Práctica 2 - Aprendizaje Automático

Arthur M. Rodríguez Nesterenko - DNI:Y1680851W

27 de abril de 2017

Ejercicio 1: Gradiente Descendente.

Implementar el algoritmo de gradiente descendente.

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

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

Para calcular el gradiente de la función E se han realizado los siguientes pasos

$\nabla E(u,v) = \frac{\partial(E)}{\partial(u)} + \frac{\partial(E)}{\partial(v)}$

Calculamos la derivada parcial de E respecto de u:

$\frac{\partial E}{\partial(u)} = 2(u^2e^v - 2v^2e^{-u})^2 * (2ue^v + 2v^2e^{-u})$

Calculamos la derivada parcial de E respecto de v:

$\frac{\partial E}{\partial(v)} = 2(u^2e^v - 2v^2e^{-u})^2 * (u^2e^v - 4ve^{-u})$

Ahora agrupamos toda la expresión del gradiente descendente

$\nabla E(u,v) = 2(u^2e^v - 2v^2e^{-u}) * (2ue^v + 2v^2e^{-u}) + 2(u^2e^v - 2v^2e^{-u}) * (u^2e^v - 4ve^{-u}))$

Una vez calculado el gradiente de E(u,v) vamos a implementar las funciones correspondientes para poder calcular a partir de los valores de u y de v el valor tanto del gradiente en cada paso como de la propia función para poder contrastar con el error que se nos pide

```
## Funcion E(u,v) que recibe dos parametros u y v y calcula el valor de la funcion
## que se nos da en el apartado a) en esos puntos
E = function(u,v){
  e=exp(1)
  valorE= ((u^2) * (e^v) - 2*(v^2) * (e^-u))^2
  valorE
}
```

```
## Funcion GradienteDeE(u,v) que a partir del calculo de las derivadas parciales
## de la funcion E con respecto de u y v, calcula el gradiente de la funcion E
GradienteDeE = function(u,v){
  e=exp(1)
  derivadaParcialRespectoU = 2*((u^2)* (e^v) - 2*(v^2)*(e^-u))*(2*u*(e^v) + 2*(v^2)* (e^-u))
  derivadaParcialRespectoV = 2*((u^2)* (e^v) - 2*(v^2)*(e^-u))*(u^2 *(e^v) - 4*v*(e^-u))
  gradiente = c(derivadaParcialRespectoU,derivadaParcialRespectoV)
  gradiente
}
```

El siguiente paso es implementar el Algoritmo del Gradiente Descendente para evaluar la función E(u,v) o cualquier otra que se le pase como argumento

```
## GradienteDescendente(valoresIniciales,funcion,gradienteFuncion,tasaAprendizaje,cotaError)
## Este algoritmo calcula el numero de iteraciones necesarias para que una funcion
## determinada partiendo de unos valoresIniciales (u y v) y siguiendo su gradienteFuncion,
## consiga un error menor que cotaErrorMin o llegue a ejecutar un maximo de iteraciones
## maxIters siguiendo una tasaDeAprendizaje prefijada.
## El algoritmo devuelve los valores de U y V tras converger a esa cota de error
GradienteDescendente = function(valoresIniciales,funcion,gradienteFuncion,
                                tasaDeAprendizaje, cotaErrorMin, maxIters){

  #Declaramos las variables que vamos a utilizar en nuestro algoritmo
  #y calculamos un primer valor de error
  iteraciones = 0
  u = valoresIniciales[1]
  v = valoresIniciales[2]
  W_old = c(u,v)
  cotaError = funcion(W_old[1],W_old[2])

  #Mientras que  error este por encima de la cota minima de error
  while(cotaError > cotaErrorMin && iteraciones < maxIters){

    #Calculamos el vector gradiente
    gradiente = gradienteFuncion(W_old[1],W_old[2])
    vectorGradiente = -gradiente

    #Actualizamos los valores de los pesos y calculamos la nueva cota de Error
    #con los valores U y V actualizados (los de W)
    W_new = W_old + (tasaDeAprendizaje*vectorGradiente)
    cotaError = abs(funcion(W_old[1],W_old[2]) - funcion(W_new[1],W_new[2]))
    iteraciones = iteraciones + 1

    #Nos quedamos con los pesos de la iteracion anterior
    #ya que seran los que nos serviran para comparar
    W_old = W_new
  }

  #Devolvemos el numero de iteraciones y los valores de U y V finales
  list(iteracionesMax = iteraciones, pesos = W_new)

}
```

Para probar el algoritmo con los datos de este apartado mostramos por pantalla el numero de iteraciones tras terminar el algoritmo y los valores u y v tras la ejecucion.

```
resultados1a = GradienteDescendente(c(1,1),E,GradienteDeE,0.1, 10^-4, 30000)
print(sprintf("El algoritmo termina tras %i iteraciones",resultados1a$iteracionesMax))
```

```
## [1] "El algoritmo termina tras 4 iteraciones"
```

```
print(sprintf("Los valores obtenidos son u = %f y v = %f",
              resultados1a$pesos[1],resultados1a$pesos[2]))
```

```
## [1] "Los valores obtenidos son u = 9.864573 y v = -24.438276"
```

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

$ f(x,y) = (x-2)^2 + 2(y-2)^2 + 2sin(2*π x) sin(2π y)$

$\nabla f(x,y) = \frac{\partial(f)}{\partial(x)} + \frac{\partial(f)}{\partial(y)}$

Calculamos la derivada parcial de f respecto de x:

$\frac{\partial(f)}{\partial(x)} = 2(x-2) + 2 * cos(2\pi x) * sin(2\pi y) * 2\pi$

Calculamos la derivada parcial de f respecto de y:

$\frac{\partial(f)}{\partial(y)} = 4(y-2) + 2 * sin(2\pi x) * cos(2\pi y) * 2\pi$

Ahora agrupamos toda la expresión del gradiente descendente

$\nabla f(x,y) = (2(x-2) + 2 * cos(2\pi x) * sin(2\pi y) * 2\pi) + (4(y-2) + 2 * sin(2\pi x) * cos(2\pi y) * 2\pi)$

Vamos a declarar nuestra funcion f(x,y) y el gradiente de la funcion F para realizar los experimentos necesarios.

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

Para calcular el gradiente de la funcion f(x,y) tenemos que empezar por calcular las derivadas parciales con respecto de x y de y. El proceso completo de muestra a continuacion.

```
## Funcion f(x,y) que recibe dos parametros (x e y) y calcula el valor de la funcion
## que se nos da en el apartado b) en esos puntos
f = function(x,y){

  valorFuncion = (x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)

}
```

```
## Funcion GradienteDeF(x,y) que a partir del calculo de las derivadas parciales
## de la funcion F con respecto de x e y, calcula el gradiente de la funcion F
GradienteDeF = function(x,y){

  derivadaParcialRespectoX = 2*(x-2) + 2*cos(2*pi*x)* sin(2*pi*y)*2*pi
  derivadaParcialRespectoY = 4*(y-2) + 2*sin(2*pi*x)* cos(2*pi*y)*2*pi

  gradiente = c(derivadaParcialRespectoX,derivadaParcialRespectoY)
  gradiente
}
```

```
## Realiza la misma funcion que la funcion GradienteDescendente solo que la condicion
## de parada ahora son solo maxIters. El algoritmo devuelve el conjunto de valores (x,y)
## en cada iteracion del mismo.
GradienteDescendenteB = function(valoresIniciales,funcion,gradienteFuncion,
                                 tasaDeAprendizaje, maxIters){

  #Declaramos las variables que vamos a utilizar en nuestro algoritmo
```

```
  #y calculamos un primer valor de error
  iteraciones = 0
  x = valoresIniciales[1]
  y = valoresIniciales[2]
  W = c(x,y)
  cotaError = funcion(W[1],W[2])
  puntosPorIteracion = numeric(0)
  valoresDeVariables = numeric(0)

  #Mientras que  error este por encima de la cota minima de error
  while(iteraciones < maxIters){

    #Calculamos el vector gradiente
    gradiente = gradienteFuncion(W[1],W[2])
    vectorGradiente = -gradiente

    #Guardamos el valor de la funcion y los valores (x,y) que
    #me dan esos valores
    puntosPorIteracion = c(puntosPorIteracion,cotaError)
    valoresDeVariables = rbind(valoresDeVariables,matrix(W,1,2,byrow=TRUE))

    #Actualizamos los valores de los pesos y calculamos la nueva cota de Error
    #con los valores U y V actualizados (los de W)
    W = W + (tasaDeAprendizaje*vectorGradiente)
    cotaError = funcion(W[1],W[2])
    iteraciones = iteraciones + 1
  }

  #Devolvemos el numero de iteraciones y los valores de U y V finales
  list(iteraciones = iteraciones, valoresFinales = W,
       puntos = puntosPorIteracion,
       valoresDeVariables = valoresDeVariables)

}
```

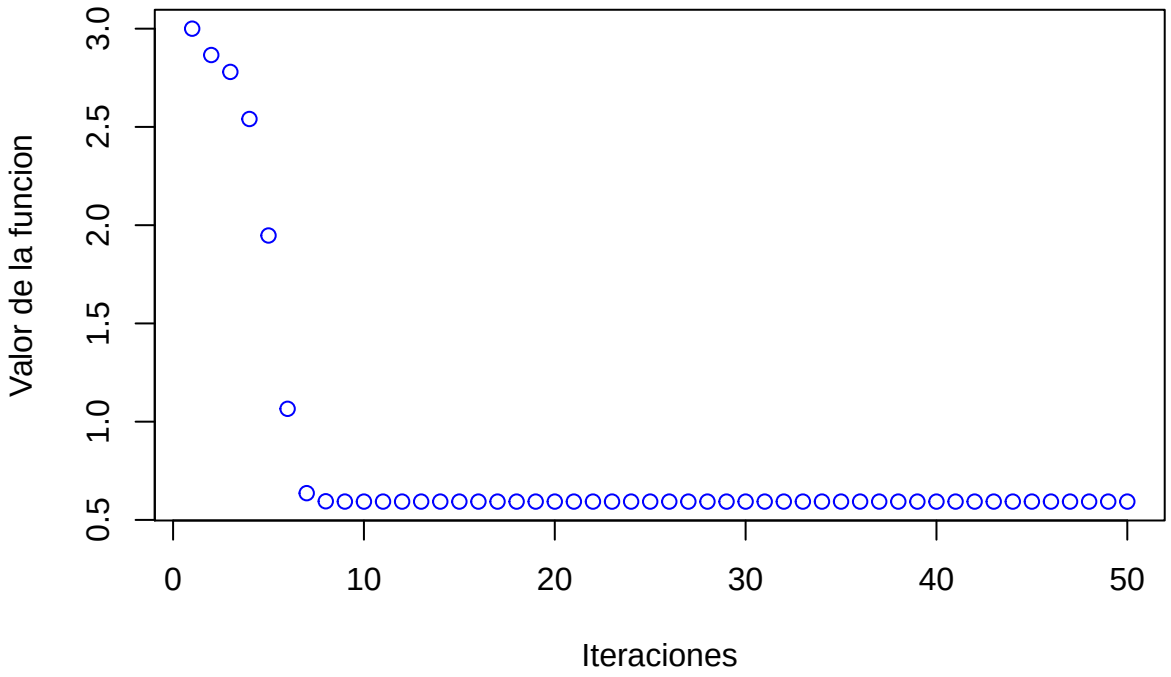
```
## Realizamos las pruebas para comparar con los resultados
## posteriores
resultados1b = GradienteDescendenteB(c(1,1),f,GradienteDeF,0.01, 50)
print(sprintf("El algoritmo termina tras %i iteraciones",
              resultados1b$iteraciones))
```

```
## [1] "El algoritmo termina tras 50 iteraciones"
```

```
print(sprintf("Los valores obtenidos son x = %f e y=%f",
resultados1b$valoresFinales[1],resultados1b$valoresFinales[2]))
```

```
## [1] "Los valores obtenidos son x = 0.781930 e y=1.287188"
```

```
plot(1:length(resultados1b$puntos), resultados1b$puntos, xlab="Iteraciones",
     ylab="Valor de la funcion", col = 4)
```



