APRENDIZAJE AUTOMÁTICO

****

TRABAJO 2. PROGRAMACIÓN.

****

Objetivos:

- **Modelos lineales.**
- **Sobreajuste.**
- **Regularización y selección de modelos.**

Autora: Cristina Zuheros Montes.

- Correo: zuhe18@gmail.com
- Github: https://github.com/cristinazuhe

Fecha: 03 Mayo 2016

********** SECCIÓN 1 **********

MODELOS LINEALES.

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

a) Considerar la función no lineal de error $E(u, v) = (ue^{v} \square - 2ve\square^{-u})^2$.

Usar gradiente descendente y minimizar esta función de error, comenzando desde el punto (u, v) = (1, 1) y usando una tasa de aprendizaje 0,1.

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

2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de E(u, v) inferior a 10^{-14} .

3) ¿Qué valores de (u, v) obtuvo cuando alcanzó el error de 10^{-14} ?

```
fs1e1f1 = function(x,y) ((x*exp(y))- (2*y*exp(-x)))^2  #defino la funcion
fs1e1f2 = function(x,y) x^2 +  2*y^2 +2*sin(2*pi*x)*sin(2*pi*y)  #defino la funcion

graddesc = function(
  FUN = function(x, y) ((x*exp(y))- (2*y*exp(-x)))^2, interx = c(-1.25, 1.25), intery=c(-1.25,1.25),
  val_ini=c(1,1), tasa=0.01, tope = 10^(-14), mimain="", maxiter=200){
    plot(NULL,NULL, xlim = interx, ylim=intery, xlab="x", ylab="y", main = mimain)
    pintar_grafica(FUN)
    coste_funcion=FUN(val_ini[1], val_ini[2])
    contador=0
    while(coste_funcion>tope && contador<maxiter){
      dxy_fs1e1 = deriv(as.expression(body(FUN)), c("x","y"), function.arg=TRUE)
      val_sig = val_ini - tasa*attr(dxy_fs1e1(val_ini[1],val_ini[2]), 'gradient')
      coste_funcion = FUN(val_sig[1], val_sig[2])
      val_ini = val_sig
      contador=contador+1
      points(val_sig, col="orange")
    }
    points(val_sig, col="red")
    print("Numero de iteraciones realizadas:")
    print(contador)
    print("Valor obtenido:")
    print(val_ini)
}
```

Lo primero que hago es definir tanto la función de este apartado como del siguiente. A continuación hago el método de gradiente descendente:

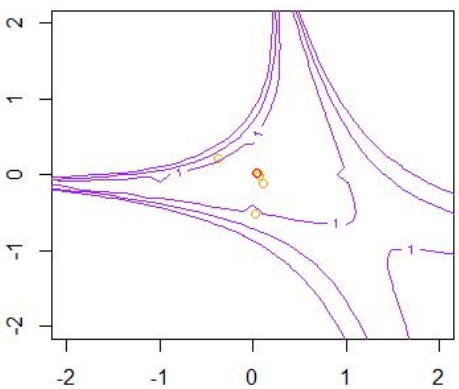
Representamos la función haciendo uso del método auxiliar pintar_grafica que usábamos en la práctica anterior y evaluamos la función en el punto de inicio val_ini=(1,1).

Mientras que la evaluación de la función en el punto que vamos obteniendo cada vez no llegue a ser menor a 10^{-14} (o bien superemos un límite de iteraciones) vamos a ir modificando el punto mínimo del siguiente modo:

Obtengo el gradiente de la función, derivando en x e y. Obtenemos el nuevo punto como resta del punto anterior y el gradiente*tasa. Volvemos a evaluar la función en dicho punto que opta a ser mínimo.

Resultados:

1.1.a)Primera funcion




```
"**********Primera funcion**********"
"Numero de iteraciones realizadas:"
10
"Valor obtenido:"
          x          y
] 0.04473629 0.02395871
```

b) Considerar ahora la función f(x, y) = x 2 + 2y 2 + 2 sin(2 π x) sin(2 π y)

1) Usar gradiente descendente para minimizar esta función.

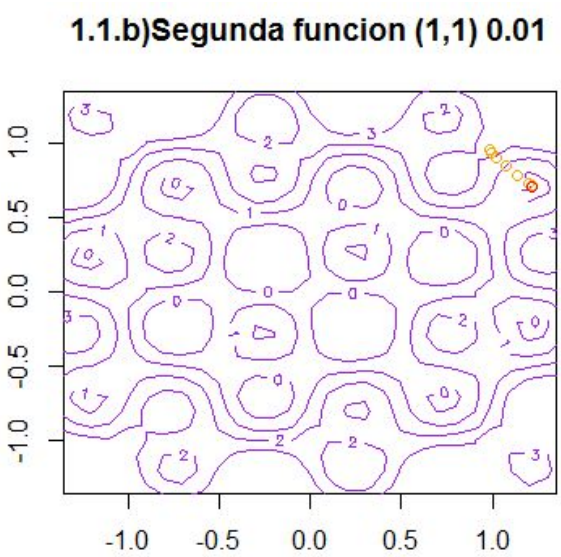
Usar como valores iniciales x0 = 1, y0 = 1, la tasa de aprendizaje η = 0,01 y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando η = 0,1, comentar las diferencias.

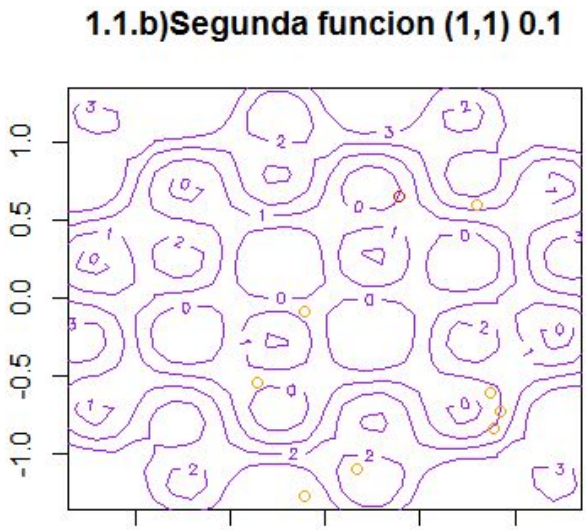
2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (0,1, 0,1), (1, 1),(−0,5, −0,5),(−1, −1). Generar una tabla con los valores obtenidos ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

<u>Solución:</u>

Hacemos uso de la función gradiente descendente que hemos implementado en el apartado anterior, modificando los parámetros según nos van pidiendo.

