Trabajo 2

Luis Suárez Lloréns

3 de mayo de 2016

Apartado 1: Modelos lineales

Ejercicio 1

Apartado a:

Lo primero, va a ser obtener el gradiente de la función $E(u,v) = (ue^v - 2ve^{-u})^2$.

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

Una vez tenemos el gradiente, el método del descenso del gradiente nos dice que debemos movernos hacia donde nos indique $\nabla E(u,v)$. Por tanto, el siguiente paso es crear funciones tanto para $E(u,v)$ como para su gradiente.

```
E = function(u,v){
  return( (u*exp(v) -  2*v*exp(-u))**2 )
}

grad.E = function(u,v){
  dx = 2*(u*exp(v) -  2*v*exp(-u))*(exp(v)+2*v*exp(-u))
  dy = 2*(u*exp(v) -  2*v*exp(-u))*(u*exp(v)-2*exp(-u))
  return(c(dx,dy))
}
```

Teniendo preparadas ya todas las funciones, creamos el procedimiento de descenso del gradiente.

```
descenso.gradiente = function(x,y,f,grad.f,tam.paso = 0.1, tolerancia = 10**-5, max.iter = 50){
  valores.f = rep(0,max.iter)
  idx = 1

  valores.f[idx] = f(x,y)

  while(f(x,y)>tolerancia && idx < max.iter){
    grad = grad.f(x,y)
    x = x - tam.paso*grad[1]
    y = y - tam.paso*grad[2]

    idx = idx + 1
    valores.f[idx] = f(x,y)
  }

  return(list(x = x, y = y, iter = idx, error = valores.f[1:idx]))
}

descenso.gradiente(1,1,E,grad.E,tolerancia = 10**-14)
```

```
## $x
## [1] 0.04473629
##
## $y
## [1] 0.02395871
##
## $iter
## [1] 11
##
## $error
##  [1] 3.930397e+00 1.159510e+00 1.007407e+00 9.900912e-02 8.660645e-03
##  [6] 1.817558e-04 1.297240e-06 7.291525e-09 4.009998e-11 2.201683e-13
## [11] 1.208683e-15
```

Apartado b

1)

El primer paso, de nuevo, es encontrar el gradiente de la función $f = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$.

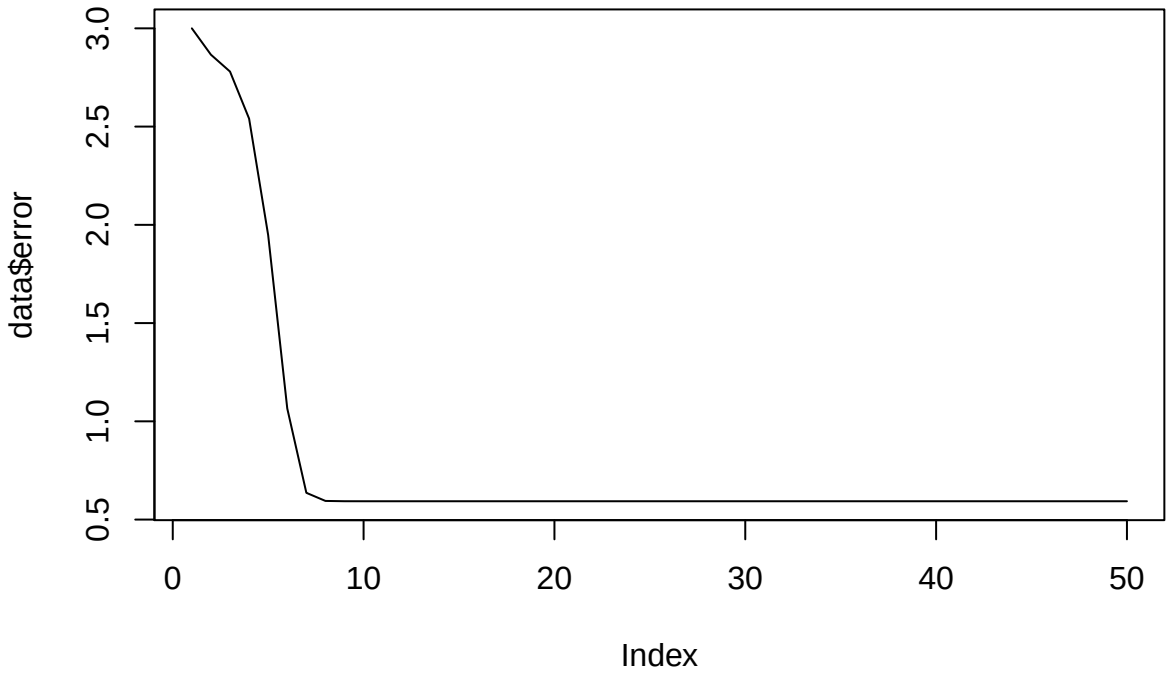
$$\nabla f(x,y) = (2x + 4\pi\sin(2\pi y)\cos(2\pi x), 4y + 4\pi\sin(2\pi x)\cos(2\pi y))$$

Definimos las funciones:

