Práctica 2 - Aprendizaje Automático

Néstor Rodriguez Vico

27 de abril de 2017

Funciones de utilidad

```
set.seed(3) # semilla para los aleatorios

# por defecto genera 2 puntos entre [0,1] de 2 dimensiones
simula_unif = function (N=2,dims=2, rango = c(0,1)){
  m = matrix(runif(N*dims, min=rango[1], max=rango[2]), nrow = N, ncol=dims, byrow=T)
  m
}

# función simula_gaus(N, dim, sigma) que genera un
# conjunto de longitud N de vectores de dimensión dim, conteniendo números
# aleatorios gaussianos de media 0 y varianzas dadas por el vector sigma.
# por defecto genera 2 puntos de 2 dimensiones
simula_gaus = function(N=2,dim=2,sigma, media){
  if (missing(sigma)) stop("Debe dar un vector de varianzas")
  sigma = sqrt(sigma)  # para la generación se usa sd, y no la varianza
  if(dim != length(sigma)) stop ("El numero de varianzas es distinto de la dimensión")
  simula_gauss1 = function() rnorm(mean = media, dim, sd = sigma) # genera 1 muestra, con las
                                                                  # desviaciones especificadas
  m = t(replicate(N,simula_gauss1())) # repite N veces, simula_gauss1 y se hace la
                                      # traspuesta
  m
}

#  simula_recta(intervalo) una funcion que calcula los parámetros
#  de una recta aleatoria, y = ax + b, que corte al cuadrado [-50,50]x[-50,50]
#  (Para calcular la recta se simulan las coordenadas de 2 ptos dentro del
#  cuadrado y se calcula la recta que pasa por ellos),
#  se pinta o no segun el valor de parametro visible
simula_recta = function (intervalo = c(-1,1), visible=F){
   ptos = simula_unif(2,2,intervalo) # se generan 2 puntos
   a = (ptos[1,2] - ptos[2,2]) / (ptos[1,1]-ptos[2,1]) # calculo de la pendiente
   b = ptos[1,2]-a*ptos[1,1]  # calculo del punto de corte

   if (visible) {  # pinta la recta y los 2 puntos
       if (dev.cur()==1) # no esta abierto el dispositivo lo abre con plot
           plot(1, type="n", xlim=intervalo, ylim=intervalo)
       points(ptos,col=3)  #pinta en verde los puntos
       abline(b,a,col=3)   # y la recta
   }
   c(a,b) # devuelve el par pendiente y punto de corte
}
```

Ejercicio 1: Gradiente Descendente

a) Considerar la función no lineal E(u, v) = (u^2 *e^v - 2v^2 * e^-u)^2. Usar gradiente descendente y para encontrar un mínimo de esta función, comenzando desde el punto (u, v) = (1, 1) y usando una tasa de aprendizaje eta = 0,1.

1) Calcular analíticamente y mostrar la expresión del gradiente de la función E(u, v).

2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de E(u, v) inferior a 10^-4. (Usar flotantes de 64 bits)

3) ¿Qué valores de (u, v) obtuvo en el apartado anterior cuando alcanzó el error de 10^-4 .

El gradiente viene dado por las derivadas con respecto a u y con respecto a v de la función dada en el enunciado.

```
f1 = function(u, v) ((u^2*exp(v) - 2*v^2*exp(-u))^2)
du1 = function(u,v) (2*((2*u*exp(v) + 2*v^2*exp(-u))*(u^2*exp(v) - 2*v^2*exp(-u))))
dv1 = function(u,v) (2*((u^2*exp(v) - 2*(2*v)*exp(-u))*(u^2*exp(v) - 2*v^2*exp(-u))))

GD = function(u = 1, v = 1, eta = 0.1, umbral = 10^(-4), max_iter = 1000, f, du, dv){
  noLlanura = T
  u_0 = 0.0
  v_0 = 0.0
  u_1 = u
  v_1 = v
  iter = 0
  # Como condición de parada tenemos el número de iteraciones
  # y que no estemos en una zona plana, es decir, que haya una diferencia
  # entre el valor de la función en el punto nuevo con respecto al anterior.
  while(iter < max_iter & noLlanura == T) {
    u_0 = u_1
    v_0 = v_1
    u_1 = u_0 - eta * du(u_0, v_0)
    v_1 = v_0 - eta * dv(u_0, v_0)
    iter = iter + 1
    if(abs(f(u_1, v_1) - f(u_0, v_0)) < umbral)
      noLlanura = F
  }
  c(iter, f(u_1, v_1), u_1, v_1, abs(f(u_1, v_1) - f(u_0, v_0)))
}

GD(f = f1, du = du1, dv = dv1)
```

```
## [1]  4.000000e+00  3.855518e-03  9.864573e+00 -2.443828e+01  5.999810e-06
```

En mi caso se ha implementado una versión en la que el gradiente no se normaliza y por lo tanto los primeros saltos son muy grandes y se van reduciendo con el paso de las iteraciones. En 4 iteraciones obtenemos el valor de E(u, v). El valor de u obtenido es 9.864573e+00 y el valor de v es -2.443828e+01.

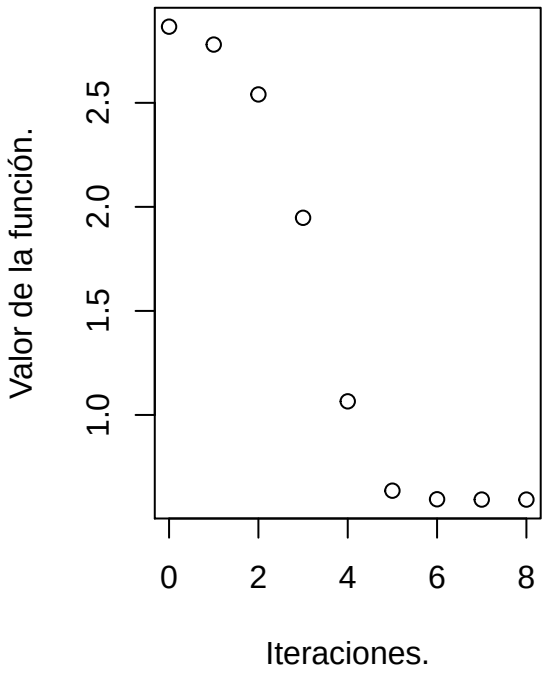
b) Considerar ahora la función f(x, y) = (x - 2)^2 + 2(y - 2)^2 + 2sin(2πx)sin(2πy)

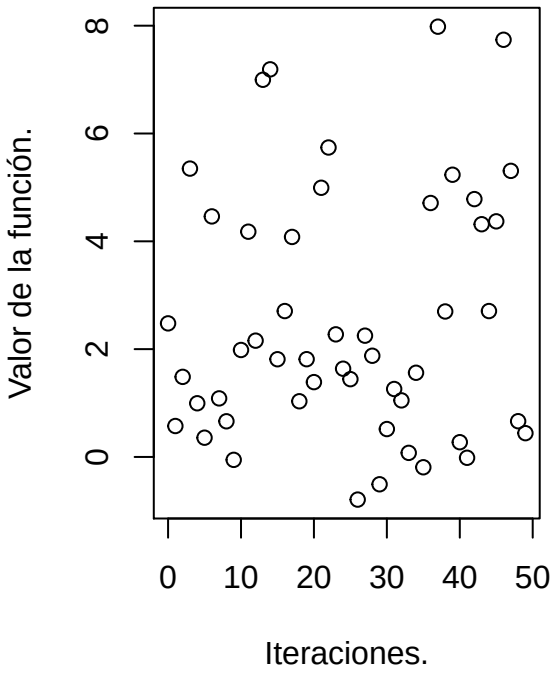
1) Usar gradiente descendente para minimizar esta función. Usar como punto inicial (x0 = 1, y0 = 1), tasa de aprendizaje eta = 0,01 y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando eta = 0,1, comentar las diferencias.

```r
f2 = function(x, y) ((x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y))
dx2 = function(x, y) (2*(x-2) + 2*(cos(2*pi*x)*(2*pi))*sin(2*pi*y))
dy2 = function(x, y) (2*(2*(y-2)) + 2*sin(2*pi*x)*(cos(2*pi*y)*(2*pi)))

GD_graf = function(u = 1, v = 1, eta = 0.1, umbral = 10^(-4), max_iter = 1000, f, du, dv){
  noLlanura = T
  valores=list()
  iteraciones=list()
  u_0 = 0.0
  v_0 = 0.0
  u_1 = u
  v_1 = v
  iter = 0
  # Como condición de parada tenemos el número de iteraciones
  # y que no estemos en una zona plana, es decir, que haya una diferencia
  # entre el valor de la función en el punto nuevo con respecto al anterior.
  while(iter < max_iter & noLlanura == T) {
    u_0 = u_1
    v_0 = v_1
    u_1 = u_0 - eta * du(u_0, v_0)
    v_1 = v_0 - eta * dv(u_0, v_0)
    valores = c(valores, f(u_1, v_1))
    iteraciones = c(iteraciones, iter)
    iter = iter + 1
    if(abs(f(u_1, v_1) - f(u_0, v_0)) < umbral)
      noLlanura = F
  }
  valores=unlist(valores)
  iteraciones=unlist(iteraciones)
  valores = matrix(valores, nrow=length(valores), ncol=1)
  iteraciones = matrix(iteraciones, nrow=length(iteraciones), ncol=1)
  salida = cbind(iteraciones, valores)
  plot(salida, xlab="Iteraciones.", ylab="Valor de la función.")
  c(iter, f(u_1, v_1), u_1, v_1, abs(f(u_1, v_1) - f(u_0, v_0)))
}

par(mfrow=c(1,2))
GD_graf(eta = 0.01, f = f2, du = dx2, dv = dy2, max_iter = 50)
```

```
## [1] 9.000000e+00 5.932713e-01 7.821293e-01 1.287100e+00 5.089406e-05
```

```r
GD_graf(eta = 0.1, f = f2, du = dx2, dv = dy2, max_iter = 50)
```






```
## [1] 50.0000000  0.4412487  1.9612982  1.3779351  0.2220038
```

```
par(mfrow=c(1,1))
```

Podemos ver que usando un valor de eta = 0.01 se converge y obtenemos un mínimo mientras que si usamos un valor de eta = 0.1 no converge, por eso podemos ver en la gráfica que los puntos parecen "aleatorios".

