Aprendizaje Automático: Práctica 2

Braulio Vargas López

1 de abril de 2016

Índice

1. Modelos Lineales

1.1. Gradiente Descendente. Implementar el algoritmo de gradiente descendiente.

a) Considerar la función no lineal de error $E(u,v) = (ue^v - 2ve^{-u})^2$. Usar gradiente descendente y minimizar esta función de error, comenzando desde el punto (u, v) = (1, 1) y usando una tasa de aprendizaje $\eta = 0,1$.

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

2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{-14} . (Usar flotantes de 64 bits).

3) ¿Qué valores de (u,v) obtuvo en el apartado anterior cuando alcanzo el error de 10^{-14}.

b) Considerar ahora la función $f(x,y) = x^2 + 2y^2 + 2 \cdot \sin(2\pi x)\sin(2\pi y)$.

***a*) Usar gradiente descendente para minimizar esta función. Usar como valores iniciales $x_0 = 1$, $y_0 = 1$, la tasa de aprendizaje $\eta = 001$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la funión con las iteraciones. Repetir el experimento pero usando $\eta = 01$, comentar las diferencias.**

***b*) 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?**

Para realizar el gradiente de la función del apartado 1, tenemos que realizar las derivadas parciales de la función $E(u,v) = (ue^v - 2ve^{-u})^2$. Para ello, derivamos en función de u y en función de v, obteniendo las siguientes expresiones:

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

$$E'(v) = 2(ue^v - 2ve^{-u}) \cdot (ue^v - 2e^{-u})$$

Una vez que tenemos las derivadas, podemos hacer una implementación del gradiente descendiente para esta función, usando las derivadas calculadas anteriormente, junto con la función inicial. Esta versión la podemos ver a continuación: \end{enumerate}

```
du <- function(u,v) 2*(u*exp(v) - 2*v*exp(-u))*(exp(v)+2*v*exp(-u))
dv <- function(u,v) 2*(u*exp(v) - 2*v*exp(-u))*(u*exp(v)-2*exp(-u))
fo <- function(u,v) (u*exp(v)-2*v*exp(-u))*(u*exp(v)-2*v*exp(-u))
fb <- function(x,y) x*x +2*y*y + 2*sin(2*PI*x)*sin(2*PI*y)

gradienteDescAnalitico <- function(x = 1,y = 1, eta = 0.1,
    prec=10^(-14), maxIter = 50, showIter = F){
    xOld = 0
    yOld = 0
    nIter = 0
    # Mientras que no se haya alcanzado el error máximo que queremos o
    # no se haya llegado al número total de iteraciones
    while (abs(du(x,y)) > prec & nIter < maxIter &
        abs(xOld - x) > prec){
        # Guardamos los valores de X e Y
        xOld = x
        yOld = y
        # Actualizamos los valores de X e Y con las derivadas de su función
        # junto con el factor de aprendizaje
        x = xOld - eta*du(xOld,yOld)
        y = yOld - eta*dv(xOld,yOld)
        # Incrementamos el número de iteraciones
        nIter = nIter+1
    }
    # Si no queremos ver el número de iteraciones, devolverá
    # el valor de la función en el punto, junto con el
    # valor de X y el valor de Y
    if(!showIter)
        c(fo(x,y),x, y)
    # Si queremos ver el número de iteraciones, realiza lo mismo que antes
    # pero devolviendo como último valor el número de iteraciones
    else
        c(fo(x,y),x, y, nIter)
}
```

En el resultado de la función, podemos ver el valor del punto en las coordenadas x e y en la función, junto con los valores de x e y.

Para ver el número de iteraciones y el resultado del gradiente descendiente, podemos llamar a la función con el parámetro `showIter` con valor *True* y obtendremos un resultado como el que se ve a continuación:

```
gradienteDescAnalitico(showIter = T)
```

```
## [1] 1.850819e-31 4.473628e-02 2.395873e-02 1.700000e+01
```

Además, para comprobar que se ha calculado bien, existe un paquete llamado `Deriv`, que se usa para derivar funciones. Usando esto, he creado la siguiente función que recibe una función y la minimiza usando gradiente descendiente, gracias al paquete `Deriv`. \end{enumerate}

```
gradDesc <- function(x = 1,y = 1, eta = 0.1, func,
    prec=10^(-14), maxIter = 50, showIter = F,
    pintarGr=F, nombre="Gradiente Descendente"){
    df = Deriv(f=func,x=formalArgs(func))
    xOld = 0
    yOld = 0
    nIter = 0
    # Estas listas contendrán los puntos que va encontrando
    # el gradiente durante el algoritmo
    xs = c()
    ys = c()
    while (abs(df(x,y)[1]) > prec & nIter < maxIter &
        abs(x - xOld) > prec){
        xOld = x
        yOld = y
        # Insertamos los puntos al final
        xs = c(xs, x)
        ys = c(ys, y)
        newValues = df(xOld,yOld)
        x = xOld - eta*newValues[1]
        y = yOld - eta*newValues[2]
        nIter = nIter+1
    }
    # Generamos la matriz de puntos
    if(pintarGr){
      # Si el flag pintarGr está a True, se pintan los puntos
      pts = cbind(xs,ys)
      pintar(puntos = pts, funcion = func, intervalo=c(-2,2), colores="red",
            verFuncion=T, nombreGrafica=nombre, k=1:20)
    }

    if(!showIter)
        c(func(x,y),x, y)
    else
        c(func(x,y),x, y, nIter)
}
```

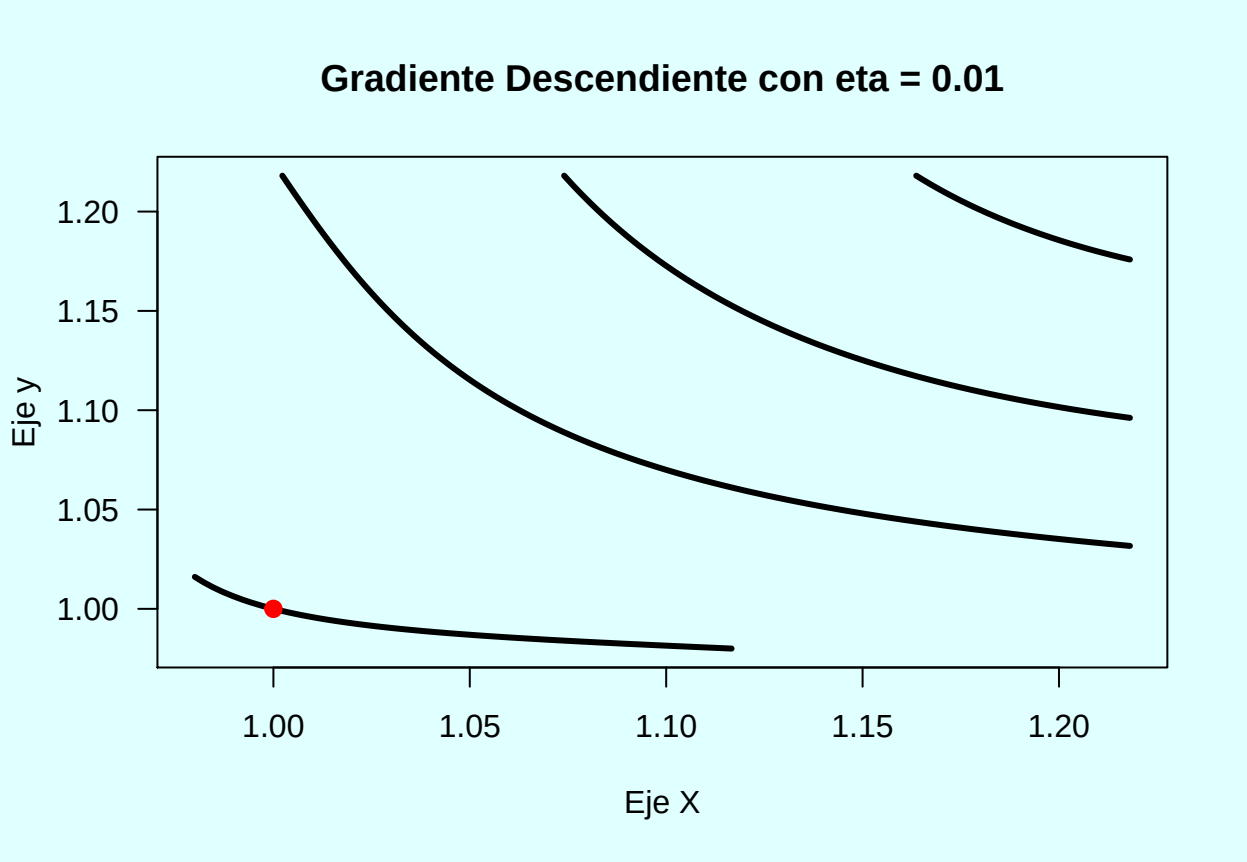
Para ejecutar este algoritmo, realizamos la siguiente llamada para calcular el gradiente a la función anterior y como podemos ver, el resultado es el mismo, y converge en el mismo número de iteraciones.

```
gradDesc(func=fo,prec=10^(-14),showIter = T)
```

```
##            u            u            v
## 1.850819e-31 4.473628e-02 2.395873e-02 1.700000e+01
```

Para el apartado b, usaremos la función `gradDesc`, donde el parámetro `func` recibe la función a minimizar, y podremos mostrar los puntos que encuentra el gradiente si queremos con el flag `pintarGr` valiendo *True*. Una vez hecho esto, pasamos a ver los valores que encuentra el gradiente para un $\eta = 01$ y un $\eta = 001$.