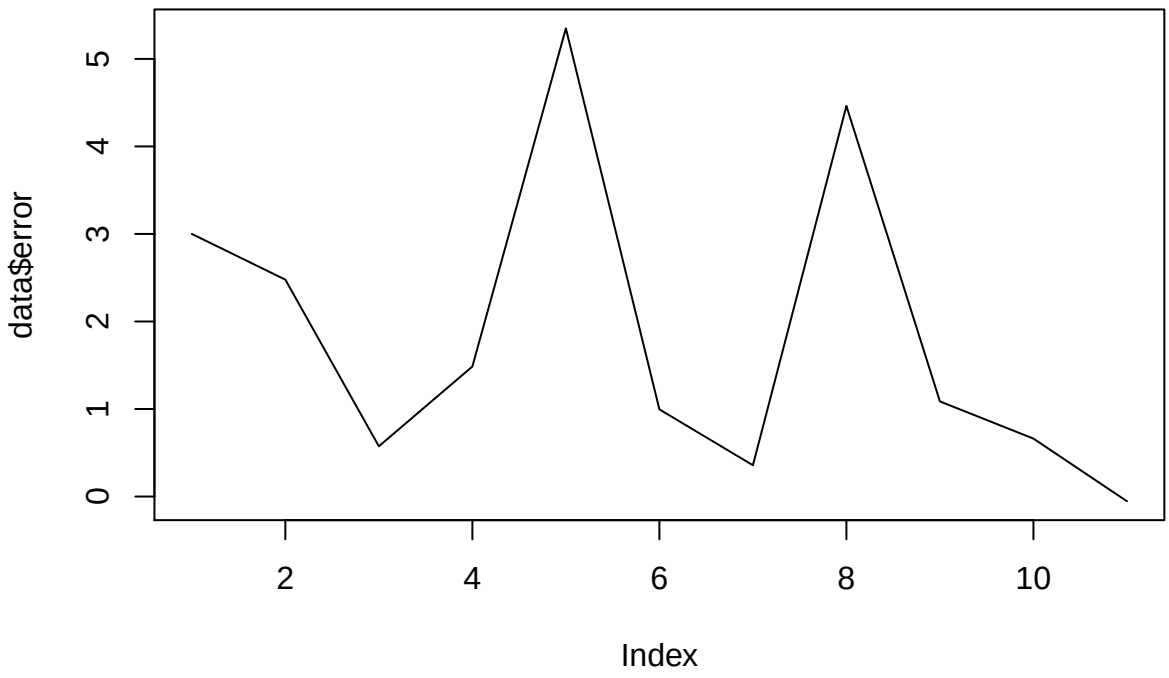
```
f = function(x,y){
  return(x**2+2*y**2+2*sin(2*pi*x)*sin(2*pi*y))
}
grad.f = function(x,y){
  dx = 2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x)
  dy = 4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y)
  return(c(dx,dy))
}
```

Volvemos a calcular el proceso del descenso del gradiente

```
data = descenso.gradiente(1,1,f,grad.f,tam.paso = 0.01, tolerancia = 10**-14)
plot(data$error,type = "l")
```




```
data = descenso.gradiente(1,1,f,grad.f,tam.paso = 0.1, tolerancia = 10**-14)
plot(data$error,type = "l")
```



Del primero, vemos como se queda atascado en un mínimo local, pues el valor se estabiliza en 0.5, y como podemos ver en la segunda gráfica, la función llega a 0. La segunda, pese a acercarse al 0 de la función, se comporta de una manera errática, pues la tasa de aprendizaje es demasiado grande.

2)

Ahora, vamos a ver los datos para diferentes puntos de partida:

```
data = descenso.gradiente(0.1,0.1,f,grad.f,tam.paso = 0.01, tolerancia = 10**-14)
cat(paste0("x:",data$x,"\ny:",data$y,"\nvalor mínimo:",data$error[data$iter]))
```

```
## x:0.00526609481014761
## y:-0.00239206934812207
## valor mínimo:-0.000955213854398881
```

```
data = descenso.gradiente(1,1,f,grad.f,tam.paso = 0.01, tolerancia = 10**-14)
cat(paste0("x:",data$x,"\ny:",data$y,"\nvalor mínimo:",data$error[data$iter]))
```

```
## x:1.21807030131108
## y:0.712811950601778
## valor mínimo:0.593269374325836
```

```
data = descenso.gradiente(-0.5,-0.5,f,grad.f,tam.paso = 0.01, tolerancia = 10**-14)
cat(paste0("x:",data$x,"\ny:",data$y,"\nvalor mínimo:",data$error[data$iter]))
```

```
## x:-0.580323406173378
## y:-0.383991863552131
## valor mínimo:-0.012440020728642
```

```
data = descenso.gradiente(-1,-1,f,grad.f,tam.paso = 0.01, tolerancia = 10**-14)
cat(paste0("x:",data$x,"\ny:",data$y,"\nvalor mínimo:",data$error[data$iter]))
```

```
## x:-1.21807030131108
## y:-0.712811950601778
## valor mínimo:0.593269374325836
```

De estos datos, podemos observar que es muy fácil caer en un mínimo local.

Este es una de las mayores dificultades que tienen que afrontar este tipo de métodos, pues como podemos ver, no tienen manera de salir de un mínimo local que no sea aumentar el tamaño de paso. Pero en ese caso, se pierde la habilidad de afinar y conseguir fácilmente el mínimo en caso de tener bien posicionado el punto de partida, es más, se puede incluso salir del mínimo y no obtener ningún buen resultado.

Además, el método no va siempre al mismo mínimo en caso de encontrar una buena solución, lo que también podría llegar a ser un problema.

Ejercicio 2

Vamos a definir la función de **Coordenada Descendente**, que realiza un proceso similar al gradiente descendente, pero avanzando sólo en una dirección cada vez.

```
coordenada.descendente = function(x,y,f,grad.f,tam.paso = 0.1, tolerancia = 10**-5, max.iter = 50){
  valores.f = rep(0,max.iter)
  idx = 1

  valores.f[idx] = f(x,y)

  while(f(x,y)>tolerancia && idx < max.iter){
    grad = grad.f(x,y)
    x = x - tam.paso*grad[1]
    grad = grad.f(x,y)
    y = y - tam.paso*grad[2]

    idx = idx + 1
    valores.f[idx] = f(x,y)
  }

  return(list(x = x, y = y, iter = idx, error = valores.f[1:idx]))
}
```

Ahora, vamos a probarlo con las funciones del ejercicio 1.1.

```
coordenada.descendente(1,1,E,grad.E,tolerancia = 10**-14)
```

```
## $x
## [1] 6.235043
##
```

```
## $y
## [1] -3.377273
##
## $iter
## [1] 50
##
## $error
##  [1]  3.93039723 34.29016311  0.53414259  0.43266083  0.36503974
##  [6]  0.31646808  0.27976342  0.25098631  0.22778330  0.20865670
## [11]  0.19260566  0.17893475  0.16714505  0.15686899  0.14782952
## [16]  0.13981379  0.13265544  0.12622253  0.12040902  0.11512872
## [21]  0.11031077  0.10589645  0.10183660  0.09808978  0.09462081
## [26]  0.09139961  0.08840030  0.08560053  0.08298084  0.08052425
## [31]  0.07821585  0.07604251  0.07399261  0.07205582  0.07022296
## [36]  0.06848580  0.06683698  0.06526986  0.06377846  0.06235738
## [41]  0.06100171  0.05970700  0.05846918  0.05728457  0.05614976
## [46]  0.05506165  0.05401739  0.05301436  0.05205014  0.05112250
```

Como podemos ver, el descenso del gradiente si llega al mínimo mientras que coordenada descendente no. Si nos fijamos en los resultados, este método nos lleva a otro mínimo, lo cual impide encontrar el mínimo que sí habíamos encontrado antes. Esto no hace que no pueda encontrar un mínimo, pero nos hace pensar que el descenso del gradiente es más sólido que la coordenada descendente.