2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (2,1, 2,1), (3, 3), (1,5, 1,5), (1, 1). Generar una tabla con los valores obtenidos

```
GD(u = 2.1, v = 2.1, eta = 0.1, f = f2, du = dx2, dv = dy2, max_iter = 50)
```

```
## [1] 50.000000  2.407525  2.392957  1.067868  1.162009
```

```
GD(u = 3, v = 3, eta = 0.1, f = f2, du = dx2, dv = dy2, max_iter = 50)
```

```
## [1] 50.00000000 -0.03433936  2.29845425  1.98968060  0.04628135
```

```
GD(u = 1.5, v = 1.5, eta = 0.1, f = f2, du = dx2, dv = dy2, max_iter = 50)
```

```
## [1] 50.000000  2.362917  1.546374  3.140278  5.842274
```

```
GD(u = 1, v = 1, eta = 0.1, f = f2, du = dx2, dv = dy2, max_iter = 50)
```

```
## [1] 50.0000000  0.4412487  1.9612982  1.3779351  0.2220038
```

Punto de Inicio	x	y	f(x, y)
(2.1, 2.1)	2.392957	1.067868	2.407525
(3, 3)	2.29845425	1.98968060	-0.03433936
(1.5, 1.5)	1.546374	3.140278	2.362917
(1, 1)	1.9612982	1.3779351	0.4412487

c) ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

Como podemos ver los valores de x e y obtenidos dependen notablemente del punto de partida escogido, por lo tanto encontrar un mínimo global de una función depende del punto de partida que elijamos. También, como podemos ver en el apartado 1 encontrar un mínimo también depende de elegir una buena tasa de aprendizaje o no.

Ejercicio 2: Regresión Logística

En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que la etiqueta y es una función determinista de x.

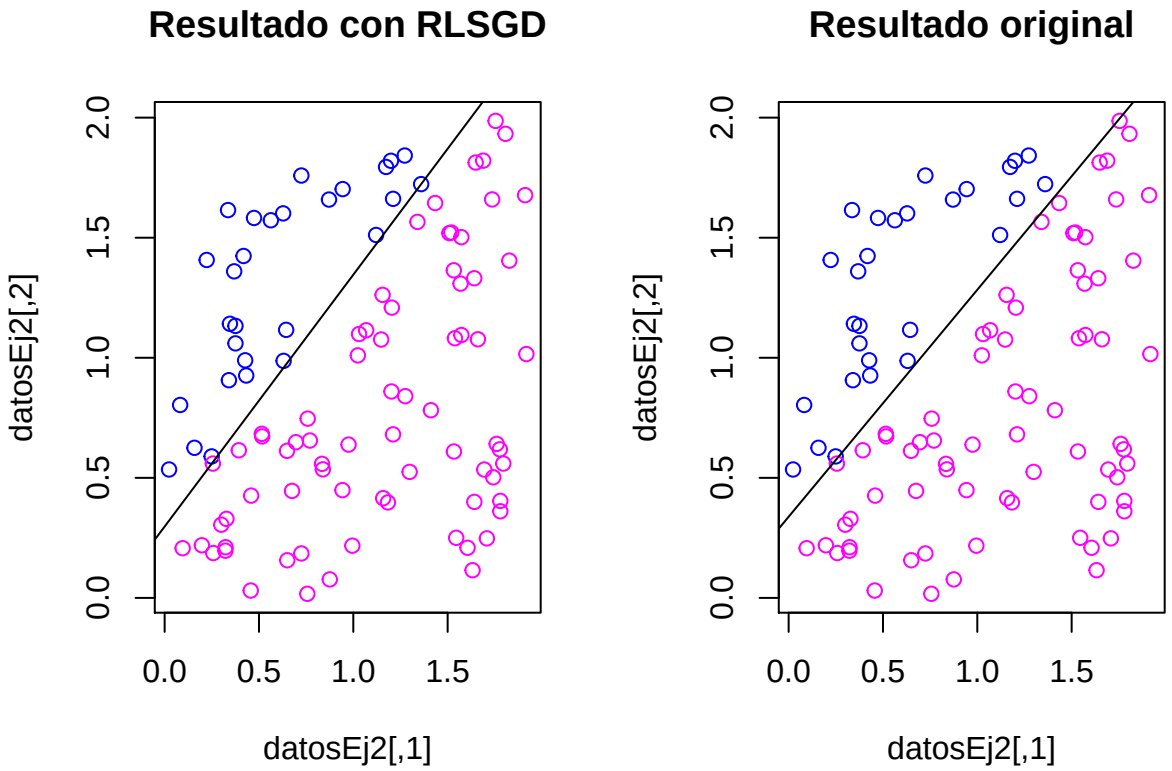
Consideremos d=2 para que los datos sean visualizables, y sea X = $[0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada x preteneciente a X . Elegir una línea en el plano que pase por X como la frontera entre f(x) = 1 (donde y toma valores +1) y f(x) = 0 (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar N = 100 puntos aleatorios $\{x_n\}$ de X y evaluar las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida.

a) Implementar Regresión Logística(RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:

- *Inicializar el vector de pesos con valores 0.*
- *Parar el algoritmo cuando* $||w^{(t-1)} - w^{(t)}|| < 0,01$*, donde w(t) denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.*
- *Aplicar una permutación aleatoria, 1, 2, . . . , N , en el orden de los datos antes de usarlos en cada época del algoritmo.*
- *Usar una tasa de aprendizaje de eta = 0,01*

```
datosEj2 = simula_unif(N = 100, dims = 2, rango = c(0,2))
rectaEj2 = simula_recta(intervalo = c(0,2))
valor_y = function(x, pendiente, valor){ pendiente*x+valor }
etiquetasEj2 = sign(valor_y(datosEj2[,1], rectaEj2[1], rectaEj2[2]) - datosEj2[,2])

norma = function(v){
  sqrt(sum(v^2))
}

RLSGD = function(datos, etiquetas, eta = 0.01, umbral = 0.01, max_iter = 550){
  indices = c(1:nrow(datos))
  datos = cbind(1,datos)
  w_1 = c(0, 0, 0)
  seguirEjecucion = T
  iter = 0

  while(iter < max_iter & seguirEjecucion){
    w_0 = w_1
    indices = sample(indices)
    for(i in indices){
      w_1 = w_1 - eta*(-etiquetas[i]*datos[i,])/as.vector((1+exp(etiquetas[i]*(w_1%*%datos[i,]))))
    }
    if(norma(w_0 - w_1) < umbral)
```

```
      seguirEjecucion = F
    iter = iter + 1
  }

  w_1
}

par(mfrow=c(1,2))
sol = RLSGD(datos = datosEj2, etiquetas = etiquetasEj2)
plot(datosEj2, col=etiquetasEj2+5, main="Resultado con RLSGD")
abline(-sol[1]/sol[3], -sol[2]/sol[3])
plot(datosEj2, col=etiquetasEj2+5, main="Resultado original")
abline(rectaEj2[2], rectaEj2[1])
```




```
par(mfrow=c(1,1))

#Calculamos el error usando el error de regresión logistica
#que aparece en la página 95 del libro "Learning from Data"
sum(log(1+exp(-etiquetasEj2*(t(sol)%*%t(cbind(1, datosEj2))))))/nrow(datosEj2)
```

```
## [1] 0.09726187
```

b) Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar E_{out} usando para ello un número suficientemente grande de nuevas muestras (>999).

```
datos_test = simula_unif(N = 1000, dims = 2, rango = c(0,2))

recta_test = rectaEj2
```

```
etiquetas_test = sign(valor_y(datos_test[,1], recta_test[1], recta_test[2]) - datos_test[,2])

#Calculamos el error usando el error de regresión logistica
#que aparece en la página 95 del libro "Learning from Data"
sum(log(1+exp(-etiquetas_test*(t(sol)%*%t(cbind(1, datos_test))))))/nrow(datos_test)
```

```
## [1] 0.1005714
```

Como podemos ver obtenemos un valor de E_{out} parecido a E_{in} y bastante bajos, lo cual es bastante bueno.

