Trabajo 2

Juan José Sierra González

30 de marzo de 2017

Ejercicio 1. Gradiente Descendente. Implementar el algoritmo de gradiente descendente.

a) Considerar la función no lineal E(u, v) = (u^{2e}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 mu = 0,1.

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

A continuación muestro la función E junto con las nuevas funciones "derivada de E respecto a u" y "derivada de E respecto a v" calculadas.

```
E = function(u,v){(u^2*exp(v)-2*v^2*exp(-u))^2}
duE = function(u,v){2*(u^2*exp(v)-2*v^2*exp(-u))*(2*exp(v)*u+2*v^2*exp(-u))}
dvE = function(u,v){2*(u^2*exp(v)-2*v^2*exp(-u))*(u^2*exp(v)-4*v*exp(-u))}
```

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)

En este ejercicio calcularemos cuántas iteraciones tarda el gradiente en encontrar una diferencia de "mínimos" menor al umbral establecido. Es decir, en qué momento la solución varía muy poco, dando a entender que podemos encontrarnos en una llanura, que es por lo que el gradiente descendente tiende a obtener mínimos locales.

```
# Función que calcula el gradiente descendente de una función dada,
# junto a sus derivadas previamente calculadas. El algoritmo actualiza
# los pesos utilizando el valor obtenido del gradiente y la tasa de
# aprendizaje que se quiera utilizar en cada caso. En el momento en el
# que la diferencia de valores en la función original para pesos(t) y
# pesos(t-1) sea menor que el umbral propuesto, el algoritmo se detendrá.
# También se detendrá si se alcanza el máximo de iteraciones establecido.
# Para facilitar la resolución de los próximos ejercicios, se devuelve
# una lista con los pesos que dan lugar al mínimo calculado y las
# iteraciones necesarias para concurrir hasta él.
gradienteDescendente = function(wini, mu, threshold, max_iter, E, duE, dvE){

  wold = wini
  iter = 0
  seguir_iterando = T

  while (iter < max_iter & seguir_iterando){
    g=c(duE(wold[1],wold[2]),dvE(wold[1],wold[2]))
    v = -g
    wnew = wold + mu*v
    if (abs(E(wold[1],wold[2]) - E(wnew[1],wnew[2])) < threshold){
```

```
      seguir_iterando = F
    }
    wold = wnew
    iter = iter+1
  }
  list(minimo=wnew, iter=iter)
}

resultado1a = gradienteDescendente(c(1,1), 0.1, 10^-4, 50, E, duE, dvE)
resultado1a$iter
```

```
## [1] 4
```

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

```
resultado1a$minimo
```

```
## [1]   9.864573 -24.438276
```

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

1) Usar gradiente descendente para minimizar esta función. Usar como punto inicial (x0 = 1, y0 = 1), tasa de aprendizaje mu = 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 mu = 0,1, comentar las diferencias.

Para realizar las gráficas he decidido adaptar la función anterior y crear una nueva que las pinte internamente, y es lo único en lo que dista de la función original del gradiente descendente. En las gráficas que se muestran a continuación, la de la izquierda corresponde a una tasa de aprendizaje 0.01, y la segunda a una 0.1.

```
fun1b = function(x,y){(x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)}
dxfun1b = function(x,y){2*(-2+x+2*pi*cos(2*pi*x)*sin(2*pi*y))}
dyfun1b = function(x,y){4*(-2+y+pi*cos(2*pi*y)*sin(2*pi*x))}

# Función equivalente al gradiente descendente pero con la particularidad
# de que dibuja en una gráfica los puntos "mínimos" que va encontrando el
# algoritmo en función del número de iteraciones que se han evaluado.
gradienteDescendenteGrafica = function(wini, mu, threshold, max_iter, E, duE, dvE){

  wold = wini
  iter = 0
  seguir_iterando = T
  iteraciones_grafica = list()
  minimos_grafica = list()

  while (iter < max_iter & seguir_iterando){
    g=c(duE(wold[1],wold[2]),dvE(wold[1],wold[2]))
    v = -g
    wnew = wold + mu*v
    iteraciones_grafica = c(iteraciones_grafica, iter)
    minimos_grafica = c(minimos_grafica, E(wnew[1],wnew[2]))
```

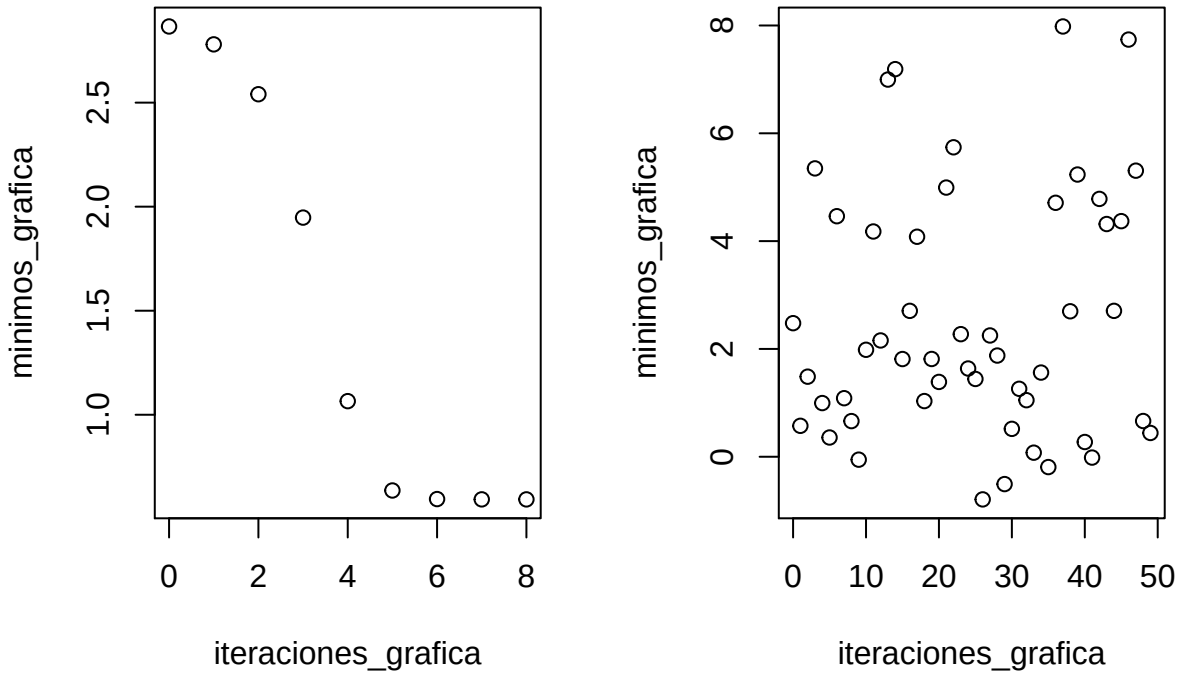
```
    if (abs(E(wold[1],wold[2]) - E(wnew[1],wnew[2])) < threshold){
      seguir_iterando = F
    }
    wold = wnew
    iter = iter+1
  }
  plot(iteraciones_grafica, minimos_grafica)
  list(minimo=wnew, iter=iter)
}

par(mfrow = c(1,2))
resultado1b = gradienteDescendenteGrafica(c(1,1), 0.01, 10^-4, 50, fun1b, dxfun1b, dyfun1b)
resultado1b$minimo
```

```
## [1] 0.7821293 1.2871002
```

```
resultado1b$iter
```

```
## [1] 9
```

```
resultado1b2 = gradienteDescendenteGrafica(c(1,1), 0.1, 10^-4, 50, fun1b, dxfun1b, dyfun1b)
```




```
resultado1b2$minimo
```

```
## [1] 1.961298 1.377935
```

```
resultado1b2$iter
```

```
## [1] 50
```

