueF3c)

plot(puntos2a, main="Apartado 3d", xlab="x", ylab="y", col=labels3d+5)
pintar_frontera(getValueF3d)
```



- Interpretación de las distintas funciones:
  - 1ª y 2ª: es una elipse ya que define una línea curva cerrada en la que la suma de la distancia a dos puntos es constante.
  - 3ª: es una hipérbola, puesto que nos aparece una curva simétrica respecto de dos ejes perpendiculares entre sí, formada por dos ramas abiertas dirigidas en sentidos opuestos.
  - 4ª: es una parábola, porque se trata de una función cuadrática.

Conclusión:

Al utilizar funciones más complejas se ha conseguido mejorar la clasificación de la muestra. Con datos reales, es muy poco probable que se pueda separar la muestra con una función lineal. Estas cuatro nuevas funciones cuadráticas nos permiten clasificar una muestra más heterogénea. Esto nos indica, que para futuras clasificiones con muestras más complejas, se podrán utilizar funciones más avanzadas que las clasifiquen correctamente.

2. Ejercicio sobre el Algoritmo Perceptron.

Apartado 1. Implementar la función ajusta_PLA(datos, label, max_iter, vini) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA...

Para realizar esta función se han seguido los pasos que podemos encontrar en los apuntes de teoría. Para las secciones técnicas se ha consultado como realizar distintas operaciones. Las referencias son:

http://www.cleveralgorithms.com/nature-inspired/neural/perceptron.html

http://r-econ.blogspot.com.es/2011/04/trabajando-con-matrices-en-r.html

- EL funcionamiento del algoritmo PLA es:
  - Bucle externo que comprueba que no se superen un máximo de iteraciones dadas y que en toda una vuelta a la muestra no haya habido ningún cambio del perceptron.
  - Bucle interno de cada dato de la muestra:

* En esta sección se comprueba si el vector de pesos por el dato que se está inspeccionando en ese momento, es igual a la etiqueta que tendría que tener. Si la etiqueta coincide no se hace nada. En caso contrario se reajusta el perceptron teniendo en cuenta la etiqueta que realmente le corresponde y el punto.

```
#######################################
# Función ajusta_PLA
# Calcula el hiperplano solución a un problema de clasificación binaria
# usando el algoritmo PLA.
# Devuelve la pendiente, el punto de corte y el número de iteraciones
# que ha necesitado para conseguirlo
# Referencias:
# - Página 33. Sesión 1. AA.
# - http://www.cleveralgorithms.com/nature-inspired/neural/perceptron.html
# - http://r-econ.blogspot.com.es/2011/04/trabajando-con-matrices-en-r.html
ajusta_PLA = function(datos, label, max_iter, vini = c(0, 0, 0)){

  # w es el vector de pesos. Comenzará con el valor establecido en vini
  w = vini

  # Bool que indica si ha encontrado el valor óptimo de w en todas las
  # iteraciones
  finded = F

  # Contador del bucle
  i = 1

  # repeat
  # Se itera un número máximo de iteraciones o hasta que ya haya encontrado
  # el ajuste óptimo sin cambiar los pesos
  while(!finded & i<=max_iter){

    # Suponemos que va a encontrar el óptimo en esta iteración
    finded = T

    # for each xi € D do
    # Se recorren todas las filas de la matriz. Se puede coger el número de
    # filas de label o el número de filas de datos
    for(j in 1:length(label)){

      # Se guarda en "punto" los valores x, y de la fila que se está tratando.
      # Como el vector de pesos tiene longitud 3, para hacer la multiplicación
      # del punto con el vector de pesos, "punto" debe tener longitud 3.
      # Se añade a "punto", 1 como tercer valor del vector.
      punto = append(datos[j,], 1)

      # Se multiplica de manera matricial (%*%) el punto con w
      res = punto%*%w

      # Interesa el signo. Si vini se ha recibido con c(0, 0, 0), el resultado
      # de la multiplicación es 0. En este caso, interesa que
      # la etiqueta sea 1.
      sign = ifelse(sign(res) >= 0, 1, -1)
```

```
      # if: sign(wT*xi) != yi
      # Si la etiqueta obtenida no coincide con la real, se debe ajustar el
      # perceptron
      if(sign != label[j]){

        # w_new = w_old + yi*xi
        # Se actualiza el vector w con los pesos para ajustarlos
        w = w + punto*label[j]

        # Si no ha encontrado un óptimo, se indica que debe seguir
        finded = F

      }

    }

    # Se aumenta el contador
    i = i+1

  }

  # Hay que convertir el vector de pesos w en la pendiente y punto de corte
  # para dibujar la recta
  c(-w[1]/w[2], -w[3]/w[2], i-1)

}
```

Apartado 2. Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección.1..

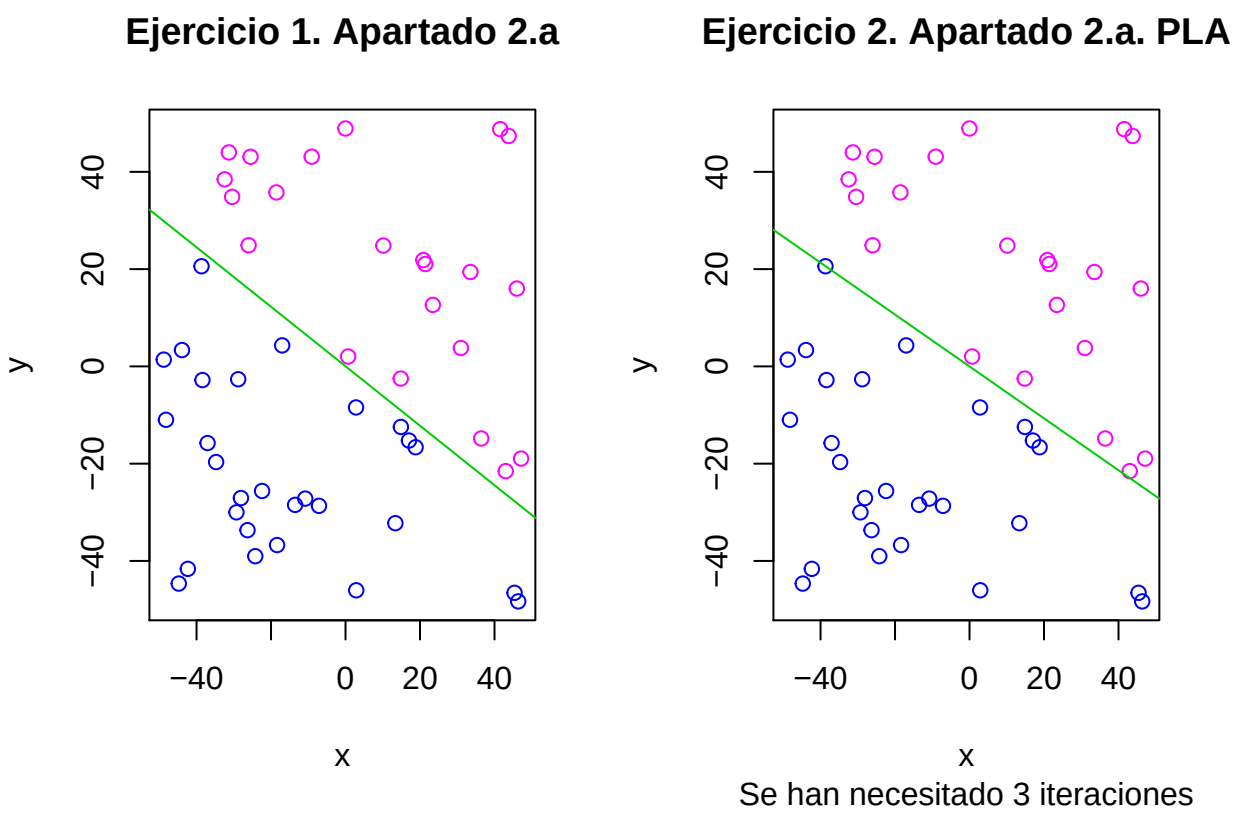
a - Dibujar la recta con el algoritmo PLA y el vector (0, 0, 0)

Para comprobar realmente que funciona el algoritmo `PLA`, se va a dibujar la muestra separada por la recta generada con la función creada en el ejercicio 1 y por la recta que se crea con el algoritmo `PLA`.

Puede no coincidir, ya que pueden separarse los datos con infinitas rectas. Lo que si debe aparecer es una clara diferencia entre los distintos puntos de la muestra (siempre que sean linealmente separables).