Aquí podemos ver un ejemplo, cambiando la tasa de aprendizaje, de como es capaz de encontrar un mínimo.

```
coordenada.descendente(1,1,E,grad.E,tam.paso = 0.05,tolerancia = 10**-14)
```

```
## $x
## [1] 0.4626781
##
## $y
## [1] 0.9519494
##
## $iter
## [1] 21
##
## $error
##  [1] 3.930397e+00 3.363428e-01 1.041846e-01 1.609311e-02 3.347256e-03
##  [6] 6.112302e-04 1.181141e-04 2.226773e-05 4.243474e-06 8.048796e-07
## [11] 1.529775e-07 2.904943e-08 5.518439e-09 1.048145e-09 1.990941e-10
## [16] 3.781652e-11 7.183082e-12 1.364386e-12 2.591583e-13 4.922575e-14
## [21] 9.350177e-15
```

Ejercicio 3

Primero, vamos a definir el método de Newton.

```
metodo.Newton = function(x,y,f,grad.f,hess.f, tolerancia = 10**-5, max.iter = 50){
  valores.f = rep(0,max.iter)
  idx = 1

  valores.f[idx] = f(x,y)
```

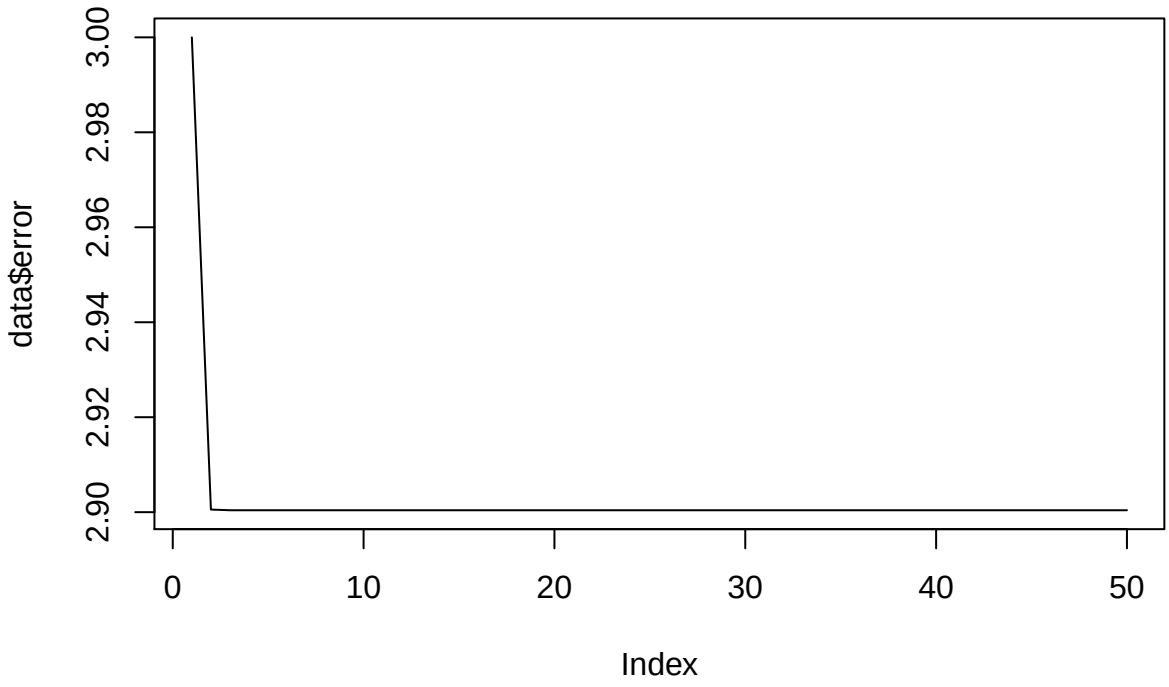
```
  while(f(x,y)>tolerancia && idx < max.iter){
    grad = grad.f(x,y)
    hess = hess.f(x,y)

    diff.w = - solve(hess) %*% grad
    x = x + diff.w[1]
    y = y + diff.w[2]

    idx = idx + 1
    valores.f[idx] = f(x,y)
  }

  return(list(x = x, y = y, iter = idx, error = valores.f[1:idx]))
}
```

Para poder usar el método de Newton, necesitamos también calcular la matriz hessiana de la función.

```
hess.f = function(x,y){
  dxx = 2 - 8*pi*pi*sin(2*pi*y)*sin(2*pi*x)
  dyy = 4 - 8*pi*pi*sin(2*pi*x)*sin(2*pi*y)
  dxy = 8*pi*pi*cos(2*pi*y)*cos(2*pi*x)
  return(matrix(c(dxx,dxy,dxy,dyy),2))
}
```

Y lo probamos:

```
data = metodo.Newton(1,1,f,grad.f,hess.f)
plot(data$error,type = "l")
```



Como podemos ver, pese a usar este método, seguimos topandonos con un mínimo local, que nos impide alcanzar el mínimo buscado. Eso sí, la convergencia a ese mínimo local esmuy rápida.

