Práctica 1 - Aprendizaje Automático

Marta Gómez Macías

7 de marzo de 2016

Índice

1 Ejercicio de Generación y Visualización de datos

1.1 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 $rango$.

La función construida es la siguiente:

```
lista = simula_unif <- function(N=5, dim=20, rango=1:50) {
    # devolvemos una matriz con N filas y dim columnas, cada una rellenada con
    # valores producidos por runif.
    matrix(runif(dim*N, min=rango[1], max=tail(rango,1)), ncol=dim, nrow=N, byrow=T)
}
```

En ella, creamos una matriz de dimensión $dim \cdot N$ y la vamos rellenando por filas con los valores generados por `runif` en el rango especificado.

Un ejemplo de ejecución de la función es el siguiente:

```
##           [,1]      [,2]      [,3]      [,4]      [,5]     [,6]      [,7]
## [1,] 44.492143  8.485252 32.635354  7.102550 24.131265 28.41726 14.661649
## [2,] 13.637791 31.431834 45.858816 25.229433 30.985046 37.73454 34.313253
## [3,]  9.098689 25.785877 23.018181  3.360890  4.741831 13.95834  5.574323
## [4,]  3.578157 39.725601 21.123086 41.024448 47.424435 46.68099  2.928674
## [5,]  7.627214  5.420168  7.520553  1.712731 27.764058 48.63843 11.830968
##          [,8]      [,9]    [,10]    [,11]     [,12]     [,13]     [,14]
## [1,] 42.21437 29.013997 49.67285  4.41517 39.784228  4.964868 32.728788
## [2,] 37.75537  8.524163 10.38782 46.97592 31.030757 16.815358  4.249192
## [3,] 29.14976 21.410769 22.74446 22.27907 44.493666 18.067875 24.802830
## [4,] 17.17704 28.897711 27.49546 42.85630 40.549511 38.962459 40.523407
## [5,] 20.38466 16.826871 32.35686 27.48887  1.062353  3.192956 32.960349
##         [,15]     [,16]     [,17]     [,18]     [,19]    [,20]
## [1,] 20.94234 45.771889 25.994083 19.844044  5.205974 14.59698
## [2,] 47.52311 45.665777 28.567035 39.462008 10.721408 40.82713
## [3,] 16.03048  8.537769 19.548397  7.014843 37.258087 37.31816
## [4,] 29.83543 49.952919  4.049419  9.250392 48.296228 18.64920
## [5,] 23.42604 18.417822 26.988310 45.378240 34.999941 25.29589
```

Se cumplen con los requisitos necesarios del ejercicio: obtener una matriz de valores aleatorios uniformes en el rango especificado. Además, la ejecución de esa instrucción es mucho más rápida que un bucle `for`.

1.2 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 $sigma$.

La función construida es la siguiente:

```
lista = simula_gaus <- function(N=5, dim=20, sigma=0.5:5.0) {
    # devolvemos una matriz en cuyas filas hay valores generados por rnorm
    matrix(rnorm(dim*N, 0, sigma), ncol=dim, nrow=N, byrow=T)
}
```

Su estructura es idéntica a la usada en el ejercicio anterior, con la diferencia de que los valores aleatorios se obtienen con la función `rnorm` ne vez de con la función `runif`.

Un ejemplo de ejecución sería el siguiente:

```
##            [,1]       [,2]       [,3]       [,4]       [,5]        [,6]
## [1,]  0.1454414  1.2694555  2.5832003  0.6308350 -0.5840066 -0.28896081
## [2,] -0.3790865 -3.0143494 -0.3666696  0.5057178  1.3103064 -0.08568469
## [3,]  0.1164233  0.4590654  2.0730163  0.5479279  0.8564388 -0.20683819
## [4,]  0.5042646  2.7268770 -1.0341061  0.9385422 -1.2981375  0.72986145
## [5,] -0.9248766 -0.7664767 -3.6347477 -3.9249870 -1.6704765  0.81992710
##            [,7]        [,8]        [,9]     [,10]      [,11]      [,12]
## [1,]  0.4915817 -0.39261366  2.69708736 -4.889608  0.2717456  0.3284815
## [2,] -1.2634147  2.69201780 -1.60377708 -3.432802 -0.9932229 -0.4818821
## [3,]  0.8307950 -3.86544988 -6.52333102  5.581627  0.4708342  0.4288140
## [4,]  1.4451639 -3.06494802  1.79875513  7.452097 -0.6615829  0.0758362
## [5,] -2.5341489  0.07425224  0.05814539 -8.515738  0.6054773 -1.4146560
##           [,13]      [,14]      [,15]      [,16]      [,17]     [,18]
## [1,]  2.5680754  0.7635562 10.6564188  0.4694082  2.0254719 -0.514292
## [2,]  3.2316774 -1.5082018 -0.8041415 -0.5969445 -4.2509977 -1.345912
## [3,] -0.6680702 -0.9837246 10.5512790 -0.6489504 -0.2628122 -1.645031
## [4,]  0.4980317  1.9218791  5.4702995 -0.1537204 -0.6586483  1.801033
## [5,] -0.4536638 -2.9562420  1.0751563  0.3453697  2.8190629 -3.715366
##            [,19]      [,20]
## [1,] -0.03618444  0.8627259
## [2,] -4.83232446  9.7835157
## [3,] -0.72293010 -2.2894642
## [4,]  3.85281541  8.0177732
## [5,] -2.03434145  0.2388333
```

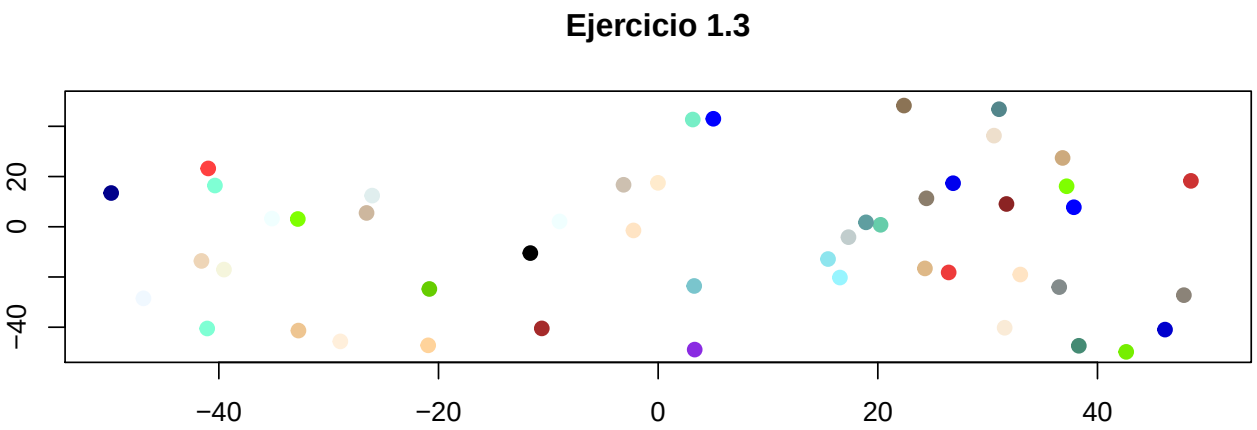
Respecto a la calidad de la solución, al haber usado la misma estructura que en el ejercicio anterior, el comentario es el mismo: se cumplen los requisitos del ejercicio y además se ha hecho con bastante eficiencia.

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

La función construida es la siguiente:

```
representa_unif <- function(N=50, dim=2, rango=-50:50) {
    # inicializamos la matriz de puntos a representar
    x = simula_unif(N, dim, rango)
    # inicializamos los limites de la grafica
    plot(rango, rango, type="n", xlab="", ylab="", main="Ejercicio 1.3")
    # representamos la matriz de puntos
    points(x=x, col=colors(), lwd=2, pch=19)
}
```

En ella, guardamos en la variable `x` una matriz aleatoria de puntos generada con `simula_unif()` y representamos la gráfica, poniendo cada punto de un color. Así, obtenemos gráficas como la siguiente:



Como se puede apreciar, la gráfica obtenida representa todos los puntos obtenidos aleatoriamente por lo que se cumple el enunciado del ejercicio. Además, la representación de puntos se hace en una sola línea, por lo que es mucho más eficiente que hacerlo con un bucle `for`, de hecho, probé a hacerlo con un bucle `for` y los puntos se iban colocando en la gráfica con cierto retraso debido a la ineficiencia que supone hacerlo con un bucle.

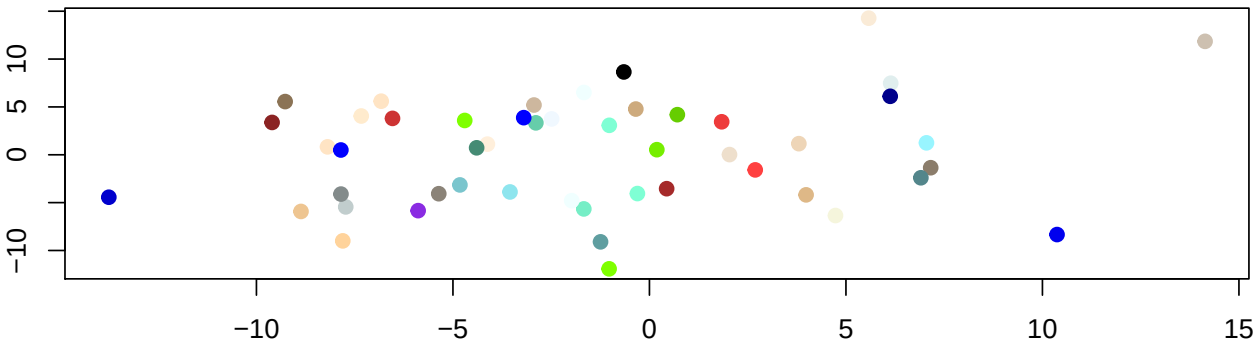
1.4 Suponer $N = 50$, $dim = 2$ y $sigma = [5, 7]$. Dibujar una gráfica de la salida de la función correspondiente.

La función construida es la siguiente:

```
representa_gaus <- function(N=50, dim=2, sigma=5:7) {
    # inicializamos la matriz de puntos a representar
    x = simula_gaus(N, dim, sigma)
    # inicializamos los límites de la gráfica
    plot(range(x[,1]), range(x[,2]), type="n", xlab="", ylab="",
        main="Ejercicio 1.4")
    points(x=x, col=colors(), lwd=2, pch=19)
}
```

Su estructura es idéntica al apartado anterior: en primer lugar obtenemos los puntos aleatoriamente, esta vez usando `simula_gaus()`, inicializamos la gráfica y por último, representamos todos los puntos con `points`.

Un ejemplo de gráfica obtenida con esta función sería:



La gráfica cumple con lo especificado en el ejercicio. Los límites seleccionados serían los límites de la función, estos límites se obtienen dinámicamente según los puntos obtenidos por lo que no se sabe el límite de antemano.

1.5 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 $[-50, 50]x[50, 50]$. (Ayuda: Para calcular la recta simular las coordenadas de dos puntos dentro del cuadrado y calcular la recta que pasa por ellos).

La función construida es:

```
calcula_recta <- function(p1x, p1y, p2x, p2y) {
    # calculamos la ecuación de la recta, para ello, lo hacemos como el siguiente
    # ejemplo: A(1,3) y B(2,-5) (http://www.vitutor.com/geo/rec/d_7.html)
    #       x - 1     y - 3
    #      ------- = --------
    #       2 - 1     -5 - 3
    a = (p2y - p1y)/(p2x - p1x)
    b = (((p2y - p1y) * -p1x) - ((p2x - p1x) *
        -p1y))/(p2x - p1x)
    # cat(sprintf("y = %fx + %f\n", a, b))
    c(a,b)      # devolvemos los valores a y b calculados
}

v = simula_recta <- function(intervalo=-50:50) {
    # calculamos dos puntos aleatorios dentro del intervalo
    puntos = sample(intervalo, 4)
    # comento tanto el print puntos como el cat para que no se impriman al
    # ejecutar el script
    # print(puntos)
    # devolvemos los parámetros a y b calculados
    calcula_recta(puntos[1], puntos[2], puntos[3], puntos[4])
}
```

En ella, tomamos dos puntos aleatorios en el intervalo dado (en este caso, el intervalo $[-50, 50]$) y calculamos la recta que pasa por ellos tal y como se explica en vitutor.

La ecuación para calcular el valor de a, al corresponder con el valor que acompaña a x sería:

$$a = \frac{By - Ay}{Bx - Ax} \qquad \text{Siendo } A \text{ el primer punto y } B \text{ el segundo}$$

La ecuación para obtener b, al ser el valor que ni acompaña a x ni acompaña a y, sería:

$$b = \frac{(-Ax) \cdot (By - Ay) - (Bx - Ax) \cdot (-Ay)}{Bx - Ax}$$

El dividir entre $Bx - Ay$ en ambos viene de que debemos dividir por el valor de y para despejarla.

Un ejemplo de ejecución, en el que vemos en primer lugar las coordenadas Ax Ay Bx By obtenidas aleatoriamente y después la ecuación final obtenida con a y b calculados sería:

Respecto a la calidad de la solución obtenida, pienso que a lo mejor hay alguna manera de hacer el cálculo de b más simplificado, pero hice varias pruebas con distintos puntos y todos se calculaban correctamente, por tanto, el ejercicio cumple con lo mínimo especificado.

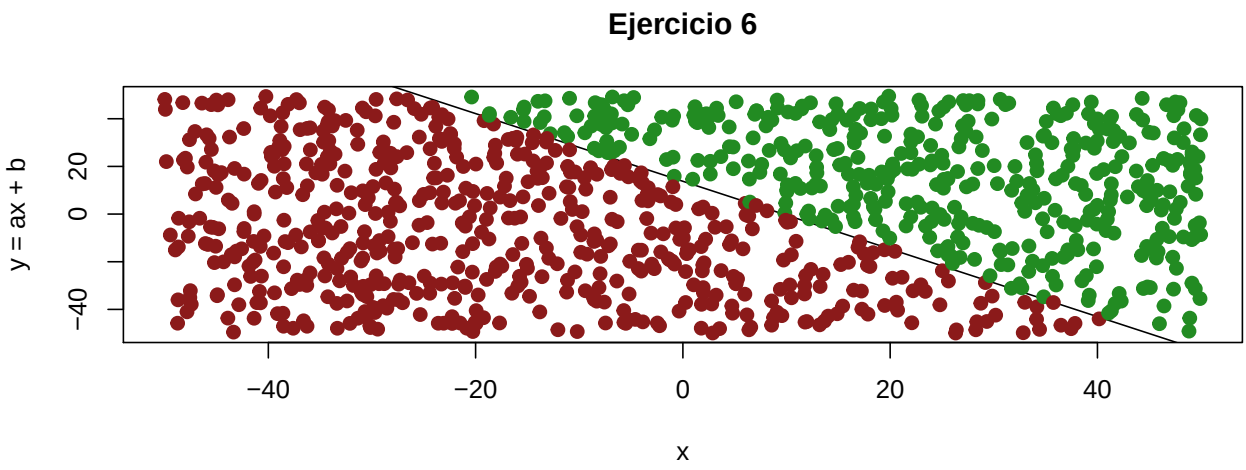
1.6 Generar una muestra 2D de puntos usando `simula_unif()` y etiquetar la muestra usando el signo de la función $f(x, y) = y - ax - b$ de cada punto a una recta simulada con `simula_recta()`. Mostrar una gráfica con el resultado de la muestra etiquetada junto con la recta usada para ello.