Ejercicio 3: Clasificación Digitos

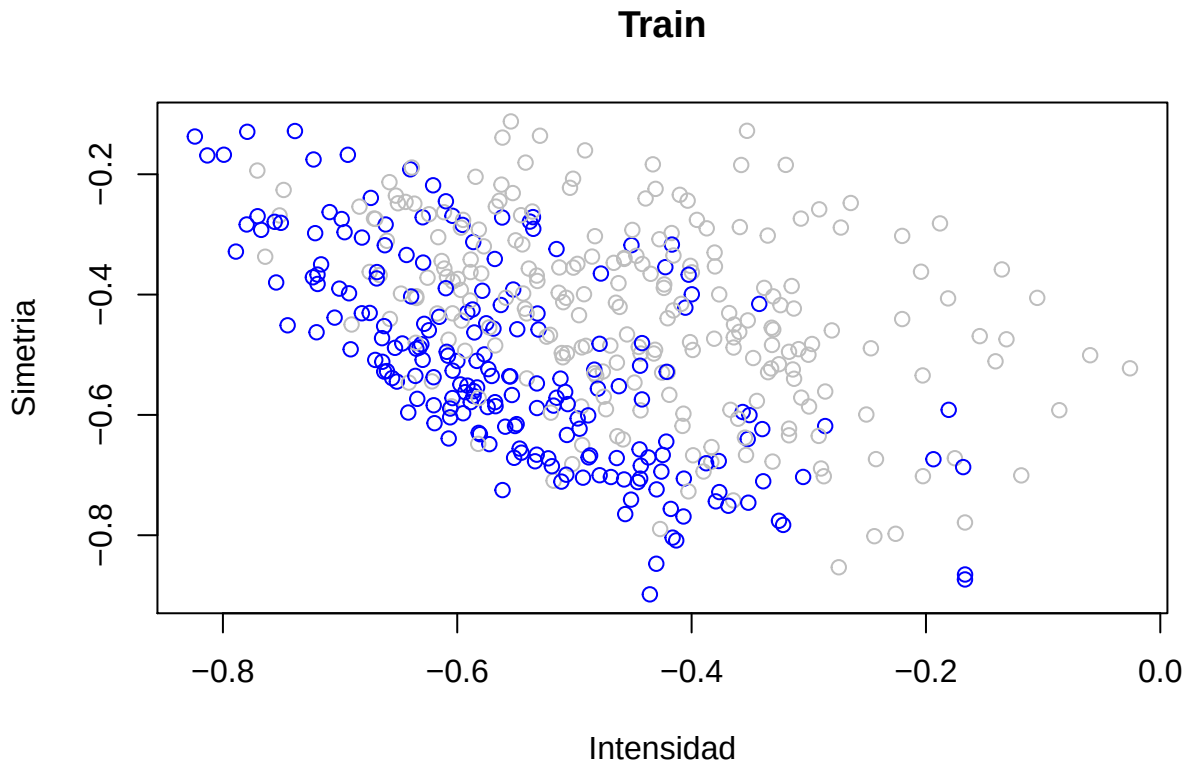
Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de intensidad promedio y simetría en la manera que se indicó en el ejercicio 3 del trabajo 1.

```
# Leemos los datos
digit.train = read.table("./datos/zip.train", quote="\"", comment.char="",
                         stringsAsFactors=FALSE)
```

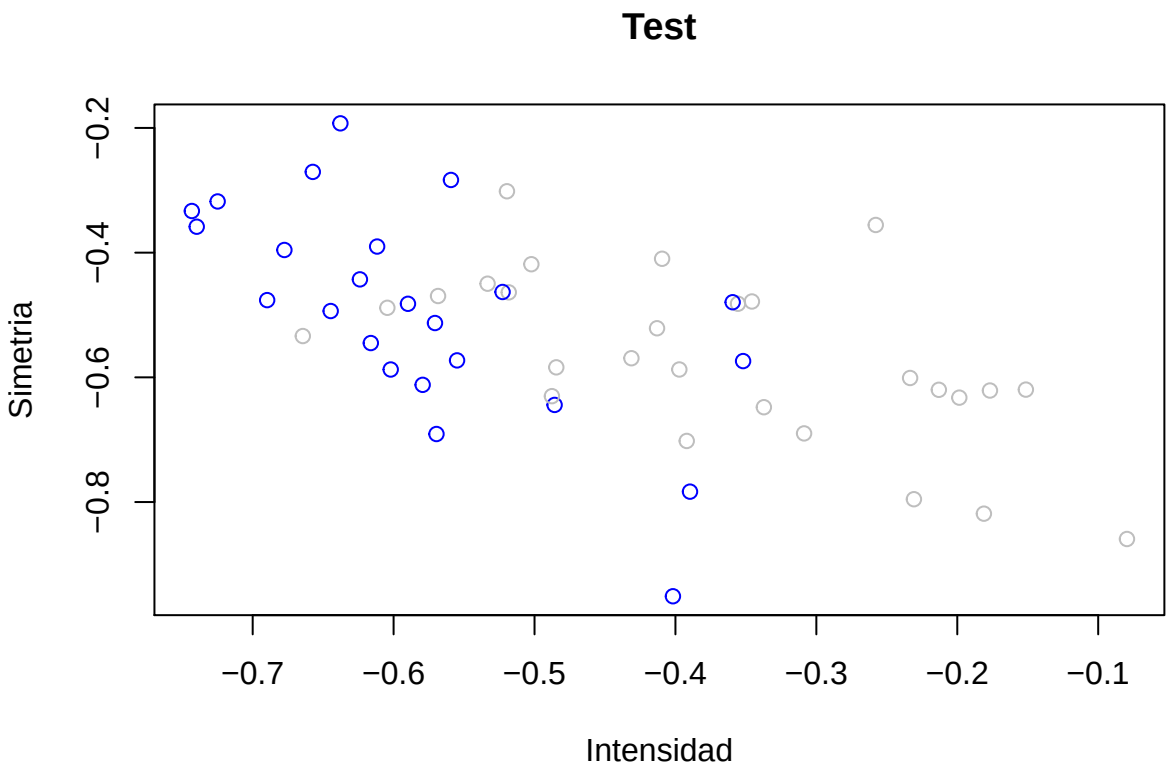
```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : número de items leídos no es múltiplo del número de columnas
```

```
digitos.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]
etiquetasTrain = digitos.train[,1]
ndigitosTrain = nrow(digitos.train)

# se retira la clase y se monta una matriz 3D: 599*16*16
datosTrain = array(unlist(subset(digitos.train,select=-V1)),c(ndigitosTrain,16,16))
rm(digit.train)
rm(digitos.train)

simetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -mean(A)
}

# Aplicamos la función para calcular la simetria y la intensidad a cada matriz
# de la lista de matrices
datosTrain.media = apply(datosTrain, 1, mean)
datosTrain.simetria = apply(datosTrain, 1, simetria)

datosTrain = matrix(c(datosTrain.media, datosTrain.simetria),
                    nrow = length(datosTrain.simetria))

# Leemos los datos de test
digit.test = read.table("./datos/zip.test", quote="\"", comment.char="",
                         stringsAsFactors=FALSE)

digitos.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]
etiquetasTest = digitos.test[,1]
ndigitosTest = nrow(digitos.test)

# se retira la clase y se monta una matriz 3D: 599*16*16
```

```
datosTest = array(unlist(subset(digitos.test,select=-V1)),c(ndigitosTest,16,16))
rm(digit.test)
rm(digitos.test)

# Aplicamos la función para calcular la simetria y la intensidad a cada matriz
# de la lista de matrices
datosTest.media = apply(datosTest, 1, mean)
datosTest.simetria = apply(datosTest, 1, simetria)

datosTest = matrix(c(datosTest.media, datosTest.simetria),
                   nrow = length(datosTest.simetria))

plot(datosTrain, xlab="Intensidad", ylab="Simetria", col=etiquetasTrain, main="Train")
```




```
plot(datosTest, xlab="Intensidad", ylab="Simetria", col=etiquetasTest, main="Test")
```



a) Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

Podemos usar cualquier algoritmo de los implementados en la práctica 1 para resolver dicho problema. La idea sería obtener un vector de pesos pasándole a los algoritmos el conjunto de datos de entrenamiento y las etiquetas, *datosTrain* y *etiquetasTrain* respectivamente en este caso. Nuestros datos de entrada, X, son los valores leidos desde los ficheros. El conjunto Y son las etiquetas leidas y nuestro objetivo es encontrar un vector de pesos que nos permita clasificar futuros datos.

b) Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

```
Regress_Lin = function(datos, label) {
  # Nos aseguramos de que vamos a trabar con matrices
  datos = as.matrix(datos)
  label = as.matrix(label)
  datos = cbind(1, datos)

  x = t(datos) %*% datos
  matrices = svd(x)
  auxiliar = matrices$v%*%diag(1/matrices$d)%*%t(matrices$v)
  pseudo = auxiliar%*%t(datos)
  w = pseudo%*%label
  unlist(w)
}

PLA_Pocket = function(datos, etiquetas, max_iter = 1000, vini = c(0,0,0)){
```