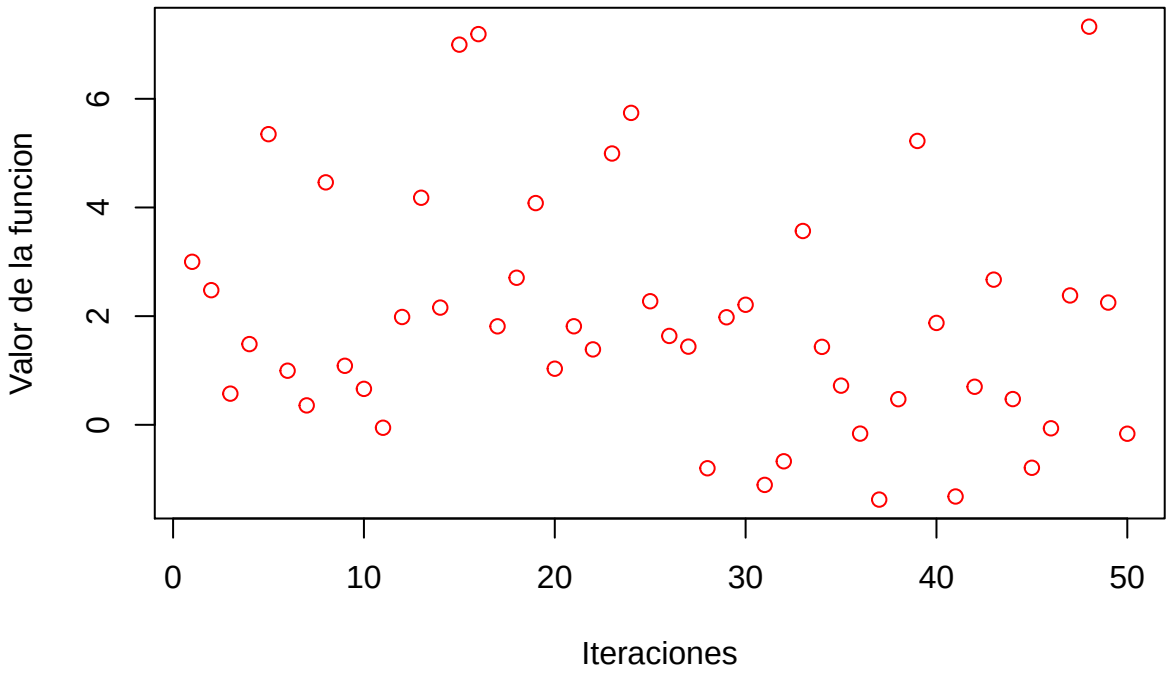

```
## Realizamos las pruebas pertinentes para comparar con los
## resultados anteriores
resultados1b2 = GradienteDescendenteB(c(1,1),f,GradienteDeF,0.1, 50)
print(sprintf("El algoritmo termina tras %i iteraciones",
              resultados1b2$iteraciones))
```

```
## [1] "El algoritmo termina tras 50 iteraciones"
```

```
print(sprintf("Los valores obtenidos son x = %f e y =%f",
resultados1b2$valoresFinales[1],resultados1b2$valoresFinales[2]))
```

```
## [1] "Los valores obtenidos son x = 3.459258 e y =1.953723"
```

```
plot(1:length(resultados1b2$puntos), resultados1b2$puntos,
     xlab="Iteraciones",ylab="Valor de la funcion", col = 2)
```



####Conclusiones sobre los dos experimentos anteriores

Segun el libro "Learning from Data", concretamente en el apartado donde se explica el Gradiente Descendente se hace hincapié en cual es valor ideal que debe tomar la tasa de aprendizaje η. Vamos a comparar los resultados de nuestros experimentos, el primero con un $\eta = 0{,}01$ y el segundo con $\eta = 0{,}1$.

En el primer experimento la funcion $f(x,y) = (x-2)^2 + 2(y-2)^2 + 2sin(2*\pi x)sin(2\pi y)$ consigue llegar a un minimo local en apenas 7 iteraciones, y es aquí donde cobra especial importancia el valor que se le asigne a la tasa de aprendizaje. Dado que el punto de inicio de ambos experimentos es el mismo (x=1, y=1) podemos asegurar que el minimo local esta cerca de nuestro punto de partida y por lo tanto la tasa de aprendizaje, que es la que controla la velocidad de avance de nuestro gradiente, ha de tener un valor lo suficientemente pequeño como para evitar que nuestro gradiente "rebote" por la función y no consiga aproximarse al minimo local tanto como debería.

Sabiendo esto, la comparación entre ambos experimentos resulta trivial. El primero, al tener una tasa de aprendizaje muy pequeña, consigue a través de un avance reducido, alcanzar el minimo local rapidamente. Mientras tanto, el valor de la tasa de aprendizaje en el segundo experimento produce un comportamiento contrario al deseado, avanzando con pasos mucho mayores y sin llegar a converger en un optimo local que se encuentra muy cerca del punto de inicio.

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

```
## Establecemos los 4 puntos de inicio
puntoDeInicio1 = c(2.1,2.1)
puntoDeInicio2 = c(3,3)
puntoDeInicio3 = c(1.5,1.5)
puntoDeInicio4 = c(1,1)

## Calculamos los resultados con los 4 puntos de inicio
resultadosPIni1 = GradienteDescendenteB(puntoDeInicio1,f,GradienteDeF,0.1, 50)
resultadosPIni2 = GradienteDescendenteB(puntoDeInicio2,f,GradienteDeF,0.1, 50)
resultadosPIni3 = GradienteDescendenteB(puntoDeInicio3,f,GradienteDeF,0.1, 50)
resultadosPIni4 = GradienteDescendenteB(puntoDeInicio4,f,GradienteDeF,0.1, 50)
```

```
## Calculamos los valores minimos de la funcion a partir de los 4 puntos
## de inicio
minimoPunto1 = min(resultadosPIni1$puntos)
minimoPunto2 = min(resultadosPIni2$puntos)
minimoPunto3 = min(resultadosPIni3$puntos)
minimoPunto4 = min(resultadosPIni4$puntos)

## Ahora calculamos, para cada valor minimo que se ha obtenido segun
## cada punto de inicio, los valores (x,y) que evaluados por la funcion
## f(x,y) nos dan esos mismos valores minimos
pesosPunto1 = which(resultadosPIni1$puntos == minimoPunto1)
pesosPunto2 = which(resultadosPIni2$puntos == minimoPunto2)
pesosPunto3 = which(resultadosPIni3$puntos == minimoPunto3)
pesosPunto4 = which(resultadosPIni4$puntos == minimoPunto4)
```

Una vez obtenidos todos los datos les damos un formato para poder expresarlo como una tabla. Utilizamos una funcion del paquete Knitr que se llama kable que permite crear tablas en Rmarkdown.

```
## Primero le damos un formato a nuestra tabla: los puntos de inicio
tablaResultadosEj1b = matrix(c(puntoDeInicio1[],minimoPunto1,
                               resultadosPIni1$valoresDeVariables[pesosPunto1,]),
                             1,5,byrow=T)

tablaResultadosEj1b = rbind(tablaResultadosEj1b,
c(puntoDeInicio2[],minimoPunto2,resultadosPIni2$valoresDeVariables[pesosPunto2,]))

tablaResultadosEj1b = rbind(tablaResultadosEj1b,
c(puntoDeInicio3[],minimoPunto3,resultadosPIni2$valoresDeVariables[pesosPunto3,]))

tablaResultadosEj1b = rbind(tablaResultadosEj1b,
c(puntoDeInicio4[],minimoPunto4, resultadosPIni2$valoresDeVariables[pesosPunto4,]))

colnames(tablaResultadosEj1b)= c("Punto Inicio (x)","Punto Inicio (y)","Valor minimo",
                "Valor de variable (x)","Valor de variable (y)")

library(knitr)
kable( tablaResultadosEj1b , caption = "Resultados Ejercicio 1b",
       align = c('c', 'c', 'c', 'c', 'c'),
       format.args = list( decimal.mark = ".")
)
```

Table 1: Resultados Ejercicio 1b

Punto Inicio (x)	Punto Inicio (y)	Valor minimo	Valor de variable (x)	Valor de variable (y)
2.1	2.1	-1.313997	2.7398091	2.218414
3.0	3.0	-0.083908	2.3087319	1.984644
1.5	1.5	-1.684806	0.3938004	1.356668
1.0	1.0	-1.375917	-0.0844732	2.030495

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

Volviendo al libro "Learning from Data" mi conclusión coincide con el problema que se plantea para un

determinado algoritmo como es el del Gradiente Descendente: cuándo debemos terminar el algoritmo. Para encontrar el minimo global de una función nuestro algoritmo deberá ser capaz de atravesar optimos locales para continuar con su busqueda, por tanto la condicion de parada del algoritmo es crucial en la obtencion de una solucion global optima. Por ende es muy importante establecer criterios de parada acordes con nuestro problema y posibles soluciones pueden ser combinar estos criterios entre si. Un numero de iteraciones lo suficientemente grande como para no restringir la busqueda del algoritmo, una variacion de error de una solucion a otra que sea minima (condicion que se cumple en un optimo global) y por ultimo una cota de error muy pequeña tambien. Por tanto la verdadera dificultad de encontrar un optimo local radica en la especificacion correcta de los criterios de parada de nuestro algoritmo, ya que es una decision que condicionará totalmente la calidad de la solucion obtenida.