La función construida es:

```
# obtenemos una lista de puntos aleatorios
p = simula_unif(N=1000, dim=2, rango=-50:50)
# generamos una recta aleatoria en el intervalo por defecto
recta = simula_recta()

# definimos la función a aplicar a cada punto
funcion_etiquetado <- function(vec,rec=recta) aux = sign(vec[2] - rec[1]*vec[1] - rec[2])

obten_param_recta <- function(puntos=p,re=recta) {
    # para cada punto, calculamos su valor para la función dada
    # y lo etiquetamos con su respectivo valor. Para ello, usamos una funcion
    # que decida si un numero es positivo o negativo
    apply(X=puntos, FUN=funcion_etiquetado, MARGIN=1, rec=re)
}

representa_recta <- function(puntos=p, re=recta, nombres, titulo) {
    # calculamos los límites de la gráfica
    lim_x = range(puntos[,1])
    lim_y = range(puntos[,2])

    # representamos la recta
    plot(lim_x, lim_y, type="n", xlab="x", ylab="y = ax + b", main=titulo)
    abline(re[2], re[1])
    # representamos cada punto calculado con simula_unif del color correspondiente
    # a su clasificación
    points(x=puntos, col=colors()[138+nombres], lwd=2, pch=19)
}

ejercicio_seis <- function(titulo, puntos=p) {
    # obtenemos el etiquetado de los puntos según la recta
    etiquetas = obten_param_recta(puntos,recta)
    # representamos la recta
    representa_recta(puntos, recta, etiquetas, titulo)
}
```

En ella, generamos en primer lugar una lista de puntos aleatorios y calculamos aleatoriamente los parámetros a y b de una recta. Tras eso, aplicamos la función $f(x, y) = y - ax - b$ a cada punto y almacenamos el signo del resultado en una lista. Esta lista la usamos para etiquetar cada punto. Todo este proceso ha sido encapsulado en varias funciones debido a que en ejercicios posteriores se vuelve a usar. Por último, representamos la recta con los puntos a y b calculados y representamos también los puntos calculados de color verde o rojo según hayan sido etiquetados. El color 139 es verde y el rojo, es el 137.

Un ejemplo de ejecución sería el siguiente:



Al principio pensé en utilizar la función `curve` para representar la recta, pero muchas veces obtenía intervalos muy grandes en el eje y (por ejemplo de -4000 a 4000) y no podía contorlar eso, por eso, pensé en utilizar `lines` para evitar este problema. Finalmente he utilizado `abline` debido a que es más cómoda de utilizar.

1.7 Usar la muestra generada en el apartado anterior y etiquetarla con +1,-1 usando el signo de cada una de las siguientes funciones: $f(x,y) = (x-10)^2+(y-20)^2-400$, $f(x,y) = 0,5(x+10)^2+(y-20)^2-400$, $f(x,y) = 0,5(x-10)^2-(y+20)^2-400$ y $f(x,y) = y-20x^2-5x+3$. Visualizar el resultado del etiquetado de cada función junto con su gráfica y comparar el resultado con el caso lineal. ¿Qué consecuencias extrae sobre la forma de las regiones positiva y negativa?

En este ejercicio, he hecho dos vector de funciones: uno que llamo al usar `apply` y cuyo único argumento es un vector con dos valores y otro con dos argumentos x e y que llamo para representar la función con `contour`.

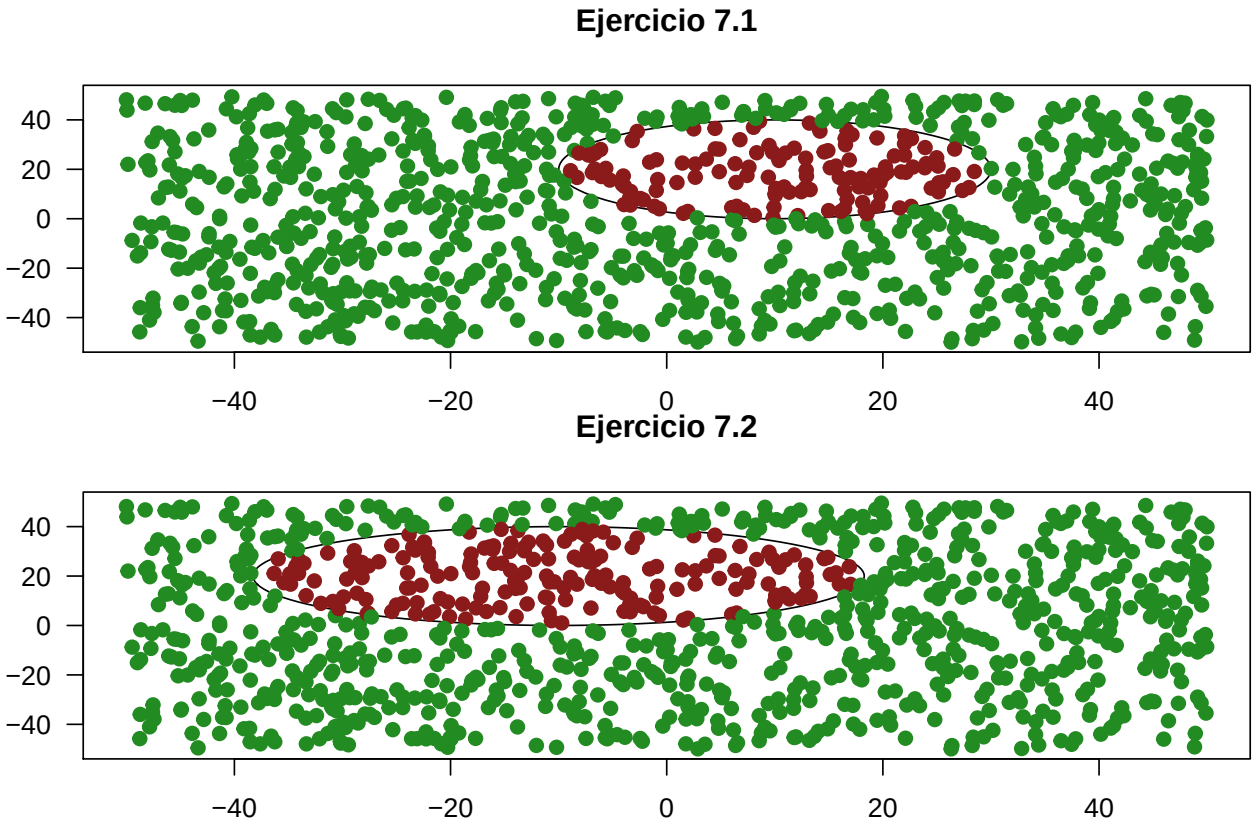
He encapsulado el proceso de dibujar la función y de etiquetar los puntos debido a que en el ejercicio 8 vuelvo a hacer uso de dichas funciones.

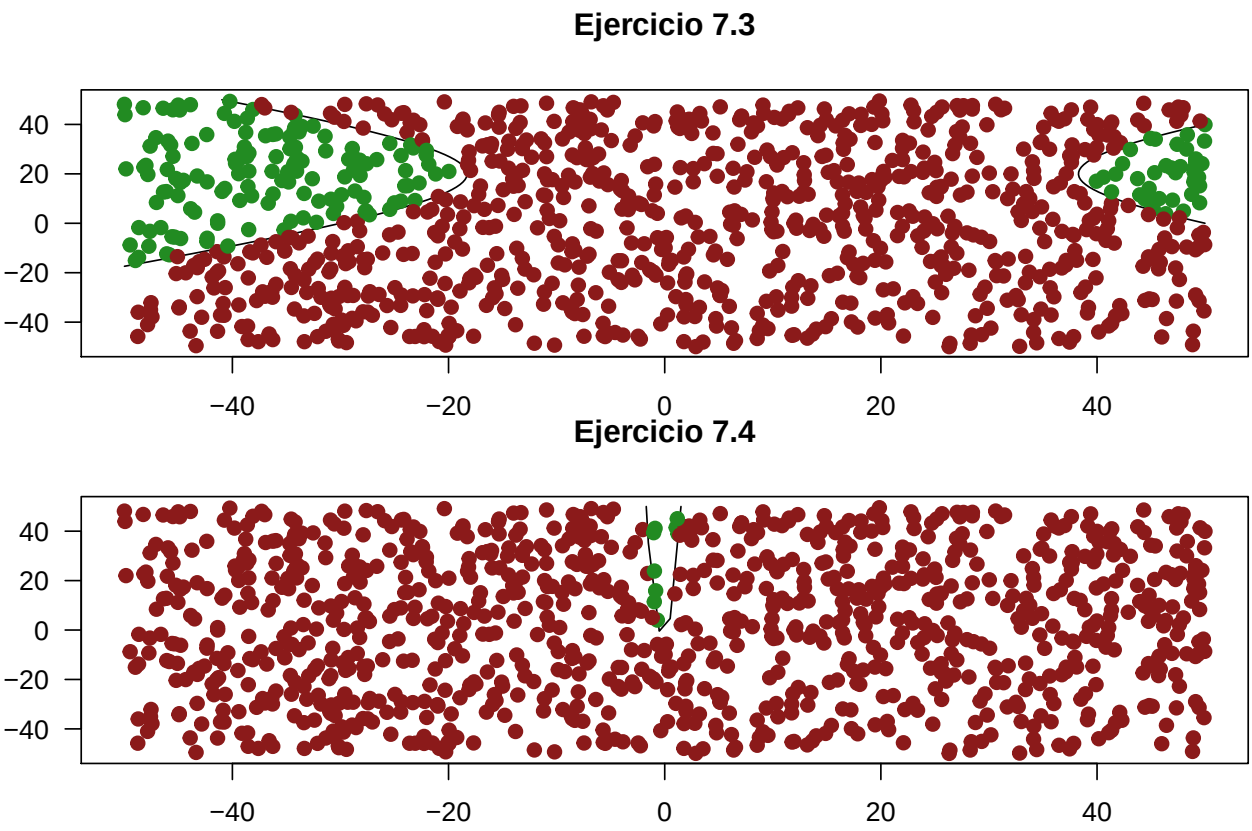
He tenido que usar un bucle `for` a la hora de representar cada función porque necesitaba tener la variable `i` y porque además, ya que en el script se necesita introducir una `s` para poder representar la siguiente función, pensé que la eficiencia del bucle no sería un problema.

```
# definimos un vector con las funciones a evaluar con modo vectorial
funciones = c(function(vec) sign((vec[1] - 10)*(vec[1] - 10) + (vec[2] - 20)*(vec[2] - 20) - 400),
              function(vec) sign(0.5*((vec[1] + 10)*(vec[1] + 10)) + (vec[2] - 20)*(vec[2] - 20) - 400),
              function(vec) sign(0.5*((vec[1] - 10)*(vec[1] - 10)) - (vec[2] - 20)*(vec[2] - 20) - 400),
              function(vec) sign(vec[2] - 20*vec[1]^2 - 5*vec[1] + 3))

# y otro con modo x,y para representar las funciones
funciones_xy = c(function(x,y) (x - 10)*(x - 10) + (y - 20)*(y - 20) - 400,
                 function(x,y) 0.5*((x + 10)*(x + 10)) + (y - 20)*(y - 20) - 400,
                 function(x,y) 0.5*((x - 10)*(x - 10)) - (y - 20)*(y - 20) - 400,
                 function(x,y) y - 20*x^2 - 5*x + 3)

dibuja_funcion <- function(puntos=p, funcion, nombres, titulo) {
    x <- y <- seq(range(puntos[,1])[1],range(puntos[,1])[2],length=100) # guardamos el intervalo a repr
    z <- outer(x,y,funcion)             # calculamos los puntos de la funcion
    contour (                           # la representamos
        x=x, y=x, z=z,
        levels=0, las=1, drawlabels=FALSE, main=titulo
    )
    # representamos los puntos
```

```
    points(x=puntos, col=colors()[138+nombres], lwd=2, pch=19)
}

# funcion que calcula el etiquetado de cada punto según la función que pasemos
# como parámetro
calcula_nombres <- function(funcion, puntos) apply(X=puntos, FUN=funcion, MARGIN=1)
etiqueta_puntos <- function(puntos=p) lapply(funciones, calcula_nombres, puntos=puntos)

ejercicio_siete <- function() {
    # calculamos el etiquetado de los puntos según cada una
    nombres <- etiqueta_puntos()

    # representamos cada función
    for (i in 1:4) {
        dibuja_funcion(funcion=funciones_xy[[i]], nombres=nombres[[i]],
                       titulo=paste("Ejercicio 7.",i, sep=""))
        # print("Pulsa s para ejecutar el siguiente ejercicio...")
        # scan(what=character(), n=1)
    }
}
```

Las gráficas obtenidas han sido:





Creo que la primera gráfica se corresponde con la función de una *elipse*, ya que la región negativa forma un círculo dentro de la región positiva. Lo mismo se aplica a la gráfica obtenida con la función siguiente, debido a que es muy similar.

En el caso de la tercera gráfica, pienso que debe tratarse de una función *hipérbola*, ya que la región positiva se encuentra a ambos lados de la negativa, obteniendo una forma bastante similar a la de la hipérbola.

Por último, pienso que la última gráfica se trata de una función *cuadrática*, debido a la curva obtenida al representar la función. Además, como se aprecia, la región negativa se queda fuera de la curva y la positiva, dentro.

1.8 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 de 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.