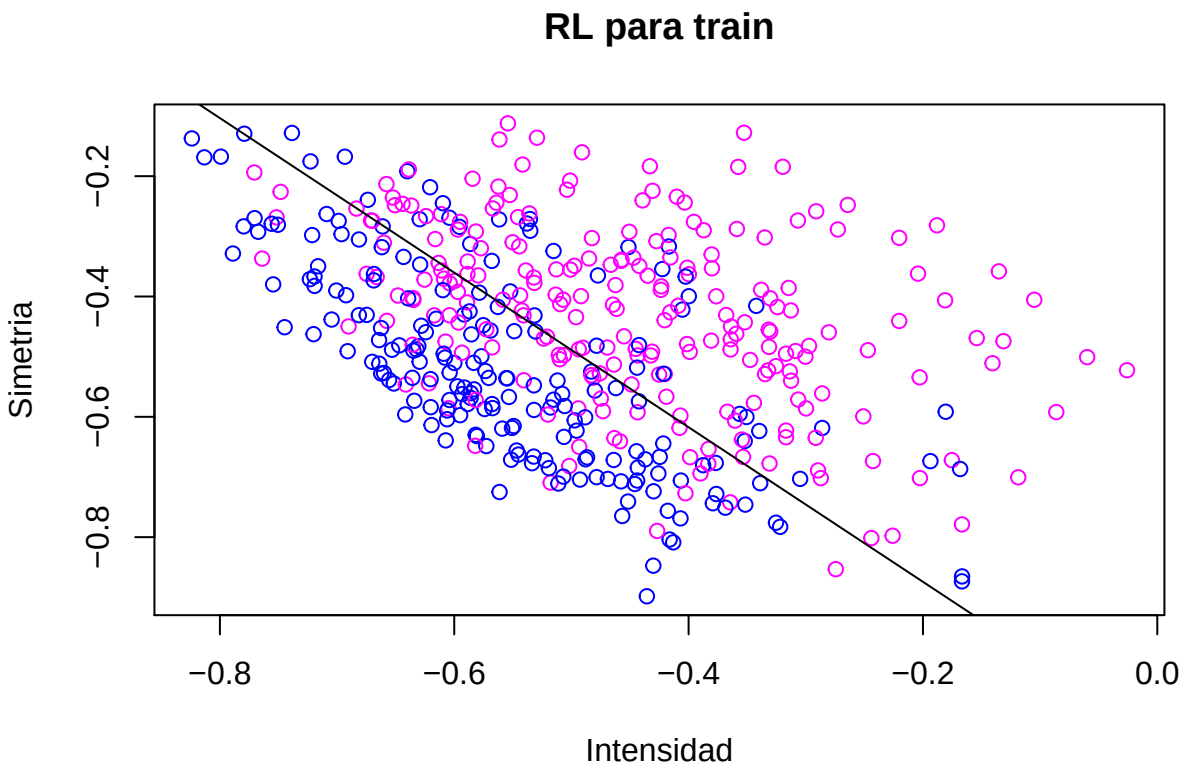
```
  seguirIterando = TRUE
  w = vini
  mejor_w = w
  iter = 0
  datos = cbind(1, datos)
  #Guardamos el error que nos da el vector de pesos inicial
  error = sum(sign(t(w)%*%t(datos)) != etiquetas) / length(etiquetas)
  #El algoritmos es igual que el PLA original
  while (seguirIterando && iter < max_iter){
    seguirIterando = FALSE
    indices = sample(1:nrow(datos))
    contador = 1
    while(contador <= length(indices) & !seguirIterando){
      i = indices[contador]
      if (sign(crossprod(w, datos[i,])) != etiquetas[i]){
        seguirIterando = TRUE
        w = w + etiquetas[i] * datos[i,]
      }
      contador = contador + 1
    }
    iter = iter + 1
    #Pero cada vez que se actualizan los pesos, calculamos el error que nos dan
    #estos pesos nuevos
    error_actual = sum(sign(t(w)%*%t(datos)) != etiquetas) / length(etiquetas)
    #Y si es menor que lo mejor que teníamos, lo guardamos
    #En este punto es donde se introduce memoria al PLA
    if(error_actual < error){
      mejor_w = w
      error = error_actual
    }
  }
  mejor_w
}
```

Al igual que pasaba en la práctica anterior, debemos cambiar las etiquetas por 1 y -1, ya que usamos el sígno para clasificar.

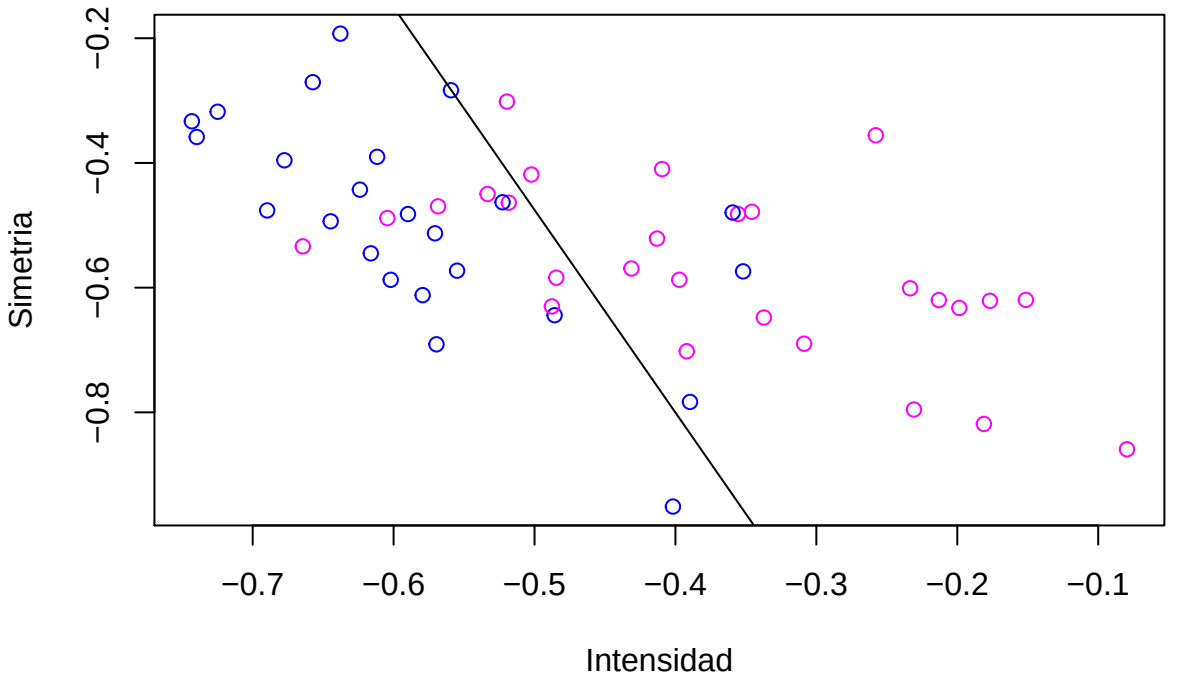
```
etiquetasTrain_2 = replace(etiquetasTrain, etiquetasTrain==4, -1)
etiquetasTrain_2 = replace(etiquetasTrain_2, etiquetasTrain_2==8, 1)

etiquetasTest_2 = replace(etiquetasTest, etiquetasTest==4, -1)
etiquetasTest_2 = replace(etiquetasTest_2, etiquetasTest_2==8, 1)
```

1) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.

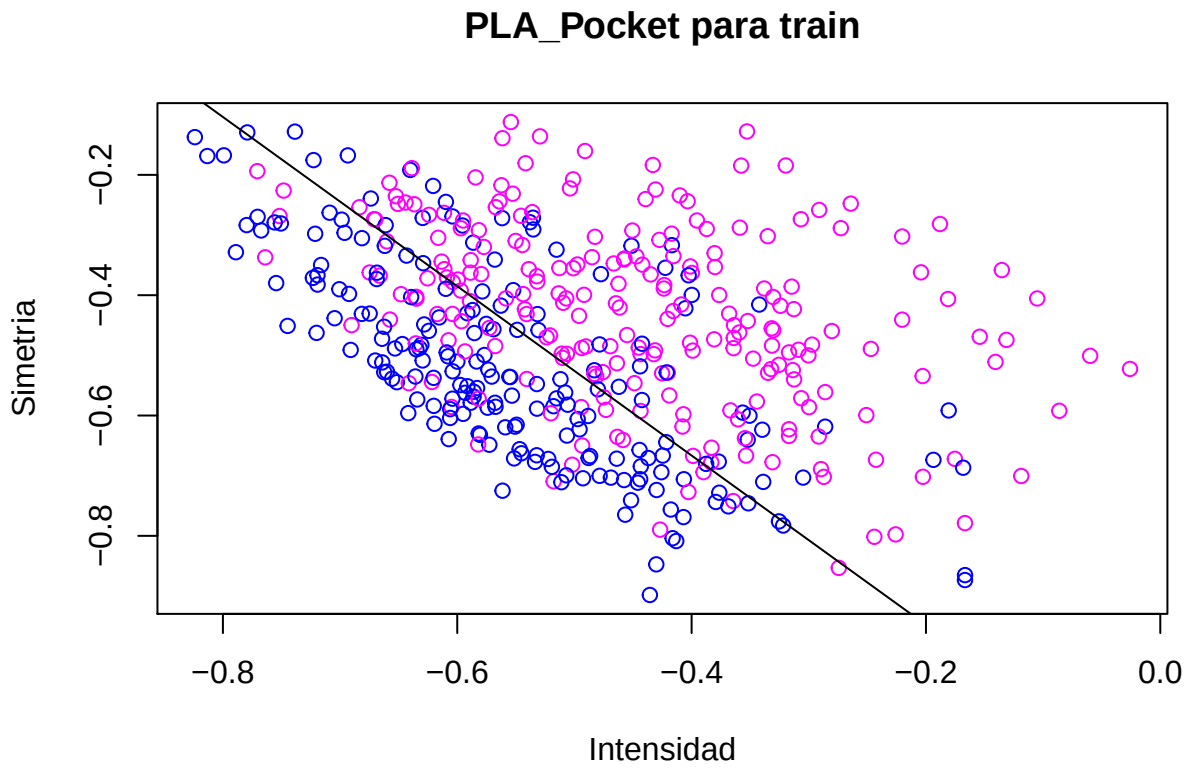
```
RLTrain = Regress_Lin(datosTrain, etiquetasTrain_2)
plot(datosTrain, xlab="Intensidad", ylab="Simetria",
     col=etiquetasTrain_2+5, main="RL para train")
abline(-RLTrain[1]/RLTrain[3], -RLTrain[2]/RLTrain[3])
```