Ejercicio 4

Para empezar, vamos a reutilizar métodos de la práctica anterior para generar los datos y la recta. Generamos los datos y la recta y los clasificamos.

```
#Funcion de generacion de datos
simula_unif <- function(N,dim,rango){
  matrix(runif(N*dim,rango[1],rango[2]),ncol = dim)
}
#Funcion de generacion de recta
simula_recta <- function(rango){
  puntos <- simula_unif(2,2,rango)
  a <- 0
  b <- 0

  if((puntos[1,1]-puntos[2,1]) != 0){
    a <- (puntos[1,2]-puntos[2,2])/(puntos[1,1]-puntos[2,1])
    b <- puntos[1,2] - a*puntos[1,1]
  }
  else{
    a <- Inf
    b <- puntos[1,1]
  }
```

```
  c(a,b)
}

#Creamos los datos y la recta.
datos.ej1.4 = simula_unif(100,2,c(-1,1))
recta.ej1.4 = simula_recta(c(-1,1))

#Funcion de evaluacion de recta
recta = recta.ej1.4
eval.recta <- function(x,y){
  y-recta[1]*x-recta[2]
}
#Funcion de clasificacion
clasifica <- function(f,x,y){
  (sign(f(x,y))+1)/2
}

label.ej1.4 = clasifica(eval.recta,datos.ej1.4[,1],datos.ej1.4[,2])
```

Una vez hemos definido los datos, vamos a generar la función que nos va a calcular la regresión logística.

```
Regresion.Logistica = function(data,label,vini,tam.paso = 0.01, tolerancia = 0.01, max.iter = 50){
  valores.f = rep(0,max.iter)
  idx = 1
  current.coef = vini
  last.coef = vini
  dist.coef = tolerancia+1
  data.extended = cbind(data,1)
  labels = label

  while(dist.coef>tolerancia && idx < max.iter){
    idx = idx + 1
    last.coef = current.coef
    samp = sample(nrow(data),nrow(data))

    for (j in samp){
      g = -(label[j]*data.extended[j,])/(1+exp(label[j]*t(current.coef)*data.extended[j,]))
      current.coef = current.coef - tam.paso*as.vector(g)
    }
    dist.coef = sqrt(sum((current.coef-last.coef)^2))

  }

  return(list(coeff = current.coef, iter = idx))
}
```

Y ahora la usamos para los datos generados antes:

```
resultados = Regresion.Logistica(datos.ej1.4,label.ej1.4,c(0,0,0),max.iter = 1000)
coef.1.4 = resultados$coeff
print(resultados)
```

```
## $coeff
```

```
## [1] -3.219816  4.617182  4.021685
##
## $iter
## [1] 211
```

Con esto, ya tenemos nuestros coeficientes. Ahora vamos a calcular el error fuera de la muestra. Creamos nuevos valores, lo etiquetamos, y consideramos el error como la media de los errores.

```
logistic.function = function(x){exp(t(coef.1.4)*x)/(1+exp(t(coef.1.4)*x))}

mean(abs(label.ej1.4-apply(cbind(datos.ej1.4,1), 1, logistic.function)))
```

```
## [1] 0.5735504
```

Ejercicio 5

Primero, cargamos los datos y los preparamos como en la práctica anterior. Además cargamos todos los métodos necesarios de esa práctica.

```
#PLA y regresion lineal
ajusta_PLA_MOD <- function(datos, label, max_iter, vini){
  cambio = TRUE
  coef = vini
  iteracion = 0
  num_datos = length(label)+1
  mejor_error = 1
  mejor_coef = c(0,0,0)

  while(cambio && iteracion < max_iter){
    i=1
    cambio = FALSE

    while(i<num_datos){

      if(sign(t(c(datos[i,],1)) %*% coef) != label[i]){
        coef <- coef + label[i]*c(datos[i,],1)
        cambio = TRUE

        error <- mean(abs(label-sign(datos[,1]*coef[1]+ datos[,2]*coef[2] + coef[3])))*0.5
        if(mejor_error > error){
          mejor_error <- error
          mejor_coef <- coef
        }
      }
      i <- i+1
    }

    iteracion <- iteracion + 1
  }

  return(list(coef = mejor_coef,iter = iteracion))
}
```

```
Regress_lin <- function(datos,label){
  s <- svd(datos)
  D <- diag(s$d)
  Dinv <- solve(D)

  p.inversa <- (s$v %*% Dinv %*% Dinv %*% t(s$v)) %*% t(datos)

  return(p.inversa%*%label)
}

trans.matriz <- function(data){
  return(matrix((data*0.5)+0.5,16))
}
simetria <- function(data){
  mat <- trans.matriz(data)
  return(-sum(abs(mat[,1:16] - mat[,16:1])))
}
```

```
rawdatatrain.num <- read.table("~/AA/data/zip.train", quote="\"", comment.char="", stringsAsFactors=FALS
```

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

```
rawdata.5 <- rawdatatrain.num[rawdatatrain.num[,1] == 5,2:257]
rawdata.1 <- rawdatatrain.num[rawdatatrain.num[,1] == 1,2:257]

data.5 = data.matrix(rawdata.5)
data.1 = data.matrix(rawdata.1)

descriptores.1 <- t(apply(data.1,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores.5 <- t(apply(data.5,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores.train <- rbind(descriptores.1,descriptores.5)
etiquetas.train <- rep(c(1,-1), c(nrow(descriptores.1),nrow(descriptores.5)))

rawdatatest.num <- read.table("~/AA/data/zip.test", quote="\"", comment.char="", stringsAsFactors=FALSE
```

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

```
rawdata.5 <- rawdatatest.num[rawdatatest.num[,1] == 5,2:257]
rawdata.1 <- rawdatatest.num[rawdatatest.num[,1] == 1,2:257]

data.5 = data.matrix(rawdata.5)
data.1 = data.matrix(rawdata.1)

descriptores.1 <- t(apply(data.1,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores.5 <- t(apply(data.5,MARGIN = 1, FUN = function(data){return(c(mean(trans.matriz(data)),sim
descriptores.test <- rbind(descriptores.1,descriptores.5)
etiquetas.test <- rep(c(1,-1), c(nrow(descriptores.1),nrow(descriptores.5)))
```