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 $X = [0, 2]Ö[0, 2]$ con probabilidad uniforme de elegir cada x ∈ 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.

El primer paso para desarrollar este ejercicio consiste en recuperar algunas de las funciones que utilizamos para la practica 1, funciones que nos ayudarán a elegir la linea que actuará como frontera y tambien a generar el conjunto de datos sobre el que aplicaremos nuestro algoritmo de Regresión Logística. Estas funciones se pueden ver a continuacion.

Funciones auxiliares

simula_unif(N,dim,rango), que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango.

```
## Funcion simula_unif(N,dim,rango) : por defecto genera 2 puntos entre
## el intervalo [0,1] de 2 dimensiones
simula_unif = function (N=2,dim=2, rango = c(0,1)){

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

simula_recta(intervalo) , que simula de forma aleatoria los parámetros, v = (a, b) de una recta, y = ax + b, que corta al cuadrado [-50, 50]x[-50, 50].

```
##  Función simula_recta(intervalo) una funcion que calcula los parámetros
##  (Para calcular la recta se simulan las coordenadas de 2 ptos dentro del
##  de una recta aleatoria, y = ax + b, que corte al cuadrado
## [-50,50]x[-50,50] 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
}
```

generarEtiquetas(muestra,rectaSimulada), evalua los puntos de la muestra de datos y los evalua teniendo en cuenta la recta simulada, asignando la etiqueta correspondiente en funcion del signo de la evaluacion en esa recta.

```
## Funcion generarEtiquetas(muestra,rectaSimulada)
## para generar las etiqueta teniendo en cuenta los valores de una muestra y
## una recta simulada siguiendo la función f(x,y)= y - ax - b

generarEtiquetas=function(muestra,rectaSimulada=simula_recta(c(-50,50)) ){

  #Comprobamos si tenemos datos de muestra
  if(missing(muestra)) stop("Error: no ha proporcionado un conjunto de muestras")

  #Ahora para cada componente de la muestra 2D comprobamos su signo segun
  #la recta que le hemos pasado por parametros
  etiquetas= sign(muestra[,2] - (rectaSimulada[1]*muestra[,1]) -rectaSimulada[2])
  etiquetas

}
```

Calcular Coeficientes de la recta a partir de un vector de pesos

```
## Funcion calcularCoeficientes(vini)que calcula los coeficientes
## de la recta (pendiente y punto de corte)
calcularCoeficientes = function(vini){

  #Calculamos la pendiente y el punto de corte según la formula
  # W1*x1 + W2*x2 -bias = 0
  puntoCorte = -vini[3]/vini[2]
  pendiente = -vini[1]/vini[2]
```

```
  coeficientes=c(puntoCorte,pendiente)
  coeficientes
}
```

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

1) Inicializar el vector de pesos con valores 0.

2) Parar el algoritmo cuando ||w(t-1) - w(t)|| < 0,01, donde w(t) denota el vector de pesos al final de la epoca t. Una epoca es un pase completo a traves de los N datos.

3) Aplicar una permutación aleatoria, 1,2,...N , en el orden de los datos antes de usarlos en cada época del algoritmo.

4) Usar una tasa de aprendizaje de η = 0,01

Vamos a utilizar la teoría consultada en el libro "Learning from Data" para llevar a cabo la realizacion de este algoritmo. Primero necesitamos de una funcion que nos calcule el gradiente de la regresion logistica y posteriormente usarla en el algoritmo. La expresion del Gradiente Descendente Estocástico (SGD) que usamos para la Regresión Logística es la siguiente: $g_t = \frac{-y_n x_n}{1+e^{y_n w(t)^T x_n}}$

```
## Funcion SGD(datoRL, evaluacionDatoRL,W)
## Calcula la expresion del gradiente que se utiliza en la
## Regresion Logistica: recibe un dato de entrada, la evaluacion
## para ese dato de entrada y el vector de pesos W_t+1
SGD=function(datoRL, evaluacionDatoRL,W){

  #Calculamos la expresion total del gradiente
  expresionGradiente = -(evaluacionDatoRL*datoRL)/(1+exp(crossprod((evaluacionDatoRL*t(W)),datoRL)))
  expresionGradiente
}
```

Una vez tenemos la funcion que computa el gradiente para la Regresion Logistica solo queda implementar el algoritmo LGRA con Gradiente Descendente Estocastico

```
## Algoritmo LGRA_SGD(Wini, datosRL, evaluacionesRL, tasaAprendizaje,
##                    cotaErrorMin)
## Algoritmo de Regresion Logistica con Gradiente Descendente Estocastico que
## partiendo de unos pesos iniciales Wini, un conjunto de datos datosRL
## y las evaluaciones de los mismos segun una recta generada en
## X= [0,2]x[0,2], calcula el minimo local siguiendo una tasaDeAprendizaje
## fijada cuando llegue a una cota minima de Error.
LGRA_SGD = function(W_antes, datosRL, evaluacionesRL, tasaAprendizaje,cotaErrorMin){

  #Declaramos las variables que vamos a utilizar en nuestro algoritmo
  #y calculamos un primer valor de error
  epoca = 0
  W_t = W_antes
  noSuperadaCotaError = F

  #Mientras que  error este por encima de la cota minima de error
  while(!noSuperadaCotaError){
```

```
    #Iniciamos una epoca y por tanto aplicamos una permutacion aleatoria
    #sobre el conjunto de datos antes de empezar a calcular el gradiente
    permutacion = 1:length(evaluacionesRL)
    permutacion = sample(permutacion)

    #Comenzamos la epoca y actualizamos los pesos
    for(i in permutacion){

      #Calculamos el vector gradiente con un dato seleccionado
      #de forma aleatoria
      gradiente = SGD(datosRL[i,], evaluacionesRL[i],W_t)
      vectorGradiente = -gradiente

      #Actualizamos los valores de los pesos
      W_t = W_t + (tasaAprendizaje*vectorGradiente)

    }

    #Finalizamos una época y realizamos la comprobacion
    #en cuanto a la cota de error
    epoca = epoca + 1
    vectorDiferencia = (W_antes - W_t)

    if(sqrt(sum((vectorDiferencia)^2)) < cotaErrorMin)
      noSuperadaCotaError = T

    #Actualizamos los pesos en cualquier caso
    W_antes = W_t
  }

  #Devolvemos el numero de epocas y los valores de W finales
  list(epoca = epoca, valoresFinales = W_t)

}
```

Una vez recuperadas las funciones necesarias para comenzar a generar nuestro conjunto de datos procedemos a generar dos valores aleatorios en $X = [0, 2]x[0, 2]$ que formaran nuestra recta. Posteriormente generaremos 100 valores aleatorios en X y los clasificaremos utilizando la recta que acabamos de obtener.

```
set.seed(24)
## El primer paso es generar los dos valores que formaran nuestra recta
## Esto se realiza internamente en la funcion simula_recta, por lo que
## es inmediato obtener la recta que servirá para clasificar nuestros datos
rectaRegresionLogistica = simula_recta(c(0,2))

## Generamos nuestro conjunto de 100 datos en el intervalo [0,2]x[0,2]
datosRegresionLogistica = simula_unif(100,2,c(0,2))
## Evaluamos las respuestas {yn} con respecto de la frontera definida
evaluacionesDatosRL = generarEtiquetas(datosRegresionLogistica,rectaRegresionLogistica)
datosRegresionLogistica = cbind(datosRegresionLogistica,1)
```

El ultimo paso consiste en realizar las pruebas pertinentes con el conjunto de datos que acabamos de generar. El objetivo es aprender unos pesos con los que podamos calcular el Eout de una muestra distinta.

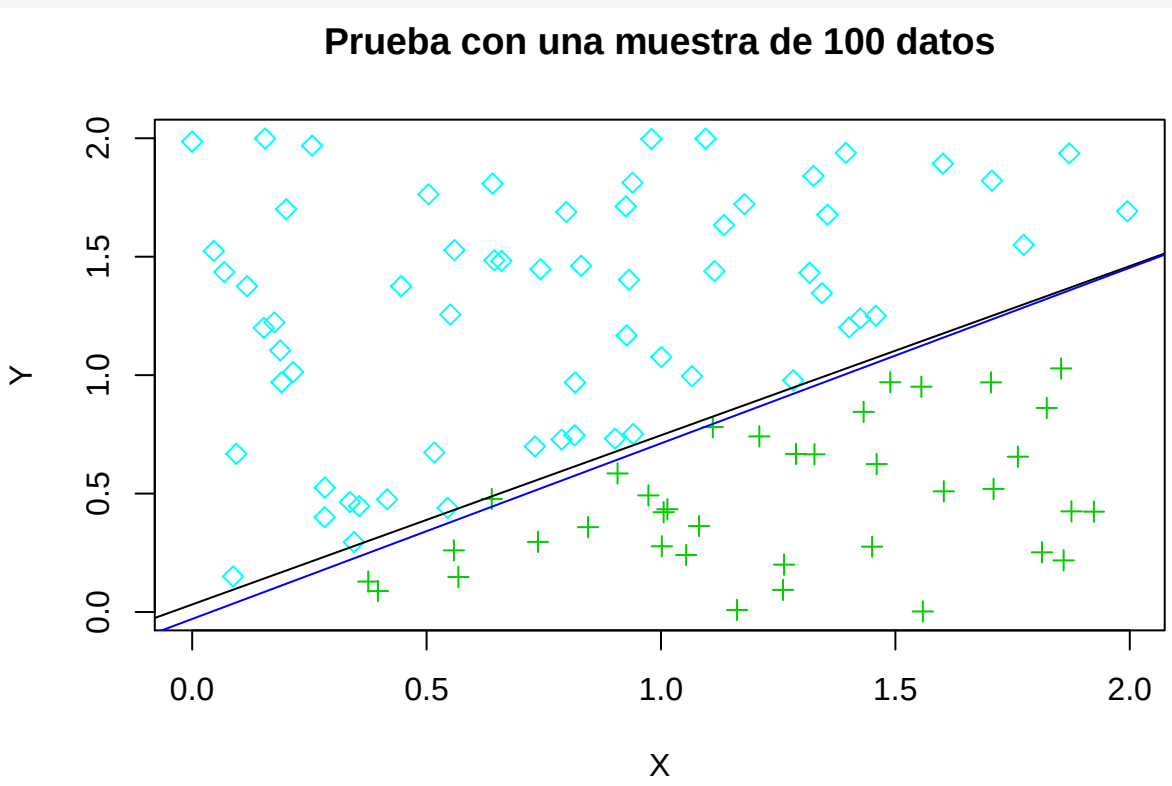
```
## Primera prueba con 100 datos generados anteriormente
## para obtener unos pesos
Wini = matrix(c(0,0,0),1,3)
tasaAprendizaje = 0.01
cotaErrorMin = 0.01
## Ejecutamos la Regresion Logistica
resultadosRL = LGRA_SGD(Wini,datosRegresionLogistica, evaluacionesDatosRL,
                        tasaAprendizaje, cotaErrorMin)
print(sprintf("El algoritmo acaba tras %i iteraciones",resultadosRL$epoca))
```