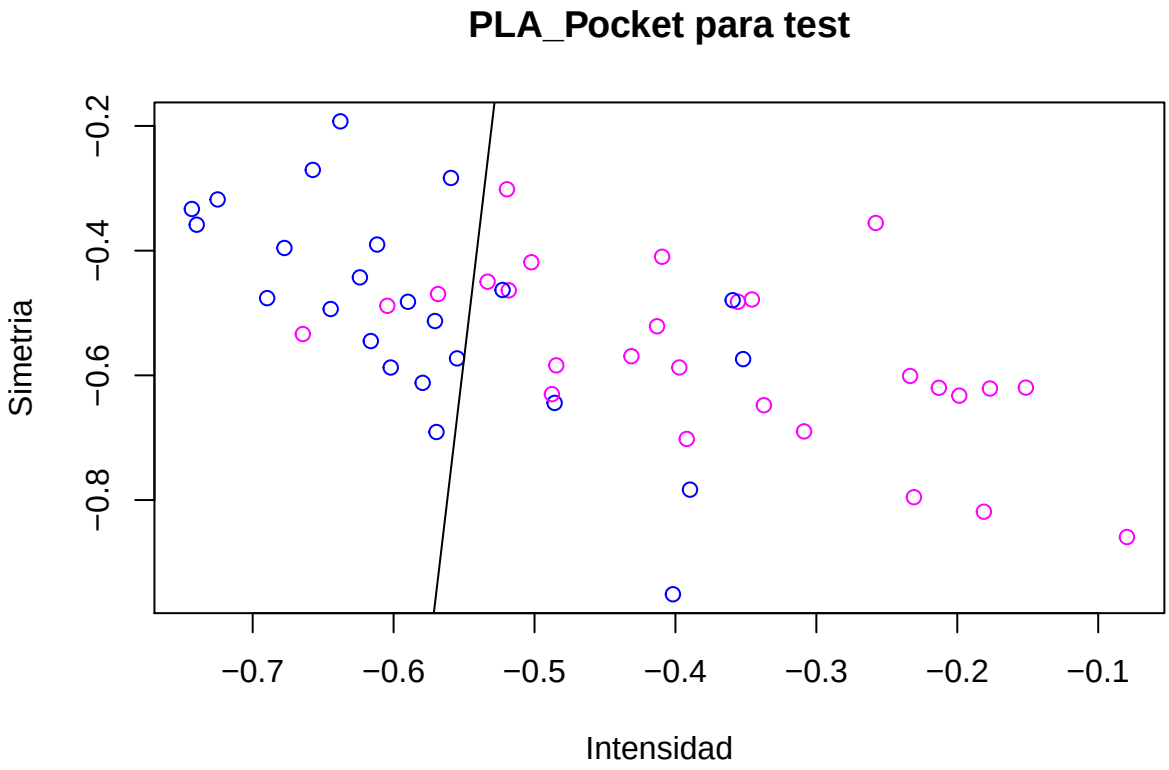

```
RLTest = Regress_Lin(datosTest, etiquetasTest_2)
plot(datosTest, xlab="Intensidad", ylab="Simetria",
     col=etiquetasTest_2+5, main="RL para test")
abline(-RLTest[1]/RLTest[3], -RLTest[2]/RLTest[3])
```



```
PLA_PocketTrain = PLA_Pocket(datos = datosTrain, etiquetas = etiquetasTrain_2)
plot(datosTrain, xlab="Intensidad", ylab="Simetria",
     col=etiquetasTrain_2+5, main="PLA_Pocket para train")
abline(-PLA_PocketTrain[1]/PLA_PocketTrain[3], -PLA_PocketTrain[2]/PLA_PocketTrain[3])
```




```
PLA_PocketTest = PLA_Pocket(datos = datosTest, etiquetas = etiquetasTest_2)
plot(datosTest, xlab="Intensidad", ylab="Simetria",
     col=etiquetasTest_2+5, main="PLA_Pocket para test")
abline(-PLA_PocketTest[1]/PLA_PocketTest[3], -PLA_PocketTest[2]/PLA_PocketTest[3])
```



2) Calcular E_{in} y E_{test} (error sobre los datos de test).

El error en regresión lineal se calcula usando mínimos cuadrados, pero para poder comparar bien los resultados con el PLA-Pocket los he calculado de la misma manera que en el PLA-Pocket, es decir, el porcentaje de datos mal clasificados.

```
RLEinTrain = sum(sign(t(RLTrain)%*%t(cbind(1, datosTrain)))
                 != etiquetasTrain_2) / nrow(datosTrain)

RLEinTest = sum(sign(t(RLTest)%*%t(cbind(1, datosTest)))
                != etiquetasTest_2) / nrow(datosTest)

PLA_PocketEinTrain = sum(sign(t(RLTrain)%*%t(cbind(1, datosTrain)))
                         != etiquetasTrain_2) / nrow(datosTrain)

PLA_PocketEinTest = sum(sign(t(RLTest)%*%t(cbind(1, datosTest)))
                        != etiquetasTest_2) / nrow(datosTest)
RLEinTrain
```

```
## [1] 0.2476852
```

```
RLEinTest
```

```
## [1] 0.1960784
```

```
PLA_PocketEinTrain
```

```
## [1] 0.2476852
```

```
PLA_PocketEinTest
```

```
## [1] 0.1960784
```

Los resultados son iguales, pero los vectores de pesos obtenidos son distintos.

```
as.vector(RLTrain)
```

```
## [1] 3.173593 3.604176 2.805622
```

```
as.vector(RLTest)
```

```
## [1] 2.813389 4.353480 1.339436
```

```
PLA_PocketTrain
```

```
## [1] 3.000000 3.433434 2.440492
```

```
PLA_PocketTest
```

```
## [1]  2.0000000  3.8472266 -0.2022969
```

Pero vayamos un paso más allá, vamos a pasarle al PLA-Pocket como pesos iniciales el vector de pesos obtenidos por la regresión.

```
PLA_PocketTrain = PLA_Pocket(datos = datosTrain, etiquetas = etiquetasTrain_2,
                             vini = RLTrain)

PLA_PocketTest = PLA_Pocket(datos = datosTest, etiquetas = etiquetasTest_2, vini = RLTest)

PLA_PocketEinTrain2 = sum(sign(t(PLA_PocketTrain)%*%t(cbind(1, datosTrain)))
                          != etiquetasTrain_2) / nrow(datosTrain)

PLA_PocketEinTest2 = sum(sign(t(PLA_PocketTest)%*%t(cbind(1, datosTest)))
                         != etiquetasTest_2) / nrow(datosTest)
RLEinTrain
```

```
## [1] 0.2476852
```

```
RLEinTest
```

```
## [1] 0.1960784
```

```
PLA_PocketEinTrain
```

```
## [1] 0.2476852
```

```
PLA_PocketEinTest
```

```
## [1] 0.1960784
```

```
PLA_PocketEinTrain2
```

```
## [1] 0.224537
```

```
PLA_PocketEinTest2
```

```
## [1] 0.1568627
```

Como podemos ver, los resultados que obtenemos pasándole un vector de pesos ya calculado al PLA-Pocket son mejores que pasándole un vector de pesos inicializado a cero.

3) Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia delta = 0,05. ¿Que cota es mejor?

Para calcular la cota podemos aplicar la fórmula que viene en la transparencia 15 de la sesion 4 de teoría, que es la misma que aparece en la página 58 del libro *Learning from data*. Debemos tener en cuenta que el Perceptron (PLA-Pocket) tiene una dimensión de Vapnik Chervonenkis igual a 3. La fórmula es la siguiente:

```
EoutTrain = RLEinTrain + sqrt((8/ndigitosTrain)*log((4*((2*ndigitosTrain)^3 + 1))/0.05))
EoutTest = RLEinTrain + sqrt((8/ndigitosTest)*log((4*((2*ndigitosTest)^3 + 1))/0.05))
EoutTrain
```

```
## [1] 0.9235488
```

```
EoutTest
```

```
## [1] 1.939972
```

Podemos ver que se trata de cotas áltísimas, llegando incluso una de ellas a ser mayor qe 1, lo cual no aporta información más allá de que, probablemente, falles siempre al clasificar.

Ejercicio 4: Regularización en la selección de modelos.