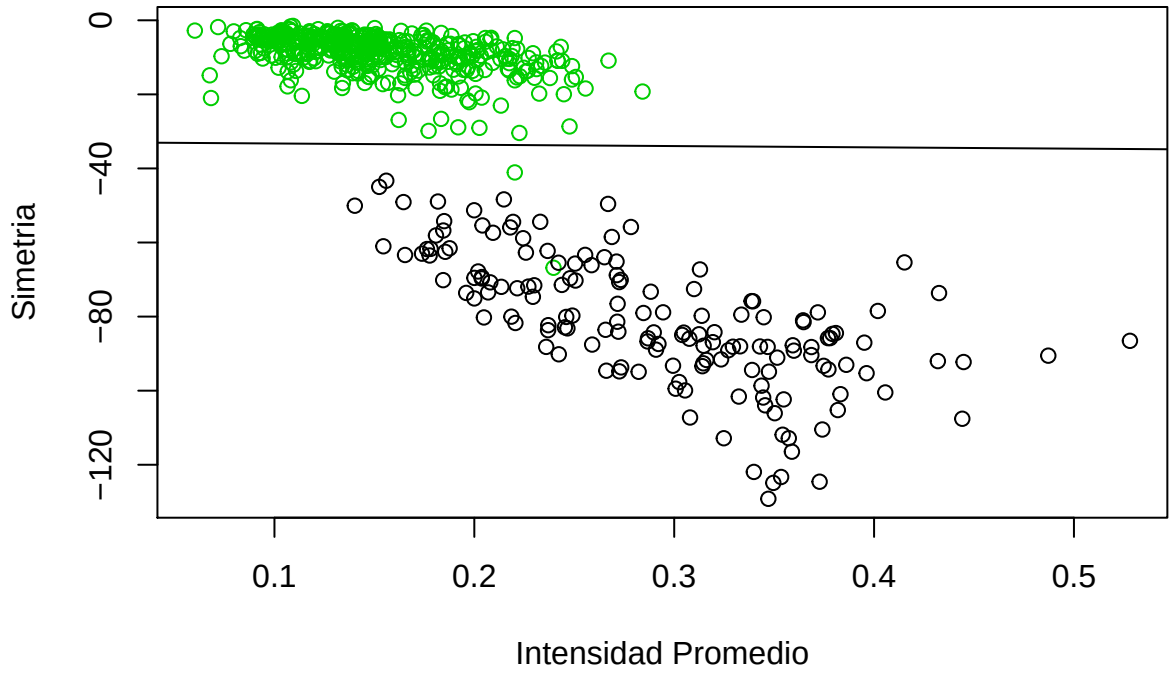
Con los datos cargados, calculamos la recta.

```
reg.coef = Regress_lin(cbind(descriptores.train,1),etiquetas.train)
result = ajusta_PLA_MOD(descriptores.train,etiquetas.train,max_iter = 50,vini = as.vector(reg.coef))
pla.coef = result$coef
print(pla.coef)
```

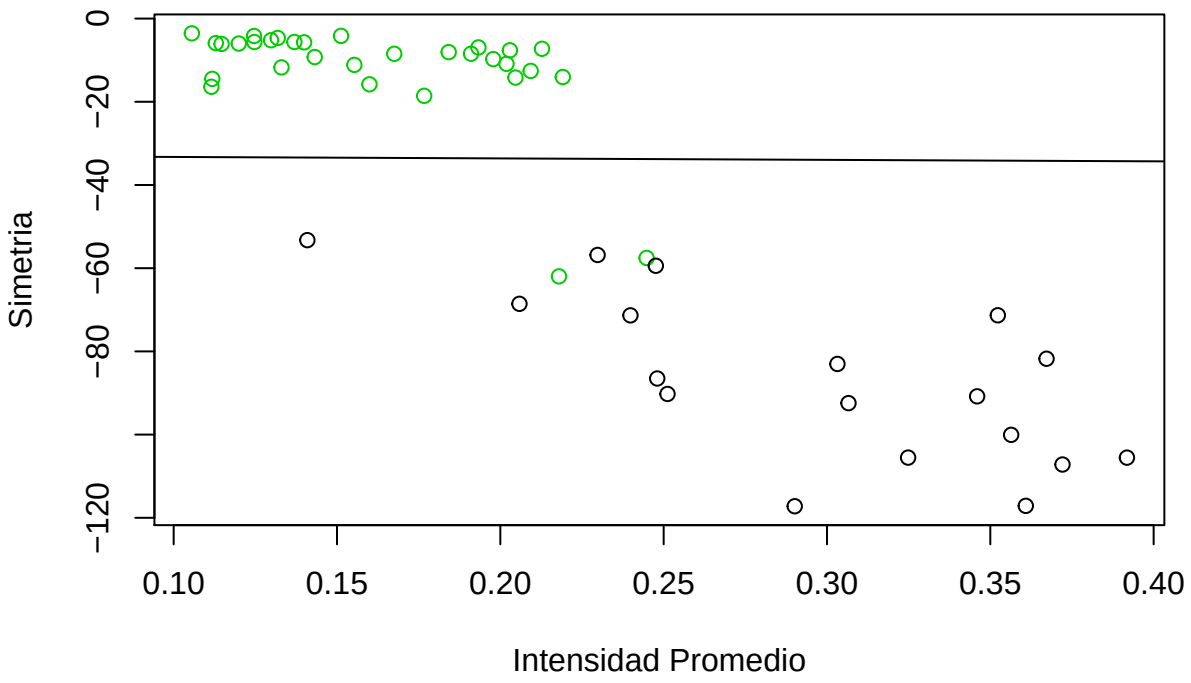
```
## [1]  37.42560  10.72404 352.98230
```

Mostramos los dos gráficos.

```
plot(descriptores.train, col = etiquetas.train+2, xlab = "Intensidad Promedio", ylab = "Simetria")
abline(-pla.coef[3]/pla.coef[2],-pla.coef[1]/pla.coef[2])
```




```
plot(descriptores.test, col = etiquetas.test+2, xlab = "Intensidad Promedio", ylab = "Simetria")
abline(-pla.coef[3]/pla.coef[2],-pla.coef[1]/pla.coef[2])
```



Calculamos el error de entrenamiento y el de test.

```
err.in = mean(abs(etiquetas.train-sign(pla.coef[1]*descriptores.train[,1]+pla.coef[2]*descriptores.trai
err.test = mean(abs(etiquetas.test-sign(pla.coef[1]*descriptores.test[,1]+pla.coef[2]*descriptores.test
cat(paste0("Error en la muestra: ",err.in,"\nError de test: ",err.test))
```

```
## Error en la muestra: 0.00333889816360601
## Error de test: 0.0408163265306122
```

Con estos valores, podemos obtener las siguientes cotas de error. La primera de las cotas que encontramos usando el error de entrenamiento es infinita, pues el error de generalización básico depende del número de funciones que consideremos, y el conjunto de todas las rectas es infinito.