```
# Se establece de nuevo dos gráficas simultáneas
par(mfrow=c(1,2))

# Se dibujan los puntos generados en el ejercicio 1, apartado 2a
plot(puntos2a, main="Ejercicio 1. Apartado 2.a", xlab="x", ylab="y",
     col=labels2a+5)

# Se dibuja la recta generada anteriormente con la función simula_recta()
abline(datosRecta[2], datosRecta[1], col=3)

max_iter = 20

# Se guarda en "pla" la pendiente y punto de corte generado con el
# algoritmo PLA
pla = ajusta_PLA(puntos2a, labels2a, max_iter, c(0, 0, 0))

# Se vuelven a dibujar los puntos y se escribe el número de iteraciones
```

```
# que ha necesitado el algoritmo para ajustarlo
plot(puntos2a, main="Ejercicio 2. Apartado 2.a. PLA", xlab="x", ylab="y",
     col=labels2a+5,
     sub=paste(c("Se han necesitado", pla[3], "iteraciones"),
               collapse = " "))

# Se dibuja la recta generada con el algoritmo PLA
abline(pla[2], pla[1], col=3)
```



b

```
# Número de vectores de inicialización
numVini = 10

# rMatrix es una matriz con los 10 valores iniciales que se le van a
# pasar a la función ajusta_PLA
rMatrix = matrix(runif(3*numVini, 0, 1), ncol=3, nrow=numVini)

# En resMatrix se van a guardar todos los datos que se usan y los que
# se obtienen en ajusta_PLA para analizarlos
resMatrix = matrix(ncol=6, nrow=numVini+1)

# Se le pone nombre a las columnas para identificarlas fácilmente
dimnames(resMatrix) = list(c(), c("w1", "w2", "w3", "a", "b", "iter"))

# Se añade el vector inicial (0, 0, 0)
resMatrix[1, ] = c(0, 0, 0, pla[1], pla[2], pla[3])

# Me habría gustado usar apply para llamar a la función ajusta_PLA,
```

```
# pero tras varios problemas lo hago con for
# Se recorre el for, cogiendo en cada iteración una fila y guardando
# el número de iteraciones necesarias para hacer la media
mean = 0

for(i in 1:numVini){

  plaI = ajusta_PLA(puntos2a, labels2a, max_iter, rMatrix[i,])

  mean = mean + plaI[3]

  resMatrix[i+1, ] = append(rMatrix[i,], plaI)

}

# Se calcula la media
mean = mean/numVini

print(paste0("Ejercicio 2. Apartado 2 -> La media de las iteraciones",
        " de los 10 vectores iniciales es:", mean))
```

```
## [1] "Ejercicio 2. Apartado 2 -> La media de las iteraciones de los 10 vectores iniciales es:2.3"
```

```
# Con N = 50 y max_iter = 20
#
#             w1         w2         w3          a           b        iter
#[1,]  0.0000000000 0.0000000 0.00000000 -0.5334547 -0.012541598    3
#[2,]  0.2212906131 0.5660934 0.94331089 -0.6834018 -0.029143716    3
#[3,]  0.0254028270 0.7768967 0.97840129 -0.6965278  0.033494951    2
#[4,]  0.9363296099 0.4108098 0.14856894 -0.6484449 -0.003150081    2
#[5,]  0.6390121812 0.5344600 0.39796440 -0.7471951 -0.008259851    2
#[6,]  0.0077158227 0.4096629 0.49665395 -0.7044295  0.049890608    2
#[7,]  0.0002158575 0.2548703 0.85154796 -0.7078167  0.038309759    2
#[8,]  0.7110067829 0.4000610 0.09108752 -0.7507837 -0.001895833    2
#[9,]  0.1248708314 0.8836167 0.86627322 -0.6973490  0.037041663    2
#[10,] 0.8655358467 0.9806928 0.55825088 -0.5678295 -0.005566791    4
#[11,] 0.5826571663 0.3152000 0.11817979 -0.7494360 -0.002464065    2

# En caso de utilizar N = 720 y max_iter = 1000 y los mismos pesos
#
#             w1         w2         w3          a          b      iter
#[1,]  0.0000000000 0.0000000 0.00000000 0.2759559 0.2495314  192
#[2,]  0.2212906131 0.5660934 0.94331089 0.2760352 0.2555508  190
#[3,]  0.0254028270 0.7768967 0.97840129 0.2762378 0.2611720  197
#[4,]  0.9363296099 0.4108098 0.14856894 0.2756697 0.2613398  205
#[5,]  0.6390121812 0.5344600 0.39796440 0.2759744 0.2651572  206
#[6,]  0.0077158227 0.4096629 0.49665395 0.2756502 0.2659347  209
#[7,]  0.0002158575 0.2548703 0.85154796 0.2759754 0.2533193  190
#[8,]  0.7110067829 0.4000610 0.09108752 0.2758261 0.2643380  206
#[9,]  0.1248708314 0.8836167 0.86627322 0.2755643 0.2631614  209
#[10,] 0.8655358467 0.9806928 0.55825088 0.2760005 0.2674554  208
#[11,] 0.5826571663 0.3152000 0.11817979 0.2751508 0.2626408  200
```

Conclusión:

Con los datos originales: `N = 50, max_iter = 20`, el número de iteraciones que ha necesitado para converger con el vector inicial *(0, 0, 0)* ha sido **3**. El número medio de iteraciones que ha necesitado para converger con los *10 vectores aleatorios* ha sido **2.3**.

Existen muchas rectas que clasifiquen todos los puntos de la muestra, se podría decir que infinitas dentro de los reales.

Según los valores iniciales con los que se va a ajustar el perceptron, el ajuste de la recta actúa de manera diferente. Es por esto, que para algunos valores iniciales necesita *2* iteraciones completas, otros *3* y otros *4*.

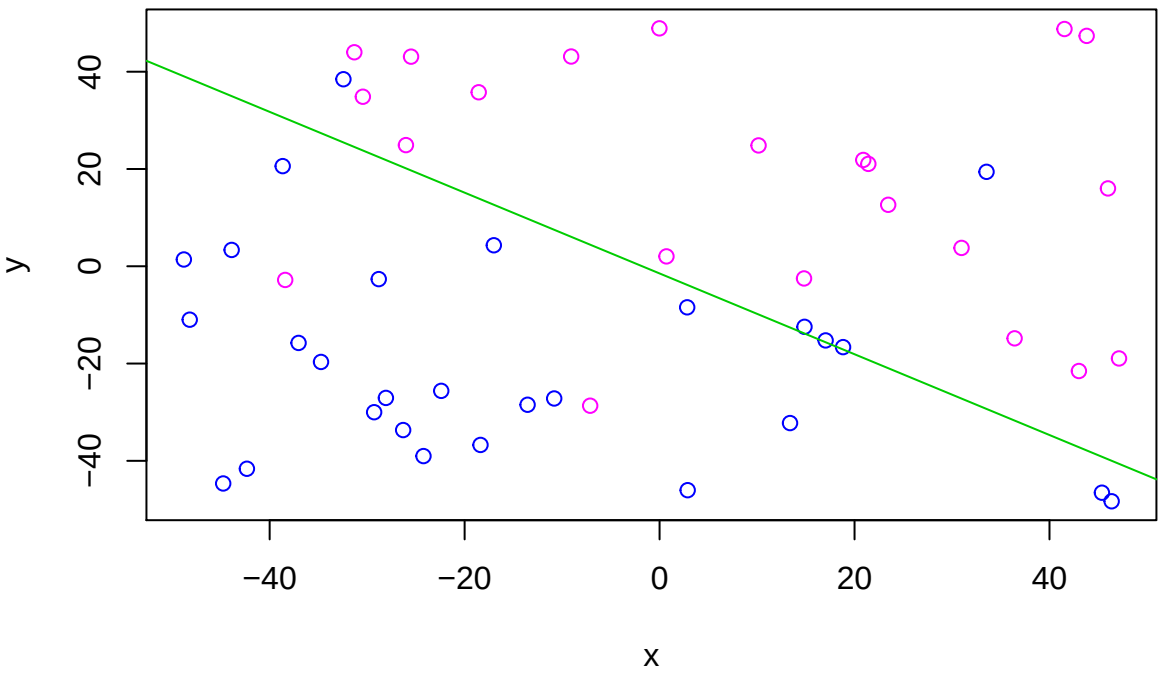
El número de iteraciones necesarias para unos valores iniciales u otros, dependen de estos valores iniciales, pero también de la muestra, ya que según sea ésta, el perceptron tardará más en ajustarse.

Se puede ver en la tabla con `N = 720`, como solo hay *2* valores iniciales que mejoren las iteraciones del vector *(0, 0, 0)*, mientras que con `N = 50`, son *8* los que lo mejoran.

La recta que se ha creado con `simula_recta()`, tampoco es la misma que la creada por el algoritmo `PLA`. Basándonos en la explicación anterior, con la función `simula_recta()` se ha obtenido una pendiente y punto de corte, y al haber infinitas rectas, el algoritmo `PLA` ha determinado otra que clasifica igual de bien.

Apartado 3. Hacer lo mismo que antes usando ahora los datos del apartado 2b de la sección.1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cuál y las razones para que ello ocurra.