Como podemos observar, si utilizamos una tasa de aprendizaje pequeña es probable que converjamos antes, debido a que prácticamente vamos a imitar la forma de la función. Si esta tiene un mínimo local o una llanura, no pasaremos de ahí, como sucede en el primer experimento. En el segundo sin embargo vemos que el espectro de los puntos varía mucho más. Al aumentar la tasa de aprendizaje, hacemos que los "saltos" que damos sobre la función gracias al aprendizaje de pesos utilizando el gradiente nos permitan salir de llanuras y mínimos que no sean excesivamente grandes. Esto por una parte facilita la exploración, en el caso del

experimento encuentra variaciones sustanciales en cada una de las 50 iteraciones, pero por otra hace fácil saltarse zonas del espacio en las que podría encontrarse un mínimo a tener en cuenta. No obstante, si nos fijamos, el mínimo valor que obtiene el experimento con tasa de aprendizaje alta es menor que el que obtiene el experimento con tasa de aprendizaje baja, pero sin embargo no se detuvo en la exploración dado que el salto es sustancialmente grande.

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.

El desarrollo de este ejercicio simplemente requiere recabar los resultados para cada uno de los puntos de interés, y una vez conocidos los datos generar una tabla como la que aparece a continuación.

```
valores_punto1 = gradienteDescendente(c(2.1,2.1), 0.1, 10^-4, 50, fun1b, dxfun1b, dyfun1b)
valores_punto2 = gradienteDescendente(c(3,3), 0.1, 10^-4, 50, fun1b, dxfun1b, dyfun1b)
valores_punto3 = gradienteDescendente(c(1.5,1.5), 0.1, 10^-4, 50, fun1b, dxfun1b, dyfun1b)
valores_punto4 = gradienteDescendente(c(1,1), 0.1, 10^-4, 50, fun1b, dxfun1b, dyfun1b)

valores_punto1$minimo
```

```
## [1] 2.392957 1.067868
```

```
print(fun1b(valores_punto1$minimo[1], valores_punto1$minimo[2]))
```

```
## [1] 2.407525
```

```
valores_punto2$minimo
```

```
## [1] 2.298454 1.989681
```

```
print(fun1b(valores_punto2$minimo[1], valores_punto2$minimo[2]))
```

```
## [1] -0.03433936
```

```
valores_punto3$minimo
```

```
## [1] 1.546374 3.140278
```

```
print(fun1b(valores_punto3$minimo[1], valores_punto3$minimo[2]))
```

```
## [1] 2.362917
```

```
valores_punto4$minimo
```

```
## [1] 1.961298 1.377935
```

```
print(fun1b(valores_punto4$minimo[1], valores_punto4$minimo[2]))
```

```
## [1] 0.4412487
```

Punto inicio	u	v	Mínimo obtenido
(2.1, 2.1)	2.392957	1.067868	2.407525
(3, 3)	2.298454	1.989681	-0.03433936
(1.5, 1.5)	1.546374	3.140278	2.362917
(1, 1)	1.961298	1.377935	0.4412487

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

Encontrar el mínimo global de una función no resulta una tarea trivial. Con el algoritmo que hemos implementado de hecho no podemos estar seguros de que lo vamos a encontrar, pero podemos buscar un balance entre el tiempo de ejecución que estemos dispuestos a admitir (que se traduce en el ancho de la función a explorar) y la bondad de la solución obtenida. Como hemos comprobado antes en este mismo ejercicio, utilizar una tasa de aprendizaje muy baja favorece seguir el comportamiento de la función y prácticamente asegurar encontrar un mínimo local en pocas iteraciones. Sin embargo, no explora en profundidad ni es capaz de pasar del primer mínimo local al tener una combinación umbral-tasa de aprendizaje poco efectiva. En estas situaciones, utilizar una tasa de aprendizaje muy alta puede provocar altibajos casi constantes si la función es compleja, pero hacerla suficientemente grande a la vez nos garantiza pasar de ese tramo en el que se ha quedado estancado en el primer caso. Por tanto, no es un problema sencillo de resolver, ni creo que tenga una opción mejor que otra necesariamente, por lo que deberemos optar por una u otra dependiendo de las condiciones que conozcamos del problema y de la función que lo compone.