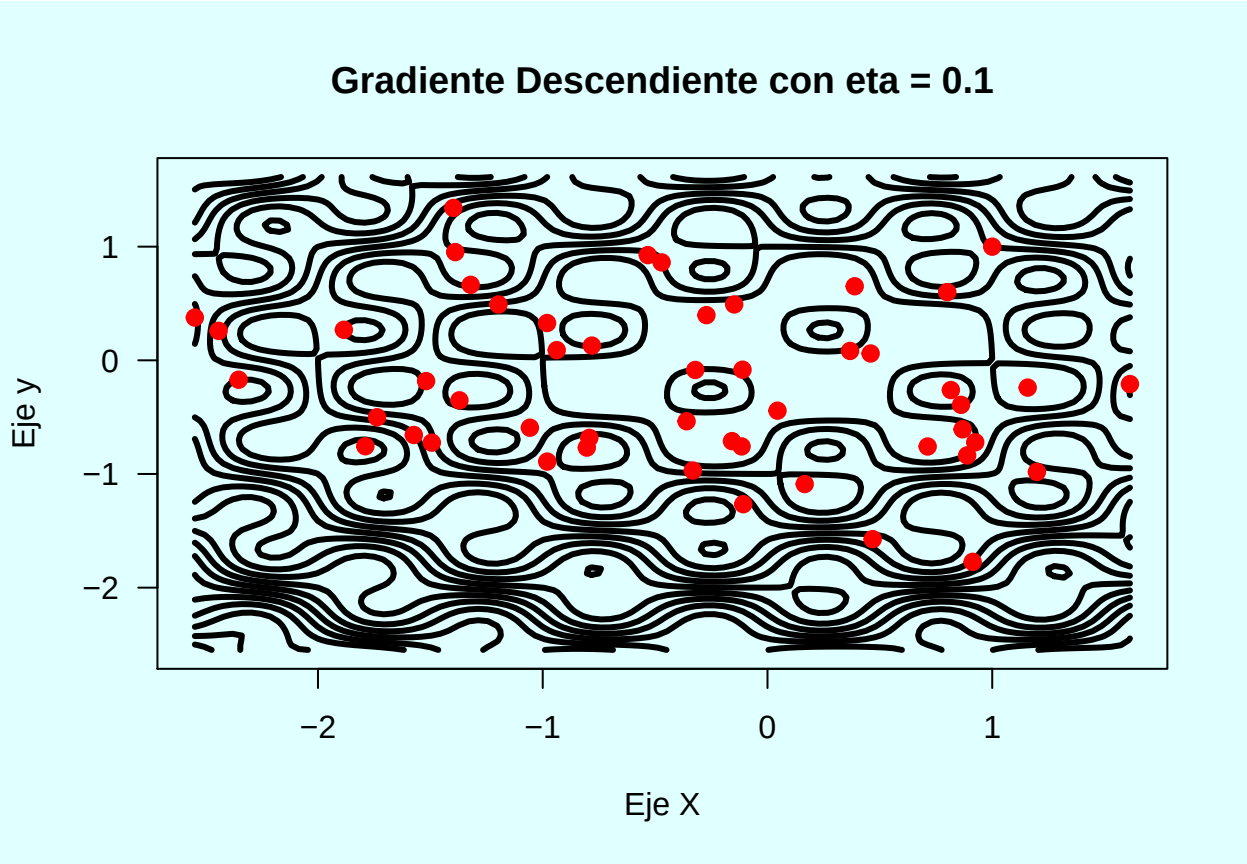
```
gradDesc(func=fb,eta = 0.01,pintarGr=T,nombre="Gradiente Descendiente con eta = 0.01")
```




```
##         x         x         y
## 0.5932694 1.2180703 0.7128120
```

En esta gráfica, podemos ver cómo el gradiente ha convergido y ha encontrado un mínimo, aunque haya sido un mínimo local. Esto se debe a que el valor de η es muy pequeño, y el algoritmo dará pasos muy pequeños, con lo que se quedará "estancado" muy pronto. Sin embargo, con una tasa de aprendizaje de 0.1, pasa algo totalmente distinto.

```
gradDesc(func=fb,eta = 0.1,showIter=T,pintarGr=T,nombre="Gradiente Descendiente con eta = 0.1")
```




```
##          x          x          y
##  1.6189421 -1.0327317 -0.4149053 50.0000000
```

En este caso, tenemos justo lo contrario. El valor η es muy grande y el algoritmo dará saltos de un lado para otro, siendo incapaz de converger en un mínimo, por lo que el valor de $f(x, y)$ en el mínimo, es mayor.

Punto de inicio	$f(x, y)$	x	y	iteraciones
(0.1,0.1)	-1.8200785	0.2438050	-0.2379258	30
(1,1)	0.5932694	1.2180703	0.7128120	27
(-0.5,-0.5)	-1.3324811	-0.7313775	-0.2378554	26
(-1,-1)	0.5932694	-1.2180703	-0.7128120	27

Visto esto, podemos ver que escoger bien el punto de inicio es bastante importante para encontrar el mínimo de la función, ya que el gradiente irá descendiento hasta que se encuentre un mínimo local, y encontrar este mínimo puede depender de dónde empecemos, como se puede ver en la tabla anterior.

1.2. 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 $\eta = 0,1$.

1. **¿Qué valor de la función $E(u, v)$ se obtiene después de 15 iteraciones completas (i.e. 30 pasos) ?**
2. **Establezca una comparación entre esta técnica y la técnica de gradiente descendente.**

A continuación podemos ver la implementación del algoritmo de la coordenada descendente:

```
coordinateDescent <- function(x = 1, y = 1, eta = 0.1, func, prec = 10^(-14),
    maxIter = 50, showIter = F, pintarGr=F, nombre = "Gradiente Descendiente"){
    df = Deriv(f=func,x=formalArgs(func))
    xOld = 0
    yOld = y
    nIter = 0
    xs = c()
    ys = c()
    while (abs(df(x,y)[1]) > prec & nIter < maxIter){
      # Almacenamos los valores de x e y en estas variables
      # auxilares para usarlas más adelante
        xOld = x
        xs = c(xs, x)
        ys = c(ys, y)
      # Calculamos el nuevo valor de X usando la derivada
        newValues = df(xOld,yOld)
        x = xOld - eta*newValues[1]
      # Calculamos el nuevo valor de Y usando la derivada,
      # pero con el valor nuevo de X
        newValues = df(x,yOld)
        y = yOld - eta*newValues[2]
        nIter = nIter+1
    }
    pts = cbind(xs,ys)
    if(pintarGr)
        pintar(puntos = pts, funcion = func, intervalo=c(-2,2), colores="red",
            verFuncion=T, nombreGrafica=nombre,k=1:20)
    if(!showIter)
        c(func(x,y),x, y)
    else
        c(func(x,y),x, y, nIter)
}
```

Para ver el resultado de este algoritmo al pasar 15 iteraciones, lo ejecutaremos de la siguiente forma:

```
coordinateDescent(func=fo,eta = 0.1,maxIter = 15,showIter = T)
```

```
##             u             u           v.u
##  1.545517e-49  6.564782e+01 -6.367828e+03  4.000000e+00
```

A su vez lo compararemos con el algoritmo anterior, usando la misma precisión y el mismo número de iteraciones:

```
gradDesc(func=fo,eta = 0.1,maxIter = 15,showIter = T)
```

```
##            u            u            v
## 6.026781e-27 4.473628e-02 2.395873e-02 1.500000e+01
```

Como se puede ver, el resultado de la coordenada descendente es mucho mejor que el gradiente descendiente clásico, ya que el mínimo que obtiene para 15 iteraciones, es de varios órdenes de magnitud menor que el del gradiente descendiente, para esa tasa de aprendizaje y número de iteraciones, pero con una tasa de aprendizaje menor y mayor número de iteraciones obtenemos lo siguiente:

```
coordinateDescent(func=fo,eta = 0.01,maxIter = 15,showIter = T)
```

```
##            u            u          v.u
## 1.018733e-04 4.655944e-01 9.999980e-01 1.500000e+01
```

```
gradDesc(func=fo,eta = 0.01,maxIter = 15,showIter = T)
```

```
##            u            u            v
##  0.001669682  0.470745574  0.842136751 15.000000000
```

```
coordinateDescent(func=fo,eta = 0.01,maxIter = 50,showIter = T)
```

```
##            u            u          v.u
## 2.813147e-16 4.630555e-01 1.000000e+00 5.000000e+01
```

```
gradDesc(func=fo,eta = 0.01,maxIter = 50,showIter = T)
```

```
##            u            u            v
## 1.873352e-11 4.586928e-01 8.427601e-01 5.000000e+01
```

Vemos que cambiando la tasa de aprendizaje, la coordenada descendente es capaz de minimizar más la función que el gradiente descendiente, ya que el mínimo que ha encontrado es menor que el gradiente descendiente en el mismo número de iteraciones, y si además, aumentamos aún más el número de iteraciones, el mínimo que encuentra es aún más pequeño, pero trabaja mucho mejor con una tasa de aprendizaje mayor.

1.3. 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.

1) Generar un gráfico de como desciende el valor de la función con las iteraciones.

2) 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.