```
## [1] "El algoritmo acaba tras 420 iteraciones"
```

```
print(sprintf("Los pesos finales son %f, %f y %f",resultadosRL$valoresFinales[1],
              resultadosRL$valoresFinales[2],resultadosRL$valoresFinales[3]))
```

```
## [1] "Los pesos finales son -6.224894, 8.396964 y 0.246382"
```

```
## Pintamos los resultados para que sean mas visibles
coefsDatosTrain = calcularCoeficientes(resultadosRL$valoresFinales)
plot(datosRegresionLogistica, pch = evaluacionesDatosRL+4,
     col = evaluacionesDatosRL+4,xlab= "X", ylab = "Y",
     main = "Prueba con una muestra de 100 datos",
     sub = "En negro: recta original. En azul: recta calculada a partir de los pesos")
abline(rectaRegresionLogistica[2],rectaRegresionLogistica[1])
abline(coefsDatosTrain[1],coefsDatosTrain[2], col = 4)
```



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).

Para realizar este apartado tenemos que generar una muestra nueva de mas de 999 datos, generar sus clasificaciones y calcular el Eout con la formula: $E_{in} = \frac{1}{N}\sum_{n=1}^{N} \ln(1 + e^{-y_n w^T x_n})$

```
## Funcion Ein(datosRL, evalucionesRL, W) que permite calcular el Error
## de una muestra de datos que tienen unas evaluaciones teniendo en
## en cuenta los pesos W que se han calculado con los datos de Train
Ein=function(datos, evaluacionesDatos,W){

  #Calculamos la dimension de datos
  N = dim(datos)[1]
  Ein = 0
  #Para todos los datos
  for(i in 1:N){

    #Calculamos el neperiano de la expresion que me define el error
    errorParcial = log(1 + exp((crossprod((-evaluacionesDatos[i]*t(W)),datos[i,]))))
    Ein = Ein + errorParcial

  }

  Ein = (1/N)*Ein
}
```

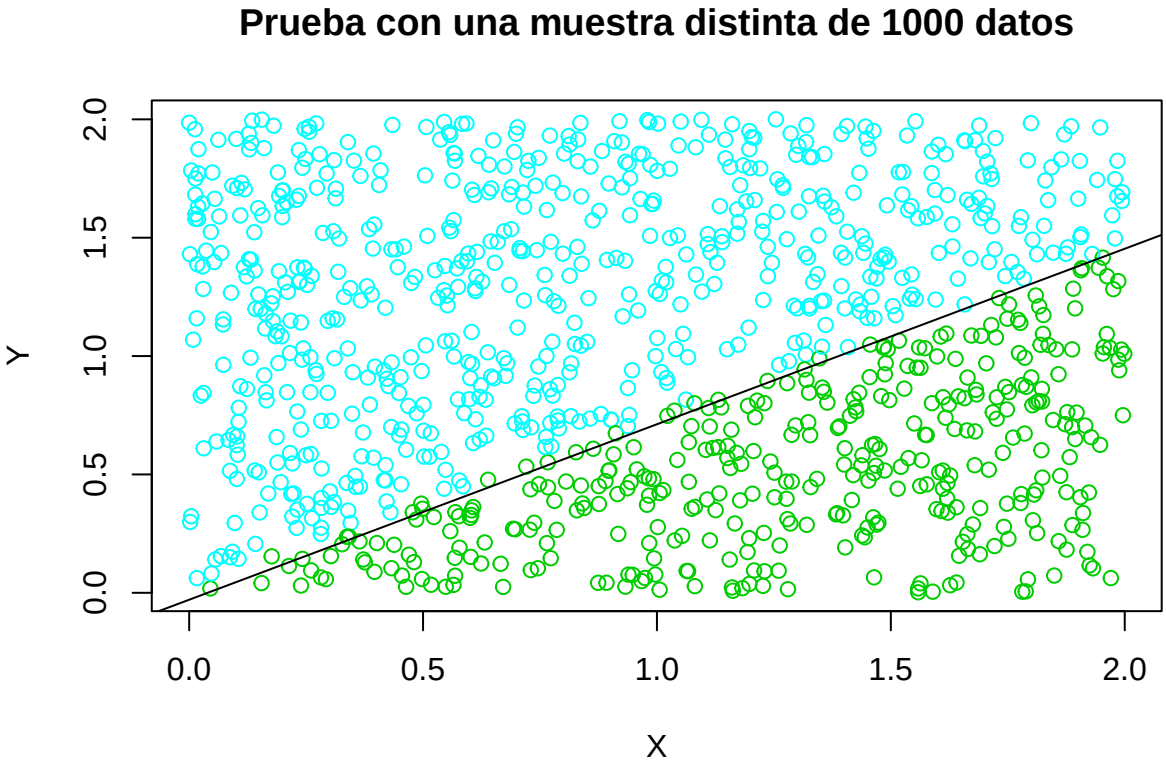
Una vez tenemos la funcion que nos calcula el Ein, vamos a calcular el Eout de una muestra de datos distinta para comprobar que tan bueno es nuestro clasificador.

```
set.seed(24)
## Generar dos nuevos datos: han de tener mas de 999 ejemplos
datos_Eout = simula_unif(1000,2,c(0,2))
## Evaluamos las respuestas {yn} con respecto de la frontera definida
## en el apartado anterior (tenemos en cuenta que las que sean 0, por tanto
## su evaluacion es 0, serán etiquetados como positivas)
evaluacionesDatos_Eout = generarEtiquetas(datos_Eout,rectaRegresionLogistica)
evaluacionesDatos_Eout[evaluacionesDatos_Eout==0]=1

## Calculamos el Eout de la nueva muestra de datos
datos_Eout = cbind(datos_Eout,1)
Eout = Ein(datos_Eout,evaluacionesDatos_Eout,resultadosRL$valoresFinales)
print(sprintf("El Eout estimado para una muestra de %i datos es de %f",
              dim(datos_Eout)[1],Eout))
```

```
## [1] "El Eout estimado para una muestra de 1000 datos es de 0.101150"
```

```
## Pintamos los resultados para que sean mas visibles
rectaEout= calcularCoeficientes(resultadosRL$valoresFinales)
plot(datos_Eout,col = evaluacionesDatos_Eout + 4, xlab= "X", ylab = "Y",
     main = "Prueba con una muestra distinta de 1000 datos",
     sub = "En azul: recta calculada a partir de los pesos anteriormente calculados")
abline(rectaEout[1],rectaEout[2])
```



Interpretar los resultados obtenidos es trivial si tenemos en cuenta que el Eout se ha estimado en funcion de las distancias de los ejemplos que se encuentran mal clasificados. El valor de Eout indica que en promedio, la distancia de aquellos ejemplos que no se encuentran bien clasificados con la recta y por tanto con la clasificacion real, es de 0.1, y por tanto se puede considerar como un buen resultado, dado que graficamente se puede comprobar que nuestra recta de Regresion Logistica con Gradiente Descendente Estocastico es capaz de clasificar muy bien un gran porcentaje de los datos.

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

La resolucion de este apartao es sencilla si consideramos las funciones con las que hemos trabajado durante la practica 1. Vamos a recuperar por tanto la funcion que me permitia leer un fichero y estructurarlo para poder utilizarlo como conjunto de datos. La funcion se modificará internamente para leer unicamente las instancias de 4s y 8s del conjunto de datos original, ya sea del zip.train o del zip.test.

```
## En una funcion que llamaremos leerDatosYEstructurar(fichero)
## englobaremos el funcionamiento de las lineas de codigo proporcionadas
## para utilizarla sobre distintos ficheros de datos
leerDatosYEstructurar=function(fichero){
```

```
  # Lectura de fichero datos.train (2780 muestras con 256 valores en escala de gris)
  # o datos.test
  datos = read.table(fichero, quote="\"", comment.char="", stringsAsFactors=FALSE)

  ## Nos limitamos a seleccionar las instancias de los numeros
  ## 4 y 8 (aquellos en los que la columna 1 - V1 tengan un 4 u 8)
  digitos_48 = datos[datos$V1==4 | datos$V1==8,]

  digitos = digitos_48[,1]  # nombre de las etiquetas (si es 4 u 8)
  ndigitos = nrow(digitos_48) # numero de muestras de 4 y 8

  ## Se retira la clase y se monta una matriz 3D: 432(51 para el train)*16*16
  ## Para cada instancia de 1 o 5, dimesionamos los 256 valores de
  ## escala de gris en una matriz 16x16
  grises = array(unlist(subset(digitos_48,select=-V1)),c(ndigitos,16,16))

  ## Eliminamos los dataframes que ya no vamos a utilizar
  rm(datos)
  rm(digitos_48)

  list(datos=grises,labels=digitos)
}
```

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

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

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

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?

Resolucion Apartado b)

Volvemos a recuperar algunas de las funciones utilizadas en la practica anterior, tales como:

FSimetria

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

Perceptron Learning Algorithm: este algoritmo ha sido modificado con respecto del utilizado en la practica anterior. Ahora recorremos todos los datos de forma aleatoria y cuando encontremos el primer ejemplo cuya etiqueta real no coincida con la calculada por el algoritmo,

los pesos se modificaran en funcion de ese unico ejemplo y el algoritmo PLA terminará devolviendo esa solucion, solucion que el algoritmo Pocket evaluara para comprobar si mejora con la solucion inicial en funcion de Ein.