La función construida para modificar el etiquetado realizado en el ejercicio 6 es la siguiente:

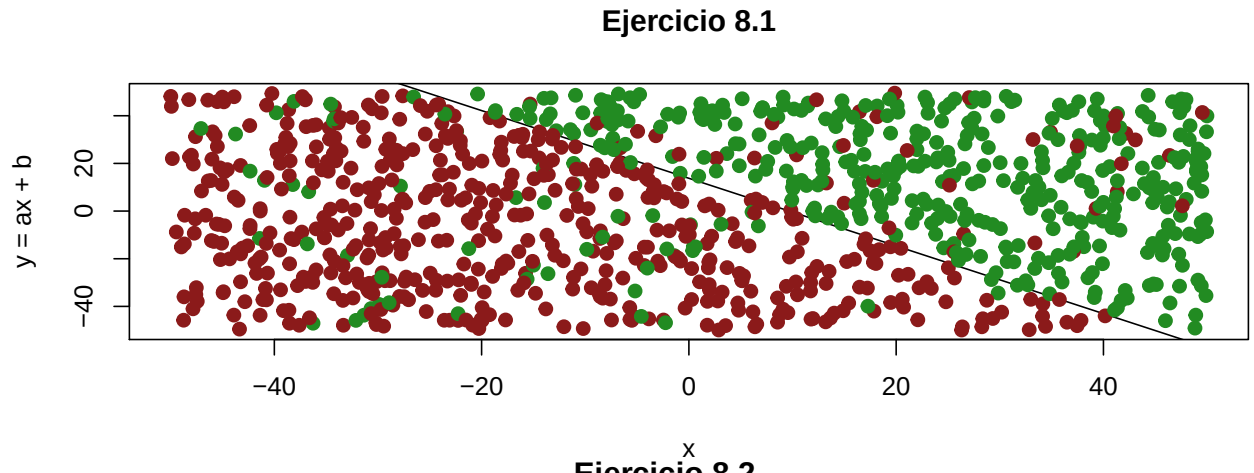
```
mete_ruido <- function(nombres=list(etiqueta_puntos()[[1]], etiqueta_puntos()[[2]],
        etiqueta_puntos()[[3]], etiqueta_puntos()[[4]], obten_param_recta())) {
    # obtenemos los puntos etiquetados como positivos
    positivos = lapply(nombres, function(nom) which(nom %in% 1))

    # y los etiquetados como negativos
    negativos = lapply(nombres, function(nom) which(nom %in% -1))

    # obtenemos indices aleatorios de un 10% de las muestras
    valores_aleatorios <- function (vec) sample(vec, 0.1*length(vec))
```

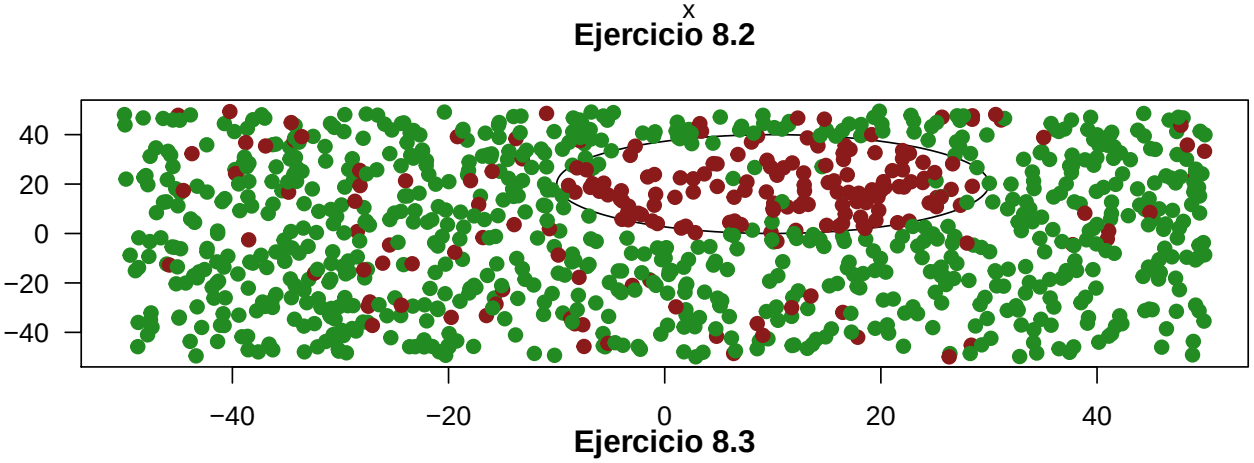
```
    ale_pos = lapply(positivos, valores_aleatorios)
    ale_neg = lapply(negativos, valores_aleatorios)

    # cambiamos el valor de los indices aleatorios obtenidos
    for(i in 1:length(nombres)) {
        nombres[[i]][ale_pos[[i]]] = -1
        nombres[[i]][ale_neg[[i]]] = +1
    }
    nombres
}

ejercicio_ocho <- function() {
    # etiquetamos los puntos del ejercicio 7
    nombres = etiqueta_puntos()
    # añadimos el etiquetado del ejercicio 6 al del ejercicio 7
    nombres = list(etiqueta_puntos()[[1]], etiqueta_puntos()[[2]],
        etiqueta_puntos()[[3]], etiqueta_puntos()[[4]], obten_param_recta())

    # calculamos el etiquetado de la funcion con ruido
    nombres = mete_ruido(nombres=nombres)

    # representamos la recta
    representa_recta(nombres=nombres[[5]], titulo="Ejercicio 8.1")

    # representamos las funciones
    for(i in 1:4) {
        # print("Pulsa s para ejecutar el siguiente ejercicio...")
        # scan(what=character(), n=1)
        dibuja_funcion(funcion=funciones_xy[[i]], nombres=nombres[[i]],
            titulo=paste("Ejercicio 8.",i+1,sep=""))
    }
}
```

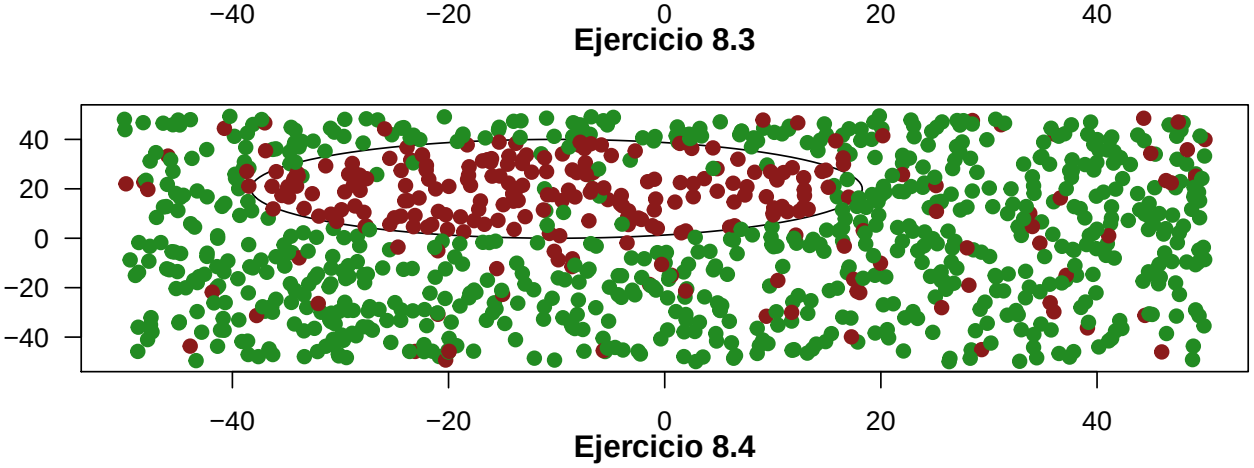
En ella, obtenemos los índices de los puntos etiquetados como positivos y tras eso, obtenemos una lista con valores aleatorios dentro del conjunto de índices obtenidos con el 10 % de tamaño con respecto a la original. Tras esto hacemos lo mismo con los puntos etiquetados como negativos. Por último, cambiamos el valor de los índices aleatorios obtenidos por su contrario. Ésto no puede hacerse antes de obtener los índices de los puntos etiquetados como negativos, ya que si no podríamos obtener también los puntos cambiados aleatoriamente por error.

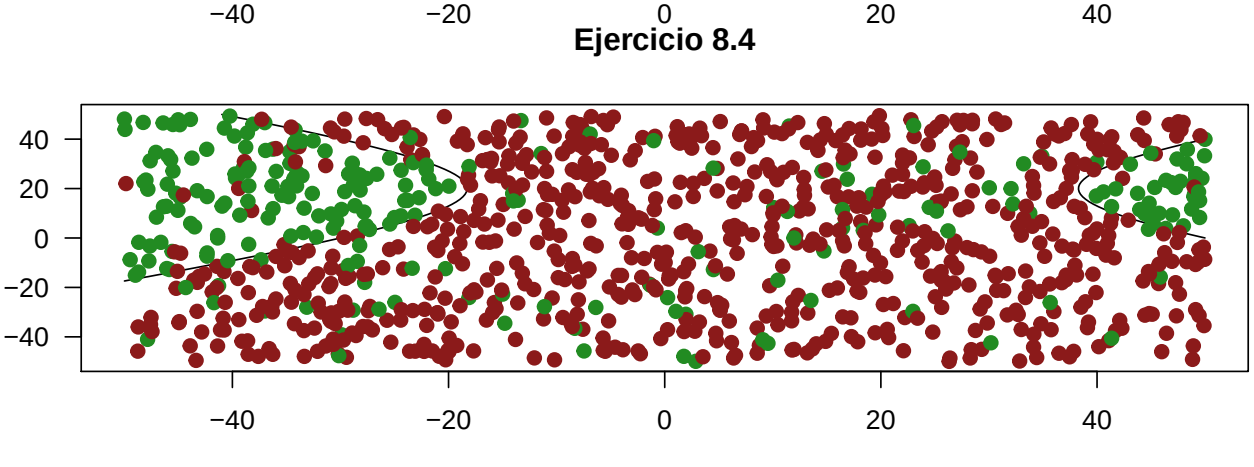
He intentado hacer con `lapply` el cambiar de valor los puntos obtenidos aleatoriamente, pero no lo he conseguido debido a que estaba trabajando con dos matrices a la vez.

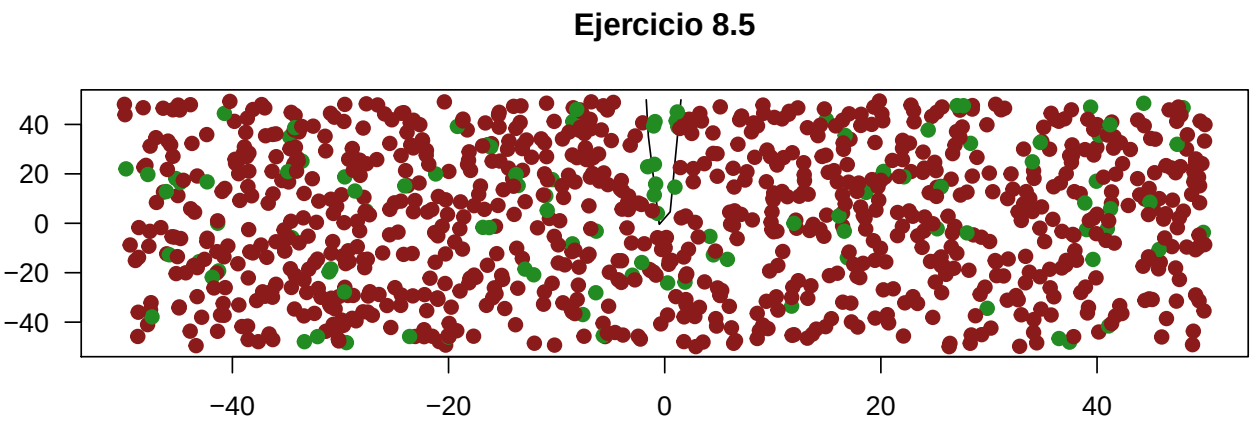
Las gráficas obtenida sería:


Ejercicio 8.1
y = ax + b
−40
0
20
−40
−20
0
20
40
x


Ejercicio 8.2
40
20
0
−20
−40
−40
−20
0
20
40


Ejercicio 8.3
40
20
0
−20
−40
−40
−20
0
20
40


Ejercicio 8.4
40
20
0
−20
−40
−40
−20
0
20
40



Como se aprecia, hay puntos etiquetados como negativos en la zona de puntos etiquetados como positivos y viceversa. Estos puntos, sin embargo, deben de considerarse puro ruido en los datos que ha usado nuestro software para aprender: nunca tendremos un conjunto de datos que aprender que sea perfecto, ya que muchas veces en la toma de una determinada decisión (la salida de nuestro algoritmo) se tiene en cuenta el contexto en el que se realiza dicha función (por ejemplo, pedir un crédito en un banco de ciudad o pedirlo en el banco de tu pueblo) y eso no se incluye en los datos de entrada que el algoritmo usa para aprender. Por eso, algunas veces tendremos datos etiquetados de forma incorrecta, aunque nuestra función aprendida aproxime de forma aceptable a la función objetivo.

2 Ejercicio de Ajuste del Algoritmo Perceptrón

2.1 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.

La función calculada es la siguiente:

```
sol = ajusta_PLA <- function(datos,label, max_iter=1000, vini=c(0,0,0)) {
    cambiado = T
    i = 1        # variable para contar el número de iteraciones
    # añadimos a la matriz datos una tercera columna para poder hacer el producto vectorial
    datos = cbind(rep(1, nrow(datos)), datos)
    # hacemos un bucle iterando hasta max_iter o hasta que deje de cambiar el perceptron
    while (i <= max_iter & cambiado) {
        cambiado = F
        # y hacemos un bucle interno iterando sobre cada dato
        for (j in 1:nrow(datos)) {
            # si el signo es cero, lo convertimos a uno
            signo = sign(datos[j,] %*% vini)
            if (signo == 0) {
                signo = 1
            }
            # comparamos el signo obtenido con sign con el signo que nos ha dado el
            # experto (etiqueta)
            if (label[j] != signo) {
                # si no coinciden, actualizamos el peso: wnew = wold + x*etiqueta
                vini = vini + datos[j,]*label[j]
                cambiado = T # actualizamos el valor de cambiado.
```

```
            }
        }
        i = 1+i # incrementamos el indice
    }
    # parámetros a y b del hiperplano del perceptrón y num iteraciones
    c((-vini[1]/vini[3]), (-vini[2]/vini[3]), i)
}
```

En primer lugar, añadimos una columna de unos al inicio de la matriz para poder hacer bien el producto vectorial con el vector de pesos. Después, hacemos un bucle mientras que o bien se agote el número de iteraciones o bien la recta haya convergido. Dentro de dicho bucle hacemos otro bucle punto por punto donde se calcula el signo con el que la recta clasificaría cada punto de la matriz. Si el signo no coincide con el que nos ha dado el experto, se recalculan los parámetros del hiperplano con la siguiente ecuación:

$$w_{new} = w_{old} + x \cdot y \quad \text{Donde } x \text{ es el punto e } y \text{ su etiqueta}$$

Por último, devolvemos los parámetros a y b del hiperplano y el número de iteraciones necesarias para converger. Dichos parámetros son:

$$a = -\frac{w_1}{w_3}$$

$$b = -\frac{w_2}{w_3}$$

Un ejemplo de ejecución se ve en el ejercicio siguiente.