```
# Se cambian las iteraciones máximas
max_iter = 1000

# Se guarda en "pla" la pendiente y punto de corte generado con el
# algoritmo PLA para las etiquetas del apartado 2b
pla = ajusta_PLA(puntos2a, changedLabels2b, max_iter, c(0, 0, 0))

# Se vuelven a dibujar los puntos y se escribe el número de iteraciones
# que ha necesitado el algoritmo para ajustarlo
plot(puntos2a, main="Ejercicio 2. Apartado 3. PLA", xlab="x", ylab="y",
     col=changedLabels2b+5,
     sub=paste(c("Se han necesitado", pla[3], "iteraciones"),
               collapse = " "))

# Se dibuja la recta generada con el algoritmo PLA
abline(pla[2], pla[1], col=3)
```



Conclusión:

El algoritmo `PLA` ha utilizado *1000* iteraciones. Incluso así no ha ajustado perfectamente el perceptron, principalmente porque no puede, ya que la muestra no se puede clasificar linealmente y muestra la recta que estaba calculando en el momento que ha llegado al límite de iteraciones.

Se utilice el número máximo de iteraciones que se utilice, en este caso, siempre va a consumirlas todas.

Tampoco importan los valores iniciales que reciba, ya que igualmente, en cada iteración, no encontrará la separación perfecta entre unas etiquetas y otras.

3. Ejercicio sobre Regresión Lineal

Apartado 1. Leemos datos...

Se hace uso del código que disponemos en `paraTrabajo1.R` para leer los archivos.

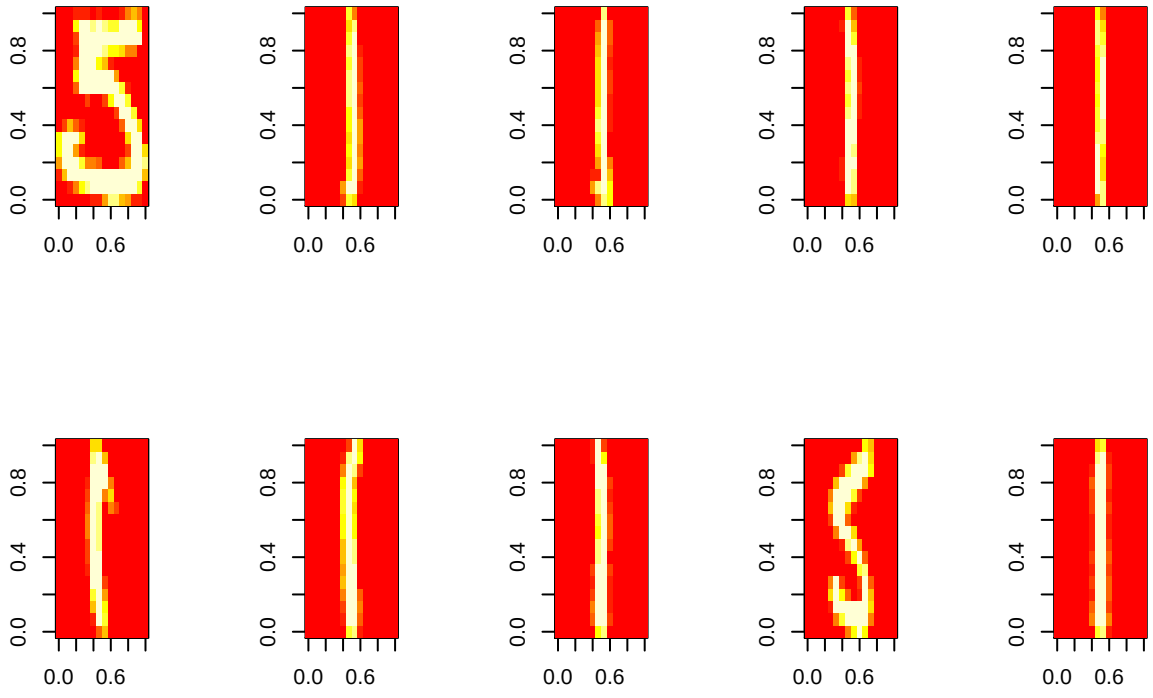
```
# Se guarda en digit.train los datos que se leen del fichero de entrenamiento
digit.train <- read.table("zip.train",
                          quote="\"", comment.char="", stringsAsFactors=FALSE)

# En digitos15 se guardan los datos de entrenamiento de los números 1 y 5
digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]

# En la primera columna de la matriz que tiene todos los datos de entrenamiento
# se indica el dígito al que corresponde. Se consigue un vector que indica en
# cada posición, el número correspondiente.
digitos = digitos15.train[,1]

# Se guarda el total de números 1 y 5 que hay
```

```
ndigitos = nrow(digitos15.train)

# Se retira la clase y se monta una matriz 3D: 599*16*16
grises = array(unlist(subset(digitos15.train, select=-V1)), c(ndigitos, 16, 16))

# Ya no es necesario tener las matrices leídas con los 1 y 5 y se eliminan
rm(digit.train)
rm(digitos15.train)
```

Una vez que se han leído las archivos y se tienen los números guardados en matrices, se utiliza la función `image()` que lee una matriz para mostrar una imagen de los datos que se encuentran en ella.

En este caso se van a mostrar *10* dígitos:

```
# Se guarda cuantas imágenes se quieren mostrar
numImageToShow = 10

# Se ajusta el plot para que muestre varios números a la vez
par(mfrow=c(2, 5))

# Se van a mostrar las "numImageToShow" primeras imágenes
for(i in 1:numImageToShow){

  # image necesita una matriz para mostrar la imagen
  # Se rota además, para mostrarla correctamente
  imagen = grises[i,,16:1]
  image(z=imagen)

