Trabajo 1

Gustavo Rivas Gervilla

23 de marzo de 2016

Nota: algunos de los resultados mostrados no serán exactamente los mismos que se vean en ejecución, esto se debe a que algunos se han copiado y pegado y otros se han ejecutado en Rmarkdown con lo cual los randoms que se dan no son los mismo exactamente, de todos modos el análisis realizado de los resultados es válido tanto para los que se aquí se muestran como para los que se verán en la ejecución.

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.

2. Construir una función lista = simula_gaus(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.

En primer lugar vamos a mostrar las funciones que generan dichas listas, no obstante para trabajar más adelante hemos hecho nuevas versiones de estas funciones de modo que lo que se genere sean matrices de datos en lugar de listas las cuales resultan muy engorrosas de manejar.

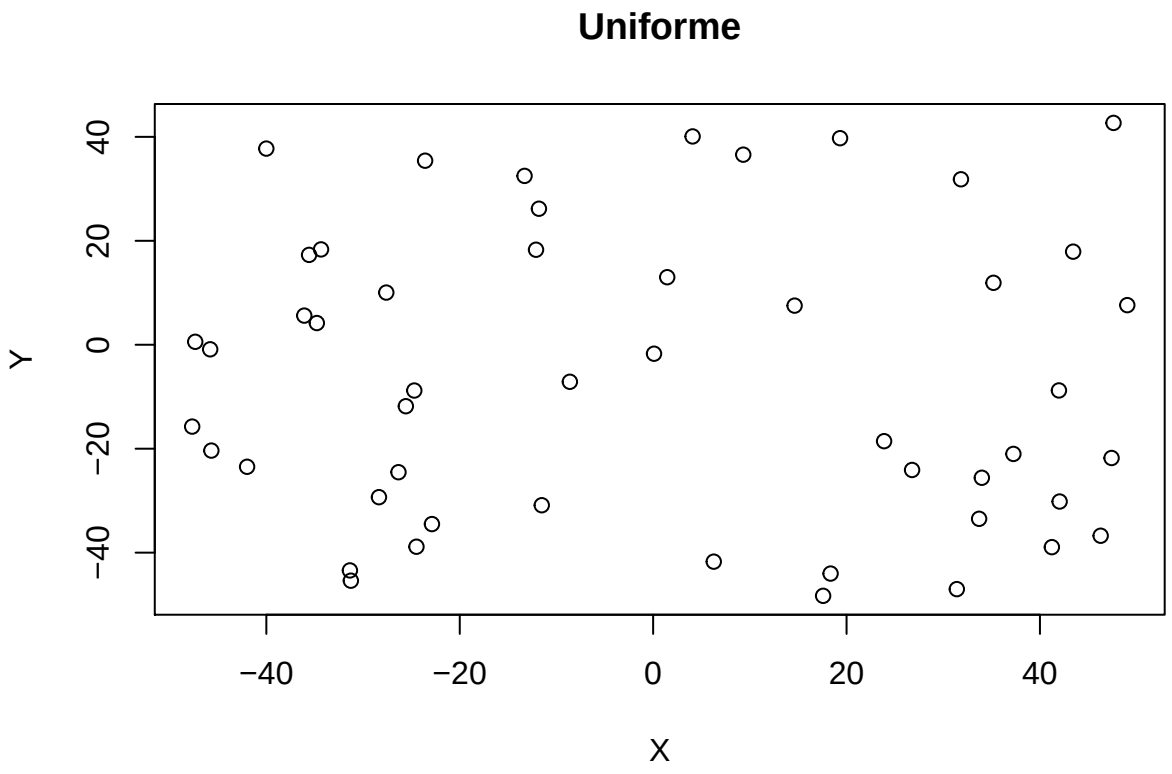
```
simula_unif <- function(N, dim, rango) {
  lista <- list() #generamos una lista vacia

  #vamos añadiendo componentes a dicha lista que son arrays de dim componente muestreando datos
  #de una uniforme con la función runif a la que le pasamos el número de datos a generar y el rango.
  for (i in seq(1:N))
    lista <- c(lista, list(array(runif(dim, rango[1], rango[2]), dim)))

  lista
}

simula_gaus <- function(N, dim, sigma) {
  lista <- list()

  for (i in seq(1:N))
    lista <- c(lista, list(array(rnorm(dim, 0, sigma), dim)))

  lista
}
```

Y aquí vemos las nuevas versiones de estas funciones que lo que generan son matrices. R es muy cómodo de usar y como vemos lo único que hacemos es generar una muestra del tamaño que deseamos de la distribución deseada la cual le damos como datos al constructor de matrices al que le decimos el número de filas y columnas que ha de tener nuestra matriz (tantas filas como datos y tantas columnas como dimensión tengan dichos datos, a través de los argumentos ncol y nrow). Con esto como ya hemos dicho obtendremos matrices de datos que nos serán más cómodas para trabajar en lo sucesivo:

```r
simula_unif <- function (N, dim, rango) {
  #generamos tantos datos como necesitemos de una uniforme y los organizamos en una matriz
  #le damos las filas y columnas que queremos que tenga con ncol y nrow
  #el primer parametro son los datos a introducir en la matriz, la muestra uniforme del tamano indicado
  datos <- matrix(runif(N*dim, rango[1], rango[2]), ncol = dim, nrow = N)

  datos
}

simula_gaus <- function(N, dim, sigma) {
  datos <- matrix(rnorm(N*dim, 0, sigma), ncol = dim, nrow = N)

  datos
}
```