2.2 Ejecutar el algoritmo PLA con los valores simulados en el apartado 6 del ejercicio anterior 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.

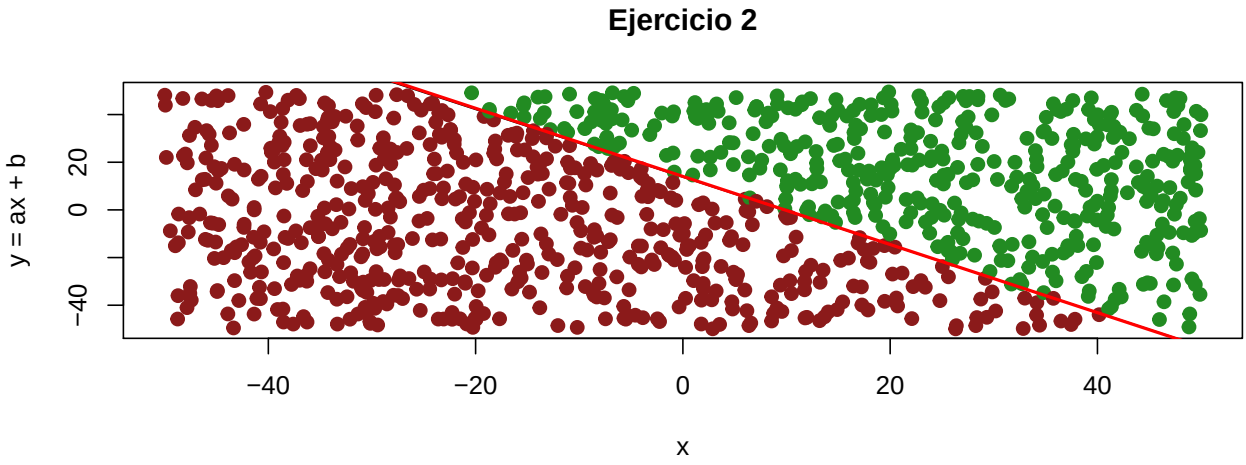
La función desarrollada es:

```
PLA_recta <- function() {
    # inicializando el vector de pesos a cero
    # representamos cada punto calculado con simula_unif del color correspondiente
    ejercicio_seis(titulo="Ejercicio 2")
    # calculamos el perceptrón para los datos del ejercicio 6
    perceptron = ajusta_PLA(datos=p, label=obten_param_recta())

    # representamos la recta hecha por el perceptron
    abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)
    cat(perceptron[3],"iteraciones para converger")

    # inicializando el vector de pesos con valores aleatorios
    iter = vector(length=10) # creamos un vector donde guardar el num iteraciones
    for (i in 1:10) {
        # print("Pulsa s para ejecutar la siguiente gráfica...")
        # scan(what=character(), n=1)
        iter[i] = ajusta_PLA(datos=p, label=obten_param_recta(),
            vini=runif(3, min=0, max=10))[3]
        # ejercicio_seis()
        # abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)
```

```
    }
    # devolvemos el número medio de iteraciones que han hecho falta
    mean(iter)
}
```

En ella guardamos los puntos del ejercicio 6 y los representamos junto a la recta. Después, calculamos el perceptrón inicializando los pesos a 0. Por último, calculamos el perceptrón 10 veces inicializando los pesos con valores aleatorios. Devolvemos la media del número de veces que han sido necesarias para converger.

Un ejemplo de ejecución es:




```
## 133 iteraciones para converger
```

```
## [1] 127.8
```

Como se ve, el perceptrón converge a la casi a la perfección cuando lo inicializamos con peso cero. Cuando inicializamos con pesos aleatorios, necesitamos un mayor número de iteraciones para converger.

2.3 Ejecutar el algoritmo PLA con los datos generados en el apartado 8 del ejercicio anterior 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.

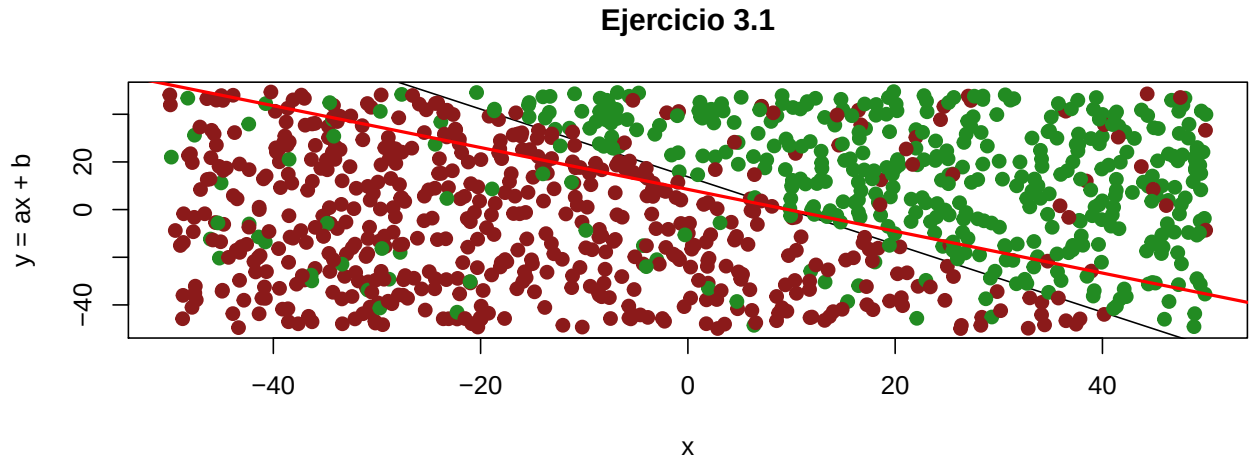
La función desarrollada es:

```
PLA_curva <- function() {
    # obtenemos el etiquetado del ejercicio 8
    nombres = mete_ruido()

    # llamamos a la función con cada una de las funciones
    for (j in 1:3) {
        # con recta
        representa_recta(puntos=p, re=recta, nombres=nombres[[5]],
            titulo="Ejercicio 3.1")
        # calculamos el perceptrón para los datos del ejercicio 8
        perceptron = ajusta_PLA(datos=p, label=nombres[[5]], max_iter=10^j)
        cat(perceptron[3],"iteraciones para converger")
```

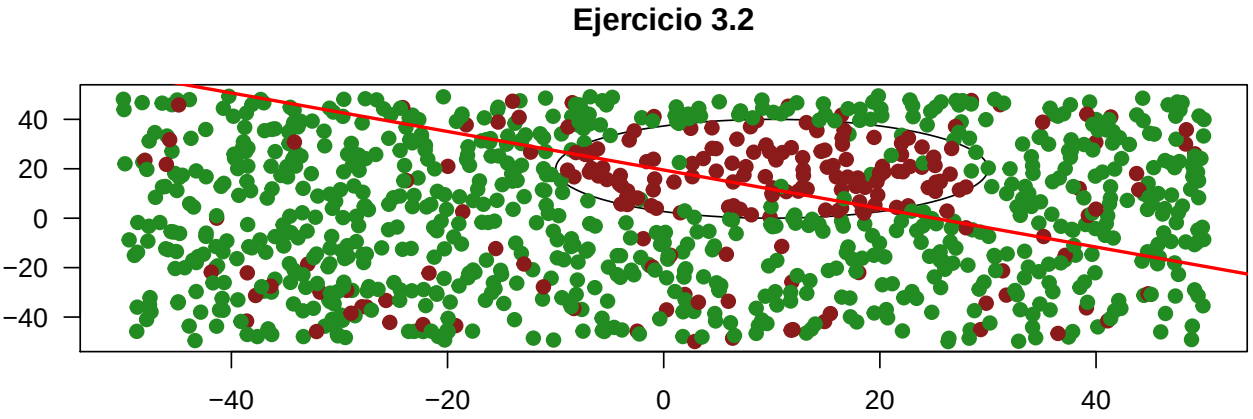
```
        # representamos la recta hecha por el perceptron
        abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)

        # con curvas
        for (i in 1:length(funciones_xy)) {
            # print("Pulsa s para ejecutar el siguiente ejercicio...")
            # scan(what=character(), n=1)
            # representamos la función número i
            dibuja_funcion(funcion=funciones_xy[[i]], nombres=nombres[[i]],
                titulo=paste("Ejercicio 3.",i+1,sep=""))
            # calculamos su perceptrón
            perceptron = ajusta_PLA(datos=p, label=nombres[[i]], max_iter=10^j)
            # imprimimos el número de iteraciones que han sido necesarias
            cat(perceptron[3],"iteraciones para converger")
            # y lo representamos.
            abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)
        }
    }
}
```

En ella obtenemos los puntos con ruido incorporado y representamos cada una de las gráficas inicializando el vector de pesos a cero.

Un ejemplo de ejecución sería:

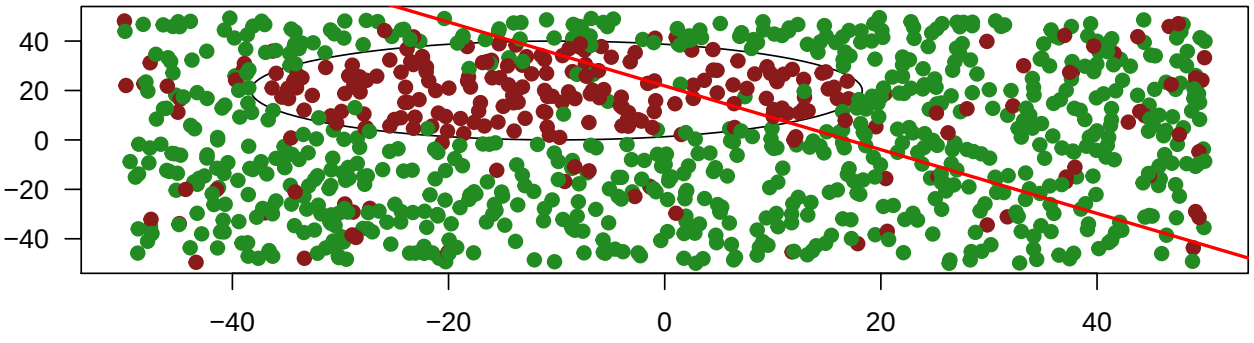



```
## 11 iteraciones para converger
```



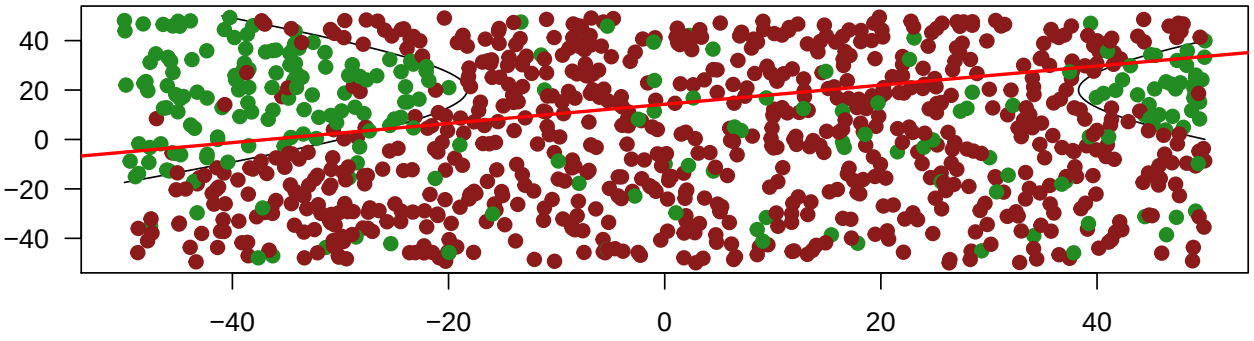
```
## 11 iteraciones para converger
```

Ejercicio 3.3



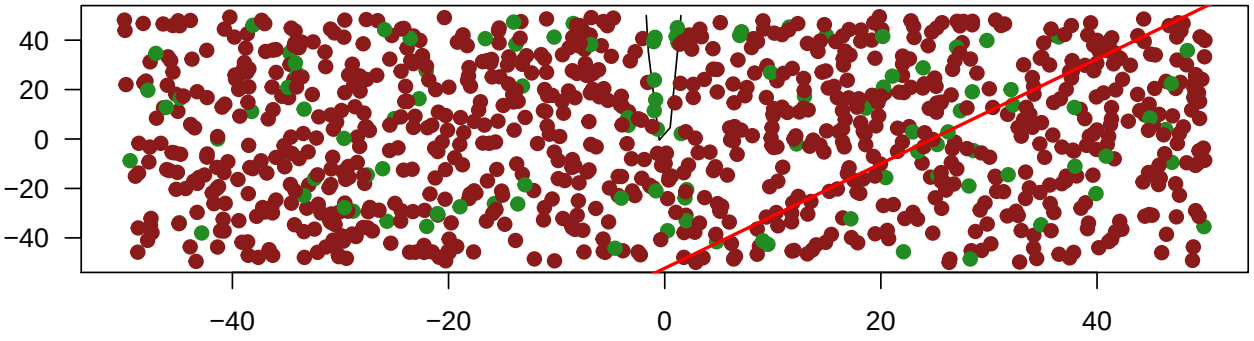

```
## 11 iteraciones para converger
```

Ejercicio 3.4



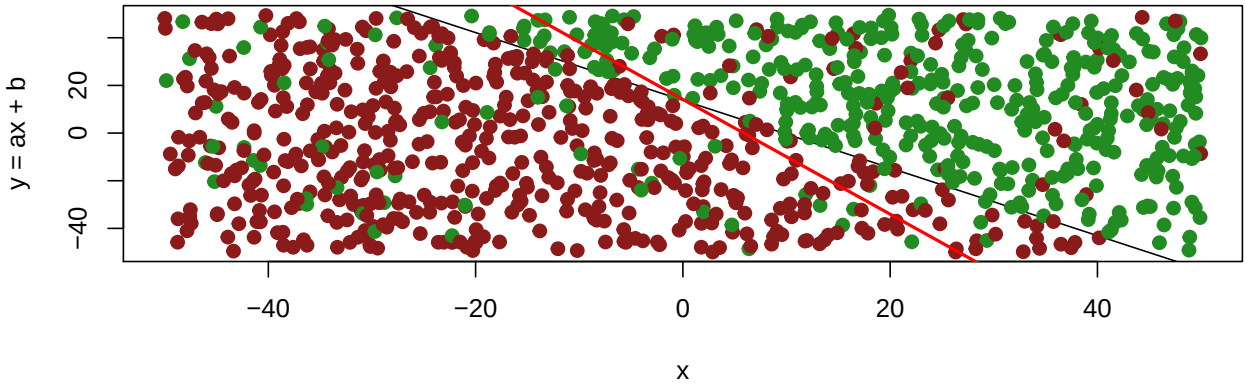

```
## 11 iteraciones para converger
```