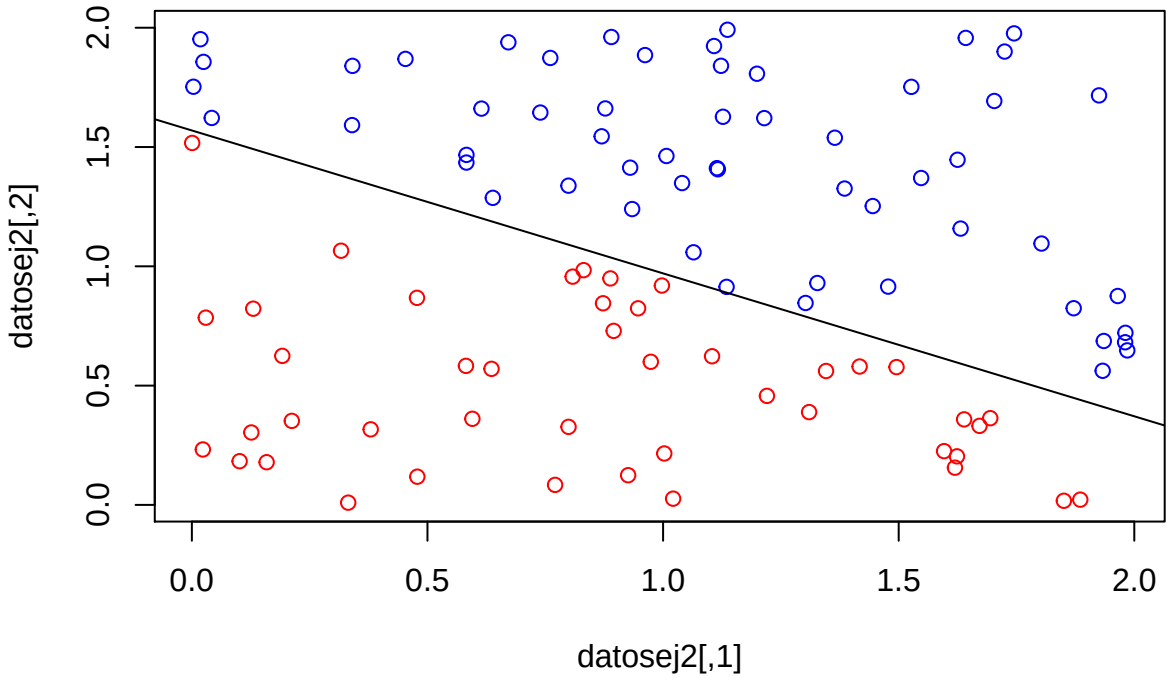
Ejercicio 2. Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que la etiqueta y es una función determinista de x.

Consideremos d = 2 para que los datos sean visualizables, y sea X = [0, 2] × [0, 2] con probabilidad uniforme de elegir cada x perteneciente 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 {xn} de X y evaluar las respuestas {yn} de todos ellos respecto de la frontera elegida.

Este párrafo nos indica que debemos partir de un conjunto de datos cuyas etiquetas están seleccionadas en función de la separación de una recta, también generada aleatoriamente. En este caso se reduce el área de trabajo al rango [0,2] tanto para x como para y en las coordenadas de los puntos.

```
# Funciones recicladas de la práctica 1, para generar datos aleatorios uniformemente
# distribuidos, rectas, y asignar etiquetas. También se incluye la función que
# calcula una recta en función de unos pesos, que se usará más adelante.
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
}

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
}

asignarEtiquetasSegunRecta = function(recta,puntos){
    etiquetas = sign(puntos[,2] - (recta[1]*puntos[,1]+recta[2]))
}

obtenerRectaPesos = function(pesos){
  c(-pesos[1]/pesos[2], -pesos[3]/pesos[2])
}

datosej2 = simula_unif(100, 2, c(0,2))
rectaej2 = simula_recta(c(0,2),F)
etiquetasej2 = asignarEtiquetasSegunRecta(rectaej2, datosej2)

plot(datosej2, col=etiquetasej2+3)
abline(rectaej2[2], rectaej2[1])
```



Aquí podemos observar que la gráfica es coherente, la línea separa perfectamente los datos y no hay ruido de ningún tipo en las etiquetas.

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 mu = 0,01

```
# Función que computa el gradiente descendente estocástico para un punto con su
# etiqueta y un vector de pesos. Esta función será llamada dentro del algoritmo
# de regresión logística cuando se deban ir actualizando los pesos.
GDEstocastico = function(punto, etiqueta, w){
  -(crossprod(etiqueta,punto)[1,] / (1 + exp(etiqueta*w%*%punto)))
}

# Función que calcula el módulo de un vector como la raíz de la sumatoria de sus
# componentes al cuadrado.
moduloVector = function(vector){
  sqrt(sum(vector^2))
}

# Función que calcula unos pesos que separan unos datos de unas etiquetas por
# regresión logística. El algoritmo itera el número de épocas que hagan falta,
# con los pesos actualizándose con cada dato y comprobándose al final de la misma
# si el módulo del nuevo vector de pesos varía más del umbral con respecto al
# de la época anterior. Este algoritmo ha sido extraído de la página 95 del libro
# "Learning from data" de Abu-Mostafa et al.
regresionLogistica = function(datos, etiquetas, threshold, mu){
  wnew = c(0,0,0)
  wold = c(0,0,0)
  datos = cbind(datos,1)
  epoca = 0
  modulo = threshold*2
  evaluado = 1
  barajados = sample(1:dim(datosej2)[1])

  while (modulo > threshold){
    wold = wnew

    while (evaluado <= length(barajados)){
      g = GDEstocastico(datos[barajados[evaluado],], etiquetas[barajados[evaluado]], wold)
      evaluado = evaluado+1
      v = -g
      wnew = wnew + mu*v
    }

    evaluado = 1
    barajados = sample(1:dim(datosej2)[1])
```

```
    modulo = moduloVector(wnew - wold)
    epoca = epoca + 1
  }

  resultados = list(pesos=wnew, epocas=epoca)
}
```