<u>Resultados 1:</u>

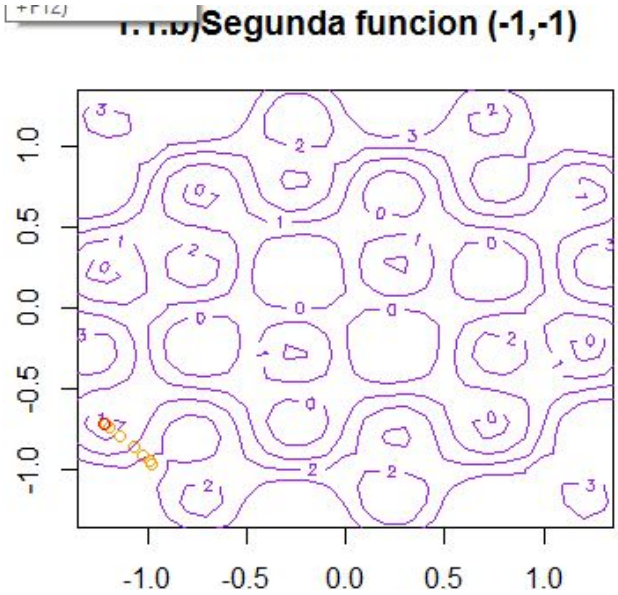


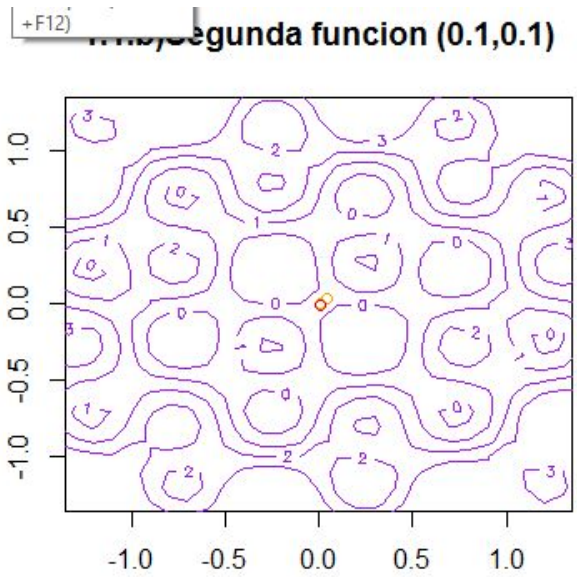


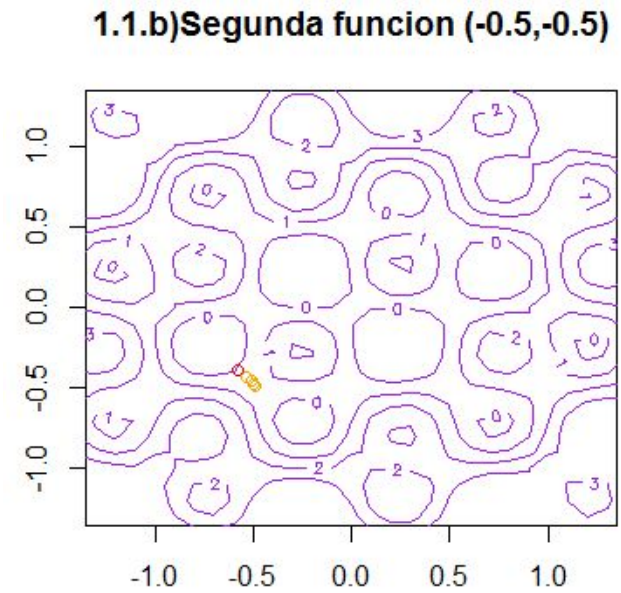

```
"************************Segunda funcion***
"************************Inicial (1,1)0.01*
"Numero de iteraciones realizadas:"
50
"Valor obtenido:"
        x         y
| 1.21807 0.712812
"************************Inicial (1,1) 0.1*
"Numero de iteraciones realizadas:"
10
"Valor obtenido:"
          x         y
| 0.3881225 0.6516421
```

Inicializando en el punto (1,1), vemos que los resultados son muy distintos en función a la tasa que imponemos. Con una tasa de 0.01, vemos que se realizan muchas iteraciones, de hecho de llega al máximo establecido, lo que nos indica que no se ha alcanzado el mínimo que se podría llegar a encontrar. Sin embargo, en la gráfica vemos que poco a poco va hacía un mínimo local, sin grandes oscilaciones por la superficie. Con una tasa de 0,1 encontramos un mínimo en 10 iteraciones.

Resultados 2:








```
"************************Inicial (-1,-1)**
"Numero de iteraciones realizadas:"
50
"Valor obtenido:"
         x          y
] -1.21807 -0.712812
"************************Inicial (0.1,0.1)**
"Numero de iteraciones realizadas:"
3
"Valor obtenido:"
            x             y
] 0.005266095 -0.002392069
"************************Inicial (-0.5,-0.5)*
"Numero de iteraciones realizadas:"
5
"Valor obtenido:"
           x          y
] -0.5803234 -0.3839919
```

Hemos usado una tasa de 0.01. Vemos que dependiendo de dónde comencemos nuestro algoritmo de gradiente descendente, llegamos un mínimo local u otro. Esta función tiene muchos mínimos locales y es normal que nuestro algoritmo encuentre unos u otros dependiendo de dónde comencemos. Además, en el caso vector_ini=(-1,-1) vemos que no conseguimos llegar un mínimo donde el valor de la función sea menor a 10^{14}

2. 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, tenemos dos pasos a lo largo de dos coordenadas.
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.

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

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

<u>Solución:</u>

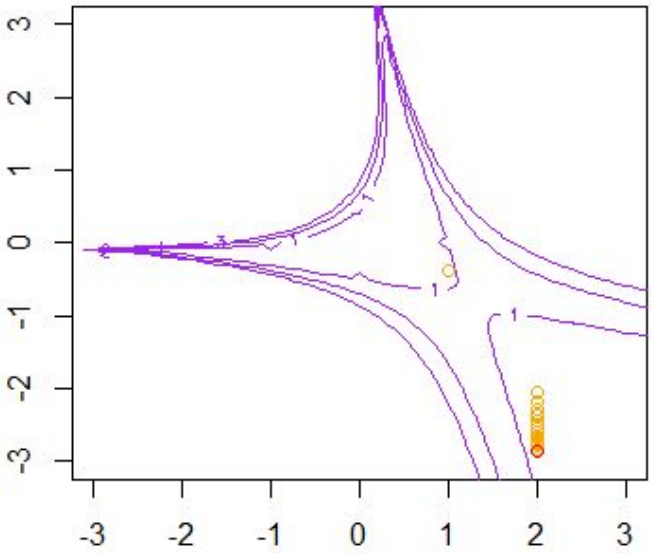
```
coorddesc = function(
  FUN = function(x, y) ((x*exp(y))- (2*y*exp(-x)))^2, interx = c(-3, 3), intery=c(-3,3),
  val_ini=c(1,1), tasa=0.1, tope = 10^(-14), mimain="", maxiter=15){
  plot(NULL,NULL, xlim = interx, ylim=intery, xlab="x", ylab="y", main = mimain)
  pintar_grafica(FUN)
  coste_funcion=FUN(val_ini[1], val_ini[2])
  contador=0
  val_sig = val_ini
  while(coste_funcion>tope && contador<maxiter){
    dxy_fs1e1 = deriv(as.expression(body(FUN)), c("x","y"), function.arg=TRUE)
    val_sig[1] = val_ini[1] - tasa*(attr(dxy_fs1e1(val_ini[1],val_ini[2]), 'gradient'))[1]
    val_sig[2] = val_ini[2] - tasa*(attr(dxy_fs1e1(val_sig[1],val_ini[2]), 'gradient'))[2]
    coste_funcion = FUN(val_sig[1], val_sig[2])
    val_ini = val_sig
    contador=contador+1
    points(val_sig, col="orange")
  }
  points(val_sig, col="red")
  print("Numero de iteraciones realizadas:")
  print(contador)
  print("Valor obtenido:")
  print(val_ini)
}
```

Dicho método es prácticamente análogo a gradiente descendente, con la diferencia de que el valor siguiente (punto que opta a ser mínimo) se va rectificando primero en coordenada x. Una vez ya tenemos la coordenada x, modificada, rectificamos la coordenada y haciendo uso de la misma expresión (coordenada -tasa*gradiente).

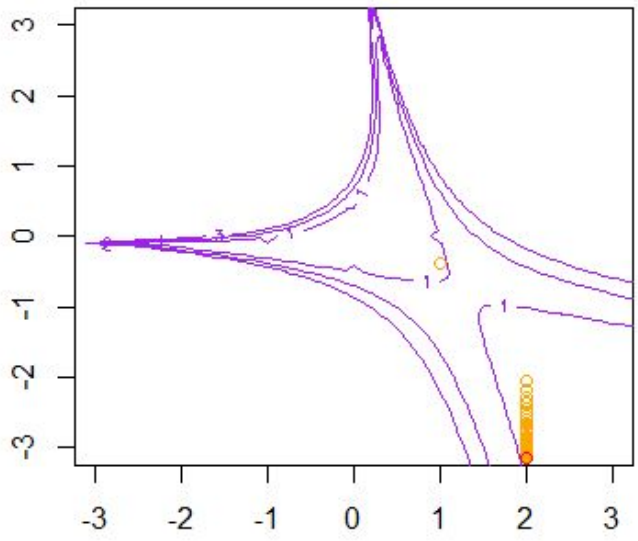
<u>Resultados:</u>

```
"##############################Ejercicio 2##
"************************15 Iteraciones*****
"Numero de iteraciones realizadas:"
15
"Valor obtenido:"
 6.297076 -2.852307
"************************30 Iteraciones*****
"Numero de iteraciones realizadas:"
30
"Valor obtenido:"
 6.259590 -3.152316
```

1.2. 15 Iteraciones.