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

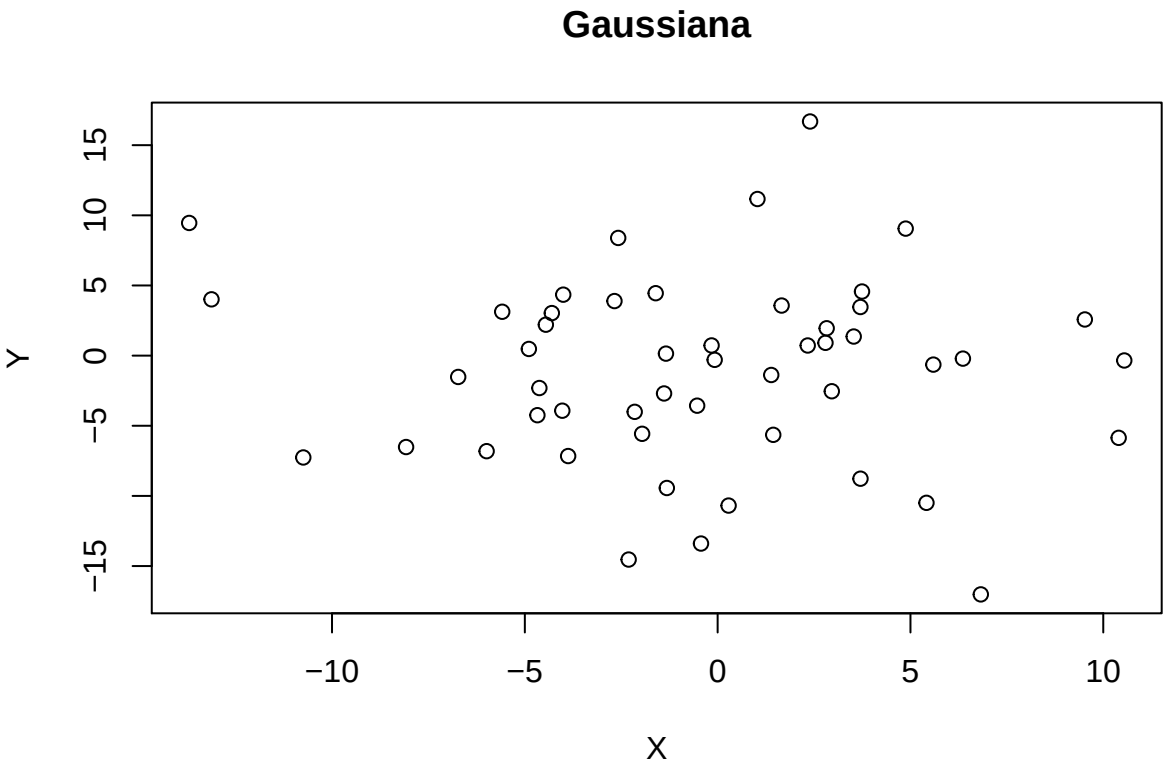
```r
ejercicio3 <- function() {
  datos <- simula_unif(50, 2, c(-50,50)) #generamos los datos con dos dimensiones
  plot(datos[,1], datos[,2], main = "Uniforme", xlab = "X", ylab = "Y") #dibujamos los datos, cada dime
}
```



Como podemos ver generamos 50 datos con dos dimensiones y los pintamos pasándole cada una de sus coordenadas, cada columna de la matriz de datos, a los dos ejes del dibujo.

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

```
ejercicio4 <- function() {
  datos <- simula_gaus(50, 2, c(5,7)) #generamos los datos con dos dimensiones
  plot(datos[,1], datos[,2], main = "Gaussiana", xlab = "X", ylab = "Y") #dibujamos los datos, cada dim
}
```



Aquí hacemos lo mismo que antes sólo que ahora para datos de una gaussiana, como podemos ver le pasamos un vector de varianzas, lo que estamos haciendo realmente es muestrear de dos gaussianas distintas, ambas con media cero pero con varianzas 5 y 7, como podemos ver la mayoría de los datos se agrupan en el intervalo [-7,7].

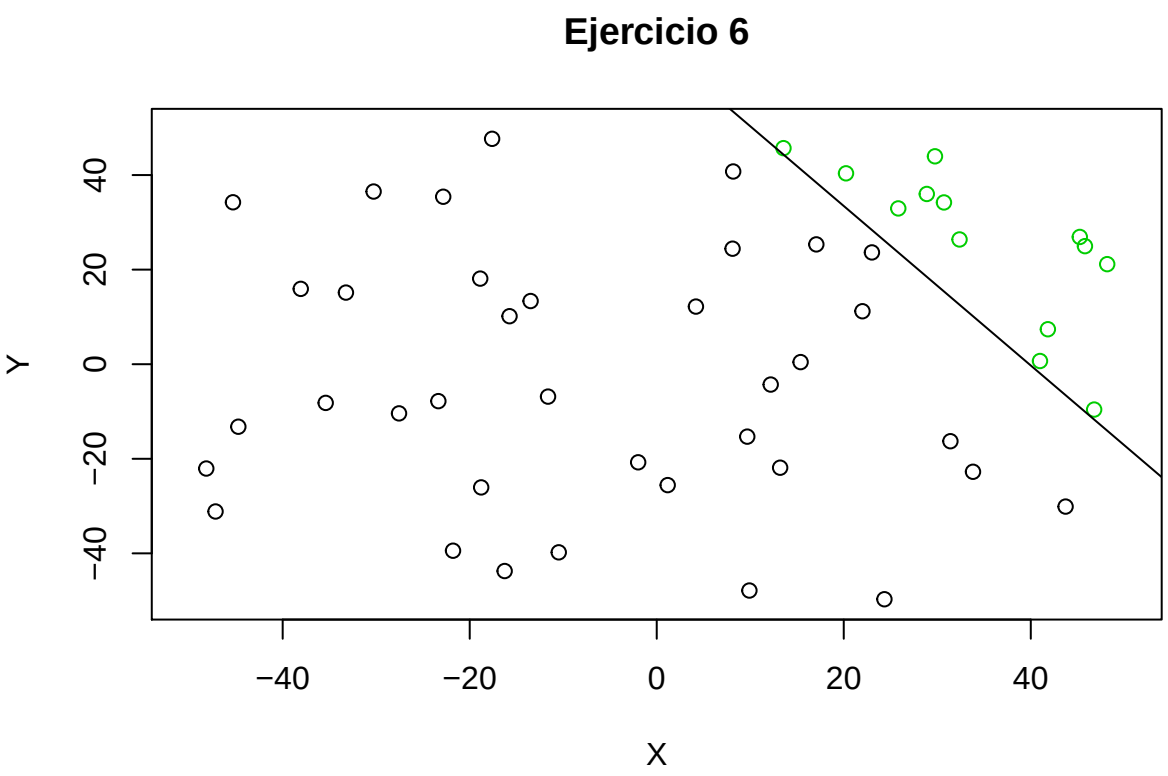
5. Construir la función v=simula_recta(intervalo) que calcula los parámetros, v = (a, b) de una recta aleatoria, y = ax + b, que corte al cuadrado [-50, +50] [-50, +50] (Ayuda: Para calcular la recta simular las coordenadas de dos puntos dentro del cuadrado y calcular la recta que pasa por ellos.)

```
simula_recta <- function(rango = c(-50,50)) {
  coordenadas = simula_unif(2,2,rango) #generamos dos ptos en el cuadrado rangoxrango
  a = (coordenadas[2,2]-coordenadas[1,2])/(coordenadas[2,1]-coordenadas[1,1])#la pendiente (m)
  b = -a*coordenadas[1,1] + coordenadas[1,2] #el factor independiente en la ecuacion y = mx+b

  c(a,b)
}
```

Lo que hacemos aquí es justo lo que se sugiere, generar dos puntos dentro del cuadrado rango x rango (que por defecto será el (-50,50)) y luego usar dichos puntos para obtener los elementos de la ecuación punto pendiente de una recta, y = mx+b. Como sabemos la pendiente se calcula como el cociente entre el incremento de la Y y el de la X y para el b no tenemos más que sustituir uno de los dos puntos generados y despejar. Observemos que tanto ahora como antes estamos considerando que la segunda coordenada de los datos, la correspondiente a la segunda columna de la matriz de datos, es la que se corresponde con el eje de ordenadas (Y).