La implementación del método de Newton la podemos ver a continuación:

```r
newtonMethod <- function(x = 0.1, y = 0.1,prec=10^{-14},
  func, maxIter = 50,showIter = T, pintarGr=F, nombre = "Metodo de Newton"){
  # Calculamos la derivada primera de la función
    df1 = Deriv(f=func,x=formalArgs(func))
  # Calculamos la derivada segunda de la función
    df2 = Deriv(f=func,x=formalArgs(func),nderiv=2)
  # Inicializamos los puntos y el contador de iteraciones
    xOld = 0
    yOld = 0
    xs = c()
    ys = c()
    nIter = 0
    itsTimeToStop1 = itsTimeToStop2 = itsTimeToStop3 = F
  # E iniciamos el bucle
    while(!itsTimeToStop1 & !itsTimeToStop2 & !itsTimeToStop3){
        xOld = x
        yOld = y

        # Calculamos el gradiente con la primera derivada
        newValues_1 = df1(xOld,yOld)

        # Calculamos el gradiente con la segunda derivada
        newValues_2 = df2(xOld,yOld)

        # Actualizamos los puntos
        xy = solve(matrix(newValues_2, ncol=2))%*%newValues_1

        x = xOld - xy[1,1]
        y = yOld - xy[2,1]


        if(abs(func(xOld, yOld) - func(x, y)) < prec){
          itsTimeToStop1 = T
          print("Me salgo porque estoy en un minimo local")
        }
        else if(nIter >= maxIter-1){
          itsTimeToStop2 = T
          print("Me salgo porque me he pasado de iteraciones")
        }
        else if(norm((as.matrix(df1(x,y))), type = "F") < prec){
          itsTimeToStop3 = T
          print("Me salgo porque he minimizado por debajo del umbral")
        }

        xs = c(xs, nIter)
        ys = c(ys, func(x,y))
        nIter = nIter + 1
    }
    if(pintarGr){
        pts = cbind(xs,ys)
        #pintar(puntos = pts, funcion = func, intervalo=c(-2,2), colores="red",
        #    verFuncion=T, nombreGrafica=nombre)
        plot(pts, type = "l")
```

```
    }
    if(!showIter)
        c(func(x,y),x, y)
    else
        c(func(x,y),x, y, nIter)
}
```

En la función, calculamos el método de Newton para minimizar la función, es decir, hacer que $\nabla f(x) = 0$. Para esto, realizamos el proceso iterativo del bucle haciendo que los pesos se actualicen de esta manera:

$$xy = \begin{pmatrix} x''_x & y''_x \\ x''_y & y''_y \end{pmatrix} \times \begin{pmatrix} x'_x \\ y'_y \end{pmatrix} \tag{1}$$

Teniendo las coordenadas que obtenemos con la segunda derivada como una matriz, podemos obtener las nuevas coordenadas como la diferencia de la coordenada anterior con el resultado de la operación anterior, quedando lo siguiente:

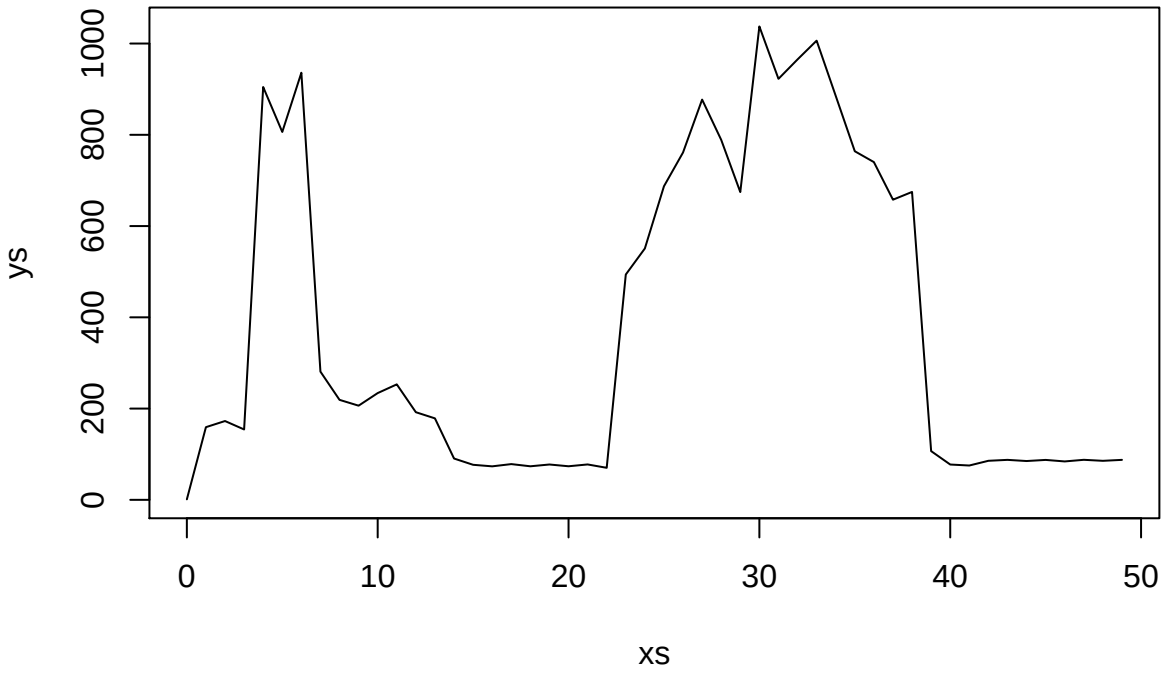
$$x_{new} = x_{old} - xy_{1,1}$$
$$y_{new} = y_{old} - xy_{2,1}$$

Con esto, el método de Newthon minimizará la función mientras que no hayamos encontrado un "mínimo local" o *flat zone*, hayamos llegado al máximo de iteraciones, o hayamos minimizado más que un ε.

Un ejemplo de ejecución para la función del ejercicio 1b es el siguiente, con los parámetros por defecto:

```
newtonMethod(func = fb, pintarGr = T)
```

```
## [1] "Me salgo porque me he pasado de iteraciones"
```



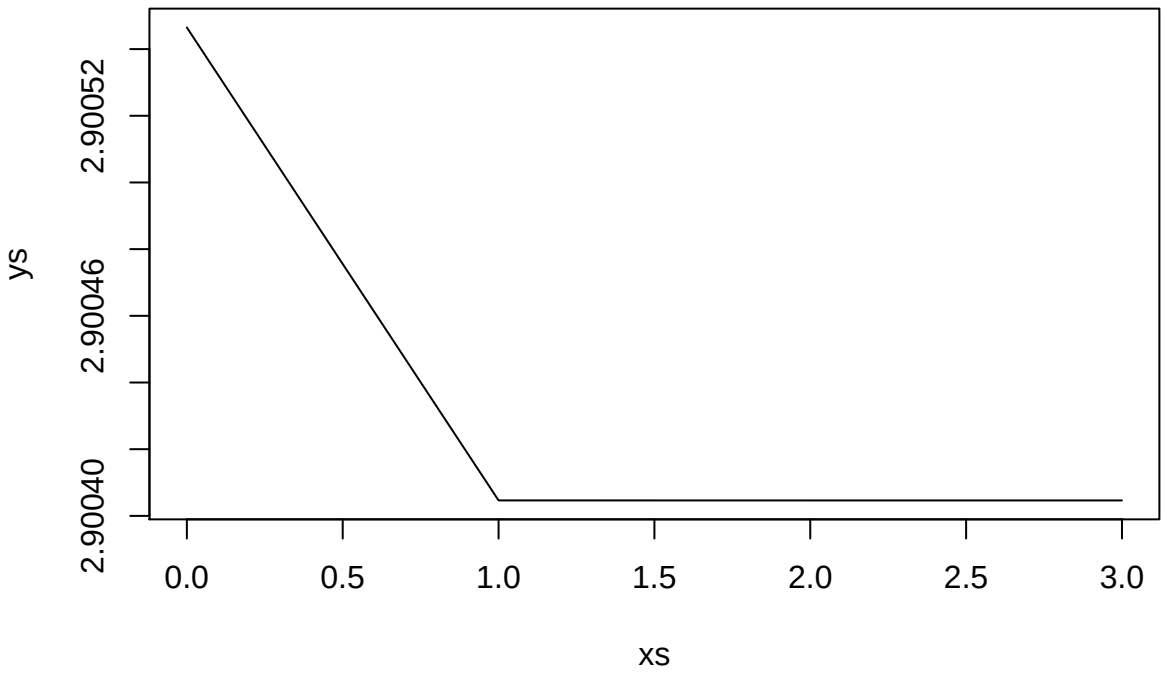

```
## [1] 87.618256 -9.079076 -1.752404 50.000000
```

En la gráfica, podemos ver como el algoritmo no es capaz de converger en ningún momento, ya que no cumple ninguna de las condiciones de parada, y se ve como el mínimo va "danzando" de un punto a otro de la función, hasta que llega al máximo de iteraciones con un valor de 87.62 aproximadamente, para un punto de inicio (0.1,0.1).

Para comparar con los otros algoritmos, vamos a mostrar la curva que realiza este algoritmo, y veremos los resultados más adelante en una tabla.

```
newtonMethod(x = 1, y = 1, func = fb, pintarGr = T, nombre = "P. inicio = (1,1)")
```

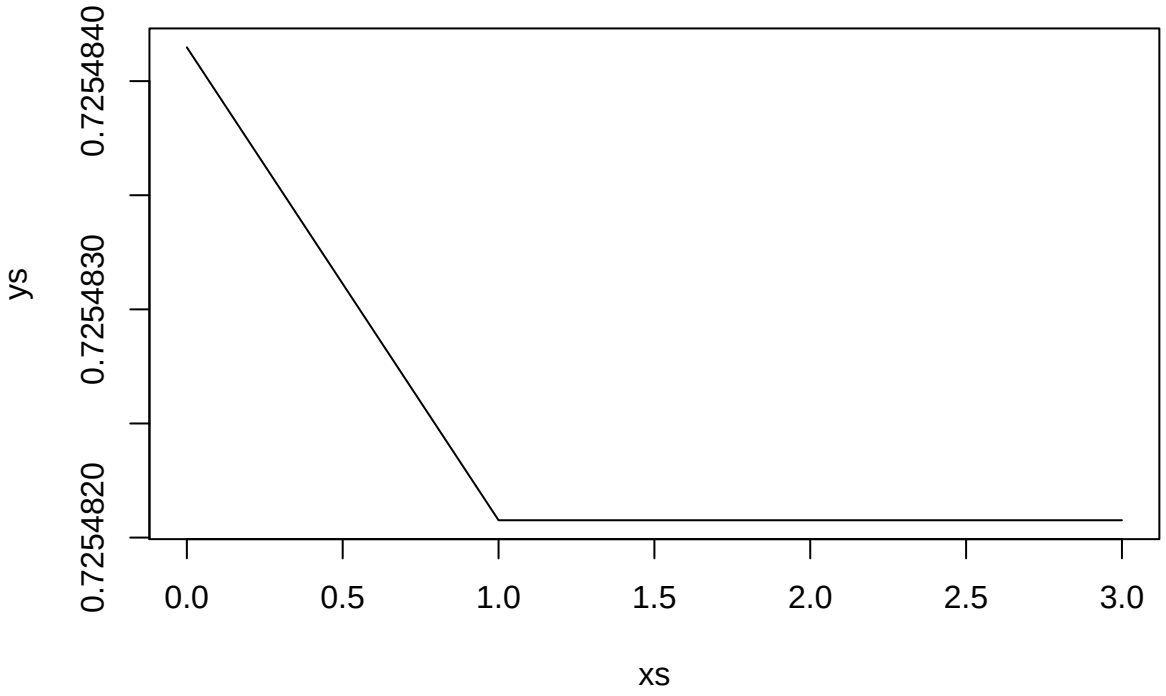
```
## [1] "Me salgo porque estoy en un minimo local"
```




```
## [1] 2.9004046 0.9491289 0.9745675 4.0000000
```

```
newtonMethod(x = -0.5, y = -0.5, func = fb, pintarGr = T, nombre = "P. inicio = (-0.5,-0.5)")
```