Para d = 3 (dimensión del vector de características) generar un conjunto de N datos aleatorios $\{x_n, y_n\}$ de la siguiente forma:

- Las coordenadas de los puntos x_n se generarán como valores aleatorios extraidos de una Gaussiana de media 1 y desviación típica 1.
- Para definir el vector de pesos w_f de la función f generamos d + 1 valores de una Gaussiana de media 0 y desviación típica 1. Al último valor le sumaremos 1.
- Usando los valores anteriores generamos la etiqueta asociada a cada punto x_n a partir del valor $y_n = w_f^T x_n + sigma^* n$, donde n es un ruido que sigue también una Gaussiana de media 0 y desviación típica 1 y $sigma^2$ es la varianza del ruido; fijar sigma = 0,5

Ahora vamos a estimar el valor de w_f usando w_{reg}, es decir los pesos de un modelo de regresión lineal con regularización "weight decay". Fijar el parámetro de regularización a 0,05/N.

a) Para N preteneciente a {d + 10, d + 20, ... , d + 110} calcular los errores de validación cruzada e_1, ... , e_N y E_{cv}. Repetir el experimento 1000 veces. Anotamos el promedio y la varianza de e_1, e_2 y E_{cv} en los experimentos.

```
RL_WD = function(datos, label, lamda) {
  # Nos aseguramos de que vamos a trabajar con matrices
  datos = as.matrix(datos)
  label = as.matrix(label)

  # El segundo termino de la sumatoria es lo que nos proporcionar el
  # wight decay
  x = t(datos) %*% datos + diag(lamda, ncol(datos), ncol(datos))
  matrices = svd(x)
  auxiliar = matrices$v%*%diag(1/matrices$d)%*%t(matrices$v)
  pseudo = auxiliar%*%t(datos)
  w = pseudo%*%label
  w
  unlist(w)
}

experimento = function(){
  datos = matrix(replicate(130, simula_gaus(N = 3, dim = 1, sigma = 1, media = 1)),
                 nrow = 130, ncol = 3, byrow = T)
```

```
  datos = cbind(1,datos)
  w_f = simula_gaus(N = 4, dim = 1, sigma = 1, media = 0)
  #Se lo sumo al primero en vez de al último ya que mi termino indepentdiente en los
  #datos es el primero, no el último
  w_f[1] = w_f[1] + 1
  ruido = simula_gaus(N = nrow(datos), dim = 1, sigma = 0.5, media = 0)

  etiquetas = as.vector(datos %*% as.vector(w_f)) + 0.5*ruido


  ini = 1
  fin = 13
  errores = list()
  for(i in 1:10){
    test = datos[ini:fin,]
    etiquetas.test = etiquetas[ini:fin]
    train = datos[-c(ini:fin),]
    etiquetas.train = etiquetas[-c(ini:fin)]
    w = RL_WD(train, etiquetas.train, lamda = 0.05/nrow(train))
    etiquetas.calculadas = as.vector(t(w)%*%t(test))
    #Calculamos el error usando minimos cuadrados
    error = sum((etiquetas.calculadas - etiquetas.test)^2) / (length(etiquetas.test))
    errores = c(errores, error)

    #Ojo, las particiones son de trece elementos
    ini = 1+i*13
    fin = 13+i*13
  }
  unlist(errores)
}

mil_experimentos = function(){
  salida = matrix(ncol = 10)
  for(i in 1:1000){
    errores = experimento()
    salida = rbind(salida, errores)
  }
  #Al crear la matriz se mete una fila de NA, así que la quitamos
  salida = salida[-1,]
  salida
}

resultados = mil_experimentos()

#Calculamos los errores e1, ..., eN para todos los experimentos (media de las columnas)
e_partes_medios = apply(resultados, c(2), mean)
#Calculamos la varianza en cada particion
e_partes_var = apply(resultados, c(2), var)
#Calculamos el error medio de validacion cruzada
e_vc_med = mean(e_partes_medios)
e_vc_var = mean(e_partes_var)

e_partes_medios[1] #e1
```

```
## [1] 0.130672
```

```
e_partes_medios[2] #e2
```

```
## [1] 0.1292624
```

```
e_vc_med  #Evc
```

```
## [1] 0.1306669
```

```
e_partes_var[1] #varianza 1
```

```
## [1] 0.002557938
```

```
e_partes_var[2] #varianza 2
```

```
## [1] 0.002565109
```

```
e_vc_var  #varianza media de los experimentos
```

```
## [1] 0.002609534
```

b) ¿Cuál debería de ser la relación entre el valor promedio de e_1 y el de E_{cv}? ¿y entre el valor promedio de e_1 y el de e_2? Argumentar la respuesta en base a los resultados de los experimentos.

Ambas relaciones son de similitud, es decir, el valor de *e_1*, de *e_2* y de E_{cv} son similares. Esto se debe a que estamos tratando con valores medios obtenidos de 1000 experimentos distintos, lo cual permite que los valores converjan y sean similares.

c) ¿Qué es lo que más contribuye a la varianza de los valores de e_1?

En este caso, como los conjuntos tienen el mismo tamaño (13 elementos) no podemos concluir nada, ya que lo que podría influir en la varianza de los datos es el tamaño del conjunto.

d) Diga que conclusiones sobre regularización y selección de modelos ha sido capaz de extraer de esta experimentación.

```
RL = function(datos, label, lamda) {
  # Nos aseguramos de que vamos a trabajar con matrices
  datos = as.matrix(datos)
  label = as.matrix(label)

  # El segundo termino de la sumatoria es lo que nos proporcionar el
  # wight decay
  x = t(datos) %*% datos
  matrices = svd(x)
  auxiliar = matrices$v%*%diag(1/matrices$d)%*%t(matrices$v)
  pseudo = auxiliar%*%t(datos)
  w = pseudo%*%label
  w
  unlist(w)
}

experimento = function(){
```

```
  datos = matrix(replicate(130, simula_gaus(N = 3, dim = 1, sigma = 1, media = 1)),
                 nrow = 130, ncol = 3, byrow = T)
  datos = cbind(1,datos)
  w_f = simula_gaus(N = 4, dim = 1, sigma = 1, media = 0)
  #Se lo sumo al primero en vez de al último ya que mi termino indepentdiente en los
  #datos es el primero, no el último
  w_f[1] = w_f[1] + 1
  ruido = simula_gaus(N = nrow(datos), dim = 1, sigma = 0.5, media = 0)

  etiquetas = as.vector(datos %*% as.vector(w_f)) + 0.5*ruido


  ini = 1
  fin = 13
  errores = list()
  for(i in 1:10){
    test = datos[ini:fin,]
    etiquetas.test = etiquetas[ini:fin]
    train = datos[-c(ini:fin),]
    etiquetas.train = etiquetas[-c(ini:fin)]
    w = RL(train, etiquetas.train, lamda = 0.05/nrow(train))
    etiquetas.calculadas = as.vector(t(w)%*%t(test))
    #Calculamos el error usando minimos cuadrados
    error = sum((etiquetas.calculadas - etiquetas.test)^2) / (length(etiquetas.test))
    errores = c(errores, error)

    #Ojo, las particiones son de trece elementos
    ini = 1+i*13
    fin = 13+i*13
  }
  unlist(errores)
}

mil_experimentos = function(){
  salida = matrix(ncol = 10)
  for(i in 1:1000){
    errores = experimento()
    salida = rbind(salida, errores)
  }
  #Al crear la matriz se mete una fila de NA, así que la quitamos
  salida = salida[-1,]
  salida
}

resultados = mil_experimentos()

#Calculamos los errores e1, ..., eN para todos los experimentos (media de las columnas)
e_partes_medios = apply(resultados, c(2), mean)
#Calculamos la varianza en cada particion
e_partes_var = apply(resultados, c(2), var)
#Calculamos el error medio de validacion cruzada
e_vc_med = mean(e_partes_medios)
e_vc_var = mean(e_partes_var)
```

```
e_partes_medios[1] #e1
```

```
## [1] 0.128434
```

```
e_partes_medios[2] #e2
```

```
## [1] 0.1284173
```

```
e_vc_med  #Evc
```

```
## [1] 0.1294941
```

```
e_partes_var[1] #varianza 1
```

```
## [1] 0.002512197
```

```
e_partes_var[2] #varianza 2
```

```
## [1] 0.002663589
```

```
e_vc_var  #varianza media de los experimentos
```

```
## [1] 0.002610656
```

Podemos ver que que los errores sin aplicar Wight Decay son menores que aplicandolo. Esto puede ser a que el parámetro de lambda que estamos usando no es el más apropiado, ya que se nos da fijado. Quizás usando otro valor de lambda obtendríamos mejores resultados.

Bonus.

Ejercicio 1: Coordenada descendente.