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

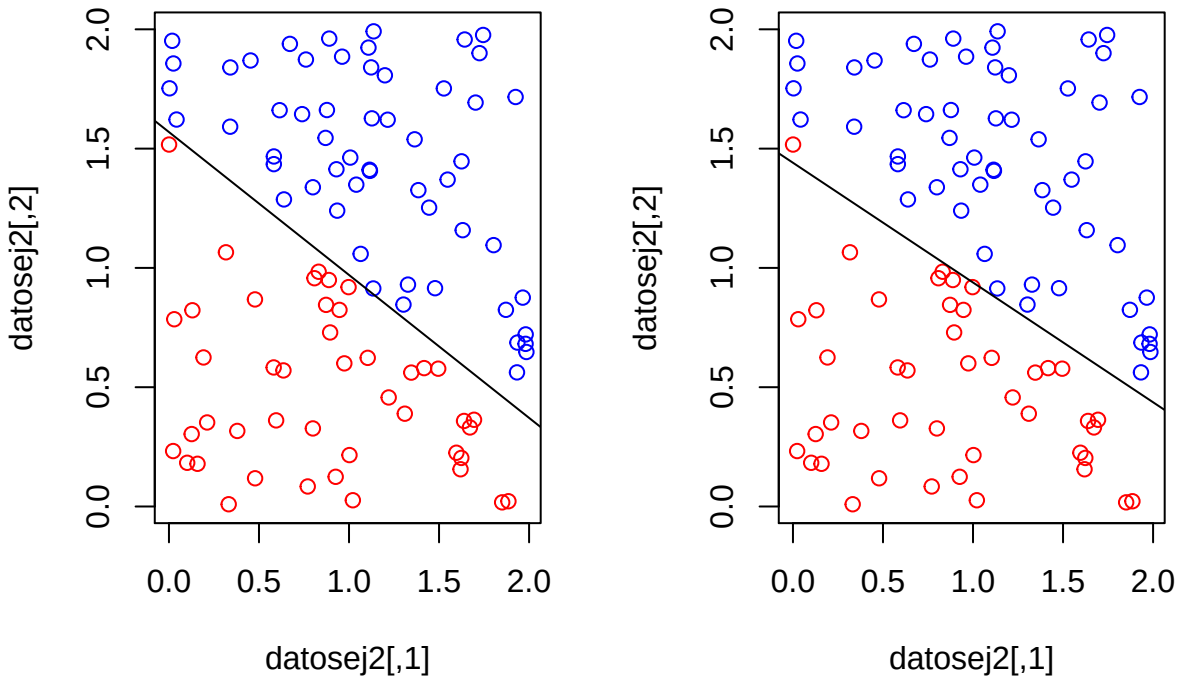
```
# Función que calcula el error para la regresión logística basándose en la fórmula
# de la página 98 del libro "Learning from data", a partir de los datos, las etiquetas
# y los pesos que se han generado. Finalmente devuelve la media del error de todos
# los datos.
errorRLog = function(datos,etiquetas,pesos){
  error = 0
  datos = cbind(datos,1)
  for (i in 1:dim(datos)[1]){
    error = error + log(1 + exp(-etiquetas[i]*(pesos%*%datos[i,])), exp(1))
  }
  error/dim(datos)[1]
}

# Función que computa el experimento pedido en el ejercicio 2b, generando N muestras
# y manteniendo los pesos calculados con regresión logística y la recta separadora
# original para comprobar el error fuera de la muestra.
ejercicio2b = function(recta, pesos, muestras){
  datos = simula_unif(muestras, 2, c(0,2))
  etiquetas = asignarEtiquetasSegunRecta(recta, datos)

  Eout = errorRLog(datos, etiquetas, pesos)
  Eout
}

resultadoRLog = regresionLogistica(datosej2, etiquetasej2, 0.01, 0.01)
rectapesosRLog = obtenerRectaPesos(resultadoRLog$pesos)
resultadoRLog$epocas
```

```
## [1] 511
```

```
resultadoRLog$pesos
```

```
## [1]  3.312706  6.599732 -9.508783
```

```
par(mfrow=c(1,2))
plot(datosej2, col=etiquetasej2+3)
abline(rectaej2[2], rectaej2[1])
plot(datosej2, col=etiquetasej2+3)
abline(rectapesosRLog[2], rectapesosRLog[1])
```




```
Eout = ejercicio2b(rectapesosRLog, resultadoRLog$pesos, 1000)
Ein = errorRLog(datosej2, etiquetasej2, resultadoRLog$pesos)
Eout
```

```
##           [,1]
## [1,] 0.123645
```

```
Ein
```

```
##            [,1]
## [1,] 0.1174677
```

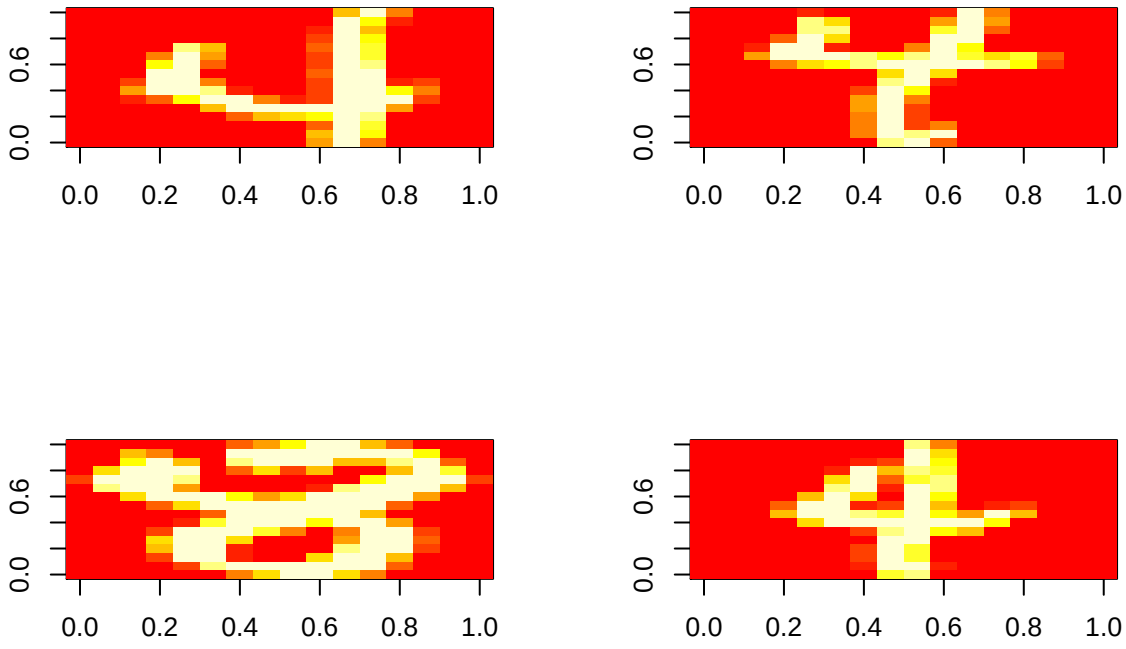
En estas dos gráficas que observamos tenemos a la izquierda la de la recta separadora original, y a la derecha la de la recta generada mediante los pesos obtenidos en regresión logística. Como se puede observar, la recta de pesos no clasifica al 100 los datos de forma correcta, pero dado que el algoritmo se detiene cuando la modificación es mínima en una pasada, eso hace que se pueda finalizar sin tener todos los datos bien colocados.

Para el experimento de la muestra de 1000 datos, he calculado el Eout manteniendo la recta y los pesos de regresión logística, y vemos que es ligeramente mayor que el Ein para los mismos pesos con los datos iniciales. Esto nos hace entender que hemos ajustado una buena función dado que para una muestra tan grande el error apenas crece una centésima.

Ejercicio 3. Clasificación de Dígitos. 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.

A continuación introduzco el código casi calcado de la primera práctica para leer los ficheros, y almacenarlos en matrices de datos, esta vez con los datos 4 y 8 en lugar de con los 1 y 5. Se incluyen el cálculo de la simetría e intensidad y se imprimen las gráficas de los datos de train y de test en base a estos datos.