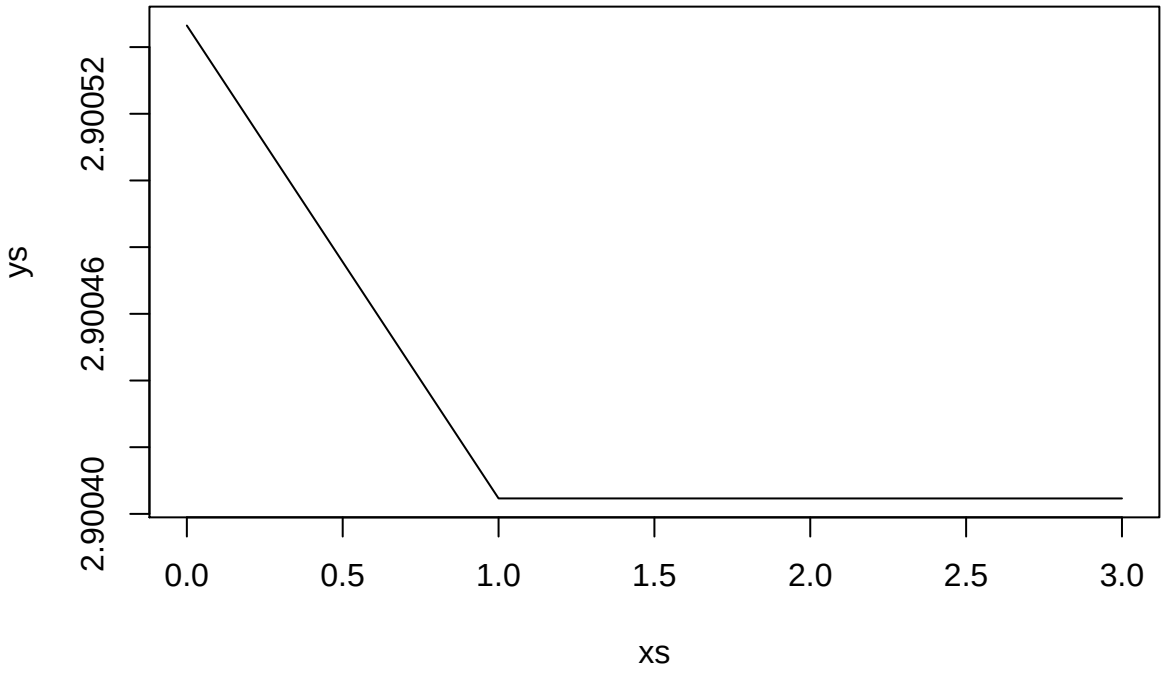
```
## [1] "Me salgo porque estoy en un minimo local"
```




```
## [1]  0.7254821 -0.4751135 -0.4878047  4.0000000
```

```
newtonMethod(x = -1, y = -1, func = fb, pintarGr = T, nombre = "P. inicio = (-1,-1)")
```

```
## [1] "Me salgo porque estoy en un minimo local"
```




```
## [1]  2.9004046 -0.9491289 -0.9745675  4.0000000
```

Punto de inicio	$f_{GD}(x, y)$	iteraciones	$f_{Nw}(x, y)$	iteraciones
(0.1, 0.1)	-1.8200785	17	87.618256	50
(1, 1)	0.5932694	12	2.9004046	4
(-0.5, -0.5)	-1.3324811	13	0.7254821	4
(-1, -1)	0.5932694	12	2.9004046	4

Como podemos ver en la tabla, el poder de minimización del método de Newton es un poco menor que el del gradiente descendiente, pero tiene la ventaja de que realiza menos iteraciones que el gradiente descendiente, y nos da un mínimo con cierta calidad, pero muy dependiente del punto de inicio de la función.

1.4. Regresión Logística

En este ejercicio crearemos nuestra propia función objetivo f (probabilidad en este caso) y nuestro conjunto de datos $\mathcal{D}$ 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. Consideremos $d = 2$ para que los datos sean visualizables, y sea $\mathcal{X} = [-1, 1] \times [-1, 1]$ con probabilidad uniforme de elegir cada $x \in \mathcal{X}$. Elegir una línea en el plano como la frontera entre $f(x) = 1$ (donde y toma valores $+1$) y $f(x) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos.

Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

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

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

b) Usar la muestra de datos etiquetada para encontrar g y estimar Eout (el error de entropía cruzada) usando para ello un número suficientemente grande de nuevas muestras.

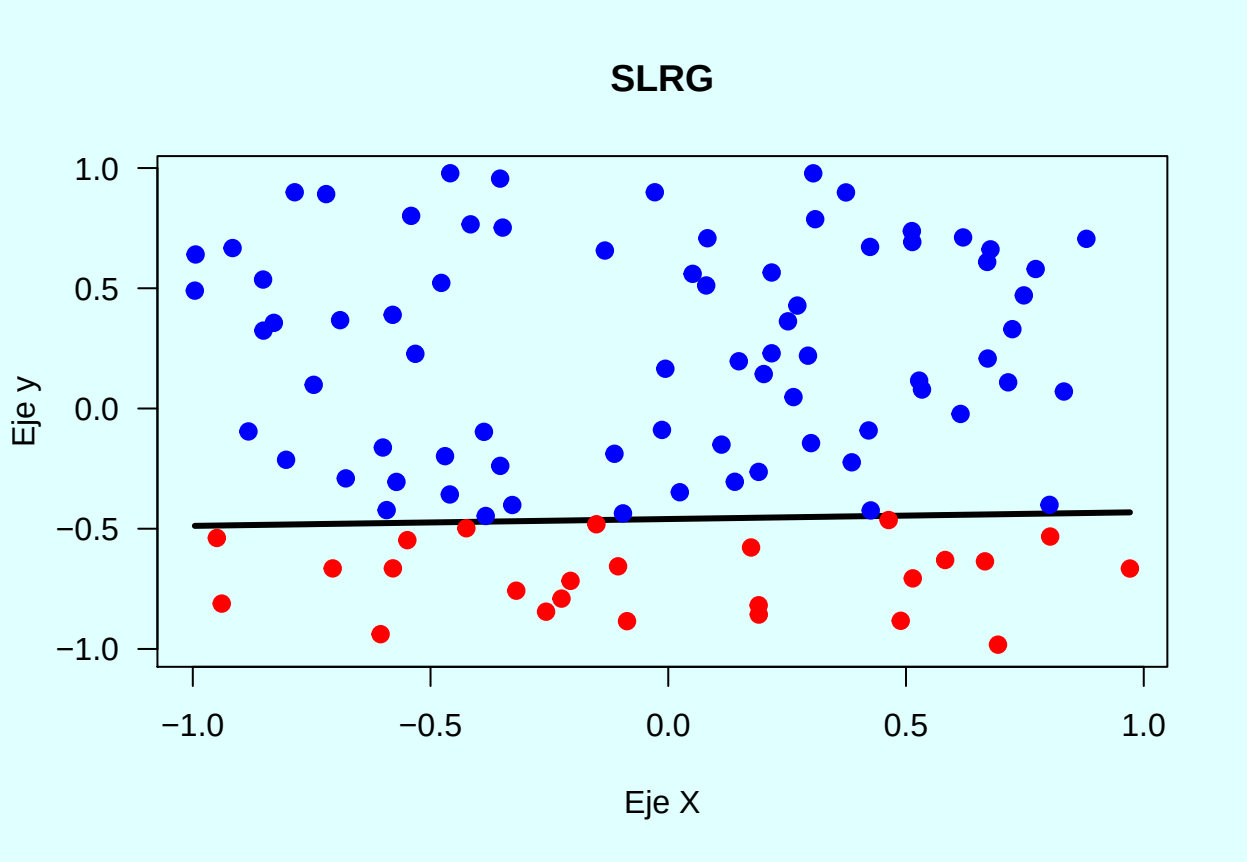
```
recta=simula_recta(dim=-1:1,verPunto=F)

logisticRegression <- function(eta = 0.01, max_iter=50,
    X = simula_unif(N=100, dim=2, rango=-1:1)) {
    w_old = c(1,1,1) # inicializamos el vector de pesos
    w_new = c(0,0,0)

    fp <- function(x,y) y - x*recta[1] - recta[2]
    f <- function(eval) eval[2] - recta[1]*eval[1] - recta[2]
    # evaluamos los puntos con respecto de la recta elegida
    Y = evaluaPuntos(puntos=X, func=f)
    # añadimos a la matriz X una tercera columma para poder hacer el producto vectorial
    i = 0
    YX = cbind(Y,X)
    # parada: modulo de la diferencia de los dos vectores
```

```
    while ((norm(as.matrix(w_old-w_new),type="F") >= 0.01 |
        norm(as.matrix(w_new),type="F") >= 0.01) & i < max_iter) {
        i = i + 1
        # hacemos una permutación aleatoria de los datos
        w_old = w_new
        sum_W = 0
        YX = YX[sample.int(nrow(YX)),]

        for (j in 1:nrow(YX)) {
            # calculamos el gradiente de error para cada punto con la fórmula
            # de la página 98 del libro de teoría
            aux = as.numeric(YX[j,])%*%(w_new)
            # actualizamos el peso usando la probabilidad de error de ese punto
            w_new = w_new - eta/nrow(YX) * (-as.numeric(YX[j,])/(1+exp(aux)))
        }

    }
    pintar(puntos = X, funcion = fp, colores=Y+3, intervalo = -1:1, nombreGrafica="SLRG",
    verFuncion = T, aniadir=FALSE, colFunc = "black")

    # Devolvemos los pesos y las iteraciones
    w_new
}
```

En el caso de la regresión logística, la función recibe una serie de puntos, que en cada generación, ajustará los pesos con cada dato. Cuando se cumpla alguna de las condiciones de parada, el algoritmo parará y delverá los pesos encontrados por la regresión.