Ejercicio 3.5



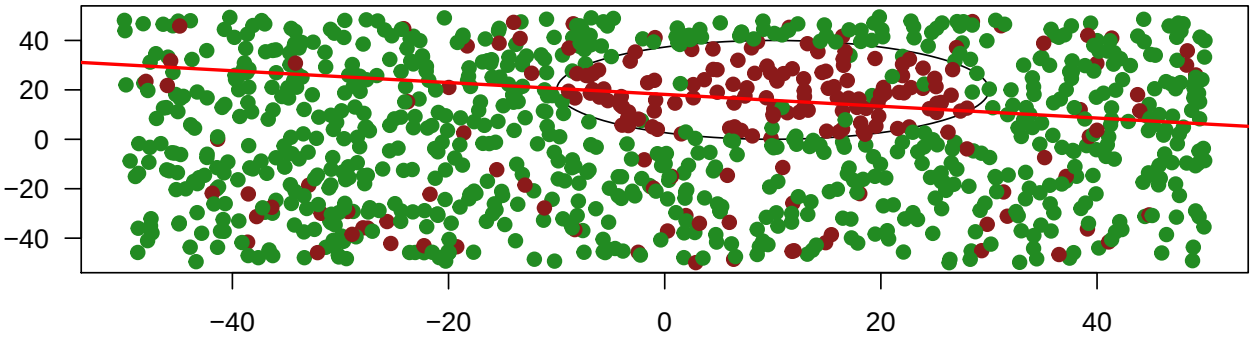

```
## 11 iteraciones para converger
```

Ejercicio 3.1



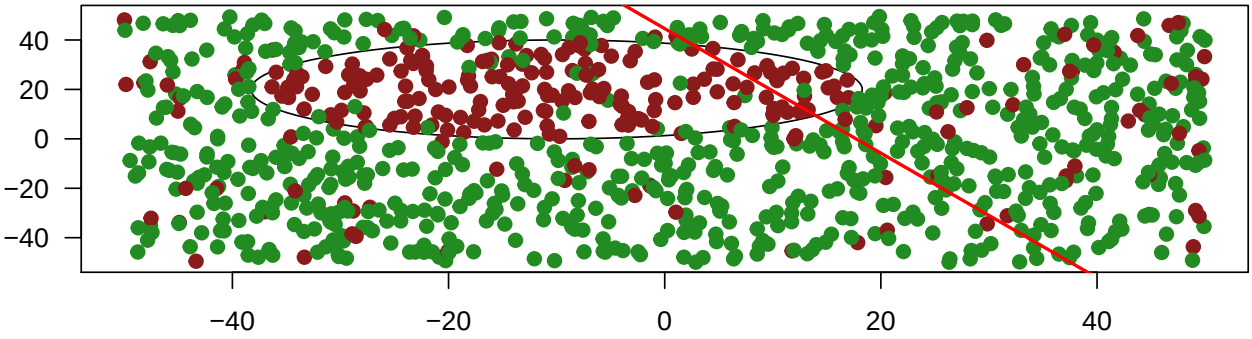

```
## 101 iteraciones para converger
```

Ejercicio 3.2



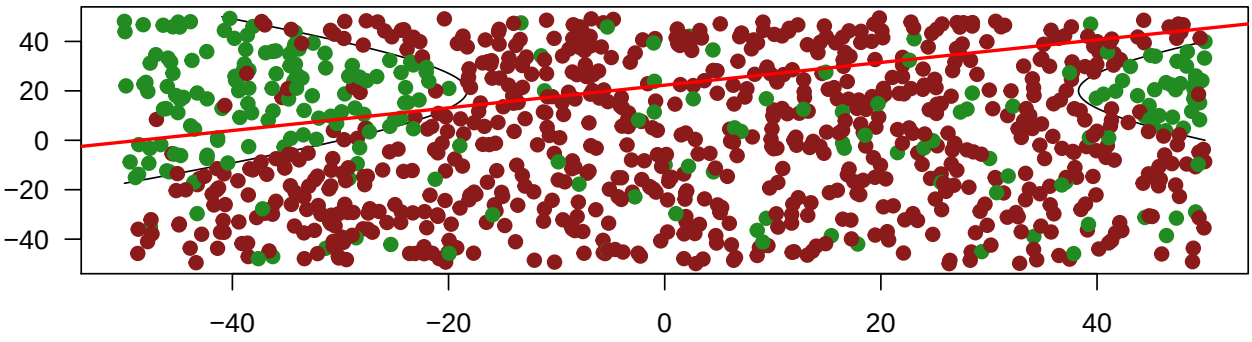

```
## 101 iteraciones para converger
```

Ejercicio 3.3



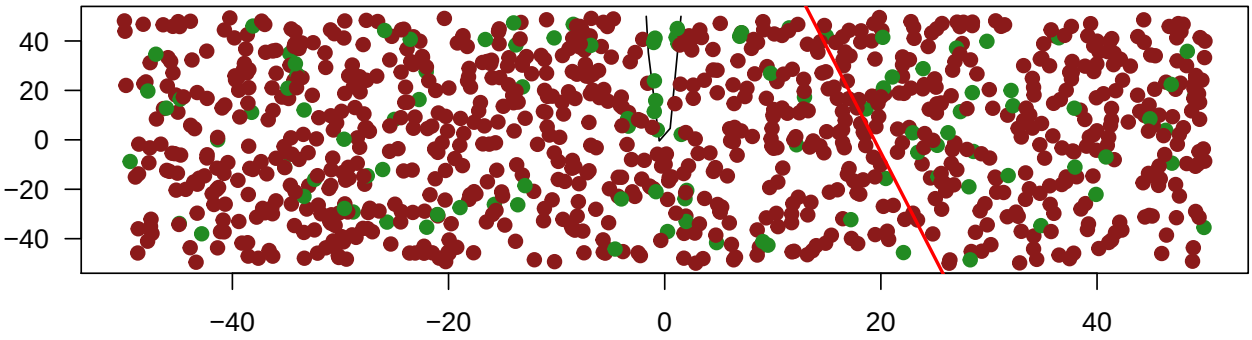

```
## 101 iteraciones para converger
```

Ejercicio 3.4



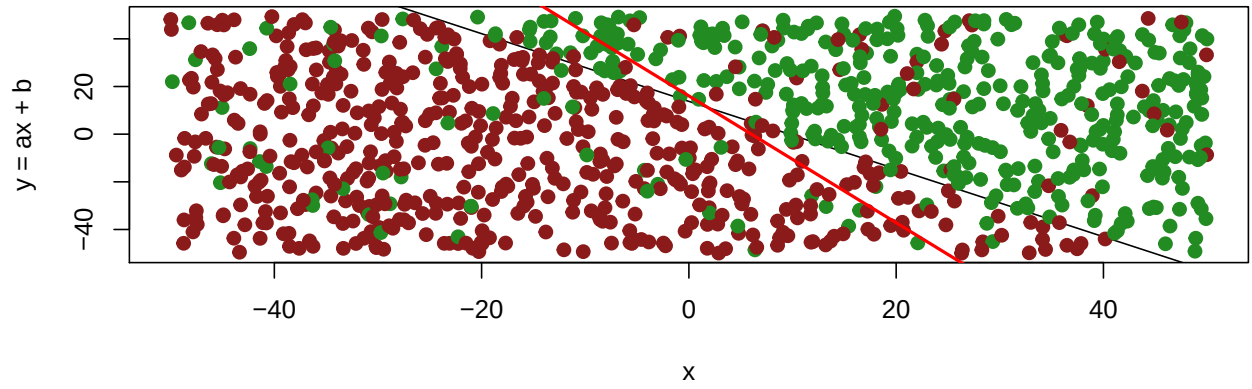

```
## 101 iteraciones para converger
```

Ejercicio 3.5



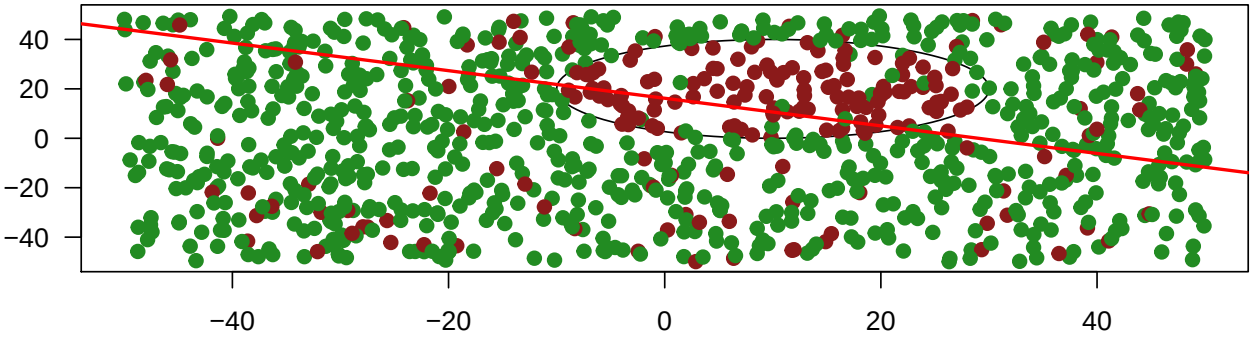

```
## 101 iteraciones para converger
```

Ejercicio 3.1



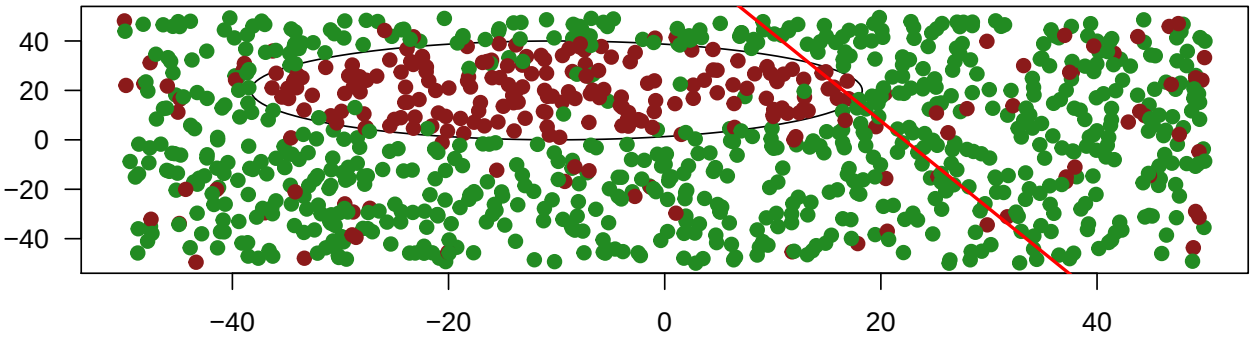

```
## 1001 iteraciones para converger
```

Ejercicio 3.2



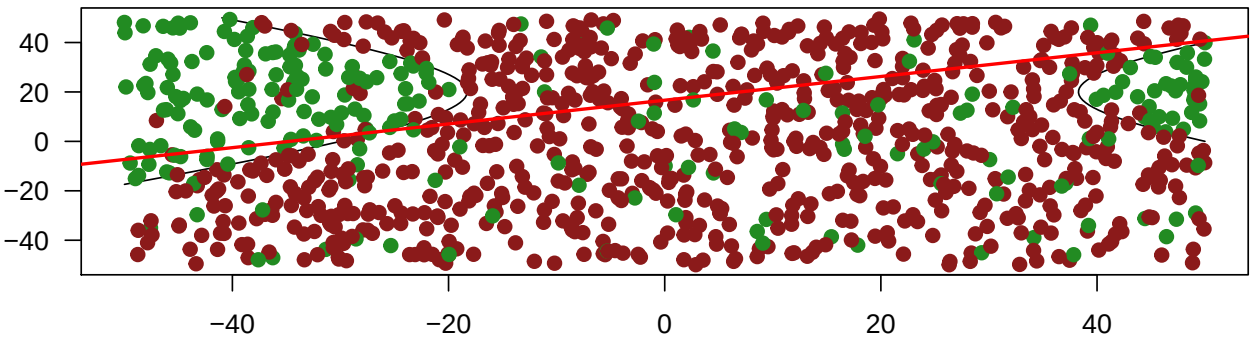

```
## 1001 iteraciones para converger
```

Ejercicio 3.3

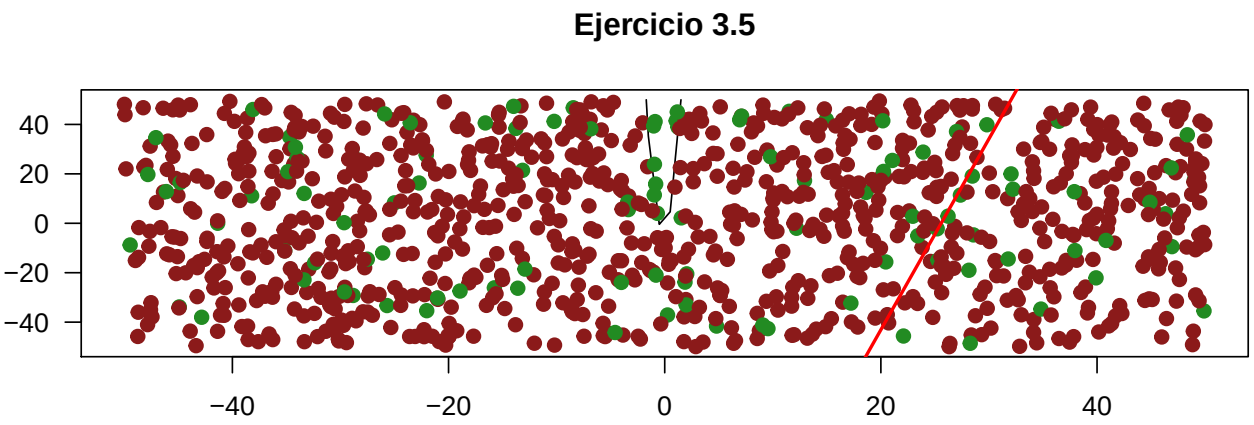



```
## 1001 iteraciones para converger
```

Ejercicio 3.4




```
## 1001 iteraciones para converger
```




```
## 1001 iteraciones para converger
```

Debido al ruido de las muestras, el perceptrón no llega a converger, sino que se agotan el número máximo de iteraciones, tanto en la recta como en las curvas.