1.2. 30 Iteraciones.



Vemos que en ambos casos tenemos un descenso bastante lineal hacia el mínimo. Al establecer 30 iteraciones, vemos que nos acercamos más al mínimo local, aunque con 15 iteraciones ya se intuía su posición.

En comparación con gradiente descendente, vemos que escapamos del mínimo local del origen que encontrábamos. Sin embargo, el mínimo que encontramos con la técnica actual, es mayor al mínimo que encontrábamos con gradiente descendente. Además, con el otro método llegábamos más rápidamente al mínimo.

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

- Generar un gráfico de cómo desciende el valor de la función con las iteraciones.
- Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.

<u>Solución:</u>

```
newton = function(
  FUN = function(x, y) x^2 +  2*y^2 +2*sin(2*pi*x)*sin(2*pi*y), interx = c(-2, 2), intery=c(-2,2),
  val_ini=as.matrix(rbind(1,1)), tasa=0.1, tope = 0.00001, mimain="", maxiter=15){

  plot(NULL,NULL, xlim = interx, ylim=intery, xlab="x", ylab="y", main = mimain)
  pintar_grafica(FUN)
  coste_funcion=FUN(val_ini[1], val_ini[2])
  contador=0
  val_sig = val_ini
  diferencia_coste = 200
  valores = coste_funcion
  while(diferencia_coste>tope && contador<maxiter){
    hess_fs1e1f1 = deriv(as.expression(body(FUN)), c("x","y"), hessian=TRUE, function.arg=TRUE)
    hessiana_fs1e1f1 = as.matrix(rbind(c((attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'hessian'))[1],
                                         (attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'hessian'))[2]),
                                       c((attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'hessian'))[3],
                                         (attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'hessian'))[4])))
    gradiente_fs1e1f1 = as.matrix(rbind((attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'grad'))[1],
                                        (attr(hess_fs1e1f1(val_ini[1],val_ini[2]), 'grad'))[2]))
    hessiana_fs1e1f1 = solve(hessiana_fs1e1f1)

    val_sig = val_ini -hessiana_fs1e1f1%*%gradiente_fs1e1f1
    diferencia_coste = abs(FUN(val_sig[1], val_sig[2]) - FUN(val_ini[1], val_ini[2]))
    val_ini = val_sig
    contador=contador+1
    valores =c(valores, FUN(val_sig[1], val_sig[2]))
  }
  points((val_sig[1]:val_sig[2]), col="red")
  print("Numero de iteraciones realizadas:")
  print(contador+1)
  print("Punto obtenido:")
  print(val_sig)
  print("Valor alcanzado")
  print(FUN(val_sig[1], val_sig[2]))
  iteraciones = 1:(contador+1)
  plot(x=iteraciones, y=valores, col="purple", main="Descenso valor función.")
```

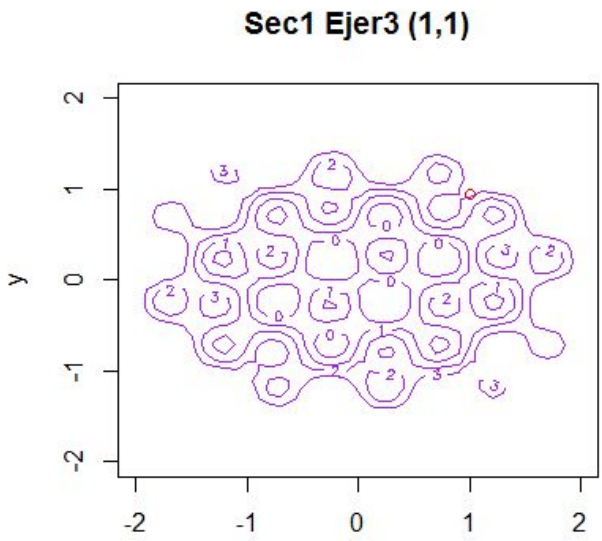
Pintamos la función que vamos a tratar de minimizar y obtenemos el valor de la función en el punto de inicio.

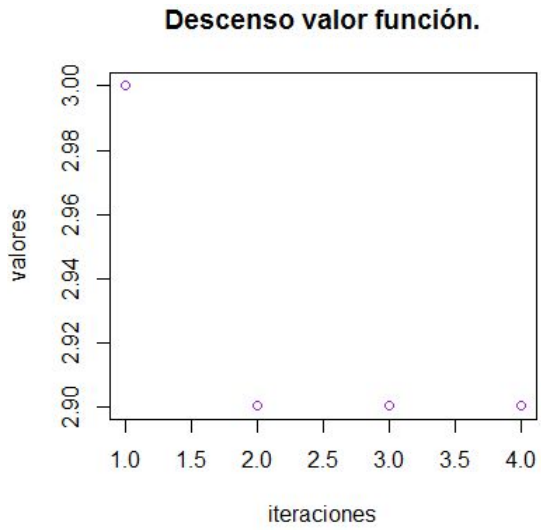
Mientras que la diferencia entre los valores que obtenemos al aplicar la función a dos puntos consecutivos obtenidos sea mayor que cierto tope (o no se supere un máximo de iteraciones), iremos buscando otro punto mínimo del siguiente modo:

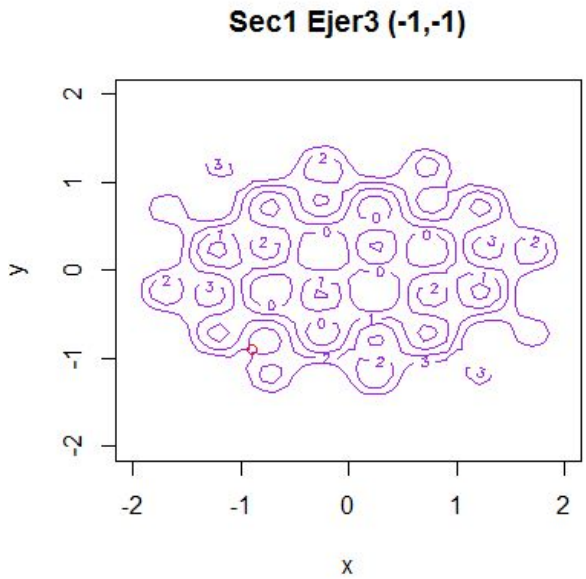
Obtenemos la hessiana (y su inversa) y el gradiente de la función. El punto que opta a ser mínimo se obtiene del punto anterior y la inversa_hessiana*gradiente

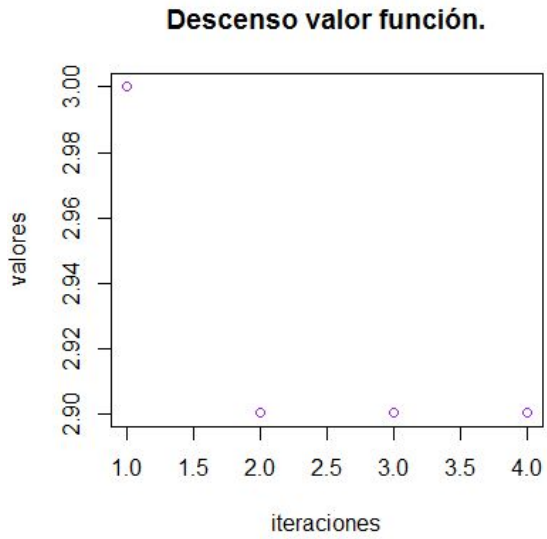
Vamos guardando los valores de la función en cada punto que opta a ser mínimo para finalmente poder representarlos en una gráfica.

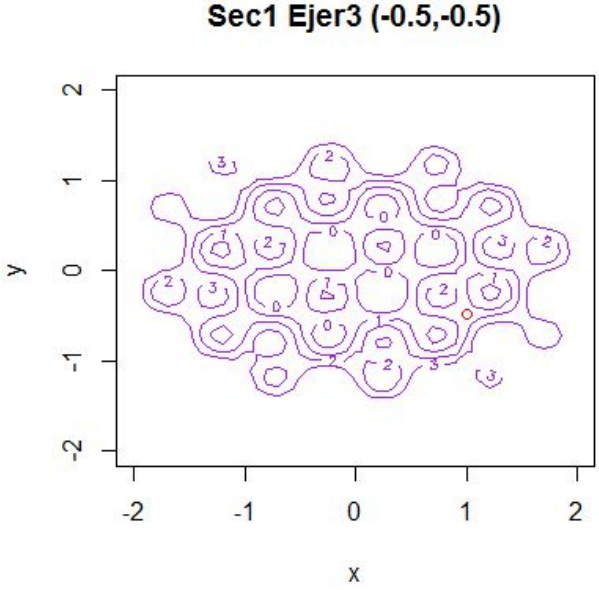
Resultados:

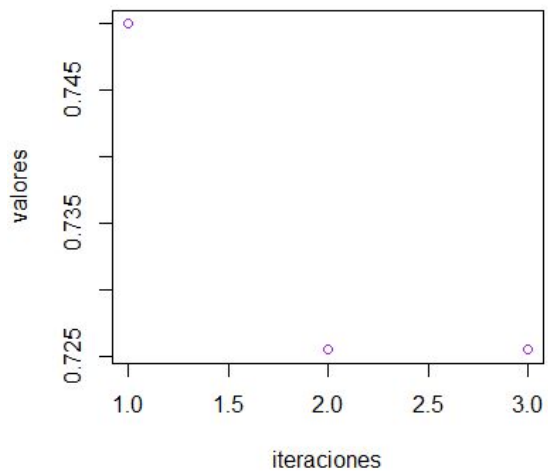

Sec1 Ejer3 (1,1)
y
x


Descenso valor función.
valores
iteraciones


Sec1 Ejer3 (-1,-1)
y
x


Descenso valor función.
valores
iteraciones


Sec1 Ejer3 (-0.5,-0.5)
y
x


Descenso valor función.
valores
iteraciones

```
"##############################Ejercicio 3##
"Newton partiendo de (1,1)"
"Numero de iteraciones realizadas:"
4
"Punto obtenido:"
      [,1]
] 0.9491289
] 0.9745675
"Valor alcanzado"
2.900405
"Newton partiendo de (-1,-1)"
"Numero de iteraciones realizadas:"
4
"Punto obtenido:"
      [,1]
] -0.9491289
] -0.9745675
"Valor alcanzado"
2.900405
"Newton partiendo de (-0.5,-0.5)"
"Numero de iteraciones realizadas:"
3
"Punto obtenido:"
      [,1]
] -0.4751135
] -0.4878047
"Valor alcanzado"
0.7254821
```

En comparación con gradiente descendente, vemos que el método de Newton se estanca más rápidamente en los mínimos locales, llegando a realizar muy pocas iteraciones. Al igual que con el otro método, al cambiar el punto de inicio, vamos obteniendo distintos mínimos locales que se encuentran en un entorno cercano al punto original.

4. Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (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 y es una función determinista de x. (...)

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

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

<u>Solución:</u>

Preparando los datos de trabajo:

```
N=100
rango = c(-1,1)
datos = simula_unifM (N,2,rango)
coef = simula_recta(rango)
f4 = function(pto,coef) {          #funcion de la recta
  pto[2]-pto[1]*coef[1]-coef[2]
}
z0 = apply(datos,1,f4,coef)        #obtiene los valores de la funcion para datos
etiqueta = sign(z0)                # apartir de ellos crea las etiquetas
pinta_puntos(datos, intervalo = rango,etiqueta=etiqueta) # utiliza las etiquetas
abline(coef[2],coef[1])            # pinta la recta
```

Lo primero que hacemos es crear un conjunto de datos uniformemente distribuidos en el intervalo (-1,1), crear una recta que divida el intervalo de trabajo y etiquetas las muestras con respecto a dicha recta (Como hacíamos en la práctica anterior)

Implementando RL+SGD

```
reglog = function(datos,etiqueta,vector_w_ini=c(0,0,0), tasa=0.01){
  N= dim(datos)[1]
  diferencia = 1
  w = vector_w_ini
  d = dim(datos)
  datos = cbind(rep(1,d[1]), datos)
  etiqueta = t(t(etiqueta))
  contador=0

  while(diferencia > 0.01 && contador<100000){
    vector_w = w
  permutacion = sample(1:N,N,replace=FALSE)
  for(i in 1:N){
    indice = permutacion[i]
    gradientelog = (-etiqueta[indice,]*datos[indice,])/(1+exp(etiqueta[indice,]*w%*%datos[indice,]))
    w = w - tasa*gradientelog
    contador=contador+1

  }
  diferencia = sqrt(sum((w-vector_w)^2))
  }
  return(w)
}
```

Lo primero será añadir una columna de 1's inicial a la matriz de los datos. Mientras que ||w(t−1) − w(t)|| > 0,01 (o bien se supere un máximo de iteraciones) vamos realizando el siguiente proceso:

Hacemos un aleatorio de N números (donde N es el número de datos de la muestra). En lugar de recorrer los datos en orden, vamos a seguir el orden que nos establezca dicha permutación, accediendo al dato (permutacion[i]) de la muestra de datos. Para dicho dato, hacemos el gradiente logístico y actualizamos el vector w como la resta de sí mismo y la tasa*gradiente. De este modo, para cada dato de la muestra vamos a ir modificando el vector w.

Resultados:

Podremos ver el funcionamiento en el siguiente apartado.

b) Usar la muestra de datos etiquetada para encontrar g y estimar Eout usando para ello un número suficientemente grande de nuevas muestras

Solución:

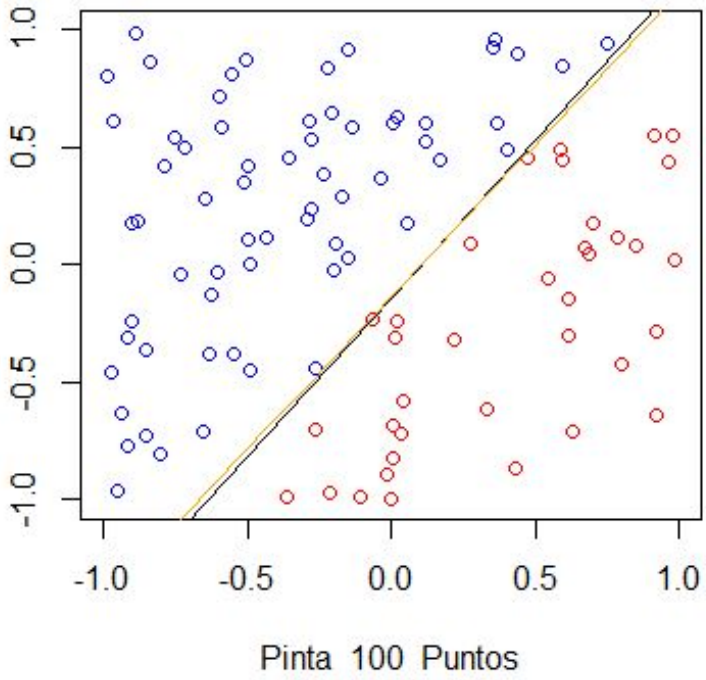
```
#Obtener g
a= -vector_final[2]/vector_final[3]
b= -vector_final[1]/vector_final[3]
curve(a*x + b, col="orange", add=T)

#Obtener Eout
nuevos_datos = simula_unifM (N,2,rango)
real_z0 = apply(nuevos_datos,1,f4,coef)          #obtie
etiquetas_real = sign(real_z0)                  # apart

coef_estimados = c(a,b)
estimado_z0 = apply(nuevos_datos,1,f4,coef_estimados)
etiquetas_estimadas = sign(estimado_z0)

error=mean(etiquetas_real!=etiquetas_estimadas)
print("Error obtenido regresion logistica:")
print(error)
```

A partir del vector w que obtenemos al hacer la regresión logística vista en el apartado anterior, podemos obtener la función g (g=wx)

Al igual que hacíamos con algoritmos como PLA, obtenemos los coeficientes de la recta g como a=-w2/w3 y b=-w1/w3 y pasamos a representarlos con curve. Ya tendríamos nuestra g que estima f.

A continuación obtenemos el error fuera de la muestra. Para ello obtengo otra muestra de datos (la he generado del mismo tamaño) y los etiquetamos con respecto a la función f original y la función g que hemos obtenido anteriormente. El error vendrá dado por las etiquetas que tengamos distintas.

Resultados:




```
"################################Ejercicio 3########
"################################Ejercicio 4########
"Error obtenido regresion logistica:"
0.01
```

Vemos que el error es muy pequeño pues la función g (en naranja) que estimamos se asimila bastante a la función f original (en negro). De modo que cuando llega otra muestra, la diferencia entre las etiquetas va a ser muy pequeña.

5. Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 1 y 5. 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.
Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g. Usando el modelo de Regresión Lineal para clasificación seguido por PLA-Pocket como mejora.
Responder a las siguientes cuestiones

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

Solución:

```
#Datos train
digit.train <- read.table("datos/zip.train", header=FALSE)
digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]
etiquetas_digitos_train = digitos15.train[,1]
ndigitos_train = nrow(digitos15.train)
matriz_Digitos_train = array(unlist(subset(digitos15.train,select=-V1)),c(ndigitos_train,16,16))
rm(digit.train)
rm(digitos15.train)

intensidad_train = apply(matriz_Digitos_train[1:ndigitos_train,,],1, mean)
simetria_train = apply(matriz_Digitos_train[1:ndigitos_train,,],1,fsimetria1)
datos_train = as.matrix(cbind(intensidad_train,simetria_train))
plot(datos_train,xlab="Intensidad Promedio",ylab="Simetria",main="SEC1:ejer5.train",
     col=etiquetas_digitos_train,pch=etiquetas_digitos_train+3)

#Regresion
hiperplanow = regresionlineal(datos_train, etiquetas_digitos_train)
hiperplanow = PLA_pocket(datos_train, etiquetas_digitos_train, vini=hiperplanow)
abline( a=-hiperplanow[1,]/hiperplanow[3,],
        b=-hiperplanow[2,]/hiperplanow[3,], col="orange")
```

Al igual que en la práctica anterior, obtenemos las muestras que representen 1's o 5's, las almacenamos en matriz_Digitos_train y obtenemos sus etiquetas (1 o 5) en etiquetas_digitos_train.

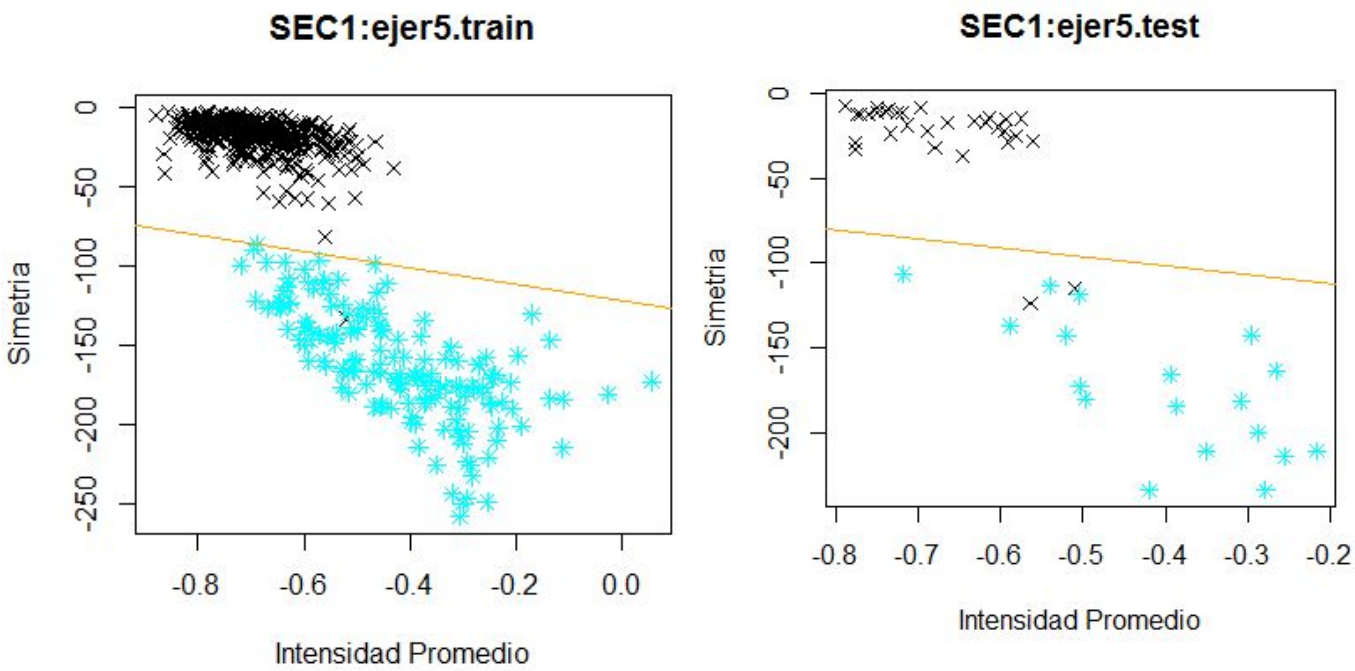
Obtenemos la intensidad y la simetría haciendo uso de la sfunciones de la práctica pasada y las almacenamos en una matriz. Será nuestra matriz de datos de trabajo. Los representamos y vemos los datos de entrenamieto. Ahora vamos a buscar la función estimada.

Para ellos hacemos regresionlineal (función que implementamos en la práctica anterior) sobre los datos y las etiquetas. Nos dará el hiperplano que pretende separar los datos. Hacemos PLA_pocket sobre los mismo datos, mismas etiquetas, pero esta vez el vector de inicio será el vector que hemos obtenido en regresión lineal. Ya tenemos el vector que representa la función estimada. Finalmente la mostramos con los datos train.

```
#Datos test
digit.test <- read.table("datos/zip.test", header=FALSE)
digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,]
etiquetas_digitos_test = digitos15.test[,1]
ndigitos_test = nrow(digitos15.test)
matriz_Digitos_test = array(unlist(subset(digitos15.test,select=-V1)),c(ndigitos_test,16,16))
rm(digit.test)
rm(digitos15.test)

intensidad_test = apply(matriz_Digitos_test[1:ndigitos_test,,],1, mean)
simetria_test = apply(matriz_Digitos_test[1:ndigitos_test,,],1,fsimetria1)
simetria_test=simetria_test #porque...
datos_test = as.matrix(cbind(intensidad_test,simetria_test))
plot(datos_test,xlab="Intensidad Promedio",ylab="Simetria",main="SEC1:ejer5.test",
    col=etiquetas_digitos_test,pch=etiquetas_digitos_test+3)
abline( a=-hiperplanow[1,]/hiperplanow[3,],
        b=-hiperplanow[2,]/hiperplanow[3,], col="orange")
```

Seguidamente, realizamos el mismo proceso con los datos test. Obtenemos la matriz de datos de intensidad y simetría, así como las etiquetas de los datos test. Representamos los datos y la función que estimamos con el conjunto de datos train.

<u>Resultados:</u>



En el conjunto train vemos que la función estimada es muy buena, pues parece ser que sólamente hay un dato mal etiquetado con respecto a dicha recta y además no es posible etiquetarlo correctamente sin empeorar el resultado general.

Al llegar el nuevo conjunto de datos, datos test, vemos que la recta estimada sigue etiquetando muy bien los datos pues parece ser que sólamente hay dos datos mal etiquetados y tampoco sería posible etiquetarlos bien sin empeorar el resultado global.

Por tanto, vemos que la estimación es bastante buena.

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

Solución:

```
#Para datos_train:
coefw = c(-hiperplanow[1,]/hiperplanow[3,],-hiperplanow[2,]/hiperplanow[3,])
f4 = function(pto,coefw) {          #funcion de la recta
  pto[2]-pto[1]*coefw[2]-coefw[1]
}
z0trainfin = apply(datos_train,1,f4,coefw)        #obtiene los valores de la
etiquetas_train_fin = sign(z0trainfin)                # apartir de ellos crea
for(i in 1:length(etiquetas_train_fin)){
  if(etiquetas_train_fin[i] == -1)
    etiquetas_train_fin[i] =5
}
distintas_train = length(which(etiquetas_train_fin!=etiquetas_digitos_train))
Ein_train = (distintas_train*100)/nrow(datos_train)
print("Ein en train:")
print(Ein_train)

#Para datos_test:
z0testfin = apply(datos_test,1,f4,coefw)        #obtiene los valores de la fu
etiquetas_test_fin = sign(z0testfin)               # apartir de ellos crea
for(i in 1:length(etiquetas_test_fin)){
  if(etiquetas_test_fin[i] == -1)
    etiquetas_test_fin[i] =5
}
distintas_test = length(which(etiquetas_test_fin!=etiquetas_digitos_test))
Ein_test = (distintas_test*100)/nrow(datos_test)
print("Ein en test")
print(Ein_test)
```

Comenzamos calculando Ein en los datos de entrenamiento. Para ello etiqueto las muestras en base a la función estimada anteriormente. Obtengo el número de muestras que hay mal etiquetas con respecto al etiquetado original. Finalmente hago el porcentaje.

Para los datos test, realizo el mismo proceso.

Resultados:

```
"################################Ejercicio 5##
"Ein en train:"
0.1669449
"Ein en test"
4.081633
```

El porcentaje de error es muy bajo, tal y como era de esperar al ver las gráficas del apartado anterior. En los datos de entrenamiento tenemos un error de 0,16%, mientras que en los datos test tenemos un 4%. Es normal que en los datos test sea mayor, aunque el resultado sigue siendo muy bueno.

1. (2 puntos) Sobreajuste. Vamos a construir un entorno que nos permita experimentar con los problemas de sobreajuste. Consideremos el espacio de entrada X = [−1, 1] con una densidad de probabilidad uniforme, P(x) = 1/2 . Consideramos dos modelos H2 y H10 representando el conjunto de todos los polinomios de grado 2 y grado 10 respectivamente.
La función objetivo es un polinomio de grado Qf que escribimos como:

$$f(x) = \sum_{q=0}^{Q_f} a_q L_q(x)$$

donde Lq(x) son los polinomios de Legendre (ver la relación de ejercicios.2).
El conjunto de datos es D = {(x1, y1), . . . ,(xN , yN)} donde yn = f(xn) + $\sigma\, \varepsilon n$ y las εn son variables aleatorias i.i.d. N (0, 1) y σ^2 la varianza del ruido.
Comenzamos realizando un experimento donde suponemos que los valores de Qf , N, σ , están especificados, para ello:

- Generamos los coeficientes a_q a partir de muestras de una distribución N (0, 1) y escalamos dichos coeficientes de manera que $E_{a,x}[f^2] = 1$
- Generamos un conjunto de datos, x1, . . . , xN muestreando de forma independiente P(x) y los valores yn = f(xn) + $\sigma\, \varepsilon n$.

Sean g2 y g10 los mejores ajustes a los datos usando H2 y H10 respectivamente, y sean Eout(g2) y Eout(g10) sus respectivos errores fuera de la muestra.

a) Calcular g2 y g10

b) ¿Por qué normalizamos f? (Ayuda: interpretar el significado de σ)

c) (Opc)¿Cómo podemos obtener Eout analíticamente para una g10 dada ?

<u>Solución:</u>

```
sobreajusteg2g10 <- function(Qf=4, N=50, sigma=0.5){
  #Obtengo los polinomios de Lagrange normalizados
  pol.Leg.Normalizados <- legendre.polynomials(n= Qf-1, normalized=T)
  #normalizo los coeficientes
  a = simula_gaus(N=Qf,1,mean=0, sd=1)
  a=a/sqrt(sum(a^2))

  #obtengo la función f(x)
  fun_f=0
  for(i in 1:Qf){
     fun_f = fun_f + pol.Leg.Normalizados[[i]]*a[i]
  }
  #print(integral(fun_f^2,limits=c(-1,1))) #Compruebo que está bien normalizado

  #Añado error a las muestras
  epsilon = simula_gaus(N=N,1,mean=0, sd=1)
  X = simula_unifM (N=N,1,rango=c(-1,1))
  f = as.function((fun_f))
  Y = f(X) + sigma*epsilon
  plot(X, Y, xlim=c(-1,1), main="Sec1 Ejer1:Ajusto con g2 y g10")
  curve(expr=f, add=T, col="red", lw=2)

  #Ya tengo la función y los datos. Ahora tengo que encontrar g2 y g10
  datos2 = as.matrix(cbind(X,X^2))
  hiperplanow2= reg_lineal(datos2,Y)
  curve(hiperplanow2[3,]*x^2 + hiperplanow2[2,]*x + hiperplanow2[1,], add=T,
        col="orange", lw=2, lty=3)

  datos10 = as.matrix(cbind(X,X^2,X^3,X^4,X^5,X^6, X^7,X^8,X^9,X^10))
  hiperplanow10= reg_lineal(datos10,Y)
  curve(hiperplanow10[11,]*x^10 + hiperplanow10[10,]*x^9 +
        hiperplanow10[9,]*x^8 + hiperplanow10[8,]*x^7 + hiperplanow10[7,]*x^6+
        hiperplanow10[6,]*x^5 + hiperplanow10[5,]*x^4 + hiperplanow10[4,]*x^3 +
        hiperplanow10[3,]*x^2 + hiperplanow10[2,]*x + hiperplanow10[1,], add=T,
        col="blue", lw=2, lty=3)
```

En primer lugar, vamos a obtener los polinomios de Legendre. Para ello hacemos uso de la librería orthopolynom: library(orthopolynom)
Obtenemos los Qf primeros polinomios de Legendre de forma que estén normalizados. Los almacenamos en pol.Leg.Normalizados:

```
> pol.Leg.Normalizados = legendre.polynomials(n=3, normalized=T)
> pol.Leg.Normalizados
[[1]]
0.7071068

[[2]]
1.224745*x

[[3]]
-0.7905694 + 2.371708*x^2

[[4]]
-2.806243*x + 4.677072*x^3
```

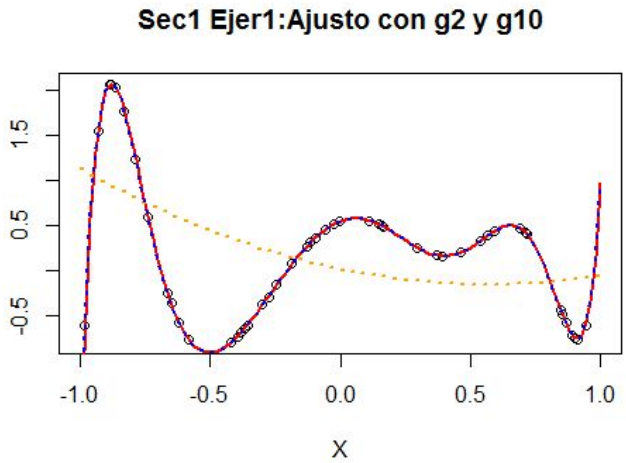
A continuación buscamos que $E_{a,x}[f^2] = 1$, es decir, $E_{a,x}[\sum_{q=o}^{Qf} (a_q L_q(x))^2] = 1$.

Visto de forma más directa vemos que buscamos que:
$\int_{-1}^{1} \sum_{q=o}^{Qf} (a_q L_q(x))^2 = 1$. Como ya tenemos los polinomios de Legendre normalizados, lo que buscamos es que $\sum_{q=o}^{Qf} a_q^2 = 1$. Para ello basta obtener la raiz de la suma de todos los a_i y dividir cada uno de ellos por dicha suma. Ya tenemos los coeficientes y los polinomios de Legendre normalizados, de modo que calculamos $f = a_q L_q(x)$. Finalmente comprobamos que la integral es efectivamente 1 y tenemos la función f bien obtenida.

Ahora vamos a generar N muestras y vamos a añadirle error. Simplemente genero N valores uniformemente distribuidos en el intervalo de trabajo (-1,1) que serán nuestras x_n y genero N valores de la distribución N(0,1) que representarán el ruido en cada punto (εn). Obtengo y_n como yn = f(x_n) + $\sigma\ \varepsilon n$. Represento la muestra de datos y la función real f.

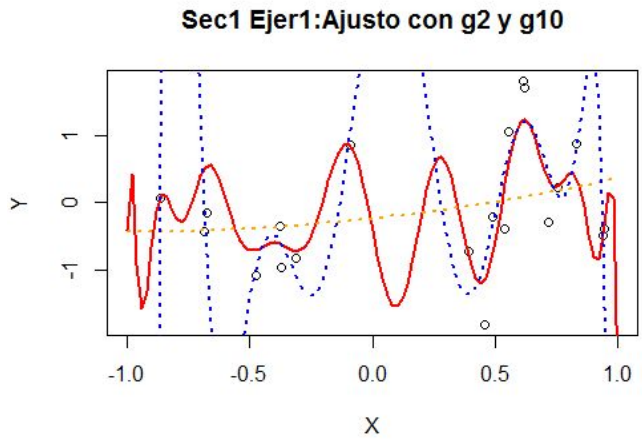
A continuación, vamos a obtener g2 y g10.
g2: Concatenamos la matriz de datos X(sería nuestro x_n) con X^2 y consideramos los valores Y (sería nuestro y_n) como las etiquetas de la muestra. Aplicamos regresión lineal a dicha matriz de datos con dichas etiquetas y obtenemos un vector de coeficientes que nos permitirá obtener la función g2. Para visualizarla, simplemente multiplicamos cada uno de los coeficientes con su término del polinomio.

g10: realizamos el mismo proceso pero con la matriz de datos llevada a dimensión;10
<u>Resultados:</u>
Voy a probar con algunos valores de Qf, N y sigma para ver correcto funcionamiento:
Por ejemplo usando sobreajusteg2g10(Qf=10, N=50, sigma=0)



En este caso al tomar sigma=0 hacemos que los datos no tenga ruido, de modo que los puntos pasan exactamente por la función f original (que visualizamos en rojo). Como es de esperar, la función g2 (visualizada en naranja) intenta ajustar la nube de puntos, pero pocos de ellos se sitúan en g2. Además vemos que la función g10 (visualizada en azul) ajusta perfectamente la nube de puntos pues no hay error y está ajustando a un conjunto de puntos que han sido generados a partir de una función con polinomio de grado 10.

Usando sobreajusteg2g10(Qf=20, N=20, sigma=0.5)



Añadimos ruido a las muestras y las generamos a partir de una función de grado 20. En este caso g10 trata de ajustarse a los puntos creando grandes oscilaciones, lo que posteriormente nos va llevar a grandes errores fuera de la muestra. g2 sigue siendo suave y ajustando bien, dentro de lo que cabe, la muestra.

b) ¿Por qué normalizamos f? (Ayuda: interpretar el significado de σ)

<u>Solución:</u>

Normalizamos f para que el valor σ^2 tenga realmente sentido. Al normalizar, σ^2 nos muestra realmente la cantidad de ruido que hay en las muestras.

2. Siguiendo con el punto anterior, usando la combinación Qf = 20, N = 50, σ = 1 ejecutar un número de experimentos (>100) calculando en cada caso Eout(g2) y Eout(g10). Promediar todos los valores de error obtenidos para cada conjunto de hipótesis, es decir

Eout(H2) = promedio sobre experimentos(Eout(g2))

Eout(H10) = promedio sobre experimentos(Eout(g10)

Definimos una medida de sobreajuste como Eout(H10) – Eout(H2).

a) Argumentar por qué la medida dada puede medir el sobreajuste.

b) (Opcional) Usando la combinación de valores de los valores Qf $\in$ {1, 2, . . . , 100}, N $\in$ {20, 25, . . . , 100}, σ $\in$ {0; 0,05; 0,1; . . . ; 2}, se obtiene una gráfica como la que aparece en la figura 4.3 del libro "Learning from data", capítulo 4. Interpreta la gráfica respecto a las condiciones en las que se da el sobreajuste. (Nota: No es necesario la implementación).

<u>Solución:</u>

Veamos cómo calculamos Eout:

```
#Error g2
fung2 = as.function(as.polynomial(c(hiperplanow2[1,], hiperplanow2[2,],hiperplanow2[3,])))
Eg2_real=(sum((fung2(X)-Y)^2))/N
print("Error con la muestra original usando g2")
print(Eg2_real)

nueva_X = simula_unifM (N=N,1,rango=c(-1,1))
nuevo_epsilon = simula_gaus(N=N,1,mean=0, sd=1)
nueva_Y = f(nueva_X) + sigma*nuevo_epsilon
Eg2_fuera=(sum((fung2(nueva_X)-Y)^2))/N
print("Error con muestra nueva usando g2")
print(Eg2_fuera)

#Error g10
fung10 = as.function(as.polynomial(c(hiperplanow10[1,], hiperplanow10[2,],
                                     hiperplanow10[3,], hiperplanow10[4,],
                                     hiperplanow10[5,], hiperplanow10[6,],
                                     hiperplanow10[7,], hiperplanow10[8,],
                                     hiperplanow10[9,], hiperplanow10[10,],
                                     hiperplanow10[11,])))
Eg10_real=(sum((fung10(X)-Y)^2))/N
print("Error con la muestra original usando g10")
print(Eg10_real)

print("Error con muestra nueva usando g10")
Eg10_fuera=(sum((fung10(nueva_X)-Y)^2))/N
print(Eg10_fuera)
print("**************************************")
return(c(Eg2_fuera,Eg10_fuera))
```

Creamos una nueva muestra de datos (nueva_X) con la que obtener Eout.

Por hacer más comparativas, hemos obtenido también el error que produce g2 y g10 con las muestras originales. El proceso es siempre el mismo:

Obtenemos la diferencia cuadrática entre la imagen por la función original y la imagen por la función g2/g10 ajustada sobre cada uno de los datos y obtenemos el promedio. Ya tendríamos los errores de g2 y de g10 para el caso particular que le pasemos a la función.

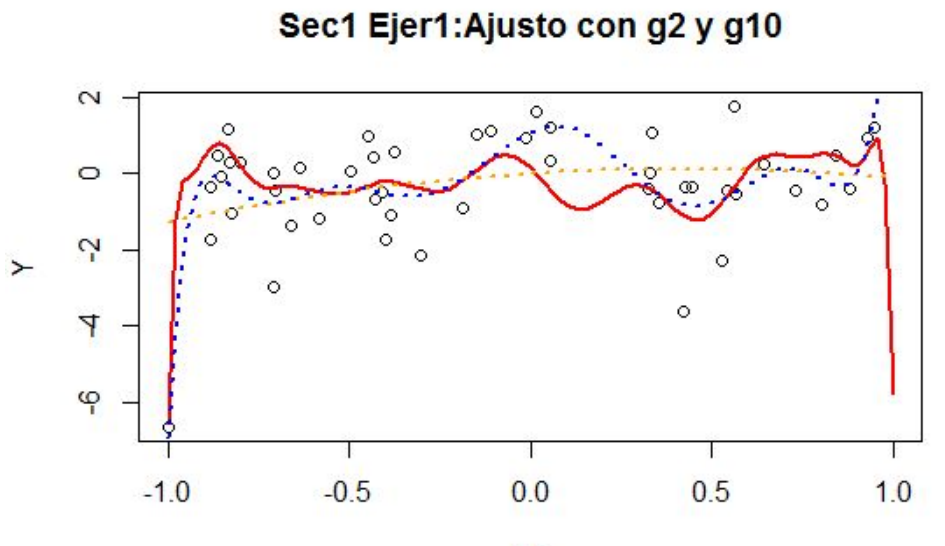
Ahora vamos a hacer el experimento que nos proponen:

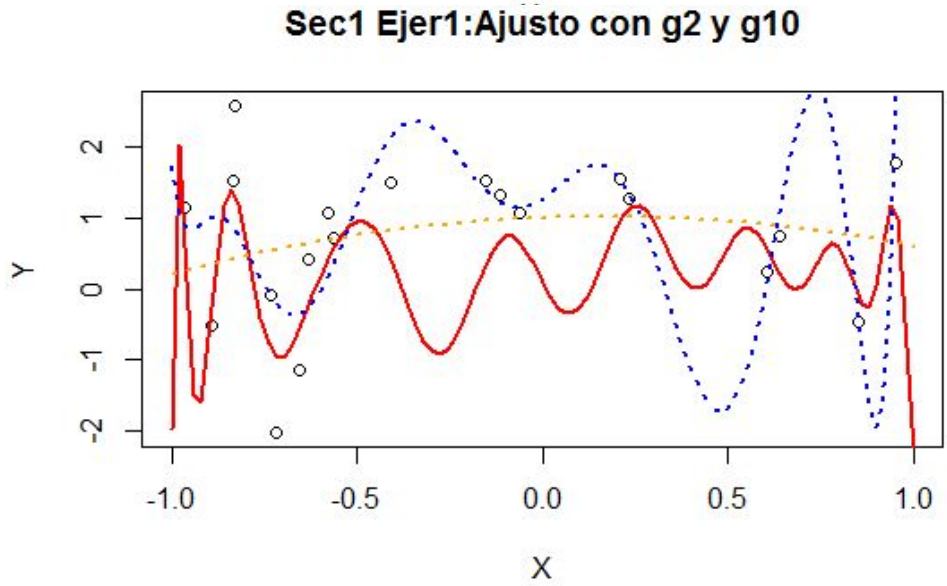
```
errores = sobreajusteg2g10(Qf=20, N=50, sigma=1)
#Hago sobreajuste 400 veces
limite=399
for(i in 1:limite){
   errores = rbind(errores,sobreajusteg2g10(Qf=20, N=50, sigma=1))
}

#Calculo la media de Eout para g2 y g10
media_errores = apply(errores,2,sum)/(limite+1)
print("Media de Eout para g2 y para g10")
print(media_errores)

sobreajuste=media_errores[2]-media_errores[1]
print("Sobreajuste")
print(sobreajuste)
```

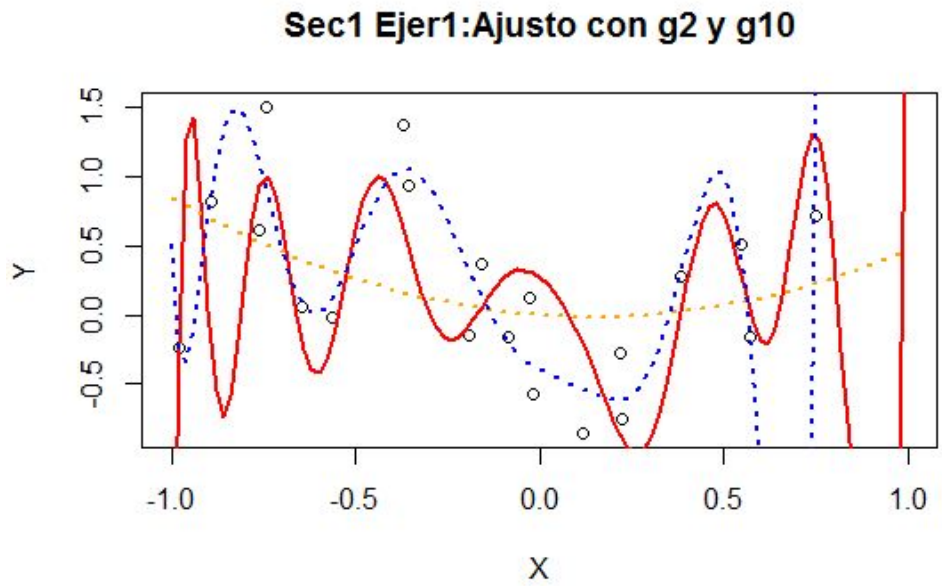
Hacemos el experimento 400 veces obteniendo los Eout para g2 y para g10. Hacemos la media y los mostramos. Finalmente obtenemos el sobreajuste.

<u>Resultados:</u>

Primero vamos a ver otro experimento que hemos realizado con sólo 3 iteraciones para que veamos también los ajustes:








```
"##############################Ejercicio 2###
"Error con la muestra original usando g2"
1.955072
"Error con muestra nueva usando g2"
2.265785
"Error con la muestra original usando g10"
0.9161099
"Error con muestra nueva usando g10"
3.93093
"######################################"
"Error con la muestra original usando g2"
1.120736
"Error con muestra nueva usando g2"
1.224672
"Error con la muestra original usando g10"
0.5960733
"Error con muestra nueva usando g10"
11.24408
"######################################"
"Error con la muestra original usando g2"
0.3418291
"Error con muestra nueva usando g2"
0.374043
"Error con la muestra original usando g10"
0.07569977
"Error con muestra nueva usando g10"
5778.991
"######################################"
"Media de Eout para g2 y para g10"
   1.288167 1931.388675
"Sobreajuste"
1930.101
```

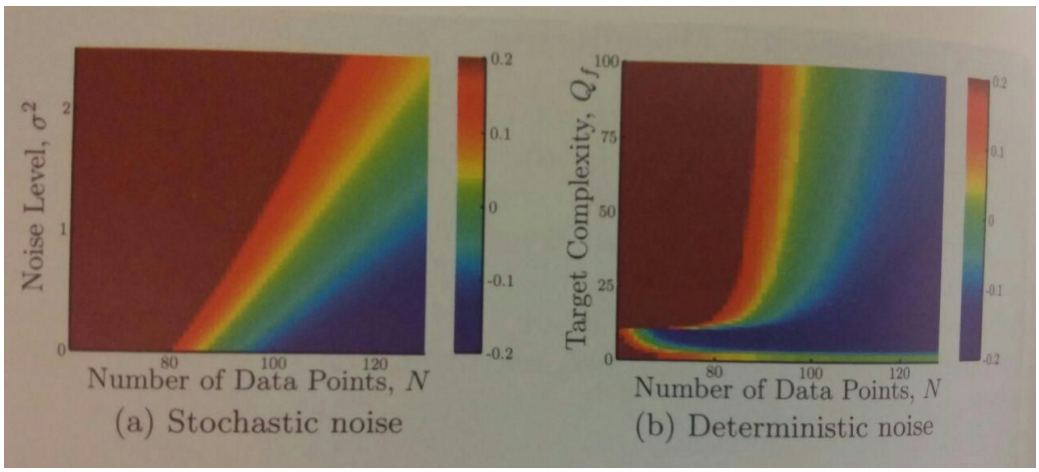
Visualmente se observa que g10 ajusta bastante bien a la muestra original, obteniendo un error con la muestra original usando g10 muy bajo en los tres casos. Sin embargo, al entrar una muestra nueva, obtenemos Eout muy grandes para g10.

Con respecto a g2, tenemos errores mayores a los que obtiene g10 dentro de la muestra (era de esperar porque estamos dando menos "flexibilidad" a nuestro ajuste), pero al obtener muestras nuevas, el error no se dispara tanto porque no tenemos oscilaciones tan grandes como con g10.

Veamos los errores para el experimento con 400 iteraciones:

```
"****************************************"
"Media de Eout para g2 y para g10"
8.395629e-01 2.456793e+06
"Sobreajuste"
2456792
```

Al igual que antes, el error fuera de la muestra para g2 es aceptable pero para g10 es inadmisible, provocando un sobreajuste muy grande.



Al ver estas gráficas vemos que podemos tener más o menos sobreajuste en función del ruido, la complejidad de la función Qf y del número de puntos que usamos como muestra. Al tener un mayor número de puntos en la muestra, vamos a conseguir un menor sobreajuste ya que los puntos tienen menos fuerza para condicionar a la función que está haciendo el ajuste. A más puntos, mayor conocimiento y menor sobreajuste.
Al tener más ruido vamos a tener más sobreajuste ya que las funciones se van a ajustar a unas muestras que no son del todo reales, que tienen ruido, de modo que estamos ajustando el ruido y al llegar una nueva muestra con distintos ruido el ajuste no va a ser bueno.
Al tener una función con mayor complejidad, vamos a provocar más sobreajuste. Esto se debe a que la función va a producir muchas oscilaciones para tratar de ajustar la muestra original, y al llegar una muestra nueva, estas oscilaciones van a producir unos errores cuadráticos muy altos.