}
```




```
# Etiquetas correspondientes a las "numImageToShow" primeras imágenes
digitos[1:numImageToShow]
```

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

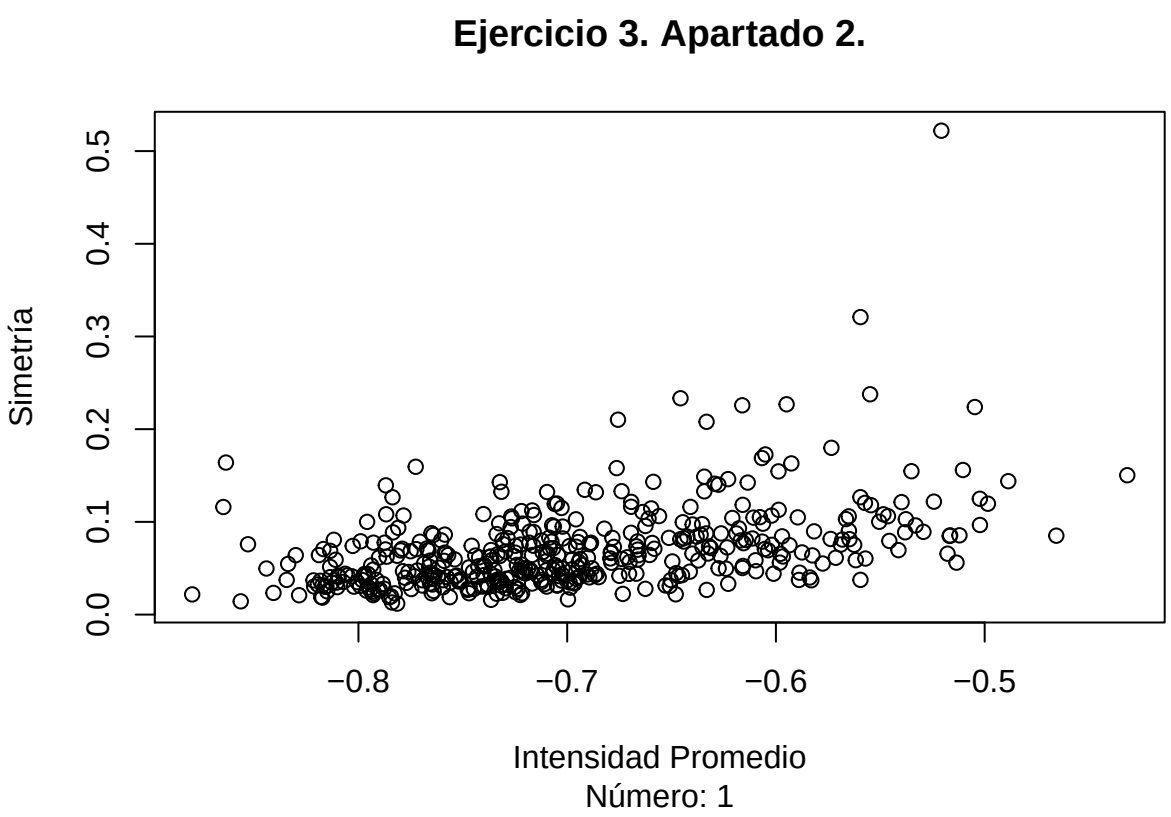
Apartado 2. De cada matriz de números (imagen) vamos a extraer dos características: a) su valor medio; y b) su grado de simetría vertical...

Se crea una función que devuelve la media y la simetría vertical de cada dígito.

En este apartado lo único que se hace es aplicar a todas las matrices de dígitos la función que se ha creado para mostrar después una gráfica con todos los puntos representados.

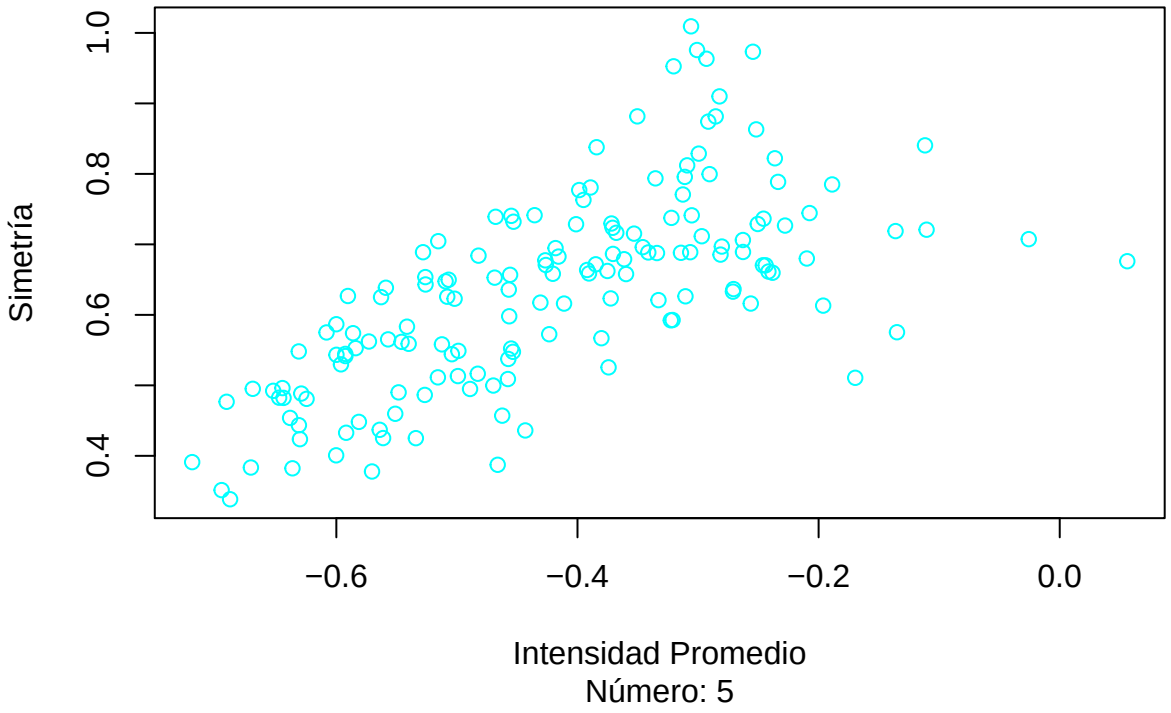
```
########################################
# Función getMeanSym()
# Función que dada una matriz devuelve su valor medio y el
# grado de simetría vertical que tiene
#
# Referencias:
# http://stackoverflow.com/questions/9135799/how-to-reverse-a-matrix-in-r
#
getMeanSym = function(matrix){

  # Se calcula la media de la matriz
  originalMatrixMean = mean(matrix)

  # Cálculo de la simetría vertical

  # a: se le da la vuelta a las columnas
  invertedMatrix = matrix[, ncol(matrix):1]

  # b: se calcula la diferencia entre la matriz original y la invertida
  matrix = matrix - invertedMatrix

  # c: se calcula la media global de los valores absolutos de la matriz
  invertedMatrixMean = mean(abs(matrix))

  c(originalMatrixMean, invertedMatrixMean)

}

# Se mostrará una sola gráfica a la vez
par(mfrow=c(1, 1))

# Se crea una matriz con 2 columnas para la media y la simetría y tantas
# filas como números 1 y 5 haya
matrixMeanSym = matrix(ncol = 2, nrow = ndigitos)

# Se le da nombre a las columnas
dimnames(matrixMeanSym) = list(c(), c("mean", "symmetry"))

# Se recorre la matriz 3d donde están todos los datos y se va creando
# la matriz con los resultados de los cálculos.
# He intentado hacerlo con apply por filas, obteniendo la matriz de cada
# número pero no conseguía reflejar los datos como debía
for(i in 1:ndigitos){

  matrixMeanSym[i, ] = getMeanSym(grises[i, , ])
```

```
}

# Gráfica con los 1
plot(matrixMeanSym[which(digitos==1), ],
    xlab="Intensidad Promedio", ylab="Simetría",
     main="Ejercicio 3. Apartado 2.", sub="Número: 1")
```



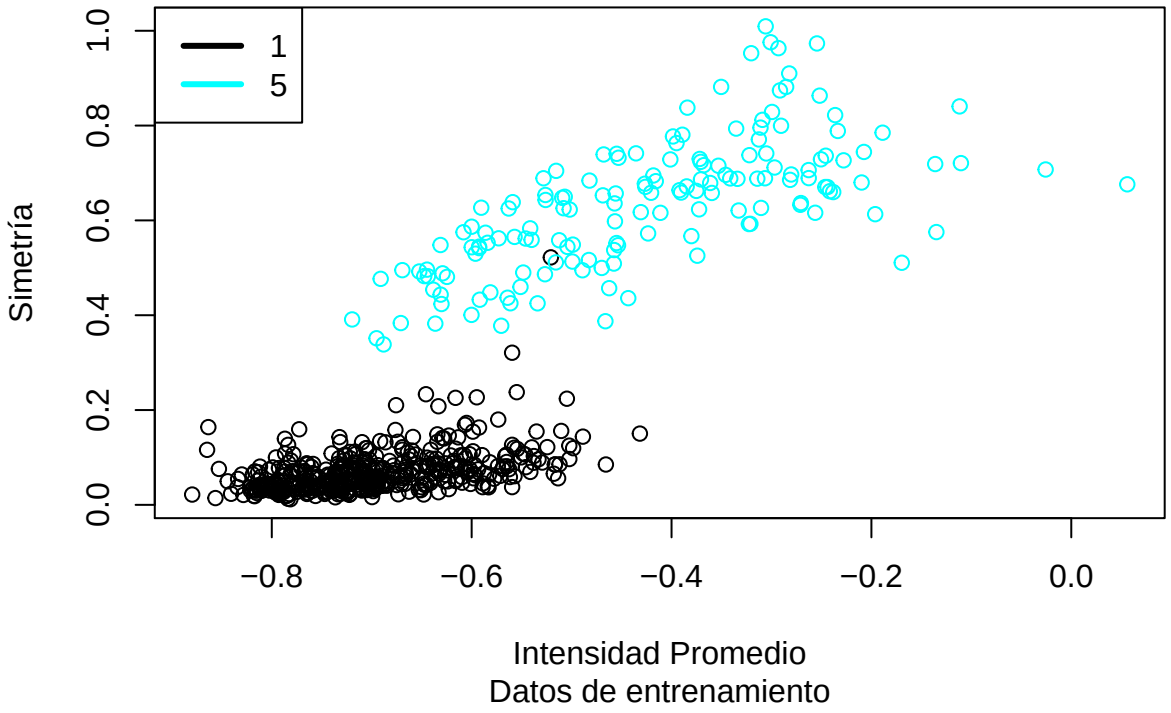

```
# Gráfica con los 5
plot(matrixMeanSym[which(digitos==5), ],
     xlab="Intensidad Promedio", ylab="Simetría", col=5,
     main="Ejercicio 3. Apartado 2.", sub="Número: 5")
```

Ejercicio 3. Apartado 2.