```
ajusta_PLA = function(datos, label,max_iter=300, vini=c(0,0,0)){

  #Comprobamos si tenemos datos
  if(missing(datos)) stop("Error: no ha proporcionado el conjunto de datos")

  #Comprobamos si tenemos etiquetas
  if(missing(label)) stop("Error: no ha proporcionado las etiquetas")

  #Booleano para controlar si en alguna de las pasadas
  #completas hemos modificado los pesos
  modificacionPesos=TRUE

  #Mientras no superemos el numero de iteraciones
  iteraciones = 0

  while(iteraciones < max_iter & modificacionPesos ){

    modificacionPesos=FALSE
    vectorDeErroneos =numeric(0)

    #Recorremos la matriz de datos, accediendo a cada
    #fila (vector de características)
    for(i in sample(1:length(datos[,1]))){

      #Calculamos el producto escalar
      valor = crossprod(datos[i,],vini)

      #Solo modificaremos el vector de pesos cuando
      # los signos sean distintos (mal clasificado)
      # y lo hacemos con el primero que encontremos
      #dada que la componente de seleccion aleatoria
      #de los datos permite que todos puedan ser elegidos
      if(sign(valor) != label[i]){
        vini = vini + label[i]*datos[i,]
        break
      }


    }
    iteraciones=iteraciones+1

  }

  #Damos como salida los pesos para luego calcular
  # los coeficientes de la recta
  list(pesos = vini,iters = iteraciones)
}
```

Error dentro de la muestra (Ein): calculado como el cociente entre el numero de etiquetas bien clasificadas y el numero de etiquetas totales asociadas a un determinado conjunto de datos

```
## Creamos una funcion para obtener el error en funcion del numero de muestras
## mal clasificadas que se llamara calculaErrorMalClasificadas(datos,label,pesos)
calcularErrorMalClasificadas = function(datos,label,pesos){

  # Calculamos el producto de los datos por los pesos
  malClasificadas = apply(datos,1,crossprod,pesos)

  #Obtenemos cuantas etiquetas han sido mal clasificadas
  numMalClasificadas = sum((sign(malClasificadas)!=label))

  #Obtenemos el error
  error = numMalClasificadas/(length(label))

}
```

Algoritmo de Regresion Lineal: calcula la Descomposicion en Valores Singulares (SVD) de un conjunto de datos X calcular los pesos como la pseudoinversa de X por el conjunto de etiquetas.

```
## Funcion Regress_Lin(datos, label)
Regress_Lin = function(datos,label){
  # Calculamos la descomposicion en valores singurales
  # y recuperamos sus valores
  descompValSing = svd(datos)
  D = descompValSing$d
  V = descompValSing$v

  #Calculamos la pseudoinversa de D
  pInversaD = diag(1/D)

  #Calculamos (X'X)^-1 = V*D^2*V'
  traspuestaXInversa = V%*%(pInversaD^2) %*%t(V)

  # Finalmente obtenemos la pseudoinversa
  pseudoInversa = traspuestaXInversa%*%(t(datos))

  # Nuestro vector de pesos final será PseudoInvX*label
  W = (pseudoInversa%*%label)

}
```

PLA Pocket: algoritmo que guarda el mejor vector solución en la iteracion t para el algoritmo PLA. El funcionamiento del algoritmo es el siguiente:

1. Inicializar el vector de pesos del Pocket (Wˆ) al valor con el que se inicializa en la iteracion 0 para el PLA

2. for i=1 to T do

2.1 Ejecutar UNA iteracion de PLA para obtener un vector de pesos w(t+1)

2.2 Calculamos el Ein para ese vector de pesos: cantidad de etiquetas mal clasificadas

2.3 Si en terminos de Ein, w(t+1) es mejor que Wˆ, actualizamos el valor de W

3. Devolver Wˆ

```
PLA_Pocket = function(datos, label, pesosRegressLin, max_iter){

  #Inicializamos el vector de pesos del Pocket y los del PLA
  W_Pocket = pesosRegressLin
  W_PLA = W_Pocket
  #Evaluamos la calidad de los pesos del pocket (los que
  #nos ofrece la Regresion Lineal)
  EinPocket = calcularErrorMalClasificadas(datos,label,W_Pocket)

  for(i in 1:max_iter){

    #Ejecutamos UNA iteracion del PLA: este no llega a converger
    #aunque ya parte de una solucion de cierta calidad dado que los
    #pesos son los que me devuelve la Regresion Lineal (sin haber
    #realizado una primera modificacion, garantizando que la solucion
    #no empeore)
    resultadosPLA = ajusta_PLA(datos,label,1,W_PLA)
    W_PLA = resultadosPLA$pesos

    #Calculamos los Ein en funcion de los pesos del PLA anterior
    EinPLA = calcularErrorMalClasificadas(datos,label,W_PLA)

    #Si el nuevo vector de Pesos calculado por la ejecucion del PLA
    #es mejor que el actual (W_Pocket) en terminos de Ein, actualizamos
    #los pesos del Pocket
    if(EinPLA<EinPocket){

      print(sprintf("Mejora Realizada"))
      EinPocket = EinPLA
      W_Pocket = W_PLA

    }

  }

  #Devolvemos los pesos del Pocket, que son los pesos de la regresion lineal
  #pero a los que se les ha aplicado una mejora sustancial
  W_Pocket
}
```

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

Una vez se han definido todas las funciones necesarias para la resolucion vamos a proceder con la lectura de los ficheros de train y test, la estructuracion de los conjuntos de datos para posteriormente aplicarle la Regresion Lineal con la mejora del PLA Pocket. Finalmente generaremos los graficos de los conjuntos train y test con la funcion estimada

```
## Leemos los datos de train y test
datosTrain_48 = leerDatosYEstructurar("datos/zip.train")
datosTest_48 =leerDatosYEstructurar("datos/zip.test")

## Mostramos los 4 primeros ejemplos del train
par(mfrow=c(2,2))
# for(i in 1:4){
#   imagen = datosTrain_48$datos[i,,16:1] # se rota para verlo bien
#   image(z=imagen)
# }

## Calculamos el grado de simetria y el valor de intensidad media
## para el conjunto de train y hacemos lo propio para el de test
gradoSimetriaTrain = apply(datosTrain_48$datos,1,fsimetria)
intensidadMediaTrain = apply(datosTrain_48$datos,1,mean)

gradoSimetriaTest = apply(datosTest_48$datos,1,fsimetria)
intensidadMediaTest = apply(datosTest_48$datos,1,mean)

## Modificamos las etiquetas de tal forma que los 4s se queden con etiqueta
## 1 y los 8s cambiarán a -1. Lo hacemos para ambos conjuntos de datos
etiquetasTrain=datosTrain_48$labels
etiquetasTest=datosTest_48$labels

etiquetasTrain[etiquetasTrain==4]=1
etiquetasTrain[etiquetasTrain==8]=-1

etiquetasTest[etiquetasTest==4]=1
etiquetasTest[etiquetasTest==8]=-1

## Creacion de la matriz de datos train a partir de la intensidad promedio y el valor medio
datosTrain=matrix(c(intensidadMediaTrain,gradoSimetriaTrain),length(etiquetasTrain),2)
# Anidimos un 1 al final de nuestro vector de caracteristicas
datosTrain=cbind(datosTrain,1)

## Creacion de la matriz de datos test a partir de la intensidad promedio y el valor medio
datosTest=matrix(c(intensidadMediaTest,gradoSimetriaTest),length(etiquetasTest),2)
# Anidimos un 1 al final de nuestro vector de caracteristicas
datosTest=cbind(datosTest,1)
```

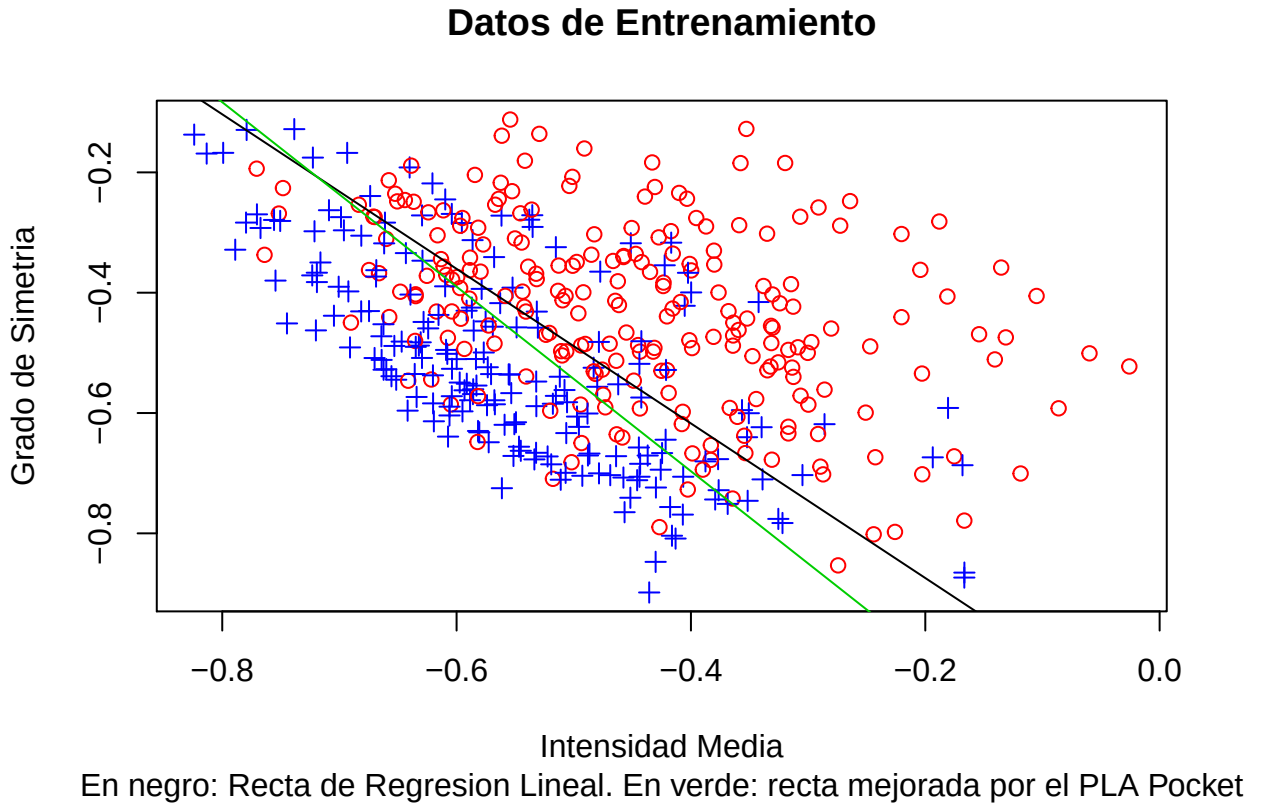
```
set.seed(24)
## Ejecutamos la Regresion Lineal para obtener unos pesos iniciales
## que serviran como punto de partida para el PLA Pocket
pesosRL_Train = Regress_Lin(datosTrain,etiquetasTrain)
pesosRL_Train = c(pesosRL_Train[1],pesosRL_Train[2], pesosRL_Train[3])
## Ejecutamos el PLA Pocket para mejorarlo
pesosPLA_Pocket_Train = PLA_Pocket(datosTrain,etiquetasTrain,pesosRL_Train,1000)
```