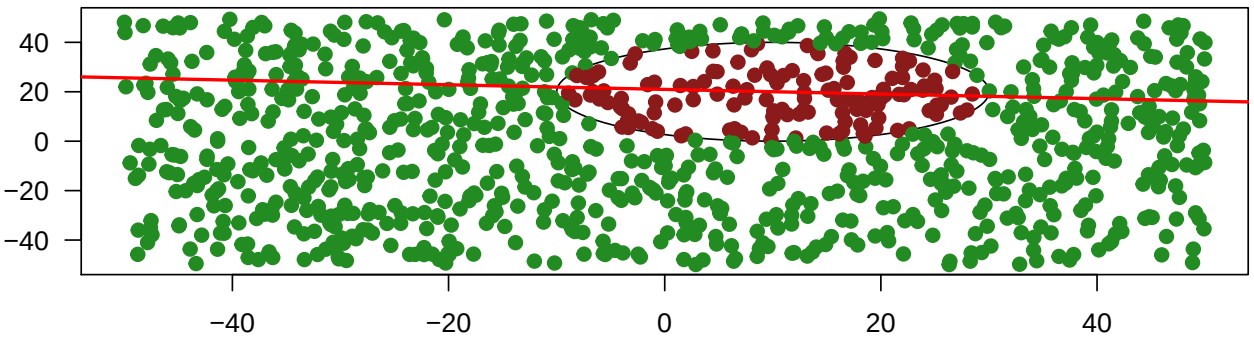
2.4 Repetir el análisis del punto anterior usando la primera función del apartado 7 del ejercicio anterior.

La función desarrollada es:

```
PLA_elipse <- function() {
    # etiquetamos los puntos
    nombres = etiqueta_puntos()[[1]]
    # representamos la función correspondiente
    dibuja_funcion(funcion=funciones_xy[[1]], nombres=nombres, titulo="Ejercicio 4")
    # calculamos su perceptrón
    perceptron = ajusta_PLA(datos=p, label=nombres)
    # y lo representamos.
    abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)
}
```

En ella, obtenemos las etiquetas correspondientes a la primera función del ejercicio 7 y dibujamos la función. Después calculamos el perceptrón y lo representamos con `abline`.

Un ejemplo de ejecución sería:



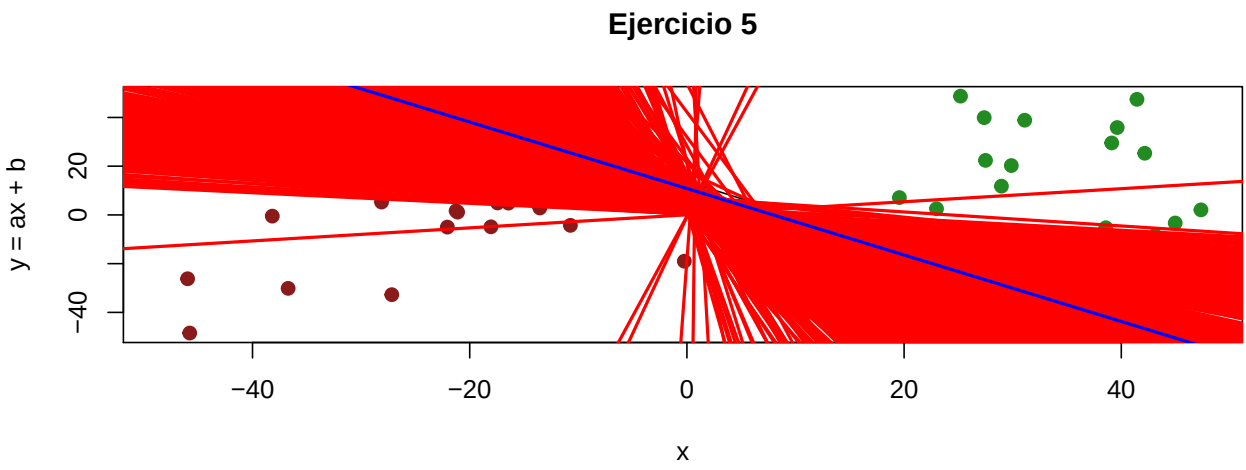
A pesar de que estos puntos no tienen ruido, al no ser una recta el perceptrón no converge, sino que obtiene una clasificación incorrecta de los puntos. Por tanto, podemos concluir que el perceptrón es sólo válido cuando la función a representar se trata de una recta.

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

La función obtenida es:

```
punt_3 = simula_unif(N=50, dim=2, rango=-50:50)

ajusta_PLA_prima <- function(datos,label, max_iter=1000, vini=c(0,0,0), recta,
    puntos) {
    cambiado = T
    i = 1        # variable para contar el número de iteraciones
    # añadimos a la matriz datos una tercera columna para poder hacer el producto vectorial
    datos = cbind(rep(1, nrow(datos)), datos)
    # representamos los puntos y la recta
    ejercicio_seis(titulo="Ejercicio 5", puntos=punt_3)
    # hacemos un bucle iterando hasta max_iter o hasta que deje de cambiar el perceptron
    while (i <= max_iter & cambiado) {
        cambiado = F
        # y hacemos un bucle interno iterando sobre cada dato
        for (j in 1:nrow(datos)) {
            # si el signo es cero, lo convertimos a uno
            signo = sign(datos[j,] %*% vini)
            if (signo == 0) {
                signo = 1
            }
            # comparamos el signo obtenido con sign con el signo que nos ha dado el
            # experto (etiqueta)
            if (label[j] != signo) {
                # si no coinciden, actualizamos el peso: wnew = wold + x*etiqueta
                vini = vini + datos[j,]*label[j]
                cambiado = T # actualizamos el valor de cambiado.
                # representamos la recta
                abline(a=(-vini[1]/vini[3]), b=(-vini[2]/vini[3]),col="red",lwd=2)
                Sys.sleep(0.01)
            }
        }
        i = 1+i # incrementamos el indice
    }
    # representamos la última recta con color azul
    abline(a=(-vini[1]/vini[3]), b=(-vini[2]/vini[3]),col="blue",lwd=2)
}

ej_cinco <- function () {
    ajusta_PLA_prima(datos=punt_3, label=obten_param_recta(puntos=punt_3), recta=recta, puntos=punt_3)
}
```

Un ejemplo de ejecución sería



Aunque no se vea de forma clara debido a la gran cantidad de rectas representadas, se ve cómo el perceptrón va "bailando" por distintos valores hasta converger.

2.6 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 permita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando los datos del apartado 7 del ejercicio anterior.

La función implementada es:

```
evaluar <- function(datos, label, pesos) {
    signos = apply(X=datos, FUN=function(fila) (pesos%*%fila - label)*(pesos%*%fila - label),
        MARGIN=1)
    # signos = sign(datos %*% pesos)
    # signos[signos == 0] = 1
    # print(typeof(signos))
    # length(datos[signos != label, ])/nrow(datos)
    sum(signos)/nrow(datos)
}

# Debemos implementar el algoritmo PLA pocket
sol = ajusta_PLA_MOD <- function(datos,label, max_iter=1000, vini=c(0,0,0)) {
    mejor_vini = vini
    cambiado = T
    i = 1        # variable para contar el número de iteraciones
    # añadimos a la matriz datos una tercera columna para poder hacer el producto vectorial
    datos = cbind(rep(1, nrow(datos)), datos)
    mejor_eval = evaluar(pesos=mejor_vini, datos=datos, label=label)
    # hacemos un bucle iterando hasta max_iter o hasta que deje de cambiar el perceptron
    while (i <= max_iter & cambiado) {
        cambiado = F
        # y hacemos un bucle interno iterando sobre cada dato
        for (j in 1:nrow(datos)) {
            # si el signo es cero, lo convertimos a uno
            signo = sign(datos[j,] %*% mejor_vini)
            if (signo == 0) {
                signo = 1
            }
            # comparamos el signo obtenido con sign con el signo que nos ha dado el
```

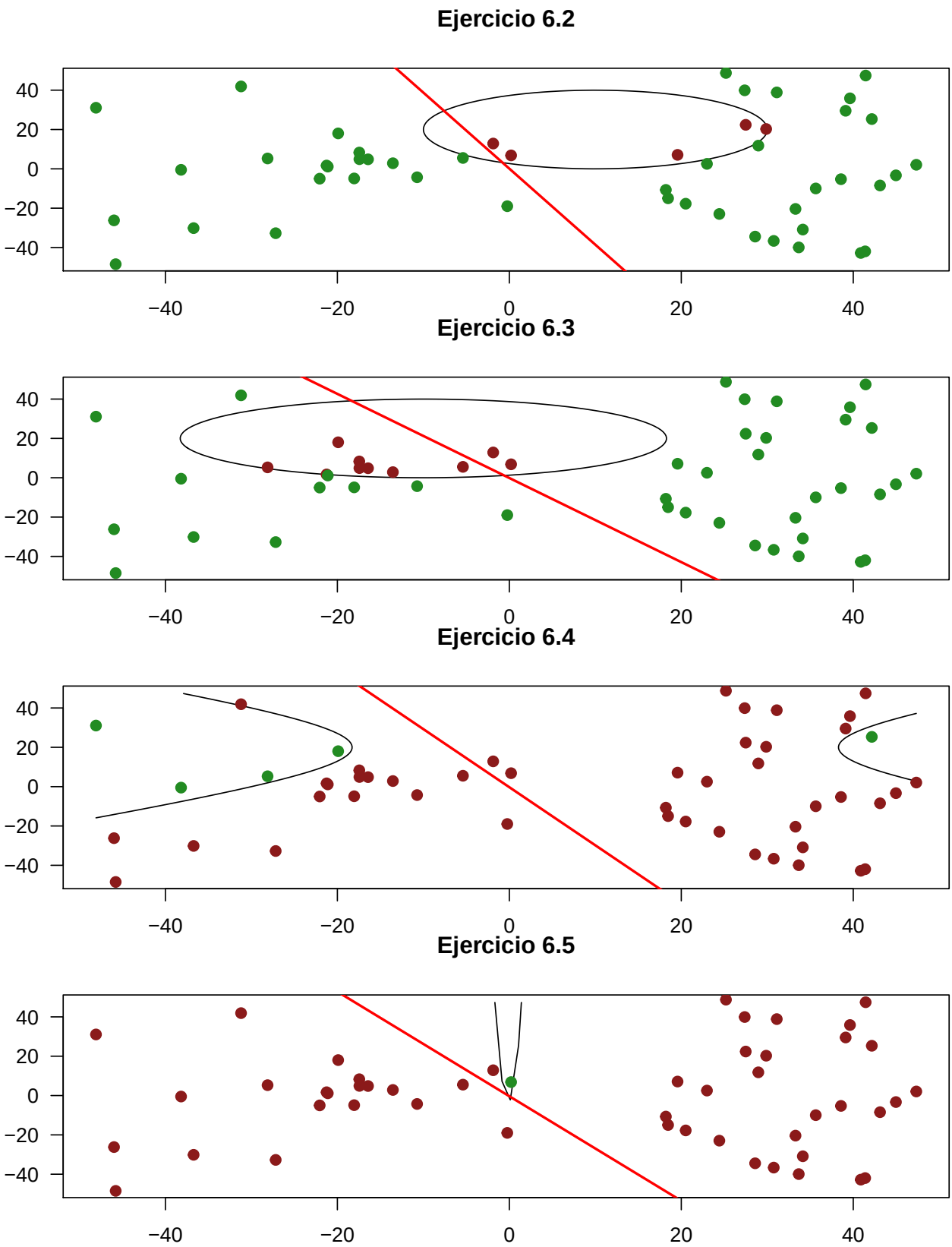
```
            # experto (etiqueta)
            if (label[j] != signo) {
                # si no coinciden, actualizamos el peso: wnew = wold + x*etiqueta
                vini = mejor_vini + datos[j,]*label[j]
                cambiado = T # actualizamos el valor de cambiado.
                # vemos si el nuevo vini es mejor que el mejor global
                eval = evaluar(pesos=vini, datos=datos, label=label)
                if (eval > mejor_eval) {
                    mejor_vini = vini
                    mejor_eval = eval
                }
            }
        }
        i = 1+i # incrementamos el indice
    }
    # parámetros a y b del hiperplano del perceptrón y num iteraciones
    c((-mejor_vini[1]/mejor_vini[3]), (-mejor_vini[2]/mejor_vini[3]), i)
}

ej_seis <- function() {
    nombres = etiqueta_puntos(puntos=punt_3)

    for (i in 1:length(funciones_xy)) {
        dibuja_funcion(puntos=punt_3, funciones_xy[[i]], nombres[[i]], paste("Ejercicio 6.",i+1,sep=""))
        perceptron = ajusta_PLA_MOD(datos=punt_3, label=nombres[[i]])
        abline(a=perceptron[1], b=perceptron[2], col="red",lwd=2)
        # print("Pulsa s para ejecutar el siguiente ejercicio...")
        # scan(what=character(), n=1)
    }
}
```

En ella, hemos definido la función que evalúa un hiperplano como el número de puntos mal etiquetados que se obtienen. La función *pocket* tiene la misma estructura que la función PLA normal, con la diferencia de que a la hora de cambiar el vector de pesos, lo evalúa y compara con un mejor global. Así, conseguimos el efecto "bolsillo".

Para evaluar, usamos la siguiente fórmula:

$$E_{in}(w) = \frac{1}{N}\sum_{n=1}^{N}(w^T \cdot x_n - y_n)^2 \qquad \text{donde } w \text{ son los pesos, } x \text{ es el punto e } y, \text{ las etiquetas}$$

Un ejemplo de ejecución es:



Como se ve, obtenemos una clasificación muchísimo mejor que la obtenida con el algoritmo PLA normal, aunque sigue sin haber convergido, debido a que las funciones a representar no son rectas.

3 Ejercicio sobre regresión lineal

3.1 Abra el fichero *ZipDigits.info* disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el fichero *ZipDigits.train*.