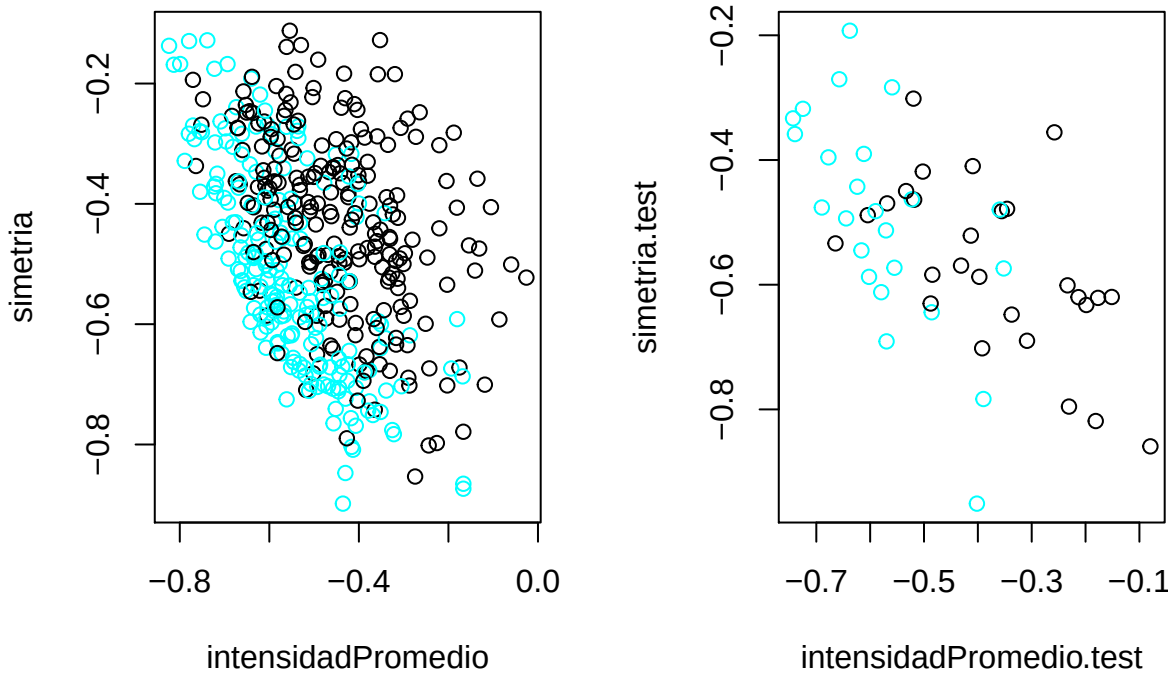
```
digit.train <- read.table("datos/zip.train",
                          quote="\"", comment.char="", stringsAsFactors=FALSE)

digitos48.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]
digitos = digitos48.train[,1]  # etiquetas
ndigitos = nrow(digitos48.train)

# se retira la clase y se monta una matriz 3D: 599*16*16
grises = array(unlist(subset(digitos48.train,select=-V1)),c(ndigitos,16,16))
rm(digit.train)
rm(digitos48.train)

# Para visualizar los 4 primeros
## ------------------------------------------------------------------------

par(mfrow=c(2,2))
for(i in 1:4){
  imagen = grises[i,,16:1] # se rota para verlo bien
  image(z=imagen)
}
```



```r
digitos[1:4] # etiquetas correspondientes a las 4 imágenes
```

```
## [1] 4 4 8 4
```

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

simetria = apply(grises, 1, fsimetria)
intensidadPromedio = apply(grises, 1, mean)
par(mfrow = c(1,2))
plot(x=intensidadPromedio, y=simetria, col=digitos+1)

digit.test <- read.table("datos/zip.test",
                         quote="\"", comment.char="", stringsAsFactors=FALSE)

digitos48.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]
digitos.test = digitos48.test[,1]  # etiquetas
ndigitos.test = nrow(digitos48.test)

# se retira la clase y se monta una matriz 3D: 599*16*16
grises.test = array(unlist(subset(digitos48.test,select=-V1)),c(ndigitos.test,16,16))
rm(digit.test)
rm(digitos48.test)

simetria.test = apply(grises.test, 1, fsimetria)
intensidadPromedio.test = apply(grises.test, 1, mean)
plot(x=intensidadPromedio.test, y=simetria.test, col=digitos.test+1)
```



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

La idea del problema a abordar es obtener un vector de pesos con el conjunto de train, y tratar ese vector como mi función g. Este vector se calculará con regresión lineal y se intentará mejorar con PLA-pocket. Una vez que he calculado la g, o mi vector de pesos, evaluaré cómo de bien clasifica los datos del test, y compararé los errores en uno y otro conjunto, esta vez usando como error el porcentaje de etiquetas mal clasificadas dado que nos enfrentamos a un problema de clasificación.

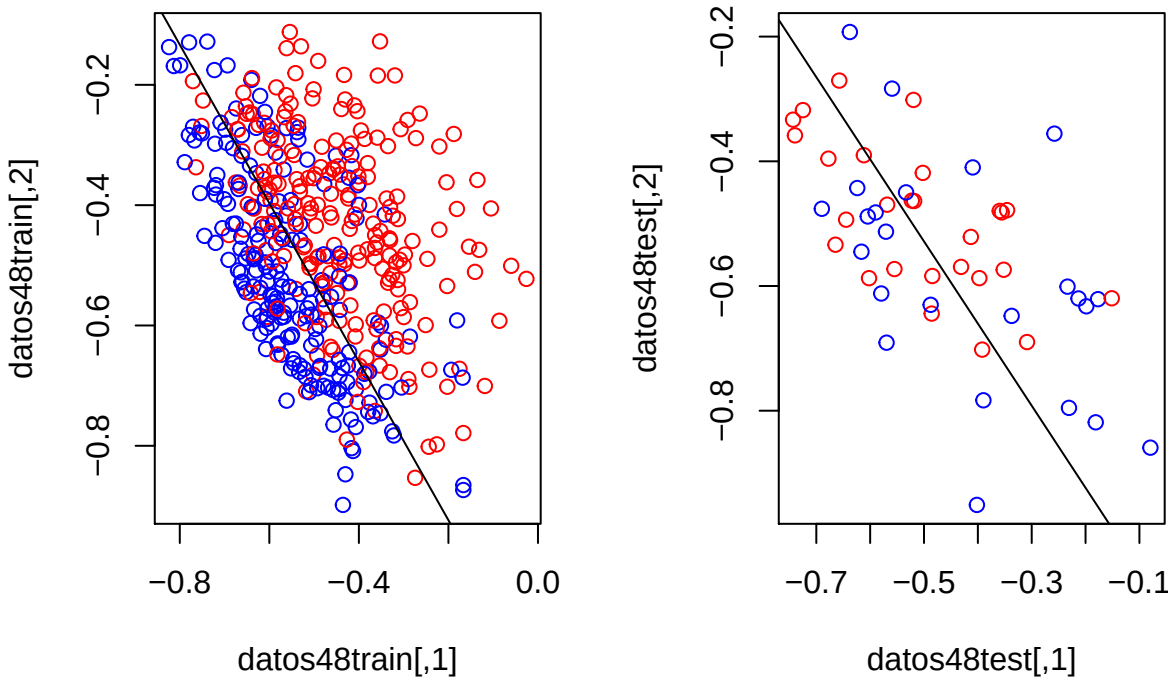
b) Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