Un ejemplo de ejecución es el siguiente:

```
w = logisticRegression()
```



En la imagen podemos ver cómo la regresión logística es capaz de separar datos que son linealmente separables sin problemas.

Para calcular la entropía cruzada, tenemos la siguiente función:

```
crossEntropyError <- function(w, X = simula_unif(N=100, dim=2, rango=-1:1),
    func = function(eval) eval[2] - recta[1]*eval[1] - recta[2]){
    Y = evaluaPuntos(puntos=X,func=func)
    X = cbind(Y, X)
    mean(apply(X = X, FUN = function(d) log(1 + exp(-d%*%w)), MARGIN=1))
}

crossEntropyError(w)
```

```
## [1] 0.6925099
```

Esta función se encarga de dado los pesos y los puntos para el cálculo del error, de evaluar los puntos con la función aprendida, y realizar el cálculo del error siguiendo lo siguiente:

$$E_{in}(w) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n w^T x_n}\right)$$

En ella podemos ver el valor que tenemos de entropía cruzada, que es de 0.69 aproximadamente, es decir, $E_{out} = 0,69$

1.5. 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.

1. **Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.**
2. **Calcular E_{in} y E_{test} (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 $\delta = 0,05$. ¿Qué cota es mejor?**
4. **Repetir los puntos anteriores pero usando una transformación polinómica de tercer orden ($\Phi_3(x)$ en las transparencias de teoría).**
5. **Si tuviera que usar los resultados para dárselos a un potencial cliente ¿usaría la transformación polinómica? Explicar la decisión.**

Para realizar el ejercicio, nos hemos basado en las funciones para leer un fichero de la práctica anterior, más aquellas que calculan la media y simetría de los distintos dígitos. Tras esto, realizamos la regresión lineal para separar los puntos y obtendremos unos pesos ω. Estos pesos, se pasarán como entrada al PLA pocket y obtendremos la separación que se ve en en el siguiente gráfico, en 3 iteraciones.

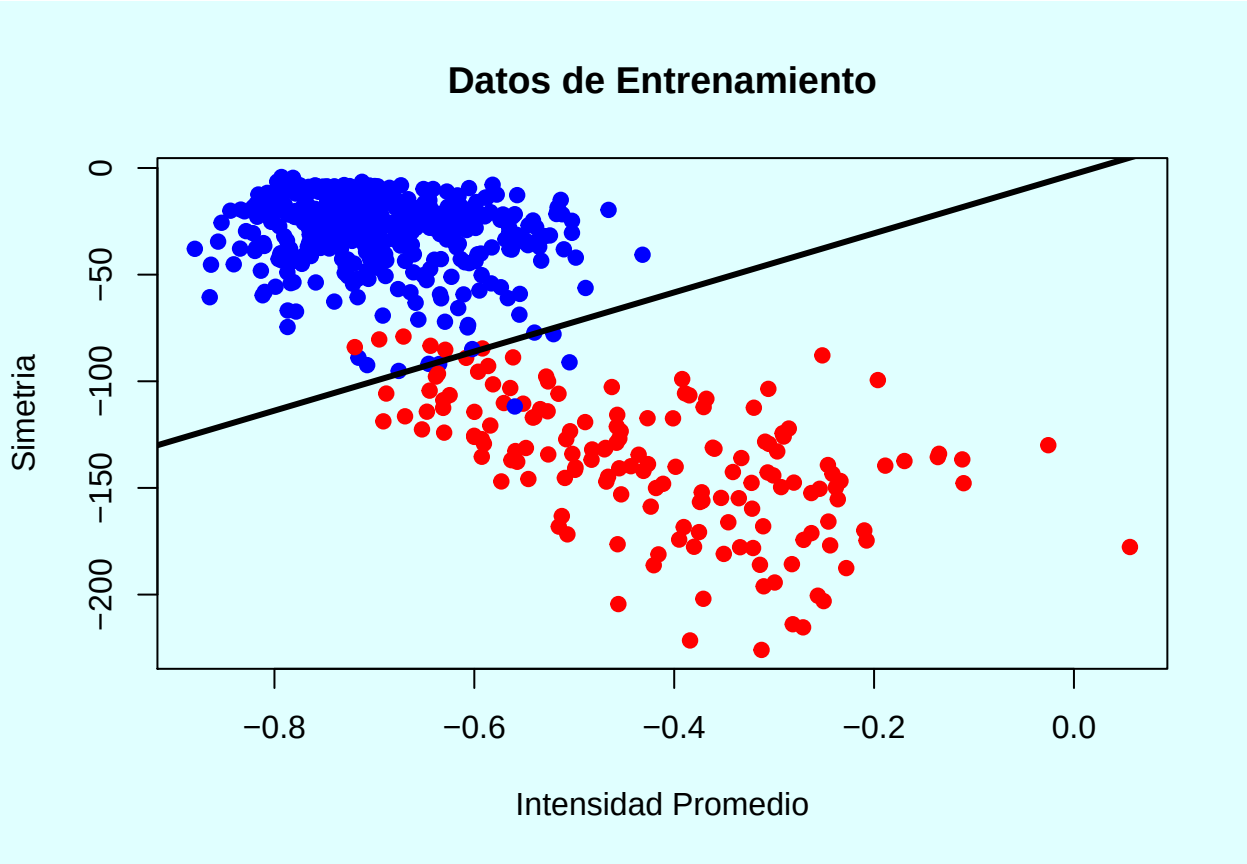
```
g = readFile()
mediaSim = intensityAndSymmetry(g[[2]])
etiquetas = g[[1]]
etiquetas[etiquetas == 1] = "blue"
etiquetas[etiquetas == -1] = "red"

reg=Regress_Lin_Effic(mediaSim[,2], mediaSim[,1])

par(bg="lightcyan")
plot(mediaSim,xlab="Intensidad Promedio",ylab="Simetria",
     pch=19,col=etiquetas, main="Datos de Entrenamiento")
perceptron = PLA(datos = mediaSim, label = g[[1]], vini = c(reg,1),
            verRecorrido = F)
```

```
## Antiguo pocentaje de error =  0.7378965
## Nuevo porcentaje de error =  0.2621035
```

```
print(perceptron)
```

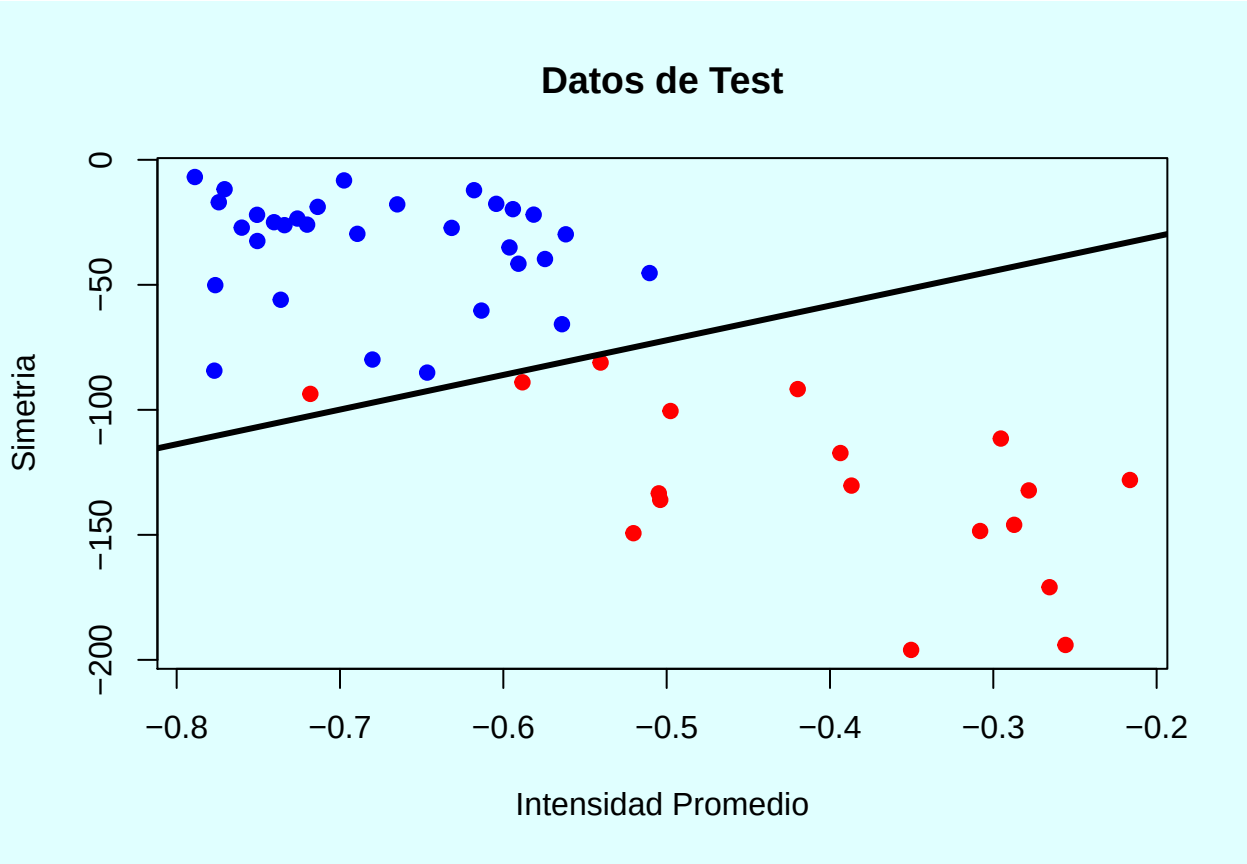
```
##               intensity    symmetry
##   1.0000000  -2.8910459 138.5877615   3.0000000   0.2621035
```

```
abline(a=perceptron[2], b=perceptron[3],col="black",lwd=3)
```