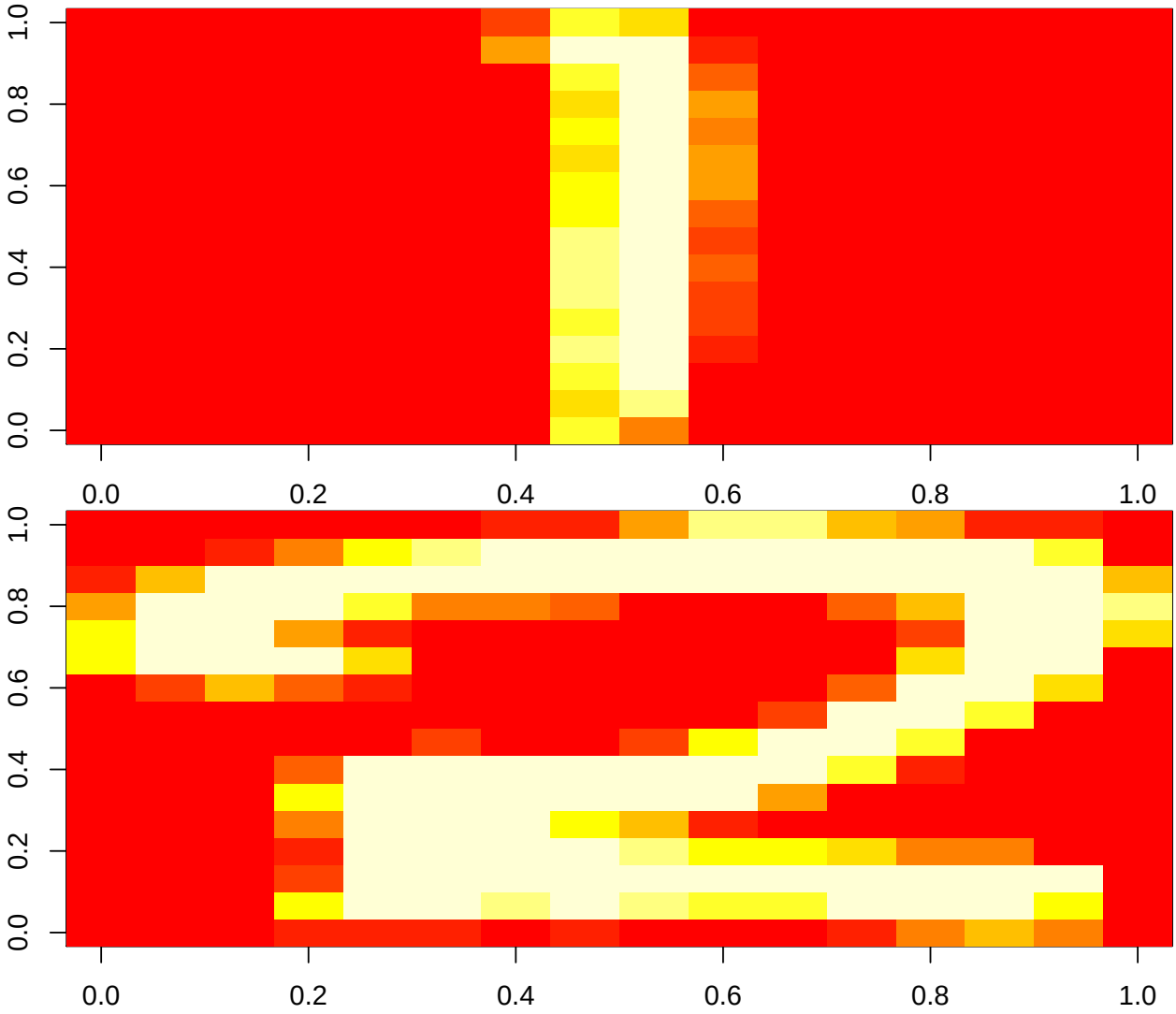
Los datos que se guardan en el fichero siguen el siguiente formato: en primer lugar se guarda el número y a continuación le siguen 256 valores que se corresponden con valores de grises en la imagen.

3.2 Lea el fichero *ZipDigits.train* dentro de su código y visualice las imágenes. Seleccione sólo las instancias de los números 1 y 5. Guárdelas como matrices de tamaño 16x16.

Las funciones desarrolladas son:

```
lee_fichero <- function () {
    # leemos el fichero con los datos de entrenamiento y lo guardamos en un data frame
    zip <- read.table("zip.train", row.names=NULL, quote="\"", comment.char="",
        stringsAsFactors=FALSE)

    # guardamos todos los unos en otro dataframe
    unos = zip[zip$V1 == 1, 2:257]

    # hacemos lo mismo con los cincos
    cincos = zip[zip$V1 == 5, 2:257]

    # devolvemos las listas de instancias
    list(unos, cincos)
}

representa_numero <- function(matriz) {
    # configuramos las dimensiones para que sean 16x16
    dim(matriz) = c(16,16)
    # representamos la matriz
    image(matriz)
}

pintar_numeros <- function(numeros=lee_fichero()) {
    # representamos un uno
    representa_numero(as.matrix(numeros[[1]][1,]))

    # print("Pulsa s para ejecutar la siguiente gráfica...")
    # scan(what=character(), n=1)
    # y un cinco
    representa_numero(as.matrix(numeros[[2]][1,]))
}
```

En la función, leemos los datos con `read.table` y filtramos según el valor de la primera columna. Para representar los números, hemos hecho una función separada en la cual tomamos las instancias de cincos y unos y representamos la primera de todas, obteniendo los siguientes resultados:



A pesar de que obtenemos los números invertidos en la imagen, a la hora de hacer los cálculos no es un detalle importante.

3.3 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 inviriendo el orden de las columnas. Finalmente cambiamos el signo.

La función implementada es:

```
calcula_media <- function(nums) apply(X=nums, FUN=mean, MARGIN=1)
calcula_sim <- function(nums) apply(X=nums,
    FUN=function(fila) -sum(abs(fila[1:256] - fila[256:1])), MARGIN=1)

valor_medio_simetrico <- function(matrices=lee_fichero()) {
    # guardamos en una variable las instancias con unos
    unos = matrices[[1]]
    # y en otra, las instancias con cincos
    cincos = matrices[[2]]
```

```
    # devolvemos una lista con las medias y simetrias de cada uno
    list(list(calcula_media(unos), calcula_sim(unos)),
        list(calcula_media(cincos), calcula_sim(cincos)))
}
```

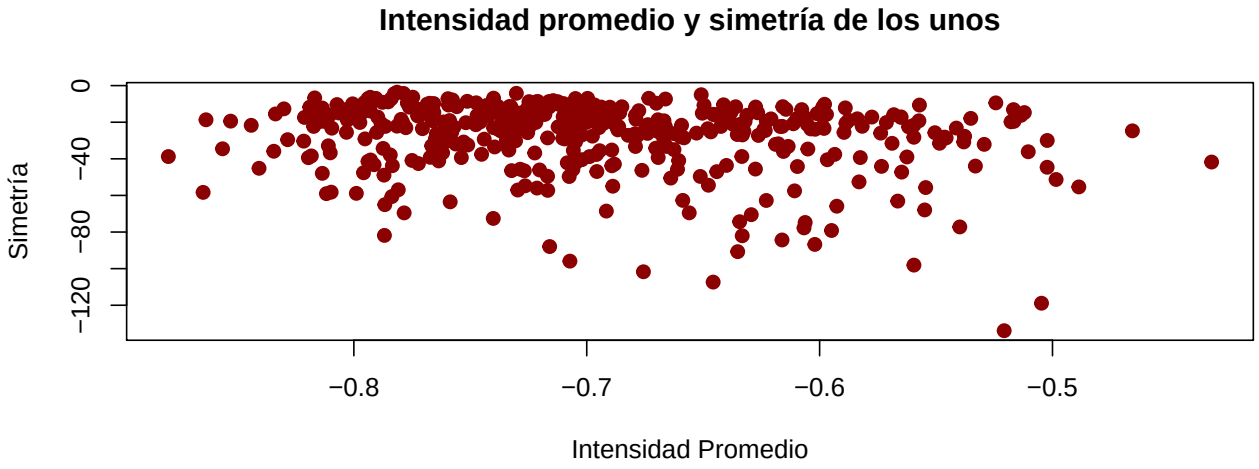
En ella, calculamos en primer lugar las medias de cada matriz usando la función `mean` y después, para el cálculo de simetrías usamos la función `abs` para hacer el valor absoluto y la función `sum` para hacer la sumatoria de las restas. Para poder restar cada número con el correspondiente si se invierte la matriz, hemos restado índices opuestos: el 1 con el 256, el 2 con el 255 y así.

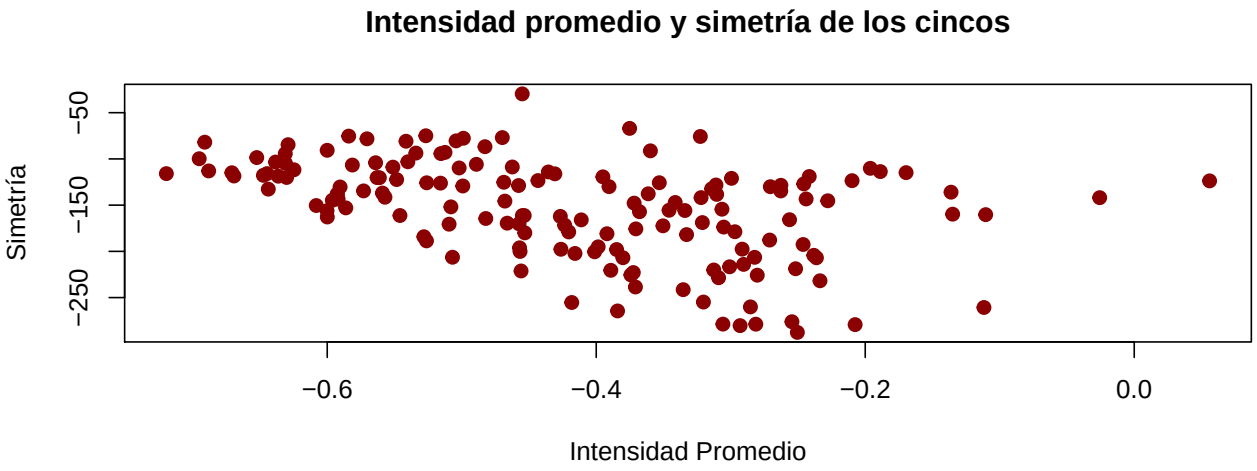
3.4 Representar en los ejes {X = Intensidad Promedio, Y = Simetría} las instancias seleccionadas de unos y de cincos.

La función desarrollada es:

```
representa_param <- function(parametros, titulo) {
    plot(x=parametros[[1]], y=parametros[[2]], pch=19, lwd=2, col=colors()[100],
        main=titulo, ylab="Simetría", xlab="Intensidad Promedio")
}

ej_cuatro <- function(parametros=valor_medio_simetrico()) {
    # representamos en una gráfica los puntos unos
    representa_param(parametros=parametros[[1]],
        titulo="Intensidad promedio y simetría de los unos")

    # print("Pulsa s para ejecutar la siguiente gráfica...")
    # scan(what=character(), n=1)

    # y otra con los cincos
    representa_param(parametros=parametros[[2]],
        titulo="Intensidad promedio y simetría de los cincos")
}
```

En ella, guardamos las simetrías y medias de los unos y los cincos y después, los representamos cada uno en una gráfica. Las gráficas obtenidas son:





Como se ve, los unos tienen la simetría ya la intensidad mucho más concentrada en el centro de la imagen. Todo lo contrario que los cincos con los que obtenemos una gráfica con valores mucho más dispersos.

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

La función implementada es:

```
sol = Regress_Lin <- function(datos, label) {
    # calculamos la pseudoinversa de datos como (X^T·X)^-1 · X^T
    # asumimos que datos*datos^T es una matriz invertible, es decir, cuadrada
    pseudoinversa = solve(t(datos)%*%datos)%*%t(datos)

    # calculamos wlin y la trasponemos
    wlin = pseudoinversa%*%label

    # devolvemos las etiquetas ŷ = X·wlin y wlin
    list(wlin, datos%*%wlin)
}
```

La función usa el algoritmo descrito en el libro *Learning from Data: A short course* de la bibliografía de la asignatura. Dicho algoritmo consiste en:

- Construir la matriz X y el vector y del conjunto de datos de prueba $(x_1, y_1), \cdots, (x_N, y_N)$. Estos datos ya se dan en los parámetros (`datos` es X y `label` es y) por lo que este paso se omite.
- Calcular la *pseudoinversa* $X^\dagger$ de la matriz X, asumiendo que $X^T \cdot X$ es invertible:

$$X^\dagger = (X^T \cdot X)^{-1} \cdot X^T$$

- Devolver $w_{lin} = X^\dagger \cdot y$.

El algoritmo desarrollado llega algo más lejos, pues en vez de devolver w_{lin}, devuelve las estimaciones de y que hacemos a partir de w_{lin}:

$$\hat{y} = X \cdot w_{lin}$$

Con estas estimaciones, representamos el hiperplano de la regresión lineal como veremos en el siguiente apartado.

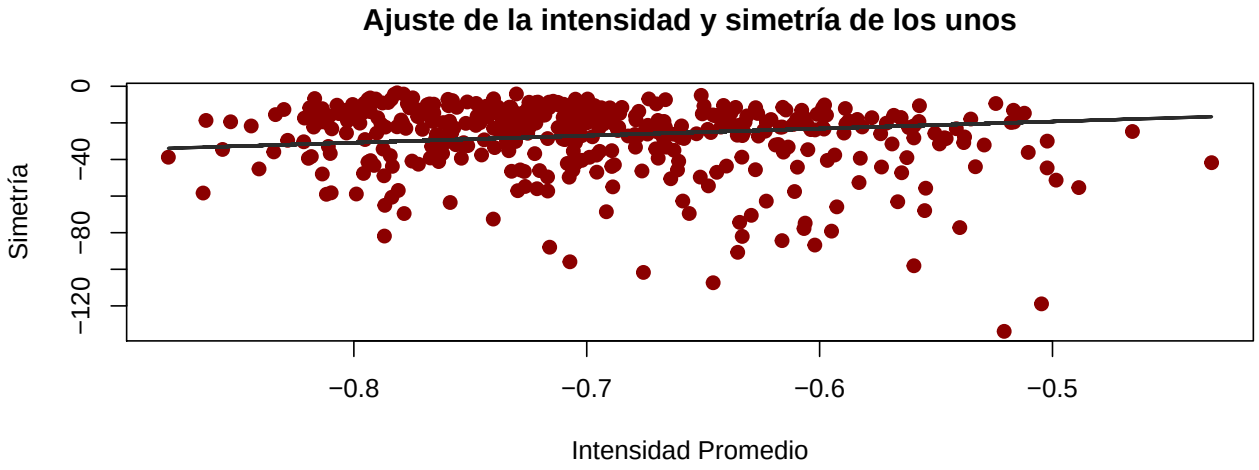
3.6 Ajustar un modelo de regresión lineal a los datos de (Intensidad promedio, Simetría) y pintar la solución junto con los datos. Valorar el resultado.