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.

```
f0 <- function(a, b, x, y) {sign(y-b*x-a)} #funcion que nos da las etiquetas para el ejercicio 6

muestra = simula_unif(50, 2, c(-50,50)) #generamos una muestra para emplearla en los sucesivos ejercici
muestra_hom = cbind(muestra, 1) #anadimos la coordenada homogenea para los ejercicios de PLA y sucesivo

ejercicio6 <- function() {
  recta = simula_recta() #obtenemos una recta
  et <- mapply(f0, recta[2], recta[1], muestra[,1], muestra[,2]) #evaluamos f0 en cada dato, en cada fi
  plot(muestra, main = "Ejercicio 6", col=et+2, xlim = c(-50,50), ylim = c(-50,50), xlab = 'X', ylab =
  abline(a = recta[2], b = recta[1]) #anadimos la recta al plot actual
}
```



En primer lugar hemos generado una muestra aleatoria como ya hemos hecho en ejercicios anteriores que la iremos usando a lo largo de los distintos ejercicios, además hemos hecho una "copia homogénea" de la misma puesto que tendremos que trabajar con los datos homogeneizados para el PLA. También hemos definido una

función que nos devuelve el signo de un dato (x,y) dado por una recta de parámetros a y b, esta función la aplicaremos sobre cada dato de la muestra generada con la funcion *mapply* a la cual le decimos qué funcion evaluar y cómo tomar los argumentos; en este caso para la x y la y toma las columnas de cada fila como podemos ver.

Con esto tendremos las etiquetas que las usaremos (sumándole dos para evitar los negativos) como color para dibujar cada uno de los datos viendo así claramente la etiqueta de cada uno de ellos. Por último usamos la función abline para agregar la recta al plot actual con cuidado de que los argumentos a y b tienen el nombre contrario a cómo lo encontramos en la ecuación del enunciado y = ax +b.

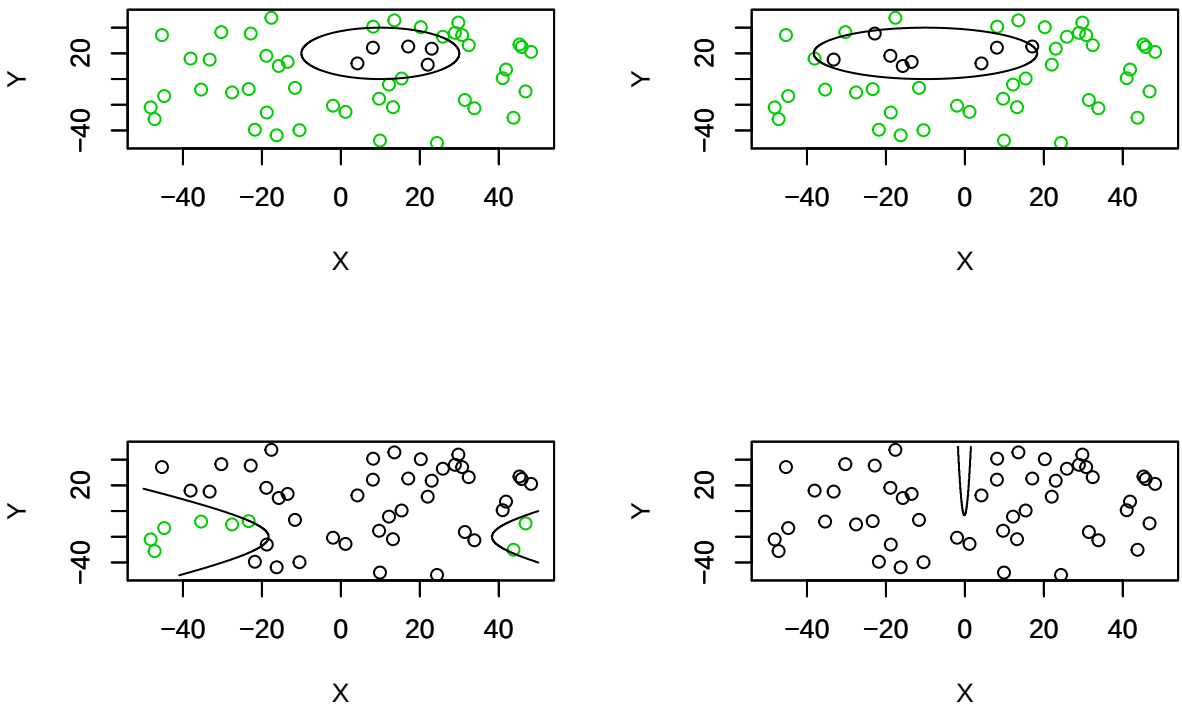
Como podemos ver lo que tenemos es que la recta divide el plot en dos regiones, a un lado tenemos puntos de un color y al otro del otro, es decir, la recta divide los dos grupos de datos según su etiqueta como debe ser.

7. Usar la muestra generada en el apartado anterior y etiquetarla con +1,-1 usando el signo de cada una de las siguientes funciones

- **f (x, y) = (x - 10) 2 + (y - 20) 2 - 400**
- **f (x, y) = 0,5(x + 10) 2 + (y - 20) 2 - 400**
- **f (x, y) = 0,5(x - 10) 2 - (y + 20) 2 - 400**
- **f (x, y) = y - 20x 2 - 5x + 3**

Visualizar el resultado del etiquetado de cada función junto con su gráfica y comparar el resultado con el caso lineal. ¿Que consecuencias extrae sobre la forma de las regiones positiva y negativa?

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

mostrar_funcion <- function(f) {
  #hacemos un submuestreo de la region donde pintaremos la funcion
  x <- seq(-50,50,length = 1000)
  y <- seq(-50,50,length = 1000)
  #calculamos el valor de la funcion en esas muestras
  z <- outer(x,y,f)

  #dibujamos la funcion con los datos indicados
  contour(x,y,z,levels=0, drawlabels = FALSE)