Para poder plantear una cota con el error dentro del muestra, tenemos que usar la dimensión de Vapnik-Chervonenkis. En el caso de las rectas en el plano, sabemos que dicha dimensión es 3. Por tanto, la cota de Vapnik-Chervonenkis queda así.

```
err.in + sqrt(8/nrow(descriptores.train)*log((4*((2*nrow(descriptores.train))^(3)+1))/0.05))
```

```
## [1] 0.5886032
```

La cota con respecto al error de test se debe a Hoeffding. Primero, calculamos ϵ tal que:

$$2e^{-2\epsilon^2 N} \leq 0.05$$

Y obtenemos que el menor epsilon que cumple la condición es $\epsilon \approx 0.194$. Entonces, el máximo error fuera de la muestra esperado es el error de test más ϵ. Obtenemos un valor entonrno a 0.25. Este valor es menor que los asociados al error dentro de la muestra, lo cual pone de manifiesto la utilidad de utilizar muestras de test para aproximar el error fuera de la muestra.

Apartado 2: Sobreajuste

Para este apartado, usamos principalmente los polinomios de Legendre. Así que como primer paso, vamos a crear una función para poder usarlos con más comodidad.

```
pol.Legendre = function(x,n){
  if(n == 0)
    return(1)
  else if(n == 1)
    return(x)
  else
    return(((2*n-1)/n)*x*pol.Legendre(x,n-1)+((n-1)/n)*pol.Legendre(x,n-2))
}
```

1.a) Los cálculos para obtener g_2 y g_10 se encuentran en el siguiente apartado, en la función *experimento.ej2*.

1.b) Es importante normalizar la función f para que σ tenga el efecto querido, introducir ruido con igual influencia en todos los experimentos. Si no se realizara, los valores de la función f podrían ser muy grandes (y el ruido casi no tendría efecto) o muy pequeños (y sólo importaría el ruido).

2 En los siguientes apartados, vamos a realizar muchas veces este experimento, así que vamos a crear una función para su realización.

```
f = function(a_q, x){
  result = 0
  for(i in 1:length(a_q)){
    result = result + a_q[i]*pol.Legendre(x,i-1)
  }
  return(result)
}

experimento.ej2 = function(Qf,N,sigma){

  a_q = rnorm(Qf+1)
  normalizacion = sqrt(sum(1/(2*(0:Qf)+1)))
  a_q = a_q/normalizacion

  datatrain = simula_unif(N,1,c(-1,1))
  datatrain.values = f(a_q,datatrain) + sigma*rnorm(nrow(datatrain))

  g2 = Regress_lin(cbind(1,datatrain,datatrain^2),datatrain.values)
  g10 = Regress_lin(cbind(1,datatrain,datatrain^2,datatrain^3,
                          datatrain^4,datatrain^5,datatrain^6,
                          datatrain^7,datatrain^8,datatrain^9,
                          datatrain^10),datatrain.values)

  dataout = simula_unif(N,1,c(-1,1))
  dataout.values = f(a_q,dataout)
```

```
  Eout.g2 = mean(abs(dataout.values - cbind(1,dataout,dataout^2)%*%g2))
  Eout.g10 = mean(abs(dataout.values - cbind(1,dataout,dataout^2,dataout^3,
                                             dataout^4,dataout^5,dataout^6,
                                             dataout^7,dataout^8,dataout^9,
                                             dataout^10)%*%g10))

  return(list(g2 = g2, Eout.g2 = Eout.g2,g10 = g10, Eout.g10 = Eout.g10))
}

experimento.ej2(5,10,0.05)
```

```
## $g2
##             [,1]
## [1,]  -1.440372
## [2,]  -8.012005
## [3,] -10.072247
##
## $Eout.g2
## [1] 1.08408
##
## $g10
##                [,1]
##  [1,]     29.46137
##  [2,]   -155.10835
##  [3,]  -2169.10297
##  [4,]  -3557.45230
##  [5,]   9247.95973
##  [6,]  19904.89902
##  [7,] -16241.66162
##  [8,] -35301.35150
##  [9,]  16627.12979
## [10,]  21286.04640
## [11,]  -9342.55951
##
## $Eout.g10
## [1] 135.8355
```

Una vez tenemos esto, podemos realizar los experimentos del apartado 2.

```
err.H2 = 0
err.H10 = 0

for(i in 1:100){
  exp = experimento.ej2(20,50,1)
  err.H2 = err.H2 + exp$Eout.g2
  err.H10 = err.H10 + exp$Eout.g10
}
cat(paste0("Error de H2: ",err.H2/100,"\nError de H10: ",err.H10/100))
```

```
## Error de H2: 258540.506556678
## Error de H10: 5865.59091445652
```

De estos datos podemos ver que el modelo de polinomios de grado 10 no está sobreajustando.

2.a) Podría ser una buena medida. Si nos diera un valor positivo, significaría que ajustar un polinomio de grado 10 nos da un mayor error que un polinomio de grado 2. Esto claramente nos indica que el modelo de grado 10 estaría sobreaprendiendo, y cometiendo grandes errores fuera de la muestra que se le proporciona.

Si da un número negativo, podemos ver que el modelo mejora, pero puede que ya esté sobreaprendiendo. Por tanto, este tipo de medida parece correcto, pero debería ser más exhaustiva, es decir, que no se compare con respecto a los polinomios de grado 2, si no con respecto a todos los polinomios de grados inferiores a 10.