```
# Función que calcula el error de unas etiquetas de unos datos para un vector de pesos
# en base a cuántas etiquetas están mal situadas con respecto a la recta que formaría
# dicho vector.
errores = function(datos,label,pesos){
  (sum(sign(datos%*%pesos) != label))/length(label)
}

# Función que calcula los pesos por regresión lineal de unos datos y unas etiquetas
# aportadas. Se hace uso de la fórmula para calcular la matriz pseudoinversa de
# los datos que aparece en las diapositivas de teoría. Según esto, la matriz
# pseudoinversa de X se puede obtener como la multiplicación de la inversa de
# la multiplicación de X traspuesta por X, por la traspuesta de X. En fórmula
# matemática, pseudoX = (XT*X)-1 * XT.
# El resto de operaciones son las mismas que aparecen en las transparencias,
# empleando la función svd para extraer las matrices U, D y V de X, y sabiendo
# que (XT*X)-1 = V * (pseudoD)^2 * VT (en las transparencias no aparece el cuadrado
# por error, pero lo corregimos en clase) y que la pseudoD es la diagonal de 1/D.
# Finalmente, los pesos se obtienen de multiplicar la pseudoinversa de X por las
# etiquetas.
Regress_Lin = function(datos,label){

  datos=cbind(datos,1)

  descomposicion = svd(datos)
  pseudoinversaD = diag(1/descomposicion$d)
  inversadatosTdatos = (descomposicion$v)%*%(pseudoinversaD^2) %*%t(descomposicion$v)
  pseudoinversa = inversadatosTdatos%*%(t(datos))

  pesos = (pseudoinversa%*%label)
}

# Función que ajusta el algoritmo Perceptron para unos datos y unas etiquetas.
# Funciona similarmente al PLA original pero este guarda la mejor solución
# por la que ha pasado (la que arrojaba el error más pequeño) y sólo la actualiza
# cuando los nuevos pesos generados en cada iteración dan un mejor resultado.
# Devuelve en este caso el mejor resultado y no el último.
PLA_pocket = function(datos, label, max_iter, vini){
  w = vini
  cambio = T
  datos = cbind(datos,1)
```

```
  iteraciones = 0

  menor_error = errores(datos,label,w)
  mejor_w = w

  while(iteraciones < max_iter & cambio){
    cambio = F
    for(j in sample(1:dim(datos)[1])){
      if (sign(crossprod(datos[j,],w)) != label[j]){
        w = w + datos[j,]*label[j]
        cambio = T
        break
      }
    }

    if (cambio){
      error_actual = errores(datos,label,w)
      if (error_actual < menor_error){
        menor_error = error_actual
        mejor_w = w
      }
    }

    iteraciones = iteraciones+1
  }

  list(w=mejor_w, iter=iteraciones)
}
```

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

```
datos48train = matrix(c(intensidadPromedio,simetria),nrow=length(intensidadPromedio),ncol=2)
digitos[digitos==4]=1
digitos[digitos==8]=-1

datos48test = matrix(c(intensidadPromedio.test,simetria.test),nrow=length(intensidadPromedio.test),ncol=
digitos.test[digitos.test==4]=1
digitos.test[digitos.test==8]=-1

pesos_train = Regress_Lin(datos48train, digitos)
resPLApocket = PLA_pocket(datos48train, digitos, 10000, pesos_train)
pesos_train = resPLApocket$w
recta_pesos_train = obtenerRectaPesos(pesos_train)

par(mfrow=c(1,2))
plot(datos48train, col=digitos+3)
abline(recta_pesos_train[2], recta_pesos_train[1])
plot(datos48test, col=digitos+3)
abline(recta_pesos_train[2], recta_pesos_train[1])
```



Vemos que la recta que se ha utilizado para separar los datos es la misma, y bajo esta misma recta (en realidad, los pesos que la generan) se calcularán los errores del apartado siguiente.

2) Calcular Ein y Etest (error sobre los datos de test).

```
Ein_3 = errores(cbind(datos48train, 1), digitos, pesos_train)
Etest_3 = errores(cbind(datos48test, 1), digitos.test, pesos_train)
Ein_3
```

```
## [1] 0.2222222
```

```
Etest_3
```

```
## [1] 0.2352941
```

Encontramos un error muy similar para los datos de entrenamiento y los de test, esto parece indicar que de momento se ha encontrado una buena solución, o al menos se tiene una muestra de entrenamiento que representa bien la población total de muestras existentes, ya que aproxima bien en nuestra prueba.

3) Obtener cotas sobre el verdadero valor de Eout. Pueden calcularse dos cotas una basada en Ein y otra basada en Etest. Usar una tolerancia delta = 0,05. ¿Que cota es mejor?

Con la fórmula de la cota de generalización Vapnik-Chervonenkis que aparece en las transparencias de la sesión 4 de teoría se puede obtener una cota para el Eout en relación a un Ein. En este caso sustituiremos ese error por el Ein de los datos de train y el Etest de los datos de test.