```
## [1] "Mejora Realizada"
## [1] "Mejora Realizada"
## [1] "Mejora Realizada"
```

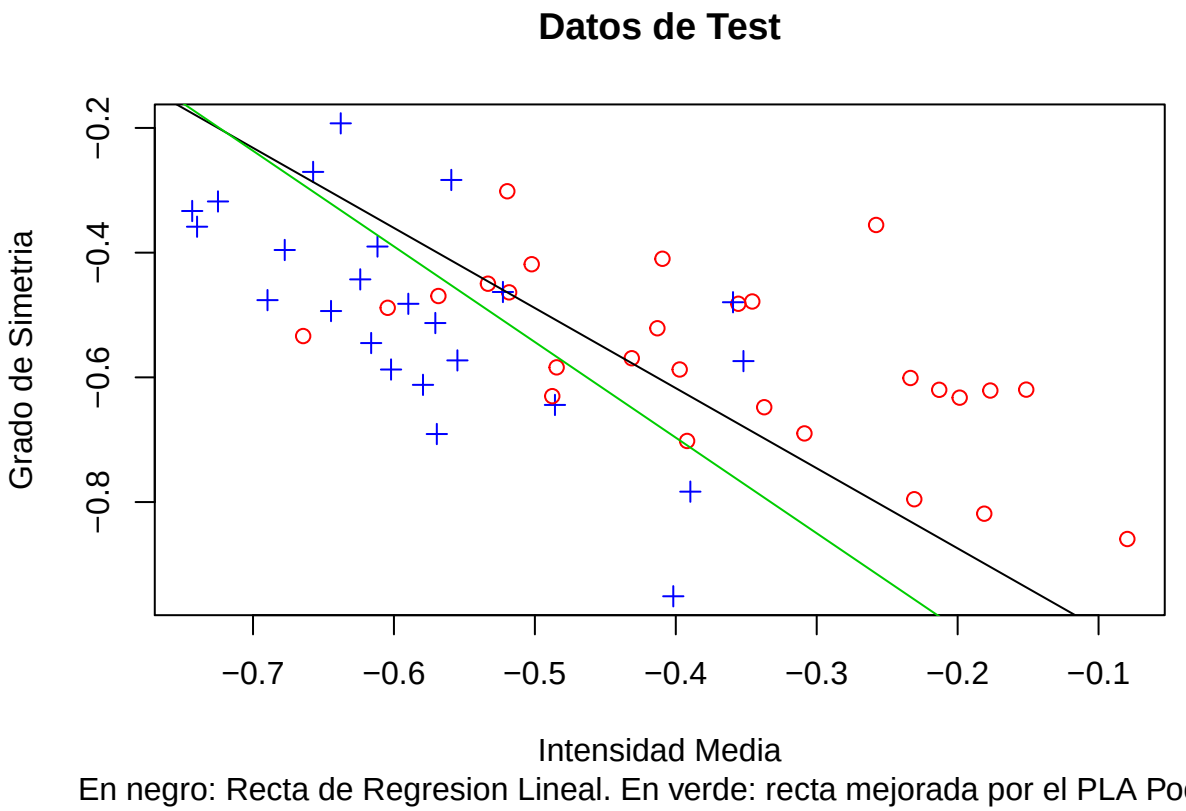
```
## Calculamos los coeficientes de la recta con los 2 pesos obtenidos
## tras ejecutar la Regresion Lineal y tras aplicar la mejora con
```

```
## PLA Pocket
coeficientesRegressLin_Train = calcularCoeficientes(pesosRL_Train)
coeficientesPLA_Pocket_Train = calcularCoeficientes(pesosPLA_Pocket_Train)

## Pintamos los resultados del conjunto de datos de entrenamiento con
## las dos rectas calculadas a partir de los pesos que proporcionaban
## tanto la Regresion Lineal como la mejora PLA Pocket
plot(datosTrain, pch=etiquetasTrain + 2, col = etiquetasTrain + 3, xlab="Intensidad Media",
     ylab = "Grado de Simetria", main="Datos de Entrenamiento",
     sub="En negro: Recta de Regresion Lineal. En verde: recta mejorada por el PLA Pocket")
abline(coeficientesRegressLin_Train[1],coeficientesRegressLin_Train[2])
abline(coeficientesPLA_Pocket_Train[1],coeficientesPLA_Pocket_Train[2],col=3)
```




```
## Pintamos los resultados del conjunto de datos de test con
## las dos rectas calculadas a partir de los pesos que proporcionaban
## tanto la Regresion Lineal como la mejora PLA Pocket
plot(datosTest, pch=etiquetasTest + 2, col = etiquetasTest + 3, xlab="Intensidad Media",
     ylab = "Grado de Simetria", main="Datos de Test",
     sub="En negro: Recta de Regresion Lineal. En verde: recta mejorada por el PLA Pocket")
abline(coeficientesRegressLin_Train[1],coeficientesRegressLin_Train[2])
abline(coeficientesPLA_Pocket_Train[1],coeficientesPLA_Pocket_Train[2],col=3)
```



En negro: Recta de Regresion Lineal. En verde: recta mejorada por el PLA Pocket

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

Para calcular el Ein y el Etest simplemente nos valemos de la funcion recuperada anteriormente que me calcula el error segun el numero de etiquetas mal clasificadas. Utilizamos los pesos optimizados por el algoritmo PLA Pocket.

```
## Calculamos el Ein y el ETest en funcion de la cantidad de datos mal clasificadas
## Para hacer una comparativa y ver si los resultados que ofrece el Pocket realmente mejoran
## el porcentaje de clasificacion, vamos a realizar una comparativa entre los pesos
## de la Regresion Lineal y los pesos del Pocket
Ein_48_RegressLin = calcularErrorMalClasificadas(datosTrain,etiquetasTrain,pesosRL_Train)
ETest_48_RegressLin = calcularErrorMalClasificadas(datosTest,etiquetasTest,pesosRL_Train)
print(sprintf("El valor del Ein (en porcentaje) es %f  y el del ETest (en porcentaje) es %f ",
              Ein_48_RegressLin*100 , ETest_48_RegressLin*100))
```

```
## [1] "El valor del Ein (en porcentaje) es 24.768519  y el del ETest (en porcentaje) es 23.529412 "
```

```
Ein_48 = calcularErrorMalClasificadas(datosTrain,etiquetasTrain,pesosPLA_Pocket_Train)
ETest_48 = calcularErrorMalClasificadas(datosTest,etiquetasTest,pesosPLA_Pocket_Train)
print(sprintf("El valor del Ein (en porcentaje) es %f  y el del ETest (en porcentaje) es %f ",
              Ein_48*100 , ETest_48*100))
```

```
## [1] "El valor del Ein (en porcentaje) es 22.453704  y el del ETest (en porcentaje) es 21.568627 "
```

Como se puede apreciar se mejora el Ein al utilizar el PLA Pocket durante 1000 iteraciones, realizando 5 modificaciones en los pesos, modificaciones que repercuten notablemente en la disminucion del Ein en cerca de un 2.3%.

Resolucion subapartado 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?

En las transparencias de la sesion 4 de teoría donde se habla de la cota de generalizacion del Eout en funcion de la d_{vc} se muestra una forma de calcular una cota sobre el verdadero valor de E_{out} teniendo en cuenta el $E_{in}(h)$ que hemos calculado a partir de una muestra de datos. Esta expresion es la que utilizaremos para calcular una cota sobre el verdadero valor de Eout y es la que aparece a continuacion:

$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} * log(\frac{4((2N)^{d_{vc}}+1)}{\delta}}$

Vamos a implementar una funcion que calcule una cota para el E_{out} en funcion de un $E_{in}(h)$ o de un $E_{test}(h)$ Utilizamos $d_{vc} = 3$ dado que es la asociada al PLA y $\delta = 0{,}05$

```
## Funcion calcularCotaEout(cotaError, tamanioDatos,Dvc, delta)
## Calcula una cota del verdadero Eout en funcion de un determinado
## error (que puede ser tanto Ein como Etest), un tamaño de los datos,
## la dimension de Vapnik-Chervonenkis y el valor delta
calcularCotaEout = function(cotaError, tamanioDatos,Dvc=3, delta=0.05){

  N = tamanioDatos
  cotaEout = cotaError + sqrt(8/N * log(4*( ((2*N)^Dvc) +1)/delta))
  cotaEout

}

## Calculamos el tamaño de los datos de Train y Test
N_DatosTrain = dim(datosTrain)[1]
N_DatosTest = dim(datosTest)[1]

## Calculamos las cotas de Eout
Eout_Ein = calcularCotaEout(Ein_48,N_DatosTrain)
Eout_Etest = calcularCotaEout(ETest_48,N_DatosTest)
print(sprintf("Las cotas de Eout calculadas son las siguientes"))
```

```
## [1] "Las cotas de Eout calculadas son las siguientes"
```

```
print(sprintf("Cota de Eout calculada con el Ein = %f", Eout_Ein))
```

```
## [1] "Cota de Eout calculada con el Ein = 0.900401"
```

```
print(sprintf("Cota de Eout calculada con el Etest = %f", Eout_Etest))
```

```
## [1] "Cota de Eout calculada con el Etest = 1.907973"
```

La cota de Eout que hemos calculado con el Etest tiene un valor mayor que 1 (valor que naturalmente truncamos a 1) y esta cota de error no nos proporciona ninguna informacion util ya que por definicion sabemos que el error se mueve entre 0 y 1, por lo que esta cota queda descartada totalmente. Nos queda unicamente la cota calculada con el Ein, valor que se encuentra cercano a 1 (0.900401) pero que si ofrece informacion que nos puede ayudar a tomar decisiones futuras: con un tamaño de la muestra de entrenamiento de N=432 datos y para un nivel de confianza de 1-tolerancia (1-0.05= 95%), el valor del Eout que vamos a obtener para un conjunto de datos que no se haya "visto" antes, nunca será mayor del 89.8%. Aqui es donde surge la idea de que cuantos mas datos utilicemos en nuestro proceso de aprendizaje, menos nos equivocaremos al intentar clasificar datos fuera de la muestra, siempre con un nivel de confianza determinado.

Como añadido a la explicacion de porque hemos escogido esta cota, consideramos la desigualdad de Hoeffding que nos dice que la cota de Eout que calculamos con el Etest ajusta mucho mejor la cota verdadera del error fuera de la muestra **siempre y cuando tengamos un conjunto de datos lo suficientemente**

grande.Esto tiene una consecuencia y es que al separar los datos en Train y Test perderemos precision en la definicion del Ein y por tanto tendremos un mayor error dentro de la muestra. En nuestro caso, elegir la cota calculada en funcion del Ein es lo mas acertado, pero si tuviesemos un conjunto de datos suficientemente grande sería mucho mas conveniente calcular la cota de Eout a partir del error en los datos de test.

Ejercicio 4 : Regularización en la Selección de Modelos

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.

Recuperamos la funcion de simular un conjunto de datos dentro de una distribucion Gaussiana utilizando los valores de media y sigma que nos indican en el ejercicio.