Apartado 3: Regularización y selección de modelos.

a) y b) Vamos a repetir la estructura del apartado anterior.

```
library(MASS)
RegLinWD = function(datos,label,lambda) {
  mat.inv <- ginv(t(datos)%*%datos + lambda * diag(nrow = ncol(datos),ncol = ncol(datos)))
  p.inversa <- mat.inv %*% t(datos)
  return(p.inversa%*%label)
}

simula_gaus <- function(N,dim,sigma){
  matrix(rnorm(N*dim,sd=sqrt(sigma)),ncol = dim,byrow = TRUE)
}

experimento.ej3 = function(N, d = 3, sigma = 0.5, lambda = 0.05){
  wf =  rnorm(d+1)

  datatrain = simula_gaus(N,d,1)
  datatrainextended = cbind(datatrain,1)
  datatrain.values = apply(datatrainextended,1,
                           function(x){wf%*%x + sigma*rnorm(1)})

  errores = sapply(1:N,FUN = function(x){
                          w = RegLinWD(datatrainextended[-x,],datatrain.values[-x],lambda)
                          return(abs(t(w)%*%datatrainextended[x,]-datatrain.values[x]))
                        })

  return(list(errores = errores, Ecv = mean(errores)))
}

experimento.ej3(10)
```

```
## $errores
##  [1] 0.79945556 0.10931285 0.55008138 0.06089836 0.49697616 0.10066604
##  [7] 0.14865813 1.10168185 1.18405858 0.39964614
##
## $Ecv
## [1] 0.4951435
```

Con esto, realizamos un experimento en concreto. Para los ejercicios, se requiere lanzar un número grande de estos experimentos y sacar estadísticas de los mismos.

Por comodidad, vamos a crear otra función, que para unos datos en concreto, nos devuelva todas las estadísticas necesarias.

```r
estadisticas.ej3 = function(N,num.repeticiones = 10^3, lambda = 0.05, verb = T){
  e1 = vector(mode = "numeric", length = num.repeticiones)
  e2 = vector(mode = "numeric", length = num.repeticiones)
  Ecv = vector(mode = "numeric", length = num.repeticiones)

  for(i in 1:num.repeticiones){
    exp = experimento.ej3(N, lambda = lambda)
    e1[i] = exp$errores[1]
    e2[i] = exp$errores[2]
    Ecv[i] = exp$Ecv
  }
  if(verb){
    cat("Estadísticas con ",N," datos.\n")
    cat(paste0("e1:\n","\tmedia: ",mean(e1),"\n\tvarianza: ",var(e1),"\n"))
    cat(paste0("e2:\n","\tmedia: ",mean(e2),"\n\tvarianza: ",var(e2),"\n"))
    cat(paste0("Ecv:\n","\tmedia: ",mean(Ecv),"\n\tvarianza: ",var(Ecv),"\n"))
    cat(paste0("Neff: ",var(e1)/var(Ecv),"\n\n"))
  }
  return(var(e1)/var(Ecv))
}

estadisticas.ej3(20)
```

```
## Estadísticas con  20  datos.
## e1:
##  media: 0.441736749473972
##  varianza: 0.111370621382397
## e2:
##  media: 0.463558020000019
##  varianza: 0.126027752368996
## Ecv:
##  media: 0.443043966402325
##  varianza: 0.00703310908269682
## Neff: 15.8351903934486
```

```
## [1] 15.83519
```

Con esto, simplemente tenemos que hacer un bucle que nos genere todos los datos que necesitamos. Guardamos N_eff para el apartado **f)**.

```r
Neff = vector(mode = "numeric", length = 11)
for(i in 1:11){
  Neff[i] = (estadisticas.ej3(3+5+10*i)/(3+5+10*i))*100
}
```

```
## Estadísticas con  18  datos.
## e1:
##  media: 0.450973522107142
##  varianza: 0.127908060443509
## e2:
##  media: 0.472318418800411
##  varianza: 0.126872364171158
```

```
## Ecv:
##  media: 0.455085647341862
##  varianza: 0.0080277107697169
## Neff: 15.9333169956769
##
## Estadísticas con  28  datos.
## e1:
##  media: 0.418491348664168
##  varianza: 0.101485726157724
## e2:
##  media: 0.422895370099274
##  varianza: 0.0992713118334302
## Ecv:
##  media: 0.43226868789987
##  varianza: 0.00431601920403646
## Neff: 23.5137336883978
##
## Estadísticas con  38  datos.
## e1:
##  media: 0.42876987307187
##  varianza: 0.094717430676759
## e2:
##  media: 0.418562278655986
##  varianza: 0.102447342599617
## Ecv:
##  media: 0.419823459039456
##  varianza: 0.00296130600815741
## Neff: 31.9850195879265
##
## Estadísticas con  48  datos.
## e1:
##  media: 0.421820111874585
##  varianza: 0.0976913257806946
## e2:
##  media: 0.404656396846664
##  varianza: 0.100367821464513
## Ecv:
##  media: 0.418623333507619
##  varianza: 0.00230508765793998
## Neff: 42.3807422004939
##
## Estadísticas con  58  datos.
## e1:
##  media: 0.413367104891051
##  varianza: 0.0994576375394564
## e2:
##  media: 0.422096751760466
##  varianza: 0.101469851395262
## Ecv:
##  media: 0.413618548665099
##  varianza: 0.00172754963111173
## Neff: 57.5715080761254
##
## Estadísticas con  68  datos.
```

```
## e1:
##  media: 0.408580879310045
##  varianza: 0.092335490868568
## e2:
##  media: 0.410019149037108
##  varianza: 0.0920624129802128
## Ecv:
##  media: 0.411226725335657
##  varianza: 0.00141679895700013
## Neff: 65.171907709528
##
## Estadísticas con  78  datos.
## e1:
##  media: 0.415098374296954
##  varianza: 0.0930928403374406
## e2:
##  media: 0.393528106055686
##  varianza: 0.0916010072101695
## Ecv:
##  media: 0.409893407645401
##  varianza: 0.00117999412696434
## Neff: 78.8926302344669
##
## Estadísticas con  88  datos.
## e1:
##  media: 0.426002612072846
##  varianza: 0.102884564749313
## e2:
##  media: 0.412285113678358
##  varianza: 0.0945572609950068
## Ecv:
##  media: 0.40780115993208
##  varianza: 0.00111675656150205
## Neff: 92.1280145521891
##
## Estadísticas con  98  datos.
## e1:
##  media: 0.404294039868494
##  varianza: 0.0923278963970905
## e2:
##  media: 0.399464047125819
##  varianza: 0.0907798971124725
## Ecv:
##  media: 0.408021414449142
##  varianza: 0.00106094683838897
## Neff: 87.0240553591624
##
## Estadísticas con  108  datos.
## e1:
##  media: 0.397084464748078
##  varianza: 0.0886653770985266
## e2:
##  media: 0.413201351720368
##  varianza: 0.102298545717672
```

```
## Ecv:
##  media: 0.404574882092188
##  varianza: 0.000920928961582384
## Neff: 96.2781938643536
##
## Estadísticas con  118  datos.
## e1:
##  media: 0.401126448281434
##  varianza: 0.0865521821539299
## e2:
##  media: 0.424061005166473
##  varianza: 0.102346614440774
## Ecv:
##  media: 0.405255950431231
##  varianza: 0.000828449149505416
## Neff: 104.474948408845
```