}
```



Aquí procedemos del mismo modo que en el ejercicio anterior, en esta ocasión para dibujar la función que nos da las etiquetas usaremos la función contour ya que estamos trabajando con funciones implícitas y entonces para cada x e y obtenemos un valor; no expresamos y en función de x. Como vemos lo que hacemos es generar una muestra de puntos (x,y) lo suficientemente extensa como para poder hacer un buen dibujado de dicha función y luego calculamos el valor de la función para esas parejas con la función *outer*. También hemos de usar la instrucción **par(new = TRUE)** para evitar que la función de alto nivel, contour, dibuje en otra área de dibujado o limpie la actual, lo que queremos es que dibuje encima de los puntos que ya hemos dibujado.

Como podemos ver las áreas dibujadas se corresponden con el interior de dos elipses, una hipérbola y una parábola; claramente esto lo vemos pues hemos dibujado las funciones que nos dan las etiquetas. Con lo cual lo que tenemos son datos que no son linealmente separables, algo que sí que podríamos ver sin necesidad de dibujar las funciones implícitas. Entonces no vamos a poder encontrar una recta que separe correctamente los datos y entonces el PLA no funcionará correctamente.

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

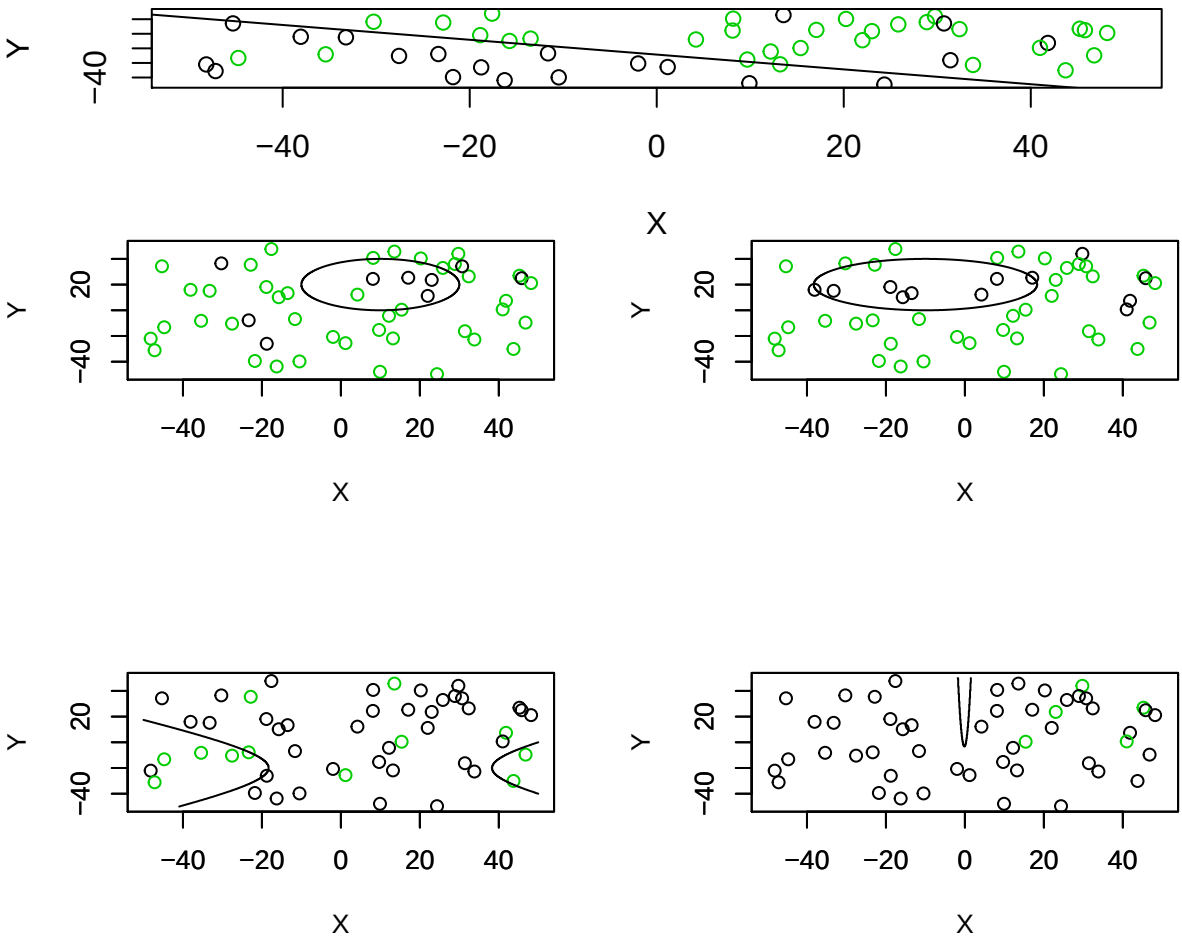
- **Visualice los puntos con las nuevas etiquetas y la recta del apartado 6.**
- **En una gráfica aparte visualice nuevo los mismos puntos pero junto con las funciones del apartado 7.**

Observe las gráficas y diga qué consecuencias extrae del proceso de modificación de etiquetas en el proceso de aprendizaje.

```
introducirRuido <- function(etiquetas_originales) {
  et_ruido <- etiquetas_originales

  #Calculamos el numero de etiquetas positivas y negativas
```

```
  n_positivos <- length(which(et_ruido == 1))
  n_negativos <- length(et_ruido) - n_positivos

  #Hacemos un muestreo aleatorio de tamaño el 10% de las etiquetas correspondientes.
  idx_positivos <- sample(which(et_ruido == 1), ceiling(n_positivos*0.1)) #positivos
  idx_negativos <- sample(which(et_ruido == -1), ceiling(n_negativos*0.1)) #negativos

  #Con los indices muestreados aleatoriamente cambiamos los valores de esas posiciones
  et_ruido[idx_positivos] = -1
  et_ruido[idx_negativos] = 1

  #devolvemos las etiquetas
  et_ruido
}
```

Aquí tenemos la función con la que introducimos ruido en las etiquetas. Lo que hacemos es en primer lugar contar cuántos negativos y positivos hay para a esto calcularle el 10% (redondeando hacia arriba) y así modificar el 10% de cada conjunto de etiquetas. Para hacer esta modificación lo que hacemos es tomar una muestra aleatoria, con *sample* del conjunto de índices de etiquetas positivas y de las negativas respectivamente, este conjunto nos lo da la función *which* con la que se pueden obtener los índices de un array de aquellas posiciones cuyo valor contenido en ellas cumplan una determinada condición. Luego no tenemos más que modificar el valor que hay en dichas posiciones.



Como podemos ver lo que ocurre es que ahora ya las funciones no separan tan bien los puntos, es decir, aunque nosotros sepamos que esa es la función correcta el ruido en las etiquetas que usaríamos para

aprendizaje provocaría que se obtuviese otra función distinta para clasificar a la que realmente es. Por ejemplo en el caso lineal no provocaría solamente que no obtuviésemos la recta correcta sino que además los datos los tomaríamos como no linealmente separables y en consecuencia no podríamos usar por ejemplo el PLA teniendo que usar técnicas más sofisticadas.

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 *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.

```
#funcion h del PLA
h <- function(w, dato) {
  sign(w%*%dato) #signo del producto escalar entre w y el dato
}

#funcion que dibujar la recta dada por los pesos de w, despejando de la ecuacion xw1+yw2+w3 = 0
plotw <- function(w, color = 'black') {
  if (w[2] != 0)
    abline(a = -w[1]/w[2], b = -w[3]/w[2], col = color)
  else if (w[1] != 0)
    abline(v = -w[3]/w[1], col = color)
  else
    print("No hay linea.")
}

ajustaPLA <- function(datos, label, max_iter, vini, plot = FALSE) {
  w <- vini
  num_datos <- nrow(datos)
  encontrado = FALSE
  iter = 1

  #mientras no superemos el max_iter y no hayamos encontrado la recta que separa los datos
  while (!encontrado && iter < max_iter) {
    encontrado <- TRUE
    for (i in seq_len(num_datos))
      #si el w actual no etiqueta bien al dato lo actualizamos
      if (h(w, datos[i,]) != label[i]) {
        w <- w + label[i]*datos[i,]
        encontrado <- FALSE
      }