En este ejercicio comparamos la eficiencia de la técnica de optimización de "coordenada descendente" usando la misma función del ejercicio 1.1a. En cada iteración, minimizamos a lo largo de cada una de las coordenadas individualmente. En el Paso-1 nos movemos a lo largo de la coordenada u para reducir el error (suponer que se verifica una aproximación de primer orden como en gradiente descendente), y el Paso-2 es para reevaluar y movernos a lo largo de la coordenada v para reducir el error (hacer la misma hipótesis que en el paso-1). Usar una tasa de aprendizaje eta $= 0{,}1$.

```
f1 = function(u, v) ((u^2*exp(v) - 2*v^2*exp(-u))^2)
du1 = function(u,v) (2*((2*u*exp(v) + 2*v^2*exp(-u))*(u^2*exp(v) - 2*v^2*exp(-u))))
dv1 = function(u,v) (2*((u^2*exp(v) - 2*(2*v)*exp(-u))*(u^2*exp(v) - 2*v^2*exp(-u))))

CD = function(u = 1, v = 1, eta = 0.1, umbral = 10^(-4), max_iter = 1000, f, du, dv){
  noLlanura = T
  u_0 = 0.0
  v_0 = 0.0
  u_1 = u
  v_1 = v
  iter = 0
  # Como condición de parada tenemos el número de iteraciones
  # y que no estemos en una zona plana, es decir, que haya una diferencia
  # entre el valor de la función en el punto nuevo con respecto al anterior.
  while(iter < max_iter & noLlanura == T) {
    u_0 = u_1
    v_0 = v_1
```

```
    #Paso 1
    u_1 = u_0 - eta * du(u_0, v_0)
    #Paso 2
    v_1 = v_0 - eta * dv(u_1, v_0)
    iter = iter + 1
    if(abs(f(u_1, v_1) - f(u_0, v_0)) < umbral)
      noLlanura = F
  }
  c(iter, f(u_1, v_1), u_1, v_1, abs(f(u_1, v_1) - f(u_0, v_0)))
}

CD(f = f1, du = du1, dv = dv1, max_iter = 15)
```

```
## [1]  3.000000e+00  0.000000e+00  1.190321e+04 -8.935536e+10  0.000000e+00
```

```
GD(f = f1, du = du1, dv = dv1, max_iter = 15)
```

```
## [1]  4.000000e+00  3.855518e-03  9.864573e+00 -2.443828e+01  5.999810e-06
```

a) ¿Qué valor de la función E(u, v) se obtiene después de 15 iteraciones completas (i.e. 30 pasos) ?

El valor de la funcion epara 15 iteraciones es 0.

b) Establezca una comparación entre esta técnica y la técnica de gradiente descendente.

Podemos ver que para los parametros elegidos obtenemos mejores resultados aplicando coordenada descendente que aplicando gradidiente descendente.

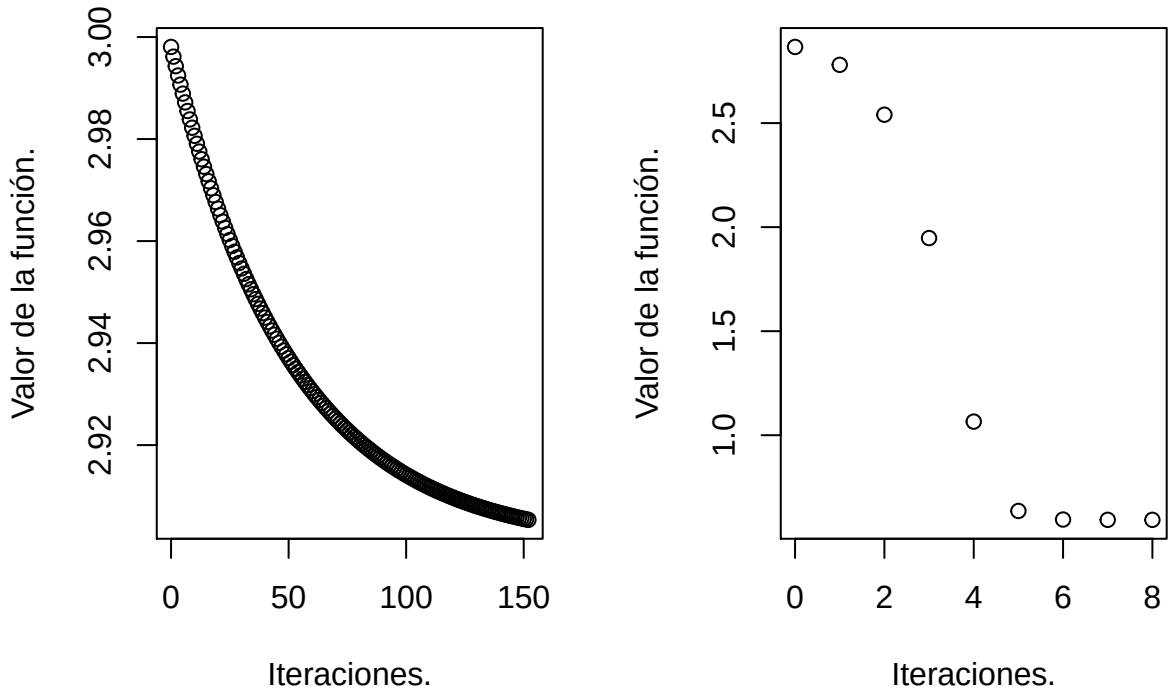
Ejercicio 2: Método de Newton.

Implementar el algoritmo de minimización de Newton y aplicarlo a la función f(x, y) dada en el ejercicio 1b. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

- *Generar un gráfico de como desciende el valor de la función con las iteraciones.*

```
fun = function(x, y) ((x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y))
dx = function(x, y) (2*(2*pi*cos(2*pi*x)*sin(2*pi*y)+x-2))
dxy = function(x, y) (8*pi^2*cos(2*pi*x)*cos(2*pi*y))
d2x = function(x, y) (2-8*pi^2*sin(2*pi*x)*sin(2*pi*y))
dy = function(x, y) (4*(pi*sin(2*pi*x)*cos(2*pi*y)+y-2))
d2y = function(x, y) (4-8*pi^2*sin(2*pi*x)*sin(2*pi*y))
dyx = function(x, y) (8*pi^2*cos(2*pi*x)*cos(2*pi*y))

MatrizNewton = function(x, y, d2x, dxy, dyx, d2y){
  matrix(c(d2x(x,y), dxy(x,y), dyx(x,y), d2y(x,y)), ncol = 2, nrow = 2, byrow = T)
}

Metodo_Newton_Graf = function(x0 = 1, y0 = 1, f, dx, dy, d2x, d2y, dxy, dyx, eta = 0.1,
                              umbral = 10^(-4), max_iter = 200) {
  w0 = c(0,0)
  w1 = c(x0,y0)

  iter = 0
  seguir = T

  valores=list()
  iteraciones=list()
```

```
  while(iter < max_iter & seguir){
    w0 = w1
    m = MatrizNewton(w0[1], w0[2], d2x, dxy, dyx, d2y)
    gradiente = c(dx(w0[1], w0[2]), dy(w0[1], w0[2]))
    w1 = w0 - eta*(solve(m)%*%gradiente)

    valores = c(valores, f(w1[1], w1[2]))
    iteraciones = c(iteraciones, iter)

    iter = iter + 1
    if(abs(f(w1[1], w1[2]) - f(w0[1], w0[2])) < umbral){
      seguir = F
    }
  }

  valores=unlist(valores)
  iteraciones=unlist(iteraciones)
  valores = matrix(valores, nrow=length(valores), ncol=1)
  iteraciones = matrix(iteraciones, nrow=length(iteraciones), ncol=1)
  salida = cbind(iteraciones, valores)
  plot(salida, xlab="Iteraciones.", ylab="Valor de la función.")
}


par(mfrow=c(1,2))
Metodo_Newton_Graf(x0 = 1, y0 = 1, f = fun, dx = dx, dy = dy, d2x = d2x, d2y = d2y, dxy =
                     dxy, dyx = dyx, eta = 0.01, umbral = 10^(-4), max_iter = 200)

GD_graf(eta = 0.01, f = f2, du = dx, dv = dy, max_iter = 200)
```



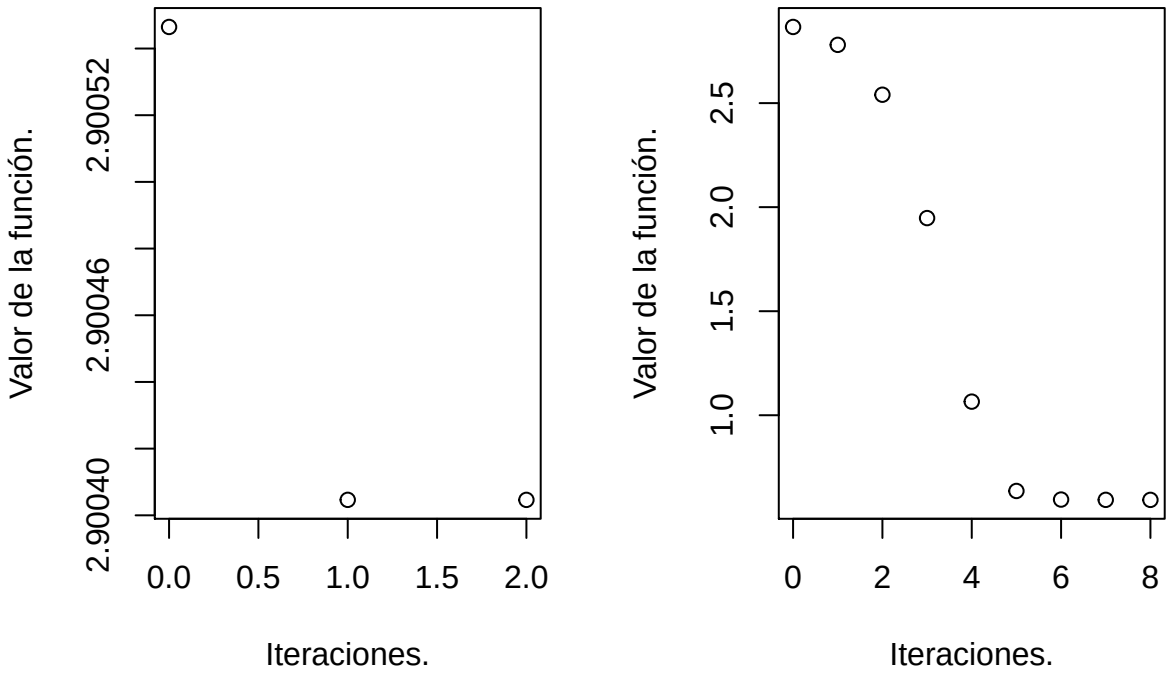

```
## [1] 9.000000e+00 5.932713e-01 7.821293e-01 1.287100e+00 5.089406e-05
```

```
par(mfrow=c(1,1))
```