Ahora pasamos a responder a las preguntas del apartado.

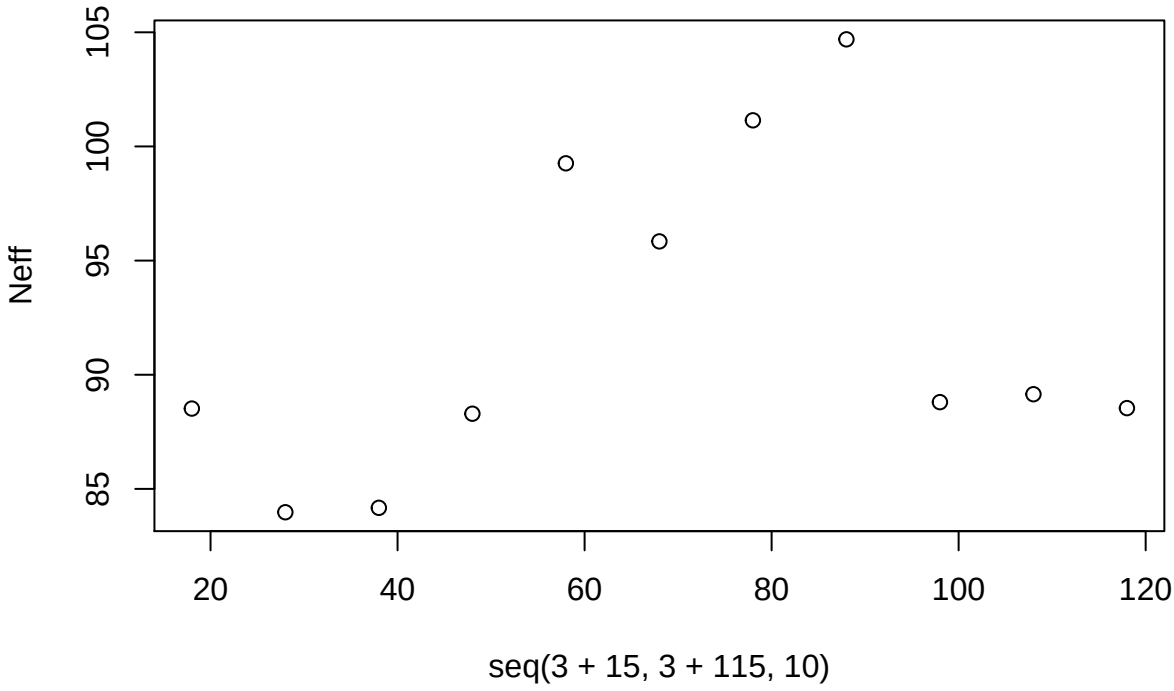
c) Los valores promedio deben de ser similares. Esto se debe a que tanto la media de e_1 y de e_2 se obtienen de la misma distribución. Para un número de experimentos grande, la media de estos valores se parecerán. E_cv no es más que la media de los e_i, luego su media también tenderá al mismo valor.

d) A la varianza de e_1 contribuyen la generación aleatoria del punto, el error introducido a la hora de evaluar el valor que va a tomar el punto y la bondad del ajuste, donde participa con especial importancia la regularización.

e) Si fueran realmente independientes, deberíamos obtener que la varianza de E_cv es $\frac{1}{n}$ veces la de e_i. Esto se debe a que podríamos ver la varianza de E_cv como la varianza de todos los datos generados de todas las iteraciones. Después separamos el cálculo de la varianza por cada e_i y obtendríamos el resultado.

f) Es un buen estimador pues, en realidad, los datos no son independientes. De serlo, el número de datos útil sería el cien por cien. Así pues, esta medida nos da una percepción de lo separados que se encuentran los datos entre ellos, es decir, la utilidad de cada dato generado.

```
plot(seq(3+15,3+115,10),Neff)
```



Como podemos ver, el valor oscila entorno al 95 por ciento, que se acerca a lo que esperaríamos si los datos fueran independientes, que sería el cien por cien.

g) Por lo que hemos visto en el apartado **e)**, si fueran independientes no supondría ninguna variación, nos daría un cien por cien. Así pues, podríamos esperar que al aumentar la regularización, bajarán los valores de la varianza, pero bajará el valor en ambos factores, tanto en la varianza de e_1 como en la de $E_c v$. Por tanto, espero que los valores no cambien mucho.

```
Neff2 = vector(mode = "numeric", length = 11)
for(i in 1:11){
  Neff2[i] = (estadisticas.ej3(3+5+10*i,lambda = 2.5,verb = F)/(3+5+10*i))*100
}
cat(paste0("La diferencia media entre los Neff es: ",mean(Neff-Neff2)))
```

```
## La diferencia media entre los Neff es: -6.61877198437868
```

Podemos observar que los nuevo valores de Neff son ligeramente mayores a los anteriores. Parece que la varianza de $E_c v$ se ve más afectada por la normalización que la de e_1, de ahí la subida.