Aquí podemos ver los datos para el conjunto de datos de *train*, mientras que los de *test*, los podemos ver a continuación.

```
# Cargamos el fichero de test
gTest = readFile(fileName = "./DigitosZip/zip.test")
# Calculamos la media y la simetría de los datos
mediaSimTest = intensityAndSymmetry(gTest[[2]])
# Y usamos las etiquetas para dar color a los puntos y
# diferenciarlos en el gráfico
etiquetasTest = gTest[[1]]
etiquetasTest[etiquetasTest == 1] = "blue"
etiquetasTest[etiquetasTest == -1] = "red"
# Pintamos los putnos
par(bg="lightcyan")
plot(mediaSimTest,xlab="Intensidad Promedio",ylab="Simetria",
     pch=19,col=etiquetasTest,main="Datos de Test")
abline(a=perceptron[2], b=perceptron[3],col="black",lwd=3)
```



Como podemos ver, el algoritmo ha aprendido bastante bien y el error de test es bastante pequeño a simple vista, pero podemos analizarlo más en profundidad. Para calcular el error dentro de la muestra, calcularemos el error dentro de la muestra como:

$$\begin{aligned} E_{in}(w) &= \frac{1}{N}\sum_{n=1}^{N}(w^T x_n - y_n)^2 \\ &= \frac{1}{N}||Xw - y||^2 \\ &= \frac{1}{N}(w^T X^T X w - 2w^T X^T y + y^T y) \end{aligned}$$

En resumen, el error dentro de la muestra se calcula como la norma de la diferencia de los puntos multiplicados por los pesos y las etiquetas. Para calcular el error fuera de la muestra, partiremos del error calculado dentro de la muestra. Este error fuera de la muestra o E_{out} se obtiene como

$$E_{out} = E_{in} + \mathcal{O}\Big(\frac{d}{N}\Big)$$

donde d es la dimensión VC y N el tamaño del conjunto de datos.

```
# Función para calcular el error fuera de la muestra
calcularEout <- function(ein, datos, tam){
    Eout = ein + (ncol(datos))/tam
    Eout
}
```

```r
# Función para calcular el error dentro de la muestra
calcularEin <- function(X, wlin, y){
    errorIN = (t(wlin)%*%t(X)%*%X%*%wlin - 2*t(wlin)%*%t(X)%*%y + t(y)%*%y)
}

y_g = g[[1]]
y_test = gTest[[1]]

datos=cbind(mediaSim,y_g)
ein = apply(X=datos, FUN=function(dato)
  calcularEin(X=cbind(dato[1],dato[2]), wlin = reg, y = dato[3]),MARGIN=1)
Ein_train=mean(ein)
Eout_train=calcularEout(Ein_train, datos, nrow(datos))
cat("Ein para datos de train = ",Ein_train,"\n")
```

```
## Ein para datos de train =  0.8405077
```

```r
cat("Eout para datos de train = ",Eout_train,"\n")
```

```
## Eout para datos de train =  0.8455161
```

```r
datosTest=cbind(mediaSimTest,y_test)
einT = apply(X=datos, FUN=function(dato)
  calcularEin(X=cbind(dato[1],dato[2]), wlin = reg, y = dato[3]),MARGIN=1)
Ein_Test=mean(einT)
Eout_Test=calcularEout(Ein_Test, datosTest, nrow(datosTest))
cat("Ein para datos de test = ",Ein_Test,"\n")
```

```
## Ein para datos de test =  0.8405077
```

```r
cat("Eout para datos de test = ",Eout_Test,"\n")
```

```
## Eout para datos de test =  0.9017322
```

Como podemos ver, para los datos de entrenamiento, tenemos que $E_{in} = 0,8405077$ y $E_{out} = 0,8455161$, y para los datos de test, tenemos que $E_{in} = 0,8405077$ y $E_{out} = 0,9017322$.

Para el cálculo de las cotas, tenemos que recurrir a la siguiente expresión:

$$E_{out} \leq E_{in}(g) + \sqrt{\frac{8}{N} \ln\left(\frac{4m_{\mathcal{H}}(2N)}{\delta}\right)}$$

Si usamos el límite polinómico basado en la dimensión de Vapnik–Chervonenkis, obtenemos la siguiente expresión, similar a la anterior, pero también válida para calcular el error fuera de la muestra

$$E_{out} \leq E_{in}(g) + \sqrt{\frac{8}{N} \ln\left(\frac{4(2N)^{d_{VC}} + 1}{\delta}\right)}$$

Como en los modelos lineales, la dimensión VC es 3, podemos simplificar un poco la expresión y desarrollar la siguiente función en R, capaz de calcular este error

```
EoutBound <- function(Ein, N, delta = 0.05, d = 3){
    Eout = Ein + sqrt( (8/N) * log( (4*(2*N)^(d+1) + 1)/delta) )
    Eout
}
```

Con esto, vamos a ver qué error obtenemos con los datos de entrenamiento y los datos de test.

```
cat("Eout real para datos de train = ",EoutBound(Ein_train, nrow(datos)),"\n")
```

```
## Eout real para datos de train =  1.501722
```

```
cat("Eout real para datos de train = ",EoutBound(Ein_Test, nrow(datosTest)),"\n")
```

```
## Eout real para datos de train =  2.766565
```

Con esto, podemos ver que los resultados para los datos de entrenamiento son $E_{out} = 1,425772$, y para los datos de test, $E_{out} = 2,561302$. Podemos ver que el error, es bastante bueno para usar un modelo lineal, y hemos conseguido ajustar un buen modelo, que se comporta bien tanto dentro, como fuera de la muestra.

2. Sobreajuste

Vamos a construir un entorno que nos permita experimentar con los problemas de sobreajuste. Consideremos el espacio de entrada $\mathcal{X} = [-1, 1]$ con una densidad de probabilidad uniforme, $\mathbb{P}(x) = \frac{1}{2}$. Consideramos dos modelos $\mathcal{H}_2$ y $\mathcal{H}_{10}$ representando el conjunto de todos los polinomios de grado 2 y grado 10 respectivamente. La función objetivo es un polinomio de grado Q_f que escribimos como $f(x) = \sum_{q=0}^{Q_f} a_q L_q(x)$, donde $L_q(x)$ son los polinomios de Legendre (ver la relación de ejercicios.2). El conjunto de datos es $\mathcal{D} = (x_1, y_1), \ldots, (x_N, y_N)$ donde $y_n = f(x_n) + \sigma \cdot \epsilon_n$ y las $\{\epsilon_n\}$ son variables aleatorias i.i.d. $\mathcal{N}(0, 1)$ y σ^2 la varianza del ruido. Comenzamos realizando un experimento donde suponemos que los valores de Q_f, N, σ, están especificados, para ello:

- **Generamos los coeficientes a_q a partir de muestras de una distribución $\mathcal{N}(0, 1)$ y escalamos dichos coeficientes de manera que $\mathbb{E}_{a,x}[f^2] = 1$ (Ayuda: Dividir los coeficientes por $\sqrt{\sum_{q=0}^{Q_f} \frac{1}{2q+1}}$).**
- **Generamos un conjunto de datos, $x_1, \ldots, x_N$ muestreando de forma independiente $\mathbb{P}(x)$ y los valores $y_n = f(x_n) + \sigma\epsilon_n$.**

Sean g_2 y g_{10} los mejores ajustes a los datos usando $\mathcal{H}_2$ y $\mathcal{H}_{10}$ respectivamente, y sean $E_{out}(g_2)$ y $E_out(g_{10})$ sus respectivos errores fuera de la muestra.