    #En caso de que queramos dibujar las soluciones que vayamos generando.
    if (plot) {
      plot.new()#reiniciamos el area de dibujo
      plot(datos, col = label+2, xlim=c(-50,50), ylim = c(-50,50)) #pintamos los datos
```

```
      plotw(w) #pintamos la recta dada por los coeficientes
      Sys.sleep(1) #esperamos un segundo antes de volver a dibujar
    }

    iter = iter + 1
  }

  #devolvemos tanto los pesos obtenidos como el numero de iteraciones empleados
  list(w,iter)
}
```

Como podemos ver aquí hemos implementado el PLA tomando como una iteración una pasada completa a los datos de entrenamiento, actualizando cuando sea necesario. Como podemos ver tenemos un argumento booleano con el que indicamos si queremos dibujar o no la función, ya que en un ejercicio posterior se nos pedirá dicha modificación.

Para hacer el dibujado de la solución como vemos lo que hacemos es en cada iteración es limpiar la region de dibujo con *plot.new()* y a continuacion dibujar tanto los datos como la recta dada por los pesos actuales.

2. Ejecutar el algoritmo PLA con los valores simulados en apartado.6 del ejercicio.5.2, inicializando el algoritmo con el vector cero y con vectores de números aleatorios en [0, 1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado

Para la muestra que nosotros hemos generado el PLA tarda en converger 167 iteraciones cuando le damos como vector inicial el constante 0 y cuando generamos vectores de números aleatorios en el [0,1] tarda en media 158.6 iteraciones. Con lo cuál o bien los 10 vectores de números aleatorios que hemos generado son mejores de inicio que el (0,0,0) o hay alguno que es excelente como punto inicial y hace que la media baje extraordinariamente. Además hemos de señalar que el (0,0,0) no clasifica ningún dato correctamente ya que h(x) con el (0,0,0) es 0 para cualquier dato x.

Aunque claro esto dependerá de la muestra que nosotros tengamos y de la recta que usemos para etiquetarlos.

3. Ejecutar el algoritmo PLA con los datos generados en el apartado.8 del ejercicio.5.2, 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.

Con 10 iteraciones max. obtenemos el w = (52.09169 13.2947 6) con el cual obtenemos 13 errores.

Con 100 iteraciones max. obtenemos el w = (41.00734 12.77309 97) con el cual obtenemos 12 errores.

Con 1000 iteraciones max. obtenemos el w = (40.55043 38.64999 162) con el cual obtenemos 7 errores.

En este ejercicio he podido observar cómo efectivamente el PLA es un algoritmo sin memoria, en esa ocasión tomando los resultados que obtenemos al ejecutar el script que se va a entregar tenemos que la solución que obtiene menos errores es aquella a la que le damos un mayor número máximo de iteraciones pero no tiene por qué pasar esto, ha habido casos en que la solución tras 1000 iteraciones ha sido peor que la solución con 100.

También cabe señalar, dado que yo cometí ese error de razonamiento, que aunque el número de iteraciones es multiplo uno de los otros las soluciones encontradas no son las mismas ya que la regla de adaptación no hace que la solución cicle. Esto es sólo un detalle que quería comentar ya que yo pensaba que esto iba a suceder así.

4. Repetir el análisis del punto anterior usando la primera función del apartado.7 del ejercicio.5.2

Con 10 iteraciones max. obtenemos el w = (-36.33307 -27.02572 139) con el cual obtenemos 19 errores.

Con 100 iteraciones max. obtenemos el w = (-7.102351 -8.6905 564) con el cual obtenemos 8 errores.

Con 1000 iteraciones max. obtenemos el w = (8.986481 -3.238176 853) con el cual obtenemos 7 errores.

Aquí nuevamente tenemos que la mejor solución es aquella que se calcula con el máximo número de iteraciones posibles, al contrario de lo que podría esperar obtenemos menos errores para esta muestra no separable que para la anterior que provenía de las etiquetas dadas por una recta, pero como trabajamos con etiquetas ruidosas puede suceder cualquier cosa.

6. A la vista de la conducta de las soluciones observada en el apartado anterior, proponga e implemente una modificación de la función original sol = ajusta_PLA_MOD(...) que ermita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando los datos del apartado.7 del ejercicio.5.2

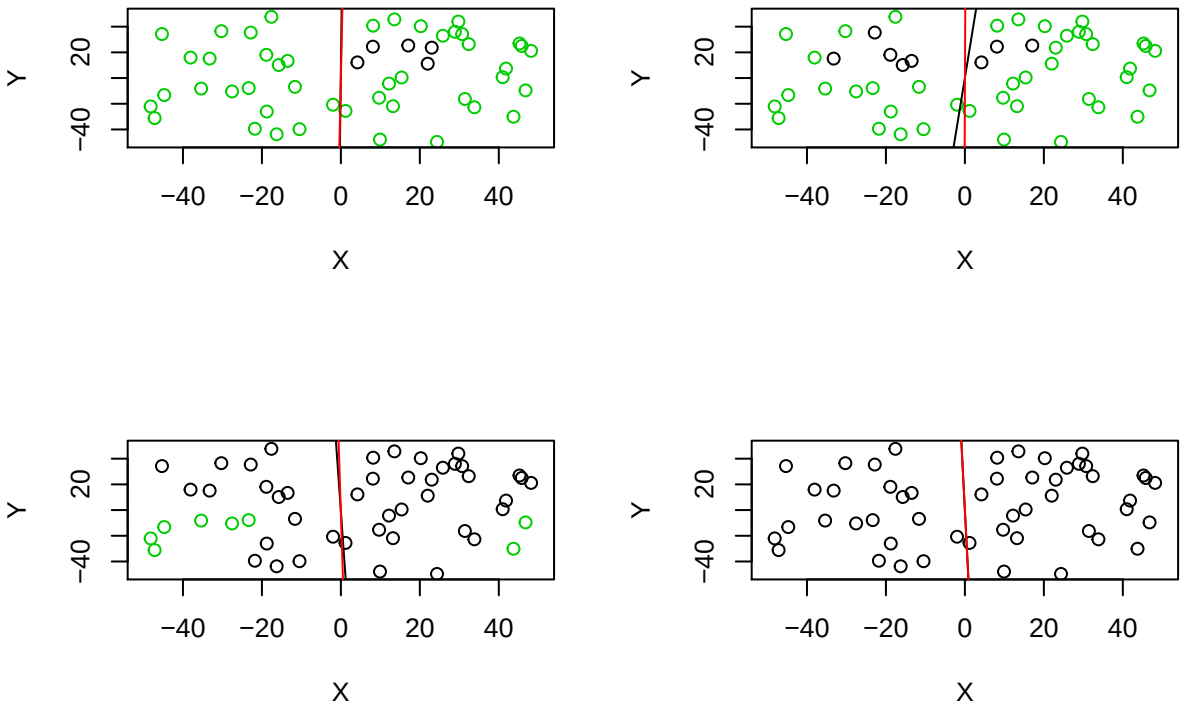
(Del ejercicio anterior no vamos a decir nada puesto que para ver las soluciones encontradas tendremos que hacerlo en ejecución, donde vamos a poder ir viendo cómo se mueve la recta que va dando el PLA al ir pasando por los datos, además ya hemos explicado anteriormente cómo funciona la modificación pedida).