```
# Función que calcula la cota de error estimada en base a la fórmula de la
# cota de generalización Vapnik-Chervonenkis. La tolerancia determinará el
# porcentaje de confianza que podemos esperar para estos resultados.
# La dimensión de Vapnik-Chervonenkis del Perceptron 2D es 3.
calcularCotaRespectoError = function(error, dvc, tolerancia, N){
  error + sqrt(8/N * log((4*((2*N)^dvc + 1)/tolerancia), exp(1)))
```

```
}

cota_segun_ein = calcularCotaRespectoError(Ein_3, 3, 0.05, dim(datos48train)[1])
cota_segun_etest = calcularCotaRespectoError(Etest_3, 3, 0.05, dim(datos48test)[1])
cota_segun_ein
```

```
## [1] 0.8980858
```

```
cota_segun_etest
```

```
## [1] 1.927581
```

Aquí vemos que hemos obtenido una cota de error 0.898 para el Ein y 1.928 para el Etest. En el segundo caso la cota no nos está aportando nada, pues tenemos pocos datos de test y no se pueden esperar un error menor a 1 para el 95% de confianza. Lo interpretamos como que no obtenemos nueva información, simplemente recalcamos que para ese tamaño de muestra de test con ese error reflejado no podemos esperar nada en particular. Cuando hablamos de la cota dada por el Ein sí obtenemos algo más sustancial. En este caso, la cota indica que para el tamaño de train que hemos estipulado, y con ese error obtenido, al 95% de confianza podremos obtener un error fuera de la muestra menor que 0.898. Esto ya nos restringe el error esperado, ya que pocas veces esperamos que sobrepase esa cota, o dicho de otra manera, ya estamos aprendiendo algo. Es una mejora pobre si tenemos en cuenta que tenemos un Ein de 0.22, pero si queremos que el Eout se reduzca más aún debemos ser capaces de obtener ese Ein para una muestra de datos de train mucho más grande, entonces se reflejará en la cota.

Ejercicio 4. En este ejercicio evaluamos el papel de la regularización en la selección de modelos. Para d = 3 (dimensión del vector de características) generar un conjunto de N datos aleatorios {xn, yn} de la siguiente forma:

Las coordenadas de los puntos xn se generarán como valores aleatorios extraidos de una Gaussiana de media 1 y desviación típica 1.

Para definir el vector de pesos wf 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 xn a partir del valor yn = wTf xn + sigma epsilon n , donde epsilon n es un ruido que sigue también una Gaussiana de media 0 y desviación típica 1 y sigmaˆ2 es la varianza del ruido; fijar sigma = 0,5

```
# Función aportada en la práctica 1, con una ligera modificación para que acepte
# la media que se le quiere dar a la distribución gaussiana.
simula_gaus = function(N=2,dim=2,sigma, mean){

  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(dim, mean=mean, sd = sigma) # genera 1 muestra, con las desviaciones
  m = t(replicate(N,simula_gauss1())) # repite N veces, simula_gauss1 y se hace la traspuesta
```

```
  m
}

# Función que calcula las etiquetas de los datos de la gaussiana en función de los pesos wf,
# y metiendo un ruido como se indica en el enunciado.
obtenerEtiquetasGauss = function(datos, pesos, sigma){
  etiquetas = as.vector(as.vector(datos%*%t(pesos))+sigma*simula_gaus(dim(datos)[1],1,sigma,0))
}

datosgauss3 = simula_gaus(130, 3, c(1,1,1), 1)

pesosgauss3 = simula_gaus(4,1,1,0)
pesosgauss3[4] = pesosgauss3[4]+1

etiquetasgauss3 = obtenerEtiquetasGauss(cbind(datosgauss3,1), pesosgauss3, 0.5)
```

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

```
# Función que calcula los pesos según la fórmula de weight decay como se indica
# en la sesión 7 de teoría, utilizando el parámetro de regularización que se
# especifica en el enunciado y empleando la técnica de cálculo de pseudoinversas
# con descomposición SVD.
weightDecay = function(datos, etiquetas, regularizacion){
  tam = dim(datos)[2]
  matriz = t(datos)%*%datos + regularizacion*diag(tam)

  descomposicion = svd(matriz)
  pseudoinversaD = diag(1/descomposicion$d)
  inversadatosTdatos = (descomposicion$v)%*%(pseudoinversaD^2) %*%t(descomposicion$v)
  pseudoinversa = inversadatosTdatos%*%(t(matriz))

  pseudoinversa %*% t(datos)%*%etiquetas
}

wreg = weightDecay(cbind(datosgauss3,1), etiquetasgauss3, 0.05/dim(datosgauss3)[1])
wreg
```

```
##            [,1]
## [1,] -0.7099581
## [2,] -0.2006722
## [3,] -1.2066331
## [4,]  1.0515881
```

a) Para N $\in$ {d + 10, d + 20, . . . , d + 110} calcular los errores de validación cruzada e1 , . . . , eN y Ecv . Repetir el experimento 1000 veces. Anotamos el promedio y la varianza de e1 , e2 y Ecv en los experimentos.

IMPORTANTE: En la realización de este ejercicio he entendido que todos los conjuntos debían tener el mismo tamaño, por lo que he generado 130 datos, a fin de tener conjuntos de 13 datos cada uno.

```
# Función que calcula el error de mínimos cuadrados como aparece en la página
# 139 del libro "Learning from data", evaluando la diferencia cuadrada entre
# las etiquetas reales y las obtenidas con los pesos del weight decay.
errorMinimosCuadrados = function(test, etiquetas_test, pesos){
  errores = (as.vector(test%*%as.vector(pesos)) - etiquetas_test)^2
  sum(errores)
}

# Función que realiza los cálculos pertinentes al experimento propuesto
# y que devuelve una matriz de errores, donde la columna i equivale al
# error de validación del conjunto i, y la fila j contiene los errores
# del experimento j. La última columna contiene la media de los errores.
experimento4 = function(datos, etiquetas){
  d = dim(datos)[2]
  datos = cbind(datos,1)
  errores = matrix(ncol=11)
  errores = errores[-1,]
  for (experimento in 1:1000){
    datos_barajados = sample(1:dim(datos)[1])
    errores_experimento = vector()
    for (indice in 1:10){
      intervalo_inf = (indice-1)*d + (indice-1)*10 + 1
      intervalo_sup = indice*d + indice*10
      train = datos[datos_barajados[-(intervalo_inf:intervalo_sup)],]
      test = datos[datos_barajados[intervalo_inf:intervalo_sup],]
      etiquetas_train = etiquetas[datos_barajados[-(intervalo_inf:intervalo_sup)]]
      etiquetas_test = etiquetas[datos_barajados[intervalo_inf:intervalo_sup]]
      pesos_decay = weightDecay(train, etiquetas_train, 0.05/dim(train)[1])

      error = errorMinimosCuadrados(test, etiquetas_test, pesos_decay)
      errores_experimento = c(errores_experimento, error)
    }
    error = sum(errores_experimento)/10
    errores_experimento = c(errores_experimento, error)