```
# Gráfica con los dos números a la vez
plot(matrixMeanSym, col=digitos, xlab="Intensidad Promedio",
     ylab="Simetría", main="Ejercicio 3. Apartado 2.",
     sub="Datos de entrenamiento")

# Se escribe en la gráfica a qué corresponde cada color
legend(x="topleft", legend=c("1","5"), col=c(1, 5), lwd=3)
```

Ejercicio 3. Apartado 2.



Apartado 3. Ajustar un modelo de regresion lineal usando el transformación SVD sobre los datos de (Intensidad promedio, Simetria) y pintar la solución obtenida...

Se va a realizar una regresión lineal, que se usará para separar los datos de la muestra generada en el apartado anterior de unos y cincos.

Para crear la función que hace la regresión lineal, se hace uso de los apuntes de clase y de la siguiente página: http://stackoverflow.com/questions/36234136/apply-svd-linear-regression-in-r

Se crea, además, una función que nos devolverá el porcentaje normalizado de los errores que ha habido al etiquetar los puntos de la muestra con `Regress_Lin()`.

Funciona de manera muy simple:

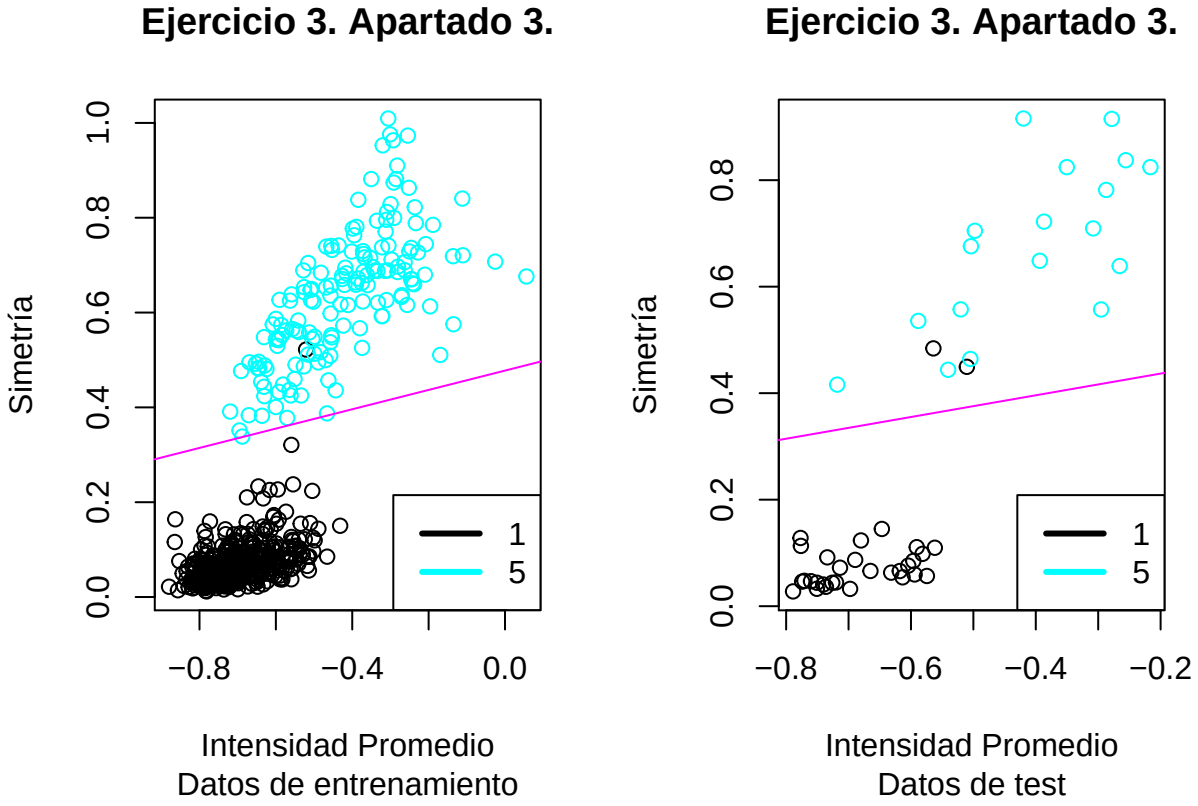
Recorre cada punto de la muestra, comprueba si según la regresión lineal tiene la etiqueta que debería y en caso contrario aumenta en uno el error. Para terminar devuelve *error/número de datos de la muestra.*

```
########################################
# Función Regress_Lin(datos, label)
#
# Referencias:
# - página 14-16, sesión 3.pdf
# - http://stackoverflow.com/questions/36234136/apply-svd-linear-regression-in-r
Regress_Lin = function(datos, label){

  # Primero se consigue SVD
  udv = svd(datos)

  # Se calcula su inversa. (xT*x)^-1 = v*d*d*vT
  # Como svd() devuelve $d, $u, $v, si se quiere acceder a "v", se hace con udv$v
```

```
  # v -> v
  # d -> diagonal d
  # uT -> traspuesta u
  datos.inverse = udv$v %*% (diag(1/udv$d))^2 %*% t(udv$v)

  # La pseudoinversa se calcula multiplicando la inversa calculada con la
  # traspuesta de la matriz original: (xT*X)^-1 * xT
  datos.pseudoinverse = datos.inverse %*% t(datos)

  # Se termina obteniendo los pesos, multiplicando la pseudoinversa con las
  # etiquetas que se recibe por parámetro
  w = datos.pseudoinverse %*% label

  w

}

########################################
# Función errorData()
# Calcula el error de una muestra de datos
errorData = function(data, label, w){

  # Se comienza sin ningún error
  error = 0

  # Se recorre todo el vector de datos
  for(i in 1:nrow(data)){

    # Si el signo que obtiene de multiplicar matricialmente el punto actual
    # con el vector de pesos, no coincide con la etiqueta que le corresponde,
    # se aumenta en 1 el error
    if(sign(data[i,]%*%w) != sign(label[i])){
      error = error + 1
    }

  }

  # Se calcula el porcentaje [0,1]
  error/nrow(data)

}

# Se copian las etiquetas de los números para cambiar los 5 por -1 y ajustarlas
# a Regress_lim con -1, 1
lD33 = digitos
lD33[which(lD33==5)] = -1

# Se calcula la regresión lineal con la matriz de medias y simetrias
# verticales y con las etiquetas de -1, 1 que representa los digitos
wLR33 = Regress_Lin(cbind(matrixMeanSym, 1), lD33)

# Se dibuja la línea que ajustaría el error mínimo
# "Se dibujará cuando ya se tengan calculados los datos de test para
```

```
# compararlas
#abline(-wLR33[3]/wLR33[2], -wLR33[1]/wLR33[2], col=6)

# Ahora se va a realizar el mismo proceso que se ha hecho para leer
# los datos de entrenamiento, pero con los datos de test
# Se guarda en digit.test los datos que se leen del fichero de test
digit.test <- read.table("zip.test",
                         quote="\"", comment.char="", stringsAsFactors=FALSE)

# En digitos15 se guardan los datos de test de los números 1 y 5
digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,]

# En la primera columna de la matriz que tiene todos los datos de test
# se indica el dígito al que corresponde. Se consigue un vector que indica en
# cada posición, el número correspondiente.
digitosTest = digitos15.test[,1]

# Se guarda el total de números 1 y 5 que hay
ndigitosTest = nrow(digitos15.test)

# Se retira la clase y se monta una matriz 3D: 599*16*16
grisesTest = array(unlist(subset(digitos15.test, select=-V1)), c(ndigitosTest, 16, 16))

# Ya no es necesario tener las matrices leídas con los 1 y 5 y se eliminan
rm(digit.test)
rm(digitos15.test)

# Se crea una matriz con 2 columnas para la media y la simetría y tantas
# filas como números 1 y 5 haya
matrixMeanSymTest = matrix(ncol = 2, nrow = ndigitosTest)

# Se le da nombre a las columnas
dimnames(matrixMeanSymTest) = list(c(), c("mean", "symmetry"))

# Se recorre la matriz 3d donde están todos los datos y se va creando
# la matriz con los resultados de los cálculos.
# He intentado hacerlo con apply por filas, obteniendo la matriz de cada
# número pero no conseguía reflejar los datos como debía
for(i in 1:ndigitosTest){

  matrixMeanSymTest[i, ] = getMeanSym(grisesTest[i, , ])

}

# Se copian las etiquetas de los números para cambiar los 5 por -1 y ajustarlas
# a Regress_lim con -1, 1
lD33Test = digitosTest
lD33Test[which(lD33Test==5)] = -1

# Se van a dibujar las dos gráficas a la vez
par(mfrow=c(1, 2));

# Gráfica con los dos números a la vez
```

```
plot(matrixMeanSym, col=digitos, xlab="Intensidad Promedio",
     ylab="Simetría", main="Ejercicio 3. Apartado 3.",
     sub="Datos de entrenamiento")

# Se escribe en la gráfica a qué corresponde cada color
legend(x="bottomright", legend=c("1","5"), col=c(1, 5), lwd=3)

# Se dibuja la recta calculada con Regress_lin para los datos de entrenamiento
abline(-wLR33[3]/wLR33[2], -wLR33[1]/wLR33[2], col=6)

# Gráfica con los dos números a la vez
plot(matrixMeanSymTest, col=digitosTest, xlab="Intensidad Promedio",
     ylab="Simetría", main="Ejercicio 3. Apartado 3.", sub="Datos de test")

# Se escribe en la gráfica a qué corresponde cada color
legend(x="bottomright", legend=c("1","5"), col=c(1, 5), lwd=3)

# Se dibuja la recta calculada con Regress_lin para los datos de entrenamiento
abline(-wLR33[3]/wLR33[2], -wLR33[1]/wLR33[2], col=6)
```



Se ha utilizado la misma recta generada por regresión para las dos gráficas, pero debido a que el conjunto de datos en ambas es diferente, puede parecer que tiene pendiente diferente.