```
## 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_gauss = function(N=2,dim=2,media,sigma){

  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")

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

- 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.

Definiremos un vector de pesos w_f generando $d+1$ valores dentro de una distribucion Gaussiana de media 0 y desviacion tipica 1, sumandole 1 a la ultima componente

- Usando los valores anteriores generamos la etiqueta asociada a cada punto x_n a partir del valor $y_n = w_f^T x_n + \sigma\epsilon_n$, donde ϵ_n es un ruido que sigue también una Gaussiana de media 0 y desviación típica 1 y σ^2 es la varianza del ruido; fijar $\sigma = 0.5$

En este apartado necesitamos de una funcion que me genere las etiquetas siguiendo la formula $y_n = w_f^T x_n + \sigma\epsilon_n$, donde ϵ_n. Posteriormente generaremos las etiquetas siguiendo la funcion determinada

```
## Generar etiquetas
generarEtiquetasRegSelMod = function(W_f, datos, sigma=0.5, ruido){

    ## Creacion de una funcion anidada que calcule una etiqeuta
```

```
    ## Yi a partir de un dato Xi. Esta funcion es crucial para
    ## poder aplicar un "apply" para todos los datos
    funcionWporX = function(Xi){
      Yi = crossprod(t(W_f),Xi) + sigma*ruido
    }

    Yn = apply(datos,1,funcionWporX)
}
```

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.

Para N ∈ {d+10, d+20,...,d+110} calcular los errores de validación cruzada e1,...en y E_{cv}. Repetir el experimento 1000 veces. Anotamos el promedio y la varianza de e1, e2 y Ecv en los experimentos.

El primer paso consiste en implementar el Weight Decay siguiendo la formula del libro "Learning from Data (pagina 133)", esta es: $w_{reg} = (Z^T Z + \lambda I)^{-1} Z^T \mathbf{y}$

```
## Funcion Weigth_Decay(datos, etiquetas, lambda)
## Calcula los pesos W_reg en funcion de unos datos de entrada
## un conjunto de etiquetas y un parametro de regularizacion, todo
## esto siguiendo la formula descrita anteriormente
Weight_Decay = function(datos, etiquetas, lambda){

  #Nombramos las variables tal y como
  #se hacen en la formula
  Z = datos
  y = etiquetas

  #Calculamos la SVD de
  SVD = svd(t(Z)%*%Z + (lambda* diag(1,ncol(Z))))
  D = SVD$d
  V = SVD$v

  #Calculamos la pseudoinversa de D
  pInversaD = diag(1/D)

  #Calculamos (X'X)^-1 = V*D^2*V'
  traspuestaXInversa = V%*%(pInversaD^2) %*%t(V)

  # Finalmente obtenemos la pseudoinversa
  W = pseudoInversa = traspuestaXInversa%*%(t(Z))%*%etiquetas

}
```

El siguiente paso consiste en generar el algoritmo que partiendo de un conjunto de datos de N=113 elementos (siguiendo las especificaciones que se nos plantean en el ejercicio) realice 11 particiones, desde {d+10, d+20,...,d+110}, y aplique la cross validation seleccionando en cada iteracion un subconjunto de datos que serviran para obtener la funcion g^- y despues validar esta funcion para el subconjunto de datos restante (de tamaño N=d+10 siendo d=3). El algoritmo calcula para cada particion un error e_i tomando en cuenta la expresion siguiente: $e(g(x), y) = (g^-(x) - y)^2$ y la media de los errores de Cross Validation calculada como:

$E_{cv} = \frac{1}{N}\sum_{n=1}^{N} e_n$. Tambien se guardarán los errores e_1 y e_2 cada vez que se llame al algoritmo. Pasamos con su implementacion

```
## Algortimo RegSeleccionModelos(datos, etiquetas)
## A partir de un conjunto de datos con unas determinadas etiquetas,
## todos ellos generados a partir de una distribucion Gaussiana
## con la media y varianza indicada en los apartados anteriores,
## y considerando un parametro de regularizacion lambda,
## devuelve los errores e1 y e2, asi como el Ecv que luego utilizaremos
## para predecir el Eout
RegSeleccionModelos_CV = function(W_f,ruido, varRuido){

  ## Generamos el conjunto de datos inicial y su etiquetado
  ## con el mismo W_f en cada experimento
  datos = simula_gauss(113,3,1,c(1,1,1))
  datos = cbind(datos,1)
  etiquetas = generarEtiquetasRegSelMod(W_f,datos,varRuido,ruido)

  #El primer paso consiste en generar un vector de indices
  #que me permitira acceder comodamente a los datos y etiquetas
  indices = 1:nrow(datos)
  indices = sample(indices)
  numeroIteraciones = 11
  E_cv = numeric(0)
  inicio = 1
  fin = 13

  for(i in 1:numeroIteraciones){

    #En la primera particion, el conjunto de validacion
    #sera de 13 elementos y el de train de 100
    #La segunda y demas particiones tendran un conjunto de
    #validacion de 10 elementos y el de train tendra 103
    datos_TrainWD = datos[-indices[inicio:fin],]
    datos_ValWD = datos[indices[inicio:fin],]
    etiquetas_TrainWD = etiquetas[-indices[inicio:fin]]
    etiquetas_ValWD = etiquetas[indices[inicio:fin]]

    #Una vez tenemos las particiones tanto de train como
    #de validacion procedemos al calculo de los pesos W_reg
    #para posteriormente validar esos pesos
    N = nrow(datos_TrainWD)
    lambda = 0.05/N
    W_reg = Weight_Decay(datos_TrainWD,etiquetas_TrainWD, lambda)
    #Procedemos a la validacion de los pesos obtenidos con la particion
    #de validacion
    diferenciaMinCuadrados = function(datos, etiquetas, W_reg){

      #Calculamos las etiquetas con los pesos obtenidos
      #durante el proceso de aprendizaje
      g_minusDatos = apply(datos,1,crossprod, W_reg)

      #Devolvemos el Error cuadratico medio
      mean( (g_minusDatos-etiquetas)*(g_minusDatos-etiquetas))
```

```
    }

    #Calculamos el error y guardamos el ei
    Error = diferenciaMinCuadrados(datos_ValWD, etiquetas_ValWD,W_reg)
    E_cv = c(E_cv,Error)

    inicio = fin+1
    fin = fin+10

  }

  #Devolvemos el E_cv como la media de los errores y los errores e1 y e2
  c(mean(E_cv),E_cv[1],E_cv[2])
}
```

A continuacion realizamos realizaremos 1000 experimentos sobre el conjunto de datos para obtener todos los E_{cv}, e_1 y e_2.

```
set.seed(24)
## Generamos los pesos con los que realizaremos
## todas las experimentaciones
W_f = simula_gauss(1,4,0,c(1,1,1,1))
W_f[4] = W_f[4]+1
ruido = simula_gauss(1,1,0,1)
varRuido = 0.5

## Ejecutamos la Cross Validation 1000 veces y guardamos todos los
## resultados para obtener los promedios y varianzas de Ecv, E1 y E2
resultados_CrossValidation = replicate(1000,RegSeleccionModelos_CV(W_f,ruido,varRuido))
```

El ultimo paso consiste en obtener la media y varianza de los errores E_{cv}, e_1 y e_2.

```
##Calculamos la media y varianza de los Ecv, E1 y E2
media_ECV = mean(resultados_CrossValidation[,1])
varianza_ECV = var(as.vector(resultados_CrossValidation[1,]))
media_E1 = mean(resultados_CrossValidation[2,])
varianza_E1 = var(resultados_CrossValidation[2,])
media_E2 = mean(resultados_CrossValidation[3,])
varianza_E2 = var(resultados_CrossValidation[3,])

print(sprintf("Media de E_cv = %f y varianza de E_cv = %f",media_ECV,varianza_ECV))
```

```
## [1] "Media de E_cv = 1.943457 y varianza de E_cv = 0.051210"
```

```
print(sprintf("Media de E_1 = %f y varianza de E_1 = %f",media_E1,varianza_E1))
```

```
## [1] "Media de E_1 = 2.298508 y varianza de E_1 = 0.435624"
```

```
print(sprintf("Media de E_2 = %f y varianza de E_2 = %f",media_E2,varianza_E2))
```

```
## [1] "Media de E_2 = 2.304469 y varianza de E_2 = 0.554583"
```

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.

La relacion entre la media de e_1 y E_{cv} es que ambos deberian ser muy parecidos debido a que E_{cv} utiliza todos los e_i para calcular un promedio de error en el Cross Validation. Los e_i tienen valores similares entre si, pero aquellos que presenten un valor de e_i mas bajo condicionaran el valor final de E_{cv}, lo que indica que con los datos de entrenamiento somos capaces de aprender de forma muy parecida para todas las particiones (en algunas mejor que en otras), hecho que se demuestra al realizar la validacion con cada particion. Es la misma relacion que existe entre e_1 y e_2, dado que nuestro modelo de regresion lineal con Weight Decay ofrece muy buenas soluciones, distando de la original tan solo por el ruido que le hemos añadido a la funcion.

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

El factor que mas contribuye a la varianza de los e_1 es el tamaño de las particiones. En cada experimento generamos unas particiones de entrenamiento y validacion, siendo la primera particion de validacion 3 elementos mas grande que las de validacion para las demas particiones. Esto puede influir notablemente en los resultados, aunque en nuestro caso al ser los tamaños de particiones muy pequeñas, no se puede asegurar que influya directamente en la varianza de los valores de e1. La aleatoriedad en la eleccion de las particiones y como se contruyen tambien es un factor determinante, dado que si existen datos muy representativos de la poblacion y estos se agrupan, la varianza de los resultados de e1 puede aumentar o disminuir notablemente.