La función implementada es:

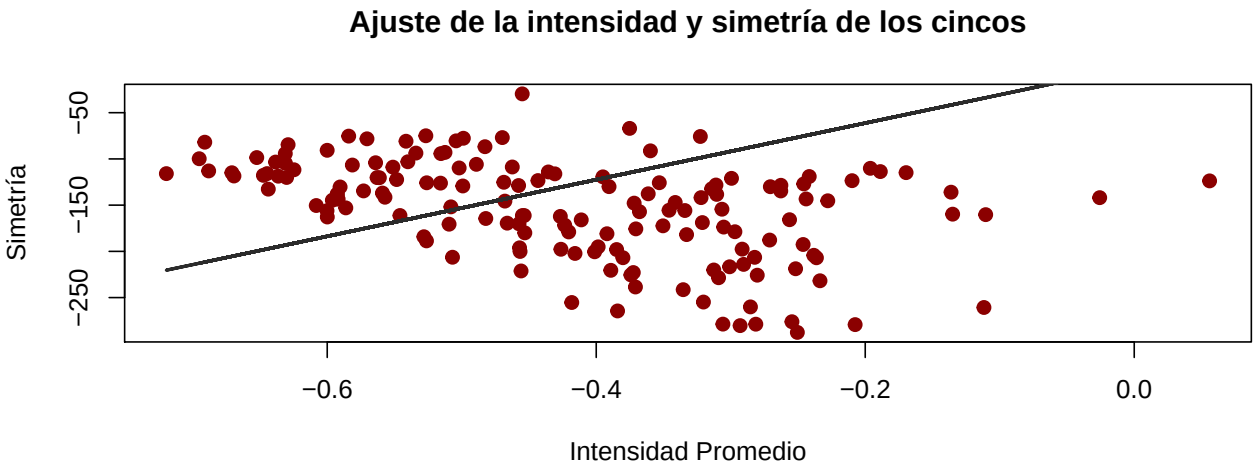
```
ajuste_reg <- function(parametros, titulo) {
    # obtenemos el ajuste con regresión lineal
    ajuste = Regress_Lin(parametros[[1]], parametros[[2]])
    # representamos los puntos
    representa_param(parametros=parametros,
        titulo=titulo)
    # y añadimos el ajuste hecho por la regresión
    lines(x=parametros[[1]], y=ajuste[[2]], lwd=2, col=colors()[278])
}

ej_seis_3 <- function (parametros=valor_medio_simetrico()) {
    # calculamos el ajuste con los unos
    ajuste_reg(parametros[[1]],titulo="Ajuste de la intensidad y simetría de los unos")

    print("Pulsa s para ejecutar la siguiente gráfica...")
    scan(what=character(), n=1)

    # y después, para los cincos
    ajuste_reg(parametros[[2]],titulo="Ajuste de la intensidad y simetría de los cincos")
}
```

En ella, obtenemos el ajuste de la regresión lineal para los datos de media y simetría. Después, representamos en una gráfica estos datos junto con el hiperplano calculado en el ajuste.

Un ejemplo de ejecución sería:




```
## [1] "Pulsa s para ejecutar la siguiente gráfica..."
```



Los hiperplanos obtenidos se ajustan adecuadamente a los datos, por ejemplo, el que obtenemos con el conjunto de las instancias de unos es casi constante mientras que el hiperplano que obtenemos con el conjunto de instancias de los cincos al estar los datos más dispersos no obtenemos una recta tan constante.

3.7 **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 $\mathcal{X} = [-10, 10] \times [-10, 10]$ 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 E_{in}, (el porcentaje de puntos incorrectamente clasificados). Repetir el experimento 1000 veces y promediar los resultados ¿Qué valor obtiene para E_{in}? b) Fijar el tamaño de muestra $N = 100$. Usar regresión lineal para encontrar g y evaluar E_{out}. Para ello generar 1000 puntos nuevos y usarlos para estimar el error fuera de la muestra, E_{out} (porcentaje de puntos mal clasificados). De nuevo, ejecutar el experimento 1000 veces y tomar el promedio. ¿Qué valor obtiene de E_{out}? Valore los resultados. c) Ahora fijamos $N = 10$, ajustamos regresión lineal y usamos el vector de pesos encontrado como un vector inicial de pesos para PLA. Ejecutar PLA hasta que converja a un vector de pesos final que separe completamente la muestra de entrenamiento. Anote el número de iteraciones y repita el experimento 1000 veces ¿Cuál es 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**

La función desarrollada es:

```
experimento_a <- function(wlin, datos, etiquetas) {
    # calculamos Ein como Ein(w) = 1/N(w^TX^TXw - 2w^TX^Ty + y^Ty)
    ein = 1/100 * (t(wlin)%*%t(datos)%*%datos%*%wlin -
        2*t(wlin)%*%t(datos)%*%etiquetas + t(etiquetas)%*%etiquetas)
    ein
}

experimento_b <- function(ein, d, N) {
    # calculamos Eout como Eout(g) = Ein(g) + O(d/N)
    eout = ein + (d/N)
    eout
}

experimento_c <- function() {
    # generamos un conjunto de datos de tamaño 10
```

```
    datos_c = simula_unif(N=10, dim=2, rango=-10:10)
    # generamos una recta aleatoria dentro de dicho intervalo
    rect_c = simula_recta(-10:10)
    # y etiquetamos los puntos según la recta generada
    etiquetas_c = obten_param_recta(puntos=datos_c, re=rect_c)
    # obtenemos las etiquetas aproximadas por la regresion y el wlim
    aprox_c = Regress_Lin(datos=datos_c, label=etiquetas_c)
    # obtenemos la recta aproximada por ŷ
    recta_aprox_c = calcula_recta(datos_c[1,1], aprox_c[[2]][1], datos_c[2,1], aprox_c[[2]][2])
    # llamamos a PLA con vini (b,a,1)
    pla = ajusta_PLA(datos=datos_c,label=etiquetas_c,
        vini=c(recta_aprox_c[2], recta_aprox_c[1], 1))
    # devolvemos el número de iteraciones que se han necesitado para converger
    pla[3]
}

ej_siete_3 <- function() {
    eins = vector(length = 1000)
    eouts = vector(length = 1000)
    for (i in 1:1000) {
        # generamos 100 puntos aleatorios en el intervalo [-10,10]
        datos = simula_unif(N=100, dim=2, rango=-10:10)
        # generamos una recta aleatoria dentro del intervalo [-10,10]
        rect = simula_recta(-10:10)
        # etiquetamos los puntos según la recta obtenida
        etiquetas = obten_param_recta(puntos=datos, re=rect)
        # obtenemos las etiquetas aproximadas por la regresión
        aprox = Regress_Lin(datos=datos, label=etiquetas)
        eins[i] = experimento_a(aprox[[1]], datos, etiquetas)
        eouts[i] = experimento_b(eins[i], ncol(datos), nrow(datos))
    }
    cat("Media de Ein = ", mean(eins),"\n")
    cat("Media de Eout = ",mean(eouts),"\n")
    # hacemos el experimento C
    iter = vector(length=1000)
    for (i in 1:1000) {
        iter[i] = experimento_c()
    }
    cat("Media de iteraciones para converger = ", mean(iter),"\n")
}
```

Para calcular E_{in} usamos la siguiente fórmula:

$$E_{in}(w) = \frac{1}{N}(w^T \cdot X^T \cdot X \cdot w - 2 \cdot w^T \cdot X^T \cdot y + y^T \cdot y) \qquad \text{donde } w \text{ es } w_{lim}$$

En el caso de E_{out} lo calculamos con la siguiente fórmula:

$$E_{out}(g) = E_{in}(g) + O\left(\frac{d}{N}\right) \qquad \text{Donde } d \text{ es la dimensión de la matriz de datos y } N \text{ el número de datos}$$

Por último, para hacer el vector de pesos para el PLA hemos calculado en primer lugar el hiperplano de la regresión con la función `calcula_recta` y para hacer el vector de pesos, hemos usado los valores b y a calculados por `calcula_recta` y un 1.

Los resultados son:

```
## Media de Ein =  0.5339201
## Media de Eout =  0.5539201
## Media de iteraciones para converger =  10.931
```

Al trabajar con un conjunto grande de datos (100) los errores tanto dentro como fuera de la muestra convergen a 05 más o menos, lo cual es un error bastante pequeño. En el caso del perceptrón, converge bastante rápido, teniendo en cuenta que el número máximo de iteraciones son 1000. Por tanto, considero que el trabajo desarrollado consigue ajustarse de manera aceptable a los datos.

3.8 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 $\mathcal{X} = [-10, 10] \times [-10, 10]$ 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 $10\,\%$ del conjunto aleatorio generado. a) Ajustar regresión lineal para estimar los pesos w. Ejecutar el experimento 1000 veces y calcular el valor promedio del error de entrenamiento E_{in}. Valorar el resultado. b) Ahora consideremos $N = 1000$ datos de entrenamiento y el siguiente vector de variables: $(1, x_1, x_2, x_1 \cdot x_2, x_1^2, x_2^2)$. Ajustar de nuevo regresión lineal y calcular el nuevo vector de pesos $\hat{w}$. Mostrar el resultado. c) 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 la función ajustada. Promediar los valores obtenidos $\hat{A}$. ¿Qué valor obtiene? Valorar el resultado.

La función implementada es:

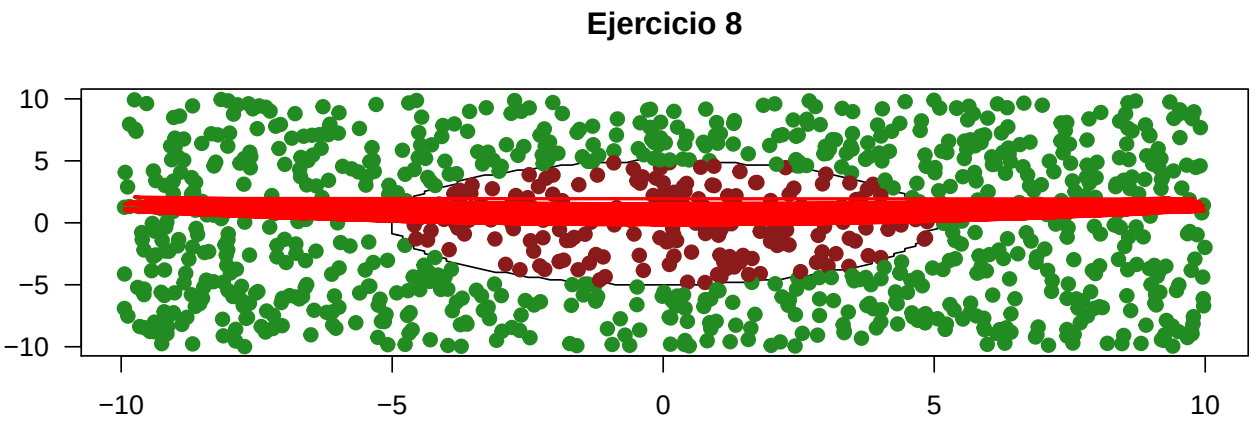
```
ej_ocho_3 <- function() {
    # procedemos de forma idéntica al ejercicio anterior
    eins = vector(length = 1000)

    for (i in 1:1000) {
        # generamos los datos de entrenamiento
        X = simula_unif(N=1000, dim=2, rango=-10:10)
        # declaramos la función
        funcion_ocho <- function(vec) sign(vec[1]*vec[1] + vec[2]*vec[2] - 25)
        # etiquetamos los puntos con la función
        etiquetas = calcula_nombres(funcion=funcion_ocho, puntos=X)
        # metemos ruido en las etiquetas generadas
        etiquetas = mete_ruido(nombres=etiquetas)
        eins[i] = experimento_a(Regress_Lin(datos=X,label=etiquetas)[[1]], X, etiquetas)
    }
    cat("Media de Ein = ",mean(eins),"\n")

    # creamos la matriz de variables con 1,x1,x2,x1x2,x1*x1,x2*x2:
    vector_variables = t(apply(X=X, MARGIN=1,
        FUN=function(p) c(1,p[1],p[2],p[1]*p[2],p[1]*p[1],p[2]*p[2])))
    # Pintamos la función
    dibuja_funcion(puntos=X, function(x,y) sign(x*x + y*y - 25), etiquetas, "Ejercicio 8")
    # le añadimos el w calculado
    lines(x=X[,1], y=Regress_Lin(datos=vector_variables,label=etiquetas)[[2]],col="red",lwd=2)

    # Repetimos lo anterior 1000 veces
    eouts = vector(length = 1000)
```

```
    for (i in 1:1000) {
        # generamos los datos de entrenamiento
        X = simula_unif(N=1000, dim=2, rango=-10:10)
        # declaramos la función
        funcion_ocho <- function(vec) sign(vec[1]*vec[1] + vec[2]*vec[2] - 25)
        # etiquetamos los puntos con la función
        etiquetas = calcula_nombres(funcion=funcion_ocho, puntos=X)
        # metemos ruido en las etiquetas generadas
        etiquetas = mete_ruido(nombres=etiquetas)
        eins[i] = experimento_a(Regress_Lin(datos=vector_variables,
            label=etiquetas)[[1]], vector_variables, etiquetas)
        eouts[i] = experimento_b(eins[i], ncol(vector_variables),
            nrow(vector_variables))
    }
    cat("Media de Eouts = ",mean(eouts),"\n")
}
```

En ella, en primer lugar repetimos 1000 veces el procedimiento de obtener datos aleatorios uniformemente y calcular la regresión lineal de dichos datos y la media del error obtenido. Después, hacemos lo mismo pero con el vector de variables indicado, $(1, x_1, x_2, x_1 \cdot x_2, x_1^2, x_2^2)$, y representamos el resultado en una gráfica. Por último, repetimos esto 1000 veces y calculamos la media del error obtenido fuera de la muestra.

Un ejemplo de ejecución sería:

```
## Media de Ein =  9.979971
```




```
## Media de Eouts =  6.276831
```

Como se ve, el ruido hace que el error dentro de la muestra sea muy grande, pero con el vector de variables indicado baja considerablemente este error fuera de la muestra. Esto se debe, tal y como se indica en la página 103 del libro de teoría *Learning from data: a short course*, a que con este vector de variables podemos tener en cuenta todas las posibles curvas cuadráticas en $\mathcal{X}$ pagando el precio de tener un vector de variables de cinco dimensiones.