    errores = rbind(errores, errores_experimento)
  }
  errores
}

matrizerrores = experimento4(datosgauss3, etiquetasgauss3)

print("Errores")
```

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

```
e1 = mean(matrizerrores[,1])
e1
```

```
## [1] 1.558396
```

```
e2 = mean(matrizerrores[,2])
e2
```

```
## [1] 1.556428
```

```
e3 = mean(matrizerrores[,3])
e3
```

```
## [1] 1.484462
```

```
e4 = mean(matrizerrores[,4])
e4
```

```
## [1] 1.504208
```

```
e5 = mean(matrizerrores[,5])
e5
```

```
## [1] 1.517118
```

```
e6 = mean(matrizerrores[,6])
e6
```

```
## [1] 1.522849
```

```
e7 = mean(matrizerrores[,7])
e7
```

```
## [1] 1.569588
```

```
e8 = mean(matrizerrores[,8])
e8
```

```
## [1] 1.489393
```

```
e9 = mean(matrizerrores[,9])
e9
```

```
## [1] 1.512584
```

```
e10 = mean(matrizerrores[,10])
e10
```

```
## [1] 1.547622
```

```
ecv = mean(matrizerrores[,11])
ecv
```

```
## [1] 1.526265
```

```
print("Varianzas")
```

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

```
v1 = var(matrizerrores[,1])
v1
```

```
## [1] 0.4456733
```

```
v2 = var(matrizerrores[,2])
v2
```

```
## [1] 0.4357502
```

```
v3 = var(matrizerrores[,3])
v3
```

```
## [1] 0.3620916
```

```
v4 = var(matrizerrores[,4])
v4
```

```
## [1] 0.4007205
```

```
v5 = var(matrizerrores[,5])
v5
```

```
## [1] 0.3914027
```

```
v6 = var(matrizerrores[,6])
v6
```

```
## [1] 0.4054183
```

```
v7 = var(matrizerrores[,7])
v7
```

```
## [1] 0.4481613
```

```
v8 = var(matrizerrores[,8])
v8
```

```
## [1] 0.3870269
```

```
v9 = var(matrizerrores[,9])
v9
```

```
## [1] 0.3819318
```

```
v10 = var(matrizerrores[,10])
v10
```

```
## [1] 0.4193247
```

```
vcv = mean(c(v1,v2,v3,v4,v5,v6,v7,v8,v9,v10))
vcv
```

```
## [1] 0.4077501
```

b) ¿Cuál debería de ser la relación entre el valor promedio de e1 y el de Ecv ? ¿y entre el valor promedio de e1 y el de e2? Argumentar la respuesta en base a los resultados de los experimentos.

Vistos los resultados, aleatorizar los datos en cada iteración del experimento para generar distintas particiones no afecta en exceso a la media de errores obtenidos. Para 1000 conjuntos de particiones distintas, el e1 ha resultado ser prácticamente similar al e2, y ambos a Ecv. Dado que Ecv viene dado en función de los valores e1, e2 y el resto de errores, es de esperar que si estos errores no varían apenas entre sí tampoco lo haga la media de ellos.

c) ¿Qué es lo que más contribuye a la varianza de los valores de e1 ?

Por el factor de aleatoriedad de los datos seleccionados en cada caso para el conjunto test se producirá una varianza entre los errores. Sin embargo, entiendo que realmente aquí el factor que influye más en la varianza

es el ruido que se incluye en los pesos, ya que es algo que afecta a la etiqueta calculada para cada uno de los puntos y a la larga influye en el error de forma más significativa.

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

Por la forma en la que he entendido el ejercicio creo que no puedo responder correctamente a la pregunta que se plantea, o al menos no de la forma esperada. Sin embargo, puedo interpretar que dando unos valores tan estables a lo largo de 1000 experimentos para distintas particiones, la regularización puede reflejar resultados aceptables y que compitan con los pesos con los que se generaron las etiquetas.

Bonus 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 mu = 0,1.

```
# Función que calcula el algoritmo de coordenada descendente de forma
# similar al algoritmo de gradiente descendente pero realizando el gradiente
# explorando primero en la coordenada x y a continuación en la coordenada y
# con la x calculada.
coordenadaDescendente = function(wini, mu, threshold, max_iter, E, duE, dvE){

  wold = wini
  iter = 0
  seguir_iterando = T
  wnew = vector(length=2)

  while (iter < max_iter & seguir_iterando){
    g=c(duE(wold[1],wold[2]),dvE(wold[1],wold[2]))
    v = -g
    wnew[1] = wold[1] + mu*v[1]
    g=c(duE(wold[1],wold[2]),dvE(wnew[1],wold[2]))
    v = -g
    wnew[2] = wold[2] + mu*v[2]
    if (abs(E(wold[1],wold[2]) - E(wnew[1],wnew[2])) < threshold){
      seguir_iterando = F
    }
    wold = wnew
    iter = iter+1
  }
```

```
  list(minimo=wnew, iter=iter)
}

resultado_coordesc = coordenadaDescendente(c(1,1), 0.1, 10^-4, 50, E, duE, dvE)
resultado_coordesc$minimo
```

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

```
E(resultado_coordesc$minimo[1], resultado_coordesc$minimo[2])
```

```
## [1] 0
```

En este caso el algoritmo de coordenadas descendentes ofrece mejores resultados, dando un mínimo de 0, que es menor que el mínimo que obtuvimos con gradiente descendente.

Bonus 3. Repetir el experimento de RL (punto.2) 100 veces con diferentes funciones frontera y calcule el promedio.

a) ¿Cuál es el valor de Eout para N = 100?

```
# Función que realiza los cálculos propuestos en el bonus 3, almacenando
# en una lista de salida el error medio y la media de épocas que han tardado
# los experimentos.
bonus3 = function(num_veces){
  errores = vector()
  epocas = vector()
  for (i in 1:num_veces){
    datos = simula_unif(100, 2, c(0,2))
    recta = simula_recta(c(0,2),F)
    etiquetas = asignarEtiquetasSegunRecta(recta, datos)

    lista_resultados = regresionLogistica(datos, etiquetas, 0.01, 0.01)
    error = errorRLog(datos, etiquetas, lista_resultados$pesos)
    errores = c(errores, error)
    epocas = c(epocas, lista_resultados$epocas)
  }
  list(epocas=mean(epocas), eout=mean(errores))
}

resultadosbonus3 = bonus3(100)
resultadosbonus3$eout
```

```
## [1] 0.1068774
```

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

```
resultadosbonus3$epocas
```

```
## [1] 409.8
```