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

En cuanto a la regularizacion, la tecnica de Weight Decay me permite reducir la varianza de las funciones que generamos con respecto de la funcion desconocida f haciendo que los pesos que calculamos sean lo mas pequeños posibles, lo que a la vez se traduce en unos errores de generalizacion mucho menores, con la contrapartida de tener un error dentro de la muestra ligeramente mayor. Para los experimentos que hemos realizado, la regularizacion ha dado buenos resultados puesto que el ruido que le añadimos a la funcion real cuando la generamos influye mucho en la funcion g que se elige y al añadir la regularizacion se garantiza un compromiso entre generalizacion y aproximacion, produciendo resultados bastante aceptables tanto dentro como fuera de la muestra.

La seleccion de modelos consiste en realizar sucesivos experimentos sobre el mismo conjunto de datos a traves del metodo de Cross Validation para calcular un conjunto de funciones que nos serviran para aproximar nuestra funcion desconocida f. Del conjunto de funciones generados elegiremos aquellas que menor E_{cv} tenga, ya que este es un buen estimador del E_{out}. Los experimentos realizados nos guian en la eleccion de una posible solucion a nuestro problema y por tanto es una tecnica muy util para cualquier problema real de aproximacion que podamos encontrarnos.

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 η= 0,1.

El funcionamiento del algoritmo es muy parecido al del Gradiente Descendente original, pero ahora tenemos dos pasos: en el primero nos movemos a lo largo de la coordenada u para reducir el error en esa coordenada. Posteriormete reevaluaremos la funcion gradiente con los nuevos pesos y despues se actualizan los pesos con el nuevo gradiente. Esto me permite reducir el error en cada una de las componentes y no solo en la totalidad de los pesos.

```
## CoordenadaDescendente(valoresIniciales,funcion,gradienteFuncion,
## tasaDeAprendizaje, cotaErrorMin, maxIters)

CoordenadaDescendente = function(valoresIniciales,funcion,gradienteFuncion,
                                 tasaDeAprendizaje, cotaErrorMin, maxIters){

  #Declaramos las variables que vamos a utilizar en nuestro algoritmo
  #y calculamos un primer valor de error
  iteraciones = 0
  u = valoresIniciales[1]
  v = valoresIniciales[2]
  W_old = c(u,v)
  W_new = W_old
  cotaError = funcion(W_old[1],W_old[2])

  #Mientras que  error este por encima de la cota minima de error
  while(cotaError > cotaErrorMin && iteraciones < maxIters){

    #Paso 1: Calculamos el vector gradiente con los primeros pesos
    gradiente = gradienteFuncion(W_old[1],W_old[2])
    vectorGradiente = -gradiente

    #Paso 1: Actualizamos la primera componente del vector de pesos
    W_new[1] = W_old[1] + (tasaDeAprendizaje*vectorGradiente[1])

    #Paso 2: Reevaluamos el gradiente con los pesos actualizados y
    #modificamos la segunda componente del vector de pesos
    gradiente = gradienteFuncion(W_new[1],W_new[2])
    vectorGradiente = -gradiente
    W_new[2] = W_new[2] + (tasaDeAprendizaje*vectorGradiente[2])
```

```
    #Calculamos la cota del error una vez hemos realizado los dos pasos
    cotaError = abs(funcion(W_old[1],W_old[2]) - funcion(W_new[1],W_new[2]))
    iteraciones = iteraciones + 1
    #Nos quedamos con los pesos de la iteracion anterior
    #ya que seran los que nos serviran para comparar
    W_old = W_new
  }

  #Devolvemos el numero de iteraciones y los valores de U y V finales
  list(iteracionesMax = iteraciones, pesos = W_new)

}
```

A continuacion realizamos las pruebas:

```
resultados1a_BONUS = CoordenadaDescendente(c(1,1),E,GradienteDeE,0.1, 10^-4, 15)
print(sprintf("El algoritmo termina tras %i iteraciones",resultados1a_BONUS$iteracionesMax))
```

```
## [1] "El algoritmo termina tras 3 iteraciones"
```

```
print(sprintf("Los valores obtenidos son u = %f y v = %f",
              resultados1a_BONUS$pesos[1],resultados1a_BONUS$pesos[2]))
```

```
## [1] "Los valores obtenidos son u = 11903.213631 y v = -89355363163.543076"
```

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

Dado que mi algoritmo mantiene la condicion de parada compuesta (maximo de iteraciones y error superior a 10^-4) no llegara a realizar las 15 iteraciones completas puesto que la diferencia de valores de E(u,v) que se obtiene en la tercera iteracion es 0, por tanto en 3 iteraciones es capaz de encontrar un minimo local de la funcion. El resultado se puede comprobar a continuacion.

```
## Comprobamos el valor de E(u,v) en
## los puntos obtenidos
print(sprintf("El valor de E(u,v) en los puntos u = %f y v = %f es de %f",
              resultados1a_BONUS$pesos[1],resultados1a_BONUS$pesos[2],
              E(resultados1a_BONUS$pesos[1], resultados1a_BONUS$pesos[2]) ))
```

```
## [1] "El valor de E(u,v) en los puntos u = 11903.213631 y v = -89355363163.543076 es de 0.000000"
```

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

Para realizar la comparativa entre el algoritmo de Coordenada Descendente y el del Gradiente Descendente vamos a ejecutar ambos algoritmos con una tasa de aprendizaje $\eta = 0.1$, con un maximo de iteraciones de 300 y con una cota de error muy cercana a 0 (10^{-14}). Ya hemos visto que nuestro algoritmo de Coordenada Descendente encuentra en 3 iteraciones un minimo local, asi que vamos a compararlo con los resultados que proporciona el Gradiente Descendente

```
resultadosGD = GradienteDescendente(c(1,1),E,GradienteDeE,0.1, 10^-14, 300)
print(sprintf("El Gradiente Descendente (GD) termina tras %i iteraciones",
              resultadosGD$iteracionesMax))
```

```
## [1] "El Gradiente Descendente (GD) termina tras 300 iteraciones"
```

```
print(sprintf("El valor de E(u,v) en los puntos u = %f y v = %f es de %f",
              resultadosGD$pesos[1],resultadosGD$pesos[2],
              E(resultadosGD$pesos[1], resultadosGD$pesos[2]) ))
```

```
## [1] "El valor de E(u,v) en los puntos u = 10.052480 y v = -24.422893 es de 0.002641"
```

```
resultados1a_BONUS = CoordenadaDescendente(c(1,1),E,GradienteDeE,0.1, 10^-14,300 )
print(sprintf("El algoritmo Coordenada Descendente (CD) termina tras %i iteraciones",
              resultados1a_BONUS$iteracionesMax))
```

```
## [1] "El algoritmo Coordenada Descendente (CD) termina tras 3 iteraciones"
```

```
print(sprintf("El valor de E(u,v) en los puntos u = %f y v = %f es de %f",
              resultados1a_BONUS$pesos[1],resultados1a_BONUS$pesos[2],
              E(resultados1a_BONUS$pesos[1], resultados1a_BONUS$pesos[2]) ))
```

```
## [1] "El valor de E(u,v) en los puntos u = 11903.213631 y v = -89355363163.543076 es de 0.000000"
```

Como se puede apreciar, la mejora que ofrece el algoritmo de Coordenada Descendente es abismal, dado que es capaz de encontrar en 3 iteraciones un error menor que 10^{-14}, mientras que el Gradiente Descendente no es capaz de encontrar un error menor que 0,0026 en 300 iteraciones. El hecho de minimizar por separado cada una de las componentes de los pesos me permite seguir la direccion de un minimo local y llegar a este mucho mas rapido, produciendo resultados con muchisima mas calidad que los que se pueden conseguir con una tecnica de GD.

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

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

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

Para repetir el experimento de Regresion Logistica realizado en el punto 2 de la practica necesitaremos generar para cada una de las 100 iteraciones una funcion frontera nueva que intentaremos ajustar con nuestra Regresion Logística. Generaremos un conjunto de puntos que utilizaremos para los N experimentos y para cada uno generaremos unas etiquetas con la funcion frontera en cada iteracion. Calcularemos la media del Eout para los N experimentos y la media de las epocas que tarda en converger.

```
set.seed(24)
## Generamos nuestro conjunto de 100 datos en el intervalo [0,2]x[0,2]
datosRL_Bonus = simula_unif(100,2,c(0,2))
datosRL_Bonus = cbind(datosRL_Bonus,1)
```

```
## Primera prueba con 100 datos generados anteriormente
## para obtener unos pesos
Wini = matrix(c(0,0,0),1,3)
tasaAprendizaje = 0.01
cotaErrorMin = 0.01
## Ejecutamos la Regresion Logistica durante 100 iteraciones
epocas = numeric(0)
EoutRL_Bonus = numeric(0)
set.seed(24)
```

```r
for(i in 1:100){

  #Generamos unas etiquetas con una funcion frontera distinta cada vez
  rectaRL_Bonus = simula_recta(c(0,2))
  etiquetasRL_Bonus = generarEtiquetas(datosRL_Bonus, rectaRL_Bonus)

  #Ejecutamos la RL con un conjunto de N=100 datos
  #y guardamos los pesos y las epocas
  resultadosRL_Bonus = LGRA_SGD(Wini,datosRL_Bonus,etiquetasRL_Bonus,
                                tasaAprendizaje,cotaErrorMin)
  pesosRL_Bonus = resultadosRL_Bonus$valoresFinales
  epocas = c(epocas,resultadosRL_Bonus$epoca)

  #Generamos un conjunto de test con N=100 datos para validar
  datosTest_RL_Bonus = simula_unif(100,2,c(0,2))
  datosTest_RL_Bonus = cbind(datosTest_RL_Bonus,1)
  etiquetasTest = generarEtiquetas(datosTest_RL_Bonus,rectaRL_Bonus)
  EoutRL_Bonus = c(EoutRL_Bonus, Ein(datosTest_RL_Bonus,etiquetasTest,pesosRL_Bonus))

}

print(sprintf("El valor de Eout medio para N=100 es de %f",mean(EoutRL_Bonus)))
```

```
## [1] "El valor de Eout medio para N=100 es de 0.114752"
```

```r
print(sprintf("El numero medio de epocas que tarda en converger para N=100 es %f",
              mean(epocas)))
```

```
## [1] "El numero medio de epocas que tarda en converger para N=100 es 403.270000"
```