```
ajusta_PLA_MOD <- function(datos, label, max_iter, vini) {
  w <- vini
  mejor_w <- w
  num_datos <- nrow(datos)
  min_errores <- num_datos
  encontrado = FALSE
  iter = 1

  while (!encontrado && iter < max_iter) {
    encontrado <- TRUE

    for (i in seq_len(num_datos))
      if (h(w, datos[i,]) != label[i]) {
        w <- w + label[i]*datos[i,]

        #por cada modificacion contamos el numero de errores del nuevo w
        errores_w = length(which(label != apply(datos, 1, h, w = w)))

        #si es menor que el que teniamos el mejor w encontrado se actualiza con el nuevo
        if (errores_w < min_errores) {
          mejor_w <- w
          min_errores = errores_w
        }

        encontrado <- FALSE
      }

    iter = iter + 1
  }
```

```
  list(mejor_w,iter)
}
```

Como podemos ver lo que estamos haciendo es una modificación muy sencilla e intuitiva del PLA original y es que cada vez que modificamos el vector de pesos, w, comprobamos si este nuevo vector comete menos errores etiquetando los datos de entrenamiento que el mejor encontrado hasta el momento. Entonces si comete menos errores el mejor vector de pesos encontrado pasa a ser este. Finalmente devolvemos el mejor w que hayamos encontrado. Esto son los resultados que obtenemos para las funciones del ejercicio7 comparándolo con los resultados obtenidos por el PLA original. En rojo mostramos la solución encontrada por el PLA modificado y en negro la del original.

```
## Sobre las etiquetas de la primera funcion. Errores de PLA:  10 Errores de PLA_MOD:  5
## Sobre las etiquetas de la segunda funcion. Errores de PLA:  14 Errores de PLA_MOD:  8
## Sobre las etiquetas de la tercera funcion. Errores de PLA:  8 Errores de PLA_MOD:  4
## Sobre las etiquetas de la cuarta funcion. Errores de PLA:  0 Errores de PLA_MOD:  0
```



Como podemos ver en todos los casos el número de errores cometidos es menor que con el PLA original, con lo cuál hemos corregido el principal problema del PLA y es la falta de memoria. Evidente seguimos cometiendo errores puesto que los datos que intentamos clasificar no son linealmente separables (aunque puede haber muestras en que si se dispongan de modo que sean separables por medio de una línea, pero esto es cuestión del azar, para la última función es muy normal que todas las muestras esten a uno de los lados) pero el error se reduce y entre otras cosas evitamos la penalización por aumentar el número de iteraciones máximas que hemos tenido en ejercicios anteriores.

Ejercicio sobre Regresión Lineal

2. Lea el fichero ZipDigits.train dentro de su código y visualize las imágenes. Seleccione solo las instancias de los números 1 y 5. Guardelas como matrices de tamaño 16x16.

Las imagenes vienen dadas como una fila de datos en la que se recogen tanto los píxeles de la imagen como su clase, para formar una matriz apartir de una fila de datos hemos implementado la siguiente función:

```
getImagen <- function(datos) {
  matrix(datos, ncol = 16)
}
```

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 invertiendo el orden de las columnas. Finalmente le cambiamos el signo.

La media de una matriz es muy fácil de calcular pues sólo tenemos que usar la función *mean* que automáticamente hace la media de los valores que componen la matriz. Para calcular la simetría sí hemos implementado una función. Esta función es muy sencilla aprovechando el funcionamiento vectorial que tiene R:

```
getSimetria <- function(m) {
  -2*sum(abs(m[,seq(1,8)]-m[,seq(16,9)]))
}
```

como vemos lo que hacemos es restar las dos mitades de la matriz (dándole a una la vuelta jugando con la lista de índices que le pasamos) y aplicando el valor absoluto obtenemos una matriz que en cada celda tiene el valor absoluto de una de las diferencias que queremos calcualr. Luego no tenemos más que sumar todos los elementos de dicha matriz con la función *sum* y multiplicar por -2 ya que queremos cambiarle el signo y los cálculos que queremos realizar sobre la matriz son simétricos.

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

Para implementar esta función hemos hecho uso de las fórmulas vistas en clase que tras una deducción nos decían cómo calcular los coeficientes del hiperplano que mejor aproxima a unas muestras de una función. Estás fórmua era la siguiente $w_{lin} = (X^T X)^{-1} X^T y$ (siendo X la matriz de datos e y el conjunto de etiquetas) y para calcular cómodamente el producto $(X^T X)^{-1}$ usamos la descomposición SVD de X, con lo que tenemos que $(X^T X)^{-1} = V D^{-2} V^T$ con $X = U D V^T$. Entonces la función para calcular la regresión lineal es la siguiente:

```
regressLin <- function(datos, label) {
  #obtenemos la descomposicion SVD de la matriz
  svd(datos) -> S

  #fabricamos una matriz diagonal con la diagonal del SVD redondeando los valores
  diag(round(S$d, digits = 5)) -> D
```

```
  #calculamos la pseudo-inversa de la matriz D con el ginv del paquete MASS
  ginv(D) -> gD
  #calculamos (t(datos) * datos)^-1 con la formula vista en clase V*D^-2*t(V)
  S$v%*%gD%*%gD%*%t(S$v) -> A
  #obtenemos el vector que nos da el hiperplano de regresion con la formula w = (t(X)X)^-1t(X)y
  A%*%t(datos)%*%label -> wlin