- *Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.*

En este caso pordemos ver que Gradiente Descendente encuentra un mínimo menor y en menos iteraciones que el Método de Newton. Debemos tener en cuenta una cosa, en las transparencias de teoría no aparece el valor de *eta*. Si no añadimos la tasa de aprendizaje (que es lo mismo que el valor sea 1) el resultado es el siguiente:

```
par(mfrow=c(1,2))
Metodo_Newton_Graf(x0 = 1, y0 = 1, f = fun, dx = dx, dy = dy, d2x = d2x, d2y = d2y, dxy =
                     dxy, dyx = dyx, eta = 1, umbral = 10^(-4), max_iter = 200)

GD_graf(eta = 0.01, f = f2, du = dx, dv = dy, max_iter = 200)
```




```
## [1] 9.000000e+00 5.932713e-01 7.821293e-01 1.287100e+00 5.089406e-05
```

```
par(mfrow=c(1,1))
```

Obtenemos un valor similar de la función pero en menos iteraciones.

Ejercicio 3: Repetir RL

Repetir el experimento de RL (punto.2) 100 veces con diferentes funciones frontera y calcule el promedio.

a) ¿Cuál es el valor de E_{out} para $N = 100$?

b) ¿Cuántas épocas tarda en promedio RL en converger para $N = 100$, usando todas las condiciones anteriormente especificadas?

```
#Version modificada para devolver las iteraciones tambien
RLSGD_aux = function(datos, etiquetas, eta = 0.01, umbral = 0.01, max_iter = 550){
  indices = c(1:nrow(datos))
  datos = cbind(1,datos)
```

```
  w_1 = c(0, 0, 0)
  seguirEjecucion = T
  iter = 0
  while(iter < max_iter & seguirEjecucion){
    w_0 = w_1
    indices = sample(indices)
    for(i in indices){
      w_1 = w_1 - eta*(-etiquetas[i]*datos[i,])/as.vector((1+exp(etiquetas[i]*
                                                              (w_1%*%datos[i,]))))
    }
    if(norma(w_0 - w_1) < umbral)
      seguirEjecucion = F

    iter = iter + 1
  }
  c(w_1, iter)
}

datos = simula_unif(N = 100, dims = 2, rango = c(0,2))
resultados_Ej3 = list()
iteraciones_Ej3 = list()
for(i in 1:100){
  recta = simula_recta(c(0,2))
  etiquetasEj2 = sign(valor_y(datosEj2[,1], rectaEj2[1], rectaEj2[2]) - datosEj2[,2])

  sol = RLSGD_aux(datos = datosEj2, etiquetas = etiquetasEj2)
  iteraciones = sol[4]
  sol = sol[-4]

  datos_test = simula_unif(N = 100, dims = 2, rango = c(0,2))
  recta_test = simula_recta(intervalo = c(0,2))
  etiquetas_test = sign(valor_y(datos_test[,1], recta_test[1], recta_test[2]) -
                          datos_test[,2])

  error = sum(sign(t(sol)%*%t(cbind(1, datos_test)))
      != etiquetas_test) / nrow(datos_test)
  iteraciones_Ej3 = c(iteraciones_Ej3, iteraciones)
  resultados_Ej3 = c(resultados_Ej3, error)
}
mean(unlist(resultados_Ej3))
```

```
## [1] 0.3836
```

```
mean(unlist(iteraciones_Ej3))
```

```
## [1] 431.47
```

El valor medio de error es 0.3912 y tarda unas 431.27 iteraciones de media en converger.

Ejercicio 4:

Considere el ejercicio 3 de la sección de Modelos Lineales de los Ejercicios de Apoyo. Repetir los puntos del mismo pero usando una transformación polinómica de tercer orden(ϕ_3(x) en las transparencias de teoría). Si tuviera que usar los resultados para dárselos a un potencial cliente ¿usaría la transformación polinómica?

Explicar la decisión.

Yo lo he hecho sobre el ejercicio 3 de esta práctica, no sobre los ejercicios de apoyo porque no me había dado cuenta de ese detalle.

```
# Leemos los datos
digit.train = read.table("./datos/zip.train", quote="\"", comment.char="",
                         stringsAsFactors=FALSE)
```

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : número de items leídos no es múltiplo del número de columnas
```

```
digitos.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]
etiquetasTrain = digitos.train[,1]
ndigitosTrain = nrow(digitos.train)

# se retira la clase y se monta una matriz 3D: 599*16*16
datosTrain = array(unlist(subset(digitos.train,select=-V1)),c(ndigitosTrain,16,16))
rm(digit.train)
rm(digitos.train)

simetria <- function(A){
  A = abs(A-A[,ncol(A):1])
  -mean(A)
}

# Aplicamos la función para calcular la simetria y la intensidad a cada matriz
# de la lista de matrices
datosTrain.media = apply(datosTrain, 1, mean)
datosTrain.simetria = apply(datosTrain, 1, simetria)

datosTrain = matrix(c(datosTrain.media, datosTrain.simetria),
                    nrow = length(datosTrain.simetria))

# Leemos los datos de test
digit.test = read.table("./datos/zip.test", quote="\"", comment.char="",
                         stringsAsFactors=FALSE)

digitos.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]
etiquetasTest = digitos.test[,1]
ndigitosTest = nrow(digitos.test)

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

# Aplicamos la función para calcular la simetria y la intensidad a cada matriz
# de la lista de matrices
datosTest.media = apply(datosTest, 1, mean)
datosTest.simetria = apply(datosTest, 1, simetria)

datosTest = matrix(c(datosTest.media, datosTest.simetria),
                    nrow = length(datosTest.simetria))

etiquetasTrain_2 = replace(etiquetasTrain, etiquetasTrain==4, -1)
```

```
etiquetasTrain_2 = replace(etiquetasTrain_2, etiquetasTrain_2==8, 1)

etiquetasTest_2 = replace(etiquetasTest, etiquetasTest==4, -1)
etiquetasTest_2 = replace(etiquetasTest_2, etiquetasTest_2==8, 1)
```

Una vez tenemos los datos listos, sólo debemos realizar la modificación de los datos y repetir las llamadas a la Regresión Lineal y al PLA-Pocket.

```
datosTrain = matrix(c(datosTrain[,1], datosTrain[,2], datosTrain[,1]*datosTrain[,2],
                    datosTrain[,1]^2, datosTrain[,2]^2, datosTrain[,1]^3,
                    datosTrain[,1]^2*datosTrain[,2],
                    datosTrain[,2]^2*datosTrain[,1], datosTrain[,2]^3),
                    nrow = length(datosTrain[,1]))

datosTest = matrix(c(datosTest[,1], datosTest[,2], datosTest[,1]*datosTest[,2],
                    datosTest[,1]^2, datosTest[,2]^2, datosTest[,1]^3,
                    datosTest[,1]^2*datosTest[,2],
                    datosTest[,2]^2*datosTest[,1], datosTest[,2]^3),
                    nrow = length(datosTest[,1]))

RLTrain = Regress_Lin(datosTrain, etiquetasTrain_2)
RLTest = Regress_Lin(datosTest, etiquetasTest_2)
PLA_PocketTrain = PLA_Pocket(datos = datosTrain, etiquetas = etiquetasTrain_2,
                             vini = c(rep(0, dim(datosTrain)[2] + 1)))
PLA_PocketTest = PLA_Pocket(datos = datosTest, etiquetas = etiquetasTest_2,
                            vini = c(rep(0, dim(datosTest)[2] + 1)))

RLEinTrain = sum(sign(t(RLTrain)%*%t(cbind(1, datosTrain)))
                 != etiquetasTrain_2) / nrow(datosTrain)

RLEinTest = sum(sign(t(RLTest)%*%t(cbind(1, datosTest)))
                != etiquetasTest_2) / nrow(datosTest)

PLA_PocketEinTrain = sum(sign(t(RLTrain)%*%t(cbind(1, datosTrain)))
                         != etiquetasTrain_2) / nrow(datosTrain)

PLA_PocketEinTest = sum(sign(t(RLTest)%*%t(cbind(1, datosTest)))
                        != etiquetasTest_2) / nrow(datosTest)
RLEinTrain
```

```
## [1] 0.2268519
```

```
RLEinTest
```

```
## [1] 0.1372549
```

```
PLA_PocketEinTrain
```

```
## [1] 0.2268519
```

```
PLA_PocketEinTest
```

```
## [1] 0.1372549
```

Como podemos ver los errores han bajado en comparación con los del experimento anterior. En el caso de tener que converncer a un potencial cliente mi idea es clara, usaría la transformación polinómica, ya que de cara al cliente no supone nada en cuanto a datos de entrada extra pero si supone obtener unos mejores resultados.