a) Calcular g_2 y g_{10}

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

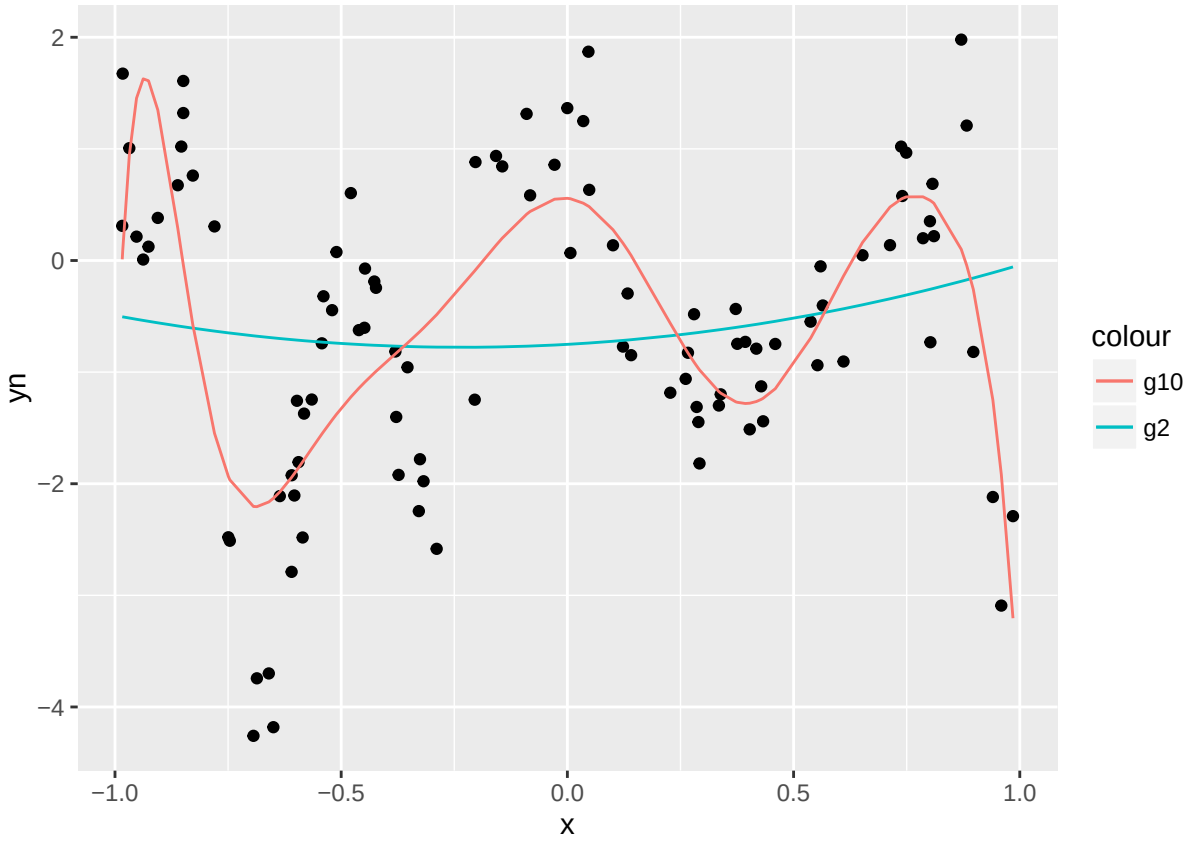
c) ¿Cómo podemos obtener E_{out} analíticamente para una g_{10} dada?

Para el desarrollo del ejercicio, he creado la siguiente función en código R, que generará todo lo necesario para el ejercicio.

```r
sobreajuste <- function(intervalo=-1:1, prob=0.5, N=100, sigma=1, Qf=15, ejercicio1=TRUE) {
    # Calculamos los coeficientes pertenecientes a la
    # distribución de probabilidad con media = 1/2
    aq = as.vector(simula_gaus(N=Qf+1, dim=1, sigma=1))
    # Aplicamos a los coeficientes la siguiente función
    #
    #                          a
    #    a =   -----------------------------
    #                 ----------------------
    #              /  ---- Qf      1
    #             /   \         ---------
    #            V    /___ q=0   2q+1
    #
    d = sqrt(sum(sapply(X=0:Qf, FUN=function(q) 1/(2*q+1))))
    aq = aq/d
    # Y realizamos el cálculo de la función objetivo, que consiste en
    #
    #               ---- Qf
    #        f =    \          aqLq(x)
    #               /___ q=0
    #
    legendre = legendre.polynomials(n=Qf, normalized=T) # generamos los 20 primeros
    f = legendre[[1]]*aq[1]
    for (i in 2:length(legendre))
        f = f + legendre[[i]] * aq[i]
    # generamos los datos
    x = as.vector(simula_unif(N=N, dim=1, rango=intervalo))
    # generamos epsilon para meter ruido
    eps = as.vector(simula_gaus(N=N, dim=1, sigma=0.5))
    # Generamos la función objetivo
    Y = sapply(X=x, FUN=as.function(f))
    # 2. le sumamos a los Y calculados, sigma*epsilon
    yn = mapply(FUN=function(valor, en) valor + sigma*en, valor=Y, en=eps)
    # generamos la matriz de datos
    datos = cbind(x, yn)
    # Transformamos los datos para realizar la regresión
    datos_predecir = poly(x, degree=Qf)
    # Realizamos la regresión para g2 y g10
    model.g2 <- lm(yn ~ poly(x, degree=2),data=datos_predecir)
    model.g10 <- lm(yn ~ poly(x, degree=10),data=datos_predecir)
    # Almacenamos los pesos
    w2 = as.vector(model.g2$'coefficients')
    w10 = as.vector(model.g10$'coefficients')
    # Calculamos el error dentro de la muestra, similar al
    # error de entropía  cruzada
    EinNL <- function(datos, w, etiquetas){
        w=as.matrix(w)
        datos = cbind(rep(1, nrow(datos)), datos)
        y_z = apply(X=datos, FUN=function(dato) t(w)%*%dato, MARGIN=1)
        mean(mapply(FUN = function(yz, y) log(1 + exp(-y * yz)), yz = y_z, y = etiquetas))

    }
    # Calculamos los errores dentro de la muestra
```

```
    eing2 = EinNL(poly(x, degree=2), w2, yn)
    eing10 = EinNL(poly(x, degree=10), w10, yn)
    # Y los errores fuera de la muestra
    eoutg2 = EoutBound(eing2, N=nrow(datos), delta = 0.05, d = ncol(poly(x,degree=2)))
    eoutg10 = EoutBound(eing10, N=nrow(datos), delta = 0.05, d = ncol(poly(x,degree=10)))

    if(ejercicio1){
        cat("Ein_g2 = ",eing2,"\tEout_g2 = ",eoutg2,"\n")
        cat("Ein_g10 = ",eing10,"\tEout_g10 = ",eoutg10,"\n")
        cat("Eout_g2 - Eout_g10 = ", eoutg2 - eoutg10,"\n")

        (ggplot() + geom_point(data=as.data.frame(datos), aes(x=x, y=yn))
          + geom_line(aes(x=x, y=predict(model.g2), color="g2"))
          + geom_line(aes(x=x, y=predict(model.g10), color="g10")))
    }
    else
            l = c(eing2, eoutg2, eing10, eoutg10)
}

sobreajuste()
```

```
## Ein_g2 =  0.5972946  Eout_g2 =  1.870934
## Ein_g10 =  0.4227431     Eout_g10 =  2.661734
## Eout_g2 - Eout_g10 =  -0.7907999
```



La función generará las coeficiente y los escalará siguiendo lo que se dice en el enunciado. Tras esto, usando el paquete ***orthopolynom***, generamos los polinomios de Legendre normalizados, para, realizando la suma de

los polinomios, obtener la función objetivo que queremos obtener, a la que posteriormente, añadiremos el ruido a la función.

Tras esto, evaluamos los puntos que hemos sacado aleatoriamente para obtener las etiquetas reales. Una vez hecho esto, realizamos la regresión con la transformación de los datos para obtener los pesos, y pasamos a calcular los errores dentro de la muestra para el modelo g_2 y el modelo g_{10}.

Como podemos ver, dentro de la muestra, g_{10} se comporta mejor que g_2, debido a que $E_{in_{g_{10}}} < E_{in_{g_2}}$, por lo que se ajusta mejor a los datos. Pero, al calcular el error fuera de la muestra, vemos que si realizamos la diferencia de $E_{in_{g_2}} - E_{in_{g_{10}}}$ vemos que es menor que 0. Esto significa que g_{10} es tan bueno dentro de la muestra que está sobreajustando los datos, con lo que el comportamiento fuera de la muestra será peor.

Respecto a por qué normalizar f, tenemos que en la función objetivo, existe un ruido σ que afecta a la función. Normalizando la función, podemos hacer que el ruido sea "el mismo" para poder acotarlo y ajustar bien la función sin que la aleatoriedad del ruido nos afecte a la función. Con esto controlamos el ruido estocástico de la función.

2.1. Medición del sobreajuste

Siguiendo con el punto anterior, usando la combinación de parámetros $Q_f = 20$, $N = 50$, $\sigma = 1$ ejecutar un número de experimentos (> 100) calculando en cada caso $E_out(g_2)$ y $E_out(g_{10})$. Promediar todos los valores de error obtenidos para cada conjunto de hipótesis, es decir

$$E_{out}(\mathcal{H}_2) = \textbf{promedio sobre experimentos}(E_{out}(g_2))$$

$$E_{out}(\mathcal{H}_{10}) = \textbf{promedio sobre experimentos}(E_{out}(g_{10}))$$

Definimos una medida de sobreajuste como $E_{out}(\mathcal{H}_{10}) - E_{out}(\mathcal{H}_2)$.

1. **Argumentar por qué la medida dada puede medir el sobreajuste.**
2. **Usando la combinación de valores de los valores $Q_f \in \{1, 2, \ldots, 100\}$, $N \in \{20, 25, \ldots, 100\}$, $\sigma \in \{0; 0,05; 0,1; \ldots; 2\}$, 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).**

Para el apartado 1, he usado el siguiente código:

```
ejercicio22 <- function(){
    EoutH2 = c()
    EoutH10 = c()

    for(i in 1:150){
        l = sobreajuste (intervalo=-1:1, prob=0.5, N=50, sigma=1, Qf=15, ejercicio1=F)
        # Insertamos los errores fuera de la muestra en cada una de las listas
        EoutH2 = c(l[[2]],EoutH2)
        EoutH10 = c(l[[4]],EoutH10)
    }
    # Restamos las medias de los errores para los dos modelos
    mean(EoutH10) - mean(EoutH2)
}
```

Haciendo uso del ejercicio anterior, realizamos 150 experimentos, para obtener los errores en media de los modelos $\mathcal{H}_2$ y $\mathcal{H}_{10}$, y después realizamos la diferencia de la media de errores de ambos modelos, para medir el sobreajuste. El resultado de esta función lo podemos ver a continuación:

```
ejercicio22()
```

```
## [1] 1.00512
```

Como podemos ver, la diferencia de medias ha salido positiva. Esto quiere decir que el error fuera de la muestra para $\mathcal{H}_{10}$ es mayor que el error de $\mathcal{H}_2$. Con esto podemos ver que $\mathcal{H}_{10}$ está sobreajustando el modelo, ya que al ser un modelo más complejo y flexible, puede ajustar los puntos y el ruido que tienen los puntos en la muestra, con lo que el error dentro de la muestra es menor que para $\mathcal{H}_2$, mientras que, si nos fijamos en el error fuera de la muestra, $\mathcal{H}_{10}$ trabaja peor que $\mathcal{H}_2$. Por lo tanto, cuanto mayor sea esta diferencia (con valor positivo), más sobreajuste produce el modelo $\mathcal{H}_{10}$.