  wlin
}
```

como podemos ver para calcular la pseudo-inversa de D usamos la función *ginv* del paquete *MASS* que calcula la preudo-inversa de la matriz que le pasamos. Instalar este módulo de R en Rsutdio es muy sencillo sin más que buscarlo en el menú *Tools > Install Packages...*.

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

6. Ajustar un modelo de regresion lineal a los datos de (Intensidad promedio, Simetria) y pintar la solución junto con los datos. Valorar el resultado.

Hemos agrupado todos los pasos previos en una sola función donde podemos ver muy claro el procedimiento que vamos siguiendo hasta aplicar la regresión lineal a la nube de puntos obtenidos:

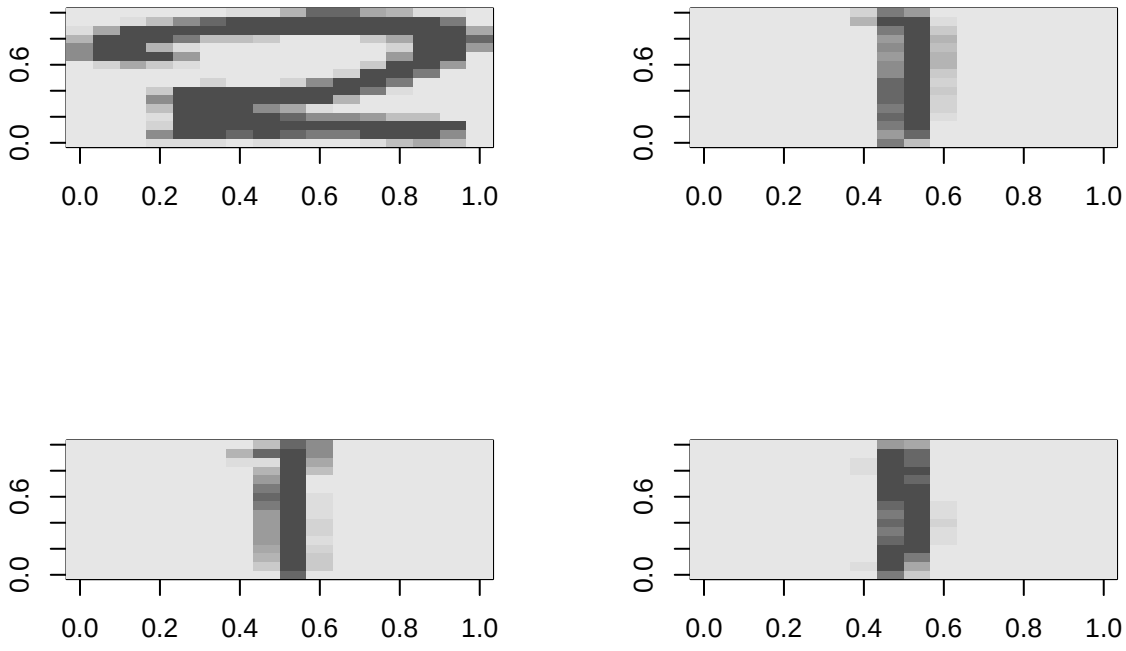
```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

```
#Funcion en la que se recogen los apartados de 1 al 6 de regresión
ejercicio4.2 <- function() {
  #tomamos solo las instancias correspondientes a 1 y 5
  instancias <- digitos_entrenamiento[(digitos_entrenamiento[,1] == 1 | digitos_entrenamiento[,1] == 5)
  #creamos una matriz con dichos datos
  m_datos <- data.matrix(instancias)
  #normalizamos los datos
  m_datos[,2:257] <- 0.5*(1-m_datos[,2:257])
  #obtenemos solo las etiquetas de los datos
  et <- m_datos[,1]

  #Guardamos cada imagen como una matriz 16x16 y creamos una lista con ellas
  imagenes <- list()

  for (i in seq(1, nrow(m_datos)))
    imagenes <- c(imagenes, list(getImagen(m_datos[i,2:257])))

  #mostramos la imagenes en escala de grises
  par(mfrow = c(2,2))
  image(imagenes[[1]], col = gray.colors(12))
  image(imagenes[[2]], col = gray.colors(12))
  image(imagenes[[3]], col = gray.colors(12))
  image(imagenes[[4]], col = gray.colors(12))

  Sys.sleep(5)
```

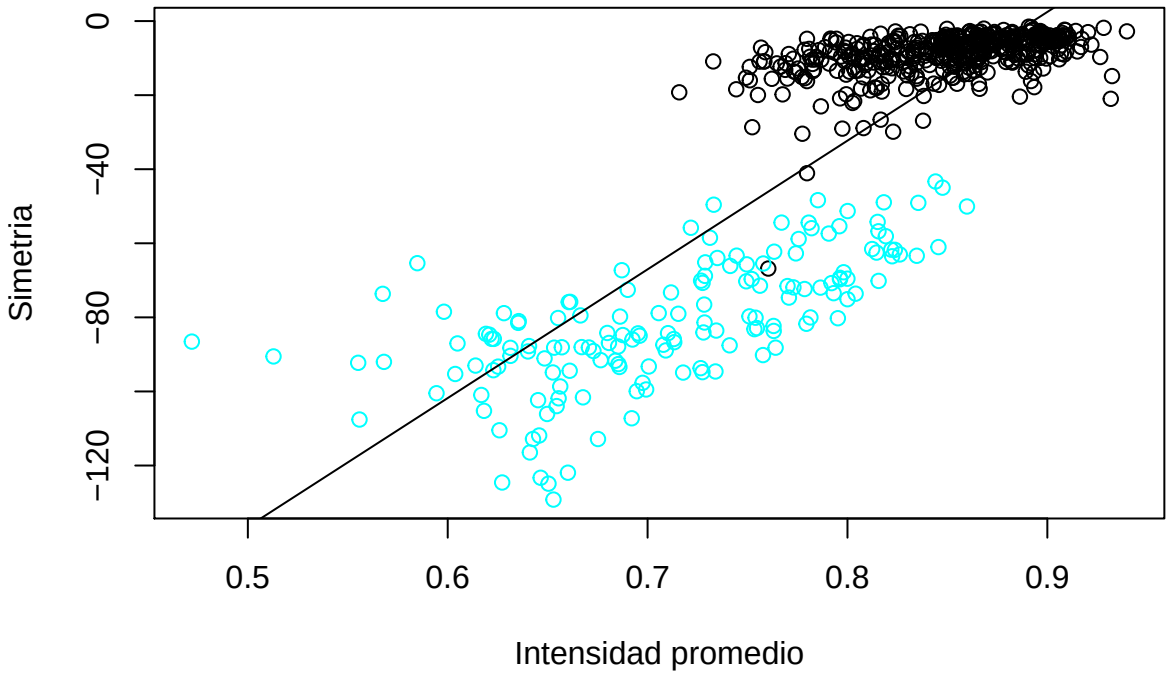
```
  print("")
  #creamos una array con las medias y simetrias de las imagenes
  medias <- unlist(lapply(imagenes, mean))
  simetrias <- unlist(lapply(imagenes, getSimetria))
  #ajustamos una recta a la una funcion que para cada media de una imagen devuelve la simetria de dicha
  #fabricamos entonces la matriz de datos X que sera el resultado de homogeneizar las medias
  datos <- matrix(medias, ncol = 1)
  datos <- cbind(datos, 1)

  par(mfrow = c(1,1))


  wlin <- t(regressLin(datos, simetrias))

  #mostramos el resultado
  plot(medias, simetrias, col = et+8, xlab = 'Intensidad promedio', ylab = 'Simetria')
  cat("wlin: ", wlin, "\n")
  abline(a = wlin[,2], b = wlin[,1])
}
```

De aquí hay que señalar qué datos son los que ajustamos, dado que queremos ajustar una recta a los pares (intensidad, simetría) lo que tenemos es que considerar que tenemos una función que para cada intensidad promedio de una imagen nos da la simetría de dicha imagene. Entonces la matriz X de datos será la compuesta por las medias de las imágenes, homogeneizadas por cuestiones computacionales y las etiquetas o los y serán las simetrías correspondientes.