```
# Se va a calcular Ein
Ein33 = errorData(cbind(matrixMeanSym, 1), lD33, wLR33);

# Se va a calcular Eout
Eout33 = errorData(cbind(matrixMeanSymTest, 1), lD33Test, wLR33);
```

```r
print("Ejercicio 3. Apartado 3.")
```

```
## [1] "Ejercicio 3. Apartado 3."
```

```r
print(paste0("El error Ein (datos de entrenamiento) es: ",
             round(Ein33*100, digits=3), "%"))
```

```
## [1] "El error Ein (datos de entrenamiento) es: 0.167%"
```

```r
print(paste0("El error Eout (datos de test)  es: ",
             round(Eout33*100, digits=3), "%"))
```

```
## [1] "El error Eout (datos de test)  es: 4.082%"
```

Conclusión: *Ein* es *0.167%* y *Eout 4.082%*. Esto prueba que la recta de regresión se ajusta mejor para los datos de entrenamiento que para los datos de test, ya que se ha calculado con los datos de entrenamiento.

Apartado 4. En este apartado exploramos como se transforman los errores E in y E out cuando aumentamos la complejidad del modelo lineal usado...

Se repetirá el mismo proceso que en el apartado 3, pero se va a utilizar una muestra aleatoria, generada con las funciones definidas al comienzo del trabajo. La función que etiqueta los puntos se define a continuación. Para el experimento 1 se utiliza un vector de características lineal y para el experimento 2 un vector de características no lineal.

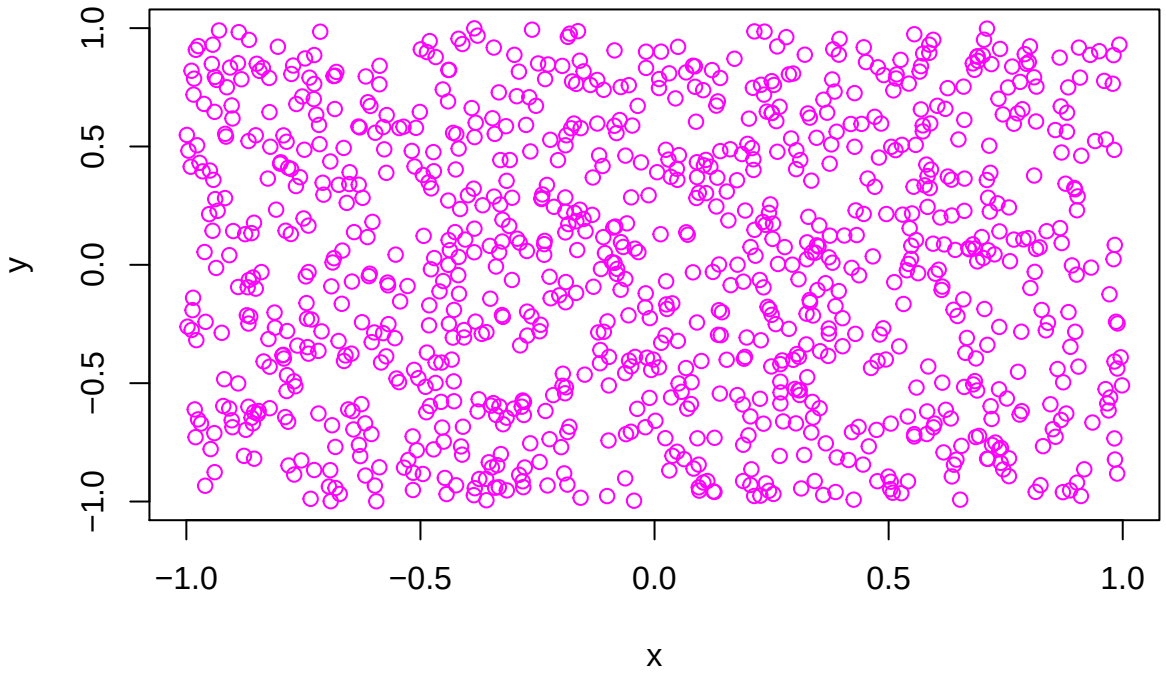
Experimento 1.

```r
########################################
# getValueF341() -> "get" el "value" de la "F" definida en el ejercicio 3,
# apartado 4, experimento 1.
# Función que devuelve el resultado de f(x,y), dependiendo de la entrada
# "punto" con respecto a la recta "recta".
# f(x, y) = sign((x+0.2)^2 + y^2 - 0.6)
getValueF341 = function(x, y){

  # Si falta algún parámetro se termina la función
  if(missing(x) | missing(y))
    stop("Faltan los puntos")

  # f(x, y) = sign((x+0.2)^2 + y^2 - 0.6)
  sign((x+0.2)^2 + y^2 - 0.6)

}
```

a

Simplemente se dibujan los puntos creados para el experimento 1.

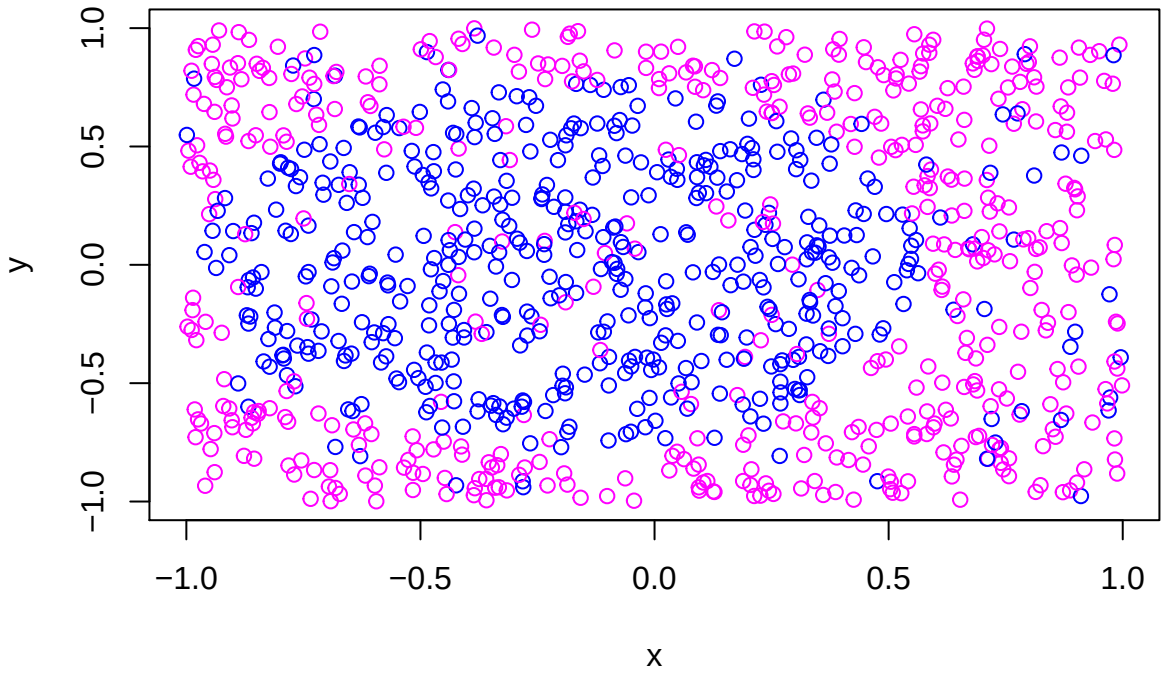
```r
experimento1 = simula_unif(1000, 2, c(-1, 1))

plot(experimento1, xlab="x", ylab = "y", main="Ejercicio 3. Apartado 4.",
     sub="Experimento 1. Subapartado a", col=6)
```



b

Como en apartados anteriores, se etiquetan los puntos de la muestra con ruido y se dibujan.

```
# Se generan las etiquetas correspondientes a la función establecida en este
# apartado
labelExperimento1 = mapply(getValueF341, experimento1[,1], experimento1[,2])

# Se obtienen las posiciones a las que se le va a cambiar el signo a las
# etiquetas para añadir ruido, por defecto es un 10%
posLabelExperimento1 = posChange(c(1:length(labelExperimento1)))

# Se cambia el signo para añadir ruido
labelExperimento1[posLabelExperimento1] = -labelExperimento1[posLabelExperimento1]

# Se dibujan los datos con las nuevas etiquetas
plot(experimento1, xlab="x", ylab = "y", main="Ejercicio 3. Apartado 4.",
     sub="Experimento 1. Subapartado b", col=labelExperimento1+5)
```



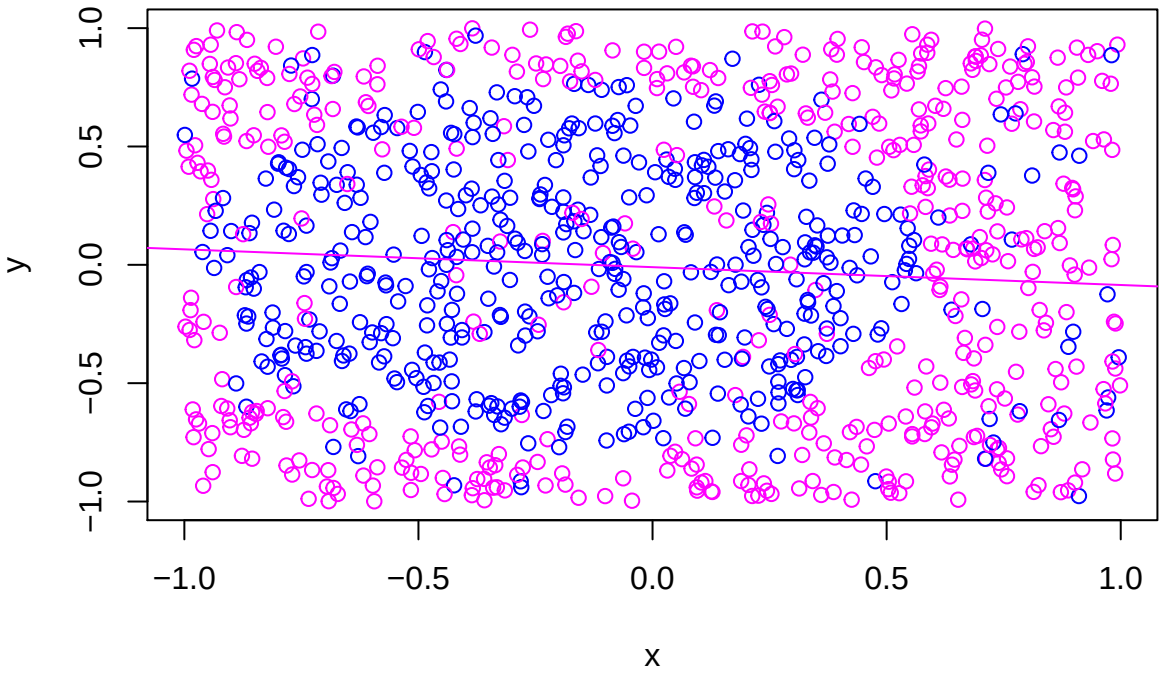
c

Ahora se va a hacer regresión lineal de los datos generados en las secciones anteriores.

Como se pide que el vector de características sea de la forma *(1, x, y)* y tenemos una matriz de dos columnas, se concatena una primera columna en la matriz con unos.

```
# Se usa como vector de características (1, x, y) y se calcula la regresión
# lineal del conjunto de datos
wRLE1 = Regress_Lin(cbind(1, experimento1), labelExperimento1)

# Se vuelven a dibujar los puntos
plot(experimento1, xlab="x", ylab = "y", main="Ejercicio 3. Apartado 4.",
     sub="Experimento 1. Subapartado c", col=labelExperimento1+5)

# Se dibuja la recta de regresión que separa los datos
abline(-wRLE1[3]/wRLE1[2], -wRLE1[1]/wRLE1[2], col=6)
```

Ejercicio 3. Apartado 4.



Experimento 1. Subapartado c

```
# Se guarda el error de los datos
EinExperimento1 = errorData(cbind(1, experimento1), labelExperimento1, wRLE1)

print(paste0("Ejercicio 3. Experimento 1. Sección c ->",
             " El error Ein del experimento 1 es: ",
             round(EinExperimento1*100, digits=3), "%"))
```

```
## [1] "Ejercicio 3. Experimento 1. Sección c -> El error Ein del experimento 1 es: 40.9%"
```

d

En esta sección hay que realizar exactamente los mismos pasos que en las tres anteriores del experimento.

Pese a que sea duplicar código, ya que la sección *d* pide volver a generar *1000* veces lo realizado en las secciones anteriores, se va a crear una función con el código anterior que vaya devolviendo los datos y sobre esto se calcula la media Ein y Eout.

```
#######################################
# Función getEinEoutDataE1()
# Devuelve el porcentaje Ein y Eout de una muestra generada de tamaño N
getEinEoutDataE1 = function(N = 1000){

  # Se generan los datos de la muestra
  data = simula_unif(N, 2, c(-1, 1))

  # Se generan las etiquetas correspondientes a la función establecida en este
  # apartado
  label = mapply(getValueF341, data[,1], data[,2])

  # Se obtienen las posiciones a las que se le va a cambiar el signo a las
```

```
  # etiquetas para añadir ruido, por defecto es un 10%
  posLabel = posChange(c(1:length(label)))

  # Se cambia el signo para añadir ruido
  label[posLabel] = -label[posLabel]

  # Se usa como vector de características (1, x, y) y se calcula la regresión
  # lineal del conjunto de datos
  w = Regress_Lin(cbind(1, data), label)

  # Se guarda el error de los datos
  Ein = errorData(cbind(1, data), label, w)

  #
  # Se repite la operación pero ahora con otra muestra para calcular Eout
  # Se generan los datos de la muestra
  dataEout = simula_unif(N, 2, c(-1, 1))

  # Se generan las etiquetas correspondientes a la función establecida en este
  # apartado
  labelEout = mapply(getValueF341, dataEout[,1], dataEout[,2])

  # Se obtienen las posiciones a las que se le va a cambiar el signo a las
  # etiquetas para añadir ruido, por defecto es un 10%
  posLabelEout = posChange(c(1:length(labelEout)))

  # Se cambia el signo para añadir ruido
  labelEout[posLabelEout] = -labelEout[posLabelEout]

  # Se guarda el error de los datos
  Eout = errorData(cbind(dataEout, 1), labelEout, w)

  # Se devuelven los errores registrados
  c(Ein, Eout)

}

# Se crea un bucle que repita este proceso 1000 veces y se consigue la media
# de los errores
nExperimento1 = 1000

# Se inicializa la matriz donde se van a guardar los resultados de los errores
resExperimento1 = matrix(nrow = nExperimento1, ncol = 2)

# Se generan todos los resultados de las muestras
for(i in 1:nExperimento1){

  resExperimento1[i, ] = getEinEoutDataE1()

}

# Se guarda la media de Ein
meanEinE1 = mean(resExperimento1[, 1])
```

```r
# Se guarda la media de Eout
meanEoutE1 = mean(resExperimento1[, 2])

print("Ejercicio 3. Experimento 1. Sección d")
```

```
## [1] "Ejercicio 3. Experimento 1. Sección d"
```

```r
print(paste0("La media Ein del experimento 1 es: ",
             round(meanEinE1*100, digits=3), "%"))
```

```
## [1] "La media Ein del experimento 1 es: 39.655%"
```

```r
print(paste0("La media Eout del experimento 1 es: ",
             round(meanEoutE1*100, digits=3), "%"))
```

```
## [1] "La media Eout del experimento 1 es: 48.744%"
```

Conclusión:

Considero que el ajuste con este modelo lineal es bastante mejorable, ya que un *39.7%* de puntos mal clasificados es demasiado.

El error fuera de la muestra es *48.7%*, algo lógico teniendo en cuenta el error dentro de la muestra. Es normal que *Eout* sea mayor, al haberse ajustado la recta de regresión con los datos de "entrenamiento".

De todos modos, el modelo lineal no es bueno y se debería conseguir mejores resultados.

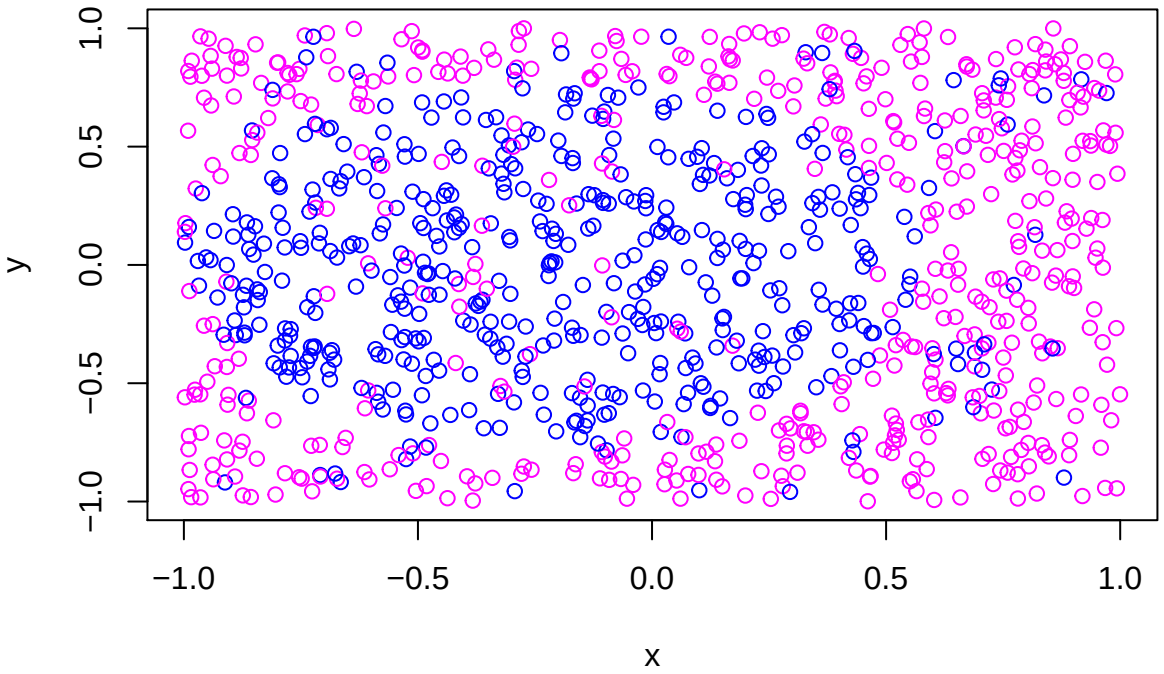
Experimento 2.

Se va a realizar un segundo experimento, con el mismo proceso que el experimento 1, pero en este caso el vector de características es (1, x, y, xy, x^2, y^2).

a

```r
# Se generan los datos del experimento 2
experimento2 = simula_unif(1000, 2, c(-1, 1))

# Se generan las etiquetas correspondientes a la función establecida en este
# apartado
labelExperimento2 = mapply(getValueF341, experimento2[,1], experimento2[,2])

# Se obtienen las posiciones a las que se le va a cambiar el signo a las
# etiquetas para añadir ruido, por defecto es un 10%
posLabelExperimento2 = posChange(c(1:length(labelExperimento2)))

# Se cambia el signo para añadir ruido
labelExperimento2[posLabelExperimento2] = -labelExperimento2[posLabelExperimento2]

# Se dibujan los datos con las nuevas etiquetas
plot(experimento2, xlab="x", ylab = "y", main="Ejercicio 3. Apartado 4.",
     sub="Subapartado b", col=labelExperimento2+5)
```

Ejercicio 3. Apartado 4.



Subapartado b

```
# Se usa como vector de características (1, x, y, xy, x^2, y^2)
# y se calcula la regresión lineal del conjunto de datos
carVecE2 = matrix(1, ncol=1, nrow=nrow(experimento2), byrow=F)
carVecE2 = cbind(carVecE2, experimento2)
carVecE2 = cbind(carVecE2, experimento2[, 1]*experimento2[, 2])
carVecE2 = cbind(carVecE2, experimento2[, 1]^2)
carVecE2 = cbind(carVecE2, experimento2[, 2]^2)

wRLE2 = Regress_Lin(carVecE2, labelExperimento2)

# Se guarda el error de los datos
EinExperimento2 = errorData(carVecE2, labelExperimento2, wRLE2)

print(paste0("Ejercicio 3. Experimento 2. Sección a ->",
       " El error Ein del experimento 2 es: ",
            round(EinExperimento2*100, digits=3), "%"))
```

```
## [1] "Ejercicio 3. Experimento 2. Sección a -> El error Ein del experimento 2 es: 14%"
```

b

Como ha pasado con el experimento 1, aunque se duplique código, para realizar esta sección se va a crear una función que agrupe el código anterior

```
########################################
# Función getEinEoutDataE2()
#
# Devuelve el porcentaje Ein y Eout de una muestra generada de tamaño N
getEinEoutDataE2 = function(N = 1000){
```

```
# Se generan los datos del experimento
data = simula_unif(N, 2, c(-1, 1))

# Se generan las etiquetas correspondientes a la función establecida en este
# apartado
label = mapply(getValueF341, data[,1], data[,2])

# Se obtienen las posiciones a las que se le va a cambiar el signo a las
# etiquetas para añadir ruido, por defecto es un 10%
posLabel = posChange(c(1:length(label)))

# Se cambia el signo para añadir ruido
label[posLabel] = -label[posLabel]

# Se usa como vector de características (1, x, y, xy, x^2, y^2)
# y se calcula la regresión lineal del conjunto de datos
carVec = matrix(1, ncol=1, nrow=nrow(data), byrow=F)
carVec = cbind(carVec, data)
carVec = cbind(carVec, data[, 1]*data[, 2])
carVec = cbind(carVec, data[, 1]^2)
carVec = cbind(carVec, data[, 2]^2)

w = Regress_Lin(carVec, label)

# Se guarda el error de los datos
Ein = errorData(carVec, label, w)

#
# Se repite la operación pero ahora con otra muestra para calcular Eout
# Se generan los datos de la muestra
dataEout = simula_unif(N, 2, c(-1, 1))

# Se generan las etiquetas correspondientes a la función establecida en este
# apartado
labelEout = mapply(getValueF341, dataEout[,1], dataEout[,2])

# Se obtienen las posiciones a las que se le va a cambiar el signo a las
# etiquetas para añadir ruido, por defecto es un 10%
posLabelEout = posChange(c(1:length(labelEout)))

# Se cambia el signo para añadir ruido
labelEout[posLabelEout] = -labelEout[posLabelEout]

# Se usa como vector de características (1, x, y, xy, x^2, y^2)
carVecEout = matrix(1, ncol=1, nrow=nrow(dataEout), byrow=F)
carVecEout = cbind(carVecEout, dataEout)
carVecEout = cbind(carVecEout, dataEout[, 1]*dataEout[, 2])
carVecEout = cbind(carVecEout, dataEout[, 1]^2)
carVecEout = cbind(carVecEout, dataEout[, 2]^2)

# Se guarda el error de los datos
Eout = errorData(carVecEout, labelEout, w)
```

```
  # Se devuelven los errores registrados
  c(Ein, Eout)

}

# Se crea un bucle que repita este proceso 1000 veces y se consigue la media
# de los errores
nExperimento2 = 1000

# Se inicializa la matriz donde se van a guardar los resultados de los errores
resExperimento2 = matrix(nrow = nExperimento2, ncol = 2)

# Se generan todos los resultados de las muestras
for(i in 1:nExperimento2){

  resExperimento2[i, ] = getEinEoutDataE2()

}

# Se guarda la media de Ein
meanEinE2 = mean(resExperimento2[, 1])

# Se guarda la media de Eout
meanEoutE2 = mean(resExperimento2[, 2])

print("Ejercicio 3. Experimento 2. Sección b.")
```

```
## [1] "Ejercicio 3. Experimento 2. Sección b."
```

```
print(paste0("La media Ein del experimento 2 es: ",
             round(meanEinE2*100, digits=3), "%"))
```

```
## [1] "La media Ein del experimento 2 es: 14.305%"
```

```
print(paste0("La media Eout del experimento 2 es: ",
             round(meanEoutE2*100, digits=3), "%"))
```

```
## [1] "La media Eout del experimento 2 es: 14.524%"
```

Conclusión:

Al usar un vector de características no lineales, se ha podido ajustar la recta de regresión al problema que se presenta mucho mejor.

Además, teniendo en cuenta que tiene un *10%* de ruido, los resultados habrían sido mucho mejores en el caso de no tenerlo, prácticamente rondando un 10% menos.

El error con los datos de test ha sido prácticamente igual de bueno, siendo levemente superior. Esto indica que la regresión se ha calculado bien y tenemos una buena medida para clasificar datos que no están en la muestra.

Conclusión final:

La diferencia entre los dos vectores de características ha hecho que el vector de características no lineales se ajuste mucho mejor a la muestra, pero debido a que la muestra que se tenía no era linealmente separable. Para esta muestra, queda demostrado que el mejor modelo es el no lineal, ajustándose éste a una muestra más heterogénea.