Respecto al apartado b, podemos ver que cuanto mayor sea el nivel de ruido σ, tendremos un mayor sobreajuste cuanto menor sea el conjunto de datos que tenemos para entrenar, ya que no podemos detectar bien el ruido, teniendo un mayor error estocástico. En la segunda imagen, vemos que cuanto más crece la complejidad del modelo, añadimos ruido determinístico a los datos si no aumentamos el tamaño del conjunto de datos para entrenar.

En resumen, tenemos lo siguiente:

Número de datos	Ruido	Complejidad del modelo	Sobreajuste
$\uparrow$	$\rightarrow$	$\rightarrow$	$\downarrow$
$\downarrow$	$\rightarrow$	$\rightarrow$	$\uparrow$
$\rightarrow$	$\uparrow$	$\rightarrow$	$\uparrow$
$\rightarrow$	$\rightarrow$	$\uparrow$	$\uparrow$

3. Regularización y selección de modelos

3.1. Regularización "weight decay"

Para $d = 3$ (dimensión) generar un conjunto de N datos aleatorios $\{x,y_n\}$ de la siguiente forma. Para cada punto, x_n generamos sus coordenadas muestreando de forma independiente una $\mathcal{N}(0,1)$. De forma similar generamos un vector de pesos de $d+1$ dimensiones w_f, y el conjunto de valores $y_n = w_f^T x_n + \sigma\epsilon_n$, donde ϵ_n es el ruido que sigue también una $\mathcal{N}(0,1)$ y σ^2 es la varianza del ruido; fijar $\sigma = 0,5$.

Usar regresión lineal con regularización "weight decay" para estimar w_f con w_{reg}. Fijar el parámetro de regularización a $0,05/N$.

a) Para $N \in \{d+15, d+25, \cdots, d+115\}$ calcular los errores $e_1, \cdots, e_N$ de validación cruzada y E_{cv}.

b) Repetir el experimento 1000 veces, anotando el promedio y la varianza de e_1, e_2 y E_{cv} en todos los experimentos.

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

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

e) Si los errores de validación-cruzada fueran verdaderamente independientes, ¿cuál sería la relación entre la varianza de los valores de e_1 y la varianza de los de E_{cv}?

f) Una medida del número efectivo de muestras nuevas usadas en el cálculo de E_{cv} es el cociente entre la varianza de e_1 y la varianza de E_{cv}. Explicar por qué, y dibujar, respecto de N, el número efectiv de nuevos ejemplos (N_{eff}) como un porcentaje de N. NOTA: Debería de encontrarse que N_{eff} está cercano a N.

g) Si se incrementa la cantidad de regularización, ¿debería N_{eff} subir o bajar? Argumentar la respuesta. Ejecutar el mismo experimento con $\lambda = 25/N$ y comparar los resultados del punto anterior para verificar la conjetura.

a) Para este apartado, he desarrollado la siguiente función

```
# Función para implementar la regresión con "weight decay"
weightDecay <- function(datos, y, lambda = 0.05/nrow(datos)){
    beta = solve(t(datos)%*%datos+lambda*diag(ncol(datos)))%*%t(datos)%*%y
}

# Función para calcular el error cuadrático
EinSquareError<-function(w, x, y){
    w=as.matrix(w)
    (t(w)*x-y)*(t(w)*x-y)
}


ejercicio3 <- function(N = 100, varianza = 1, lambda = 0.5, d = 2){
    # Generamos el vector aleatorio de pesos
    w = as.vector(simula_gaus(N=1, dim=d+1, sigma = 1))
    # Y una lista vacía para almacenar los errores
    CVErrorsList = c()
    e1 = c()
    e2 = c()
    # La lista (d+15,d+25,...,d+115)
    Ns = seq(from = 15, to = 115, by = 10)
    # Iniciamos la validación cruzada
    for (i in Ns){
        # Con el nuevo conjunto de datos de tamaño Ns_i + d
        n = i+d
        # Generamos una nueva muestra, el ruido y los evaluamos en función
        # de los pesos que tenemos
        newDS = simula_gaus(N=n, dim=d, sigma = 1)
        newDSaux=cbind(rep(1,n),newDS)
        epsilon = as.vector(simula_gaus(N=n, dim=1, sigma=1))
        ynDS = apply(X = newDSaux, FUN = function(dato) dato%*%w,MARGIN=1)
        ynDS = mapply(FUN=function(e,y) y + e*lambda/n, e = epsilon, y=ynDS)
        # Tras esto, aplicamos la validación cruzada e insertamos el error
        # en la lista CVErrorsList
        ei = sapply(X=1:n,FUN=function(k) EinSquareError(
            as.vector(weightDecay(newDSaux[-k,],ynDS[-k])),
            x=newDSaux[k,], y = ynDS[k]))
        CVErrorsList = c(mean(ei), CVErrorsList)
        e1 = c(ei[1],e1)
        e2 = c(ei[2],e2)
    }
    list(CVErrorsList,e1,e2)
}
```

En la función, calculamos el *weight decay* o w_{reg} de la siguiente forma:

$$w_{reg} = (Z^T Z + \lambda I)^{-1} Z^T y$$

. Con este peso que obtenemos, pasamos a calcular el error de validación cruzada, donde para dado un conjunto $\mathcal{D}_n = (x_1, y_1), \ldots, (x_N, y_N)$, entrenamos el model con $N-1$ puntos, y usaremos uno para validar. En concreto, hacemos lo siguiente

$$\mathcal{D}_n = (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}), \cancel{(x_n, y_n)}, (x_{n+1}, y_{n+1}), \ldots, (x_N, y_N)$$

donde (x_n, y_n) es el dato que usaremos para validar la función g^- calculada.

Con esto, cada función g^- tendrá un error e_n, que podemos calcular de la siguiente manera:

$$e_n = E_{val}(g_n^-) = e(g_n^-(x_n), y_n)$$

. Teniendo esto en cuenta, la validación cruzada lo que hará será calcular el error en media para g^-, es decir

$$E_{cv} = \frac{1}{N} \sum_{n=1}^{N} e_n$$

```
ejercicio3()[1]
```

```
## [[1]]
##  [1] 1.560291 1.707365 1.693381 1.829736 1.601042 1.522263 1.420277
##  [8] 1.311996 1.670578 1.285179 2.143937
```

En la función devolvemos la lista de errores de validación cruzada para los distintos tamaños de N.

b) En este caso, tenemos el siguiente código

```
ejercicio3b <- function(times=1000) {
    resultados = matrix(ncol=6, nrow=times)

    for (i in 1:times) {
        l = ejercicio3()
        resultados[i,] = c(mean(l[[1]]), var(l[[1]]),
            mean(l[[2]]), var(l[[2]]),mean(l[[3]]), var(l[[3]]))
    }

    colnames(resultados)=c("Media Ecv", "Varianza Ecv", "Media e_1",
        "Varianza e_1", "Media e_2", "Varianza e_2")
    resultados
}

ejercicio3b(10)
```

```
##       Media Ecv Varianza Ecv   Media e_1 Varianza e_1  Media e_2
##  [1,] 5.8412744 0.5159112158 13.37417057 3.479162e+02 4.22307471
##  [2,] 0.6877315 0.0004154399  0.05969877 5.397676e-03 0.75068037
##  [3,] 1.3267779 0.0480440590  0.61410816 7.661655e-01 0.78097803
##  [4,] 0.5079017 0.0054647997  0.52122437 3.835033e-01 0.48903968
##  [5,] 0.3354674 0.0019638974  0.33860299 3.947228e-01 0.46225901
##  [6,] 1.5636645 0.0956919270  0.48339720 4.302156e-01 1.20279097
##  [7,] 1.1098919 0.0163122663  2.31439282 8.791187e+00 2.25584790
##  [8,] 0.6561061 0.0041322919  0.20728391 8.539185e-02 0.42383954
##  [9,] 0.2988033 0.0004600530  0.07457699 4.240038e-03 0.31622676
```

```
## [10,] 2.0160951 0.2679960316  3.44081461 1.096485e+01 0.07467383
##       Varianza e_2
##  [1,]  56.22896624
##  [2,]   0.12428440
##  [3,]   1.01700316
##  [4,]   0.25107516
##  [5,]   0.13542393
##  [6,]   0.75977164
##  [7,]   6.69156664
##  [8,]   0.08605573
##  [9,]   0.03265576
## [10,]   0.01041953
```

c) Por simplicidad, se ha realizado el experimento 10 veces, para poder apreciar bien los datos. La relación que existe entre los datos e_1, e_2 y E_{cv} es bastante fuerte ya que, para evaluar E_{cv}, haremos el promedio de las distintas evaluaciones e_n. Estas evaluaciones, en el caso de e_1, usaremos para aprender todos los puntos menos (x_1, y_1) que lo usaremos para validar. En los puntos que usamos para aprender, se encuentra (x_2, y_2), que en el caso de e_2 se usará para validar, mientras que (x_1, y_1) se usará para aprender. Por lo que existe una relación entre ambos en la media del error, siendo parecido el error, y la variabilidad entre los resultados será puede ser grande, al evaluar solo con un dato, mientras que con la validación cruzada lo hacemos con N datos, tal y como se puede ver en los resultados.

c) La varianza puede llegar a ser muy alta, ya que el conjunto de datos que usamos para aprender es bastante pequeño, con lo que el error fuera de la muestra, crecerá bastante, y tendrá una alta variabilidad.

c) Si los datos fueran realmente independientes, obtendríamos una varianza mayor, ya que los resultados dependerían del punto que cojamos, si es completamente aleatorio, pero obtendríamos errores más representativos.