Con esto obtenemos el siguiente resultado:




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




```
## wlin:  347.4851 -310.2742
```

Como podemos ver la recta se ajusta los datos en base a las distancias al cuadrado a cada punto. Podríamos pensar que la recta va a tener a ser horizontal para ajustar bien la nube de puntos negra donde da la sensación de estar concentrada la mayor cantidad de puntos. Lo que ocurre es que estos puntos están muy compactos con lo que las distancias a la recta de dichos puntos son muy similares o no pesan tanto que la de los puntos azules que forman una nube más dispersa. Por esto la recta de regresión opta por pasar "a través" de la nube de puntos azul que es la que aporta mayores distancias.

Señalar que el color qué imagen es cada punto si la de un 1 o la de un 5.

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

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

b) Fijar el tamaño de muestra N = 100. Usar regresión lineal para encontrar g y evaluar Eout . 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 Eout? Valore los resultados.

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 1.000 veces ¿Cuál es 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

[1] "Apartado a" Numero medio de E_in: 3.874 Porcentaje medio de E_in: 3.874 %

[1] "Apartado b" Numero medio de E_out: 50.639 Numero medio de porcentaje de E_out: 5.0639 %

[1] "Apartado c" Numero medio de iteraciones PLA: 7.844

Como podemos ver los errores obtenidos por regresión no son demasiado elevados pues no llegan al 5%, claro que todo dependerá de para qué queramos esta estimación y por ende qué importancia tenga la cantidad de errores. Lo que sí observamos algo que parece lógico de esperar y es que el error dentro de la muestra es menor que el fuera de la muestra, pues no es información que se tenga en cuenta en el proceso de aprendizaje.

El número de iteraciones, de pasadas a los datos que ha de dar el PLA para obtener una recta que separe correctamente los datos es también pequeño comparándolo con los resultados obtenidos en ejercicios anteriores con lo cuál confirmamos que aplicar regresión es un buen método para obtener un vector de pesos inicial para el PLA. También es cierto que el número de datos a separar es muy pequeño, tan solo 10 datos.

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 $\mathcal{X}$ = [-10, 10] × [-10, 10] muestreando cada punto x $\in \mathcal{X}$ 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.

Ajustar regresion lineal, para estimar los pesos w. Ejecutar el experimento 1.000 y calcular el valor promedio del error de entrenamiento E in . Valorar el resultado.

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 w. Mostrar el resultado.

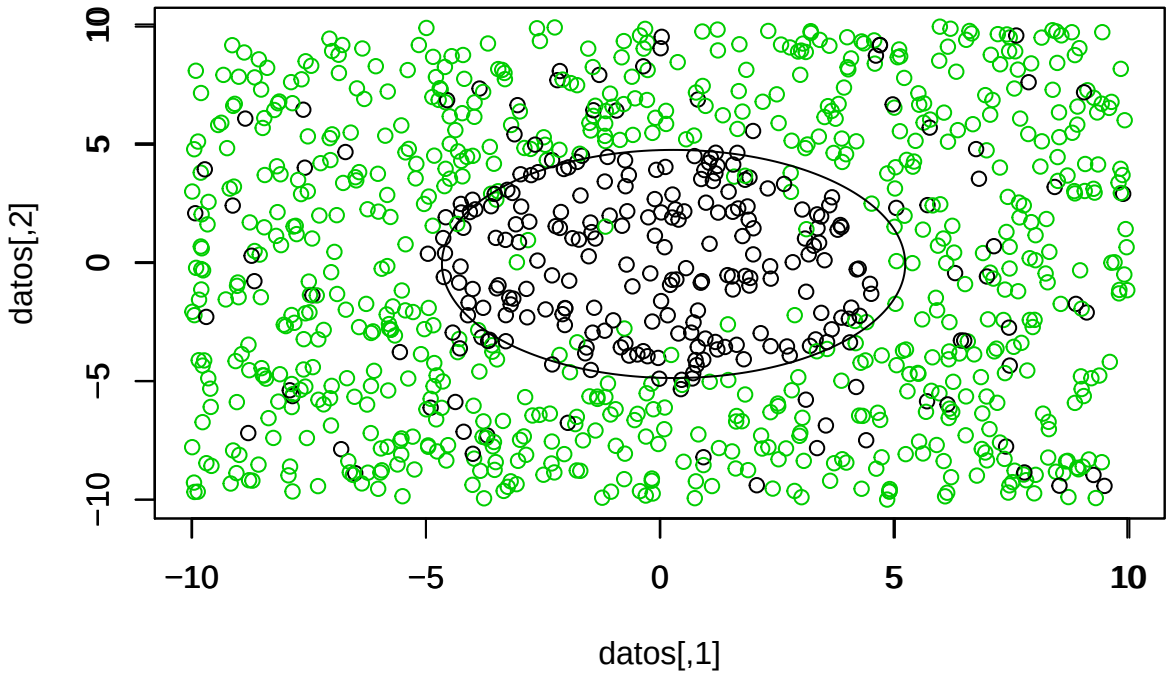
Repetir el experimento anterior 1.000 veces calculando en cada ocasión el error fuera de la muestra. Para ello generar en cada ejecución 1.000 puntos nuevos y valorar sobre ellos la función ajustada. Promediar los valores obtenidos. ¿Qué valor obtiene? Valorar el resultado.

Aquí lo que tenemos que hacer es definir tanto la función que evaluaremos para obtener las etiquetas como la función que a partir de un punto (x,y) nos devuelve el punto $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2\ )$ que es con el que trabajaremos para hacer regresión.

```
fi <- function(dato) {
  c(1, dato[1], dato[2], dato[1]*dato[2], dato[1]^2, dato[2]^2)
}

f5 <- function(dato) {
  sign(dato[1]*dato[1]+dato[2]*dato[2]-25)

}
```

Una vez hecho esto el cómo implementamos el experimento es claro visto los códigos que hemos visto anteriormente y puede verse en el código adjunto con la práctica. Entonces pasamos a mostrar los resultados:

```
## [1] "Apartado a"
## El error promedio de entrenamiento es:  256.417
## [1] "Apartado b"
## El w de pesos obtenido:  -0.2538589 -0.006147658 0.0007713895 2.374181e-05 0.01038188 0.01094983
## El error dentro de la muestra es:  121
```




```
## [1] "Apartado c"
## E_out promedio:  73.215
```

Para el primer apartado del ejercicio vemos cómo la tasa de error es muy elevada (256.417) ya que estamos tratando de clasificar datos no linealmente separables por medio de una recta.

Cuando transformamos los datos de modo que pasemos a datos linealmente separables como podemos ver el error se reduce a (121) que sigue siendo un error grande pero es menor que antes. Si nos fijamos en lo que obtenemos es una elipse, la función objetivo es una circuferencia pero dado que estamos introduciendo error a las etiquetas es normal que obtengamos algo así.

Ahora encontramos un resultado extraño y es que el error promedio fuera de la muestra es mejor que el obtenido en el apartado anterior dentro de la muestra, supongo que será porque en algunas de las 1000 iteraciones la muestra de test se ajusta muy bien a la función ajustada y